UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Skjemaekstraksjon fra MongoDBs dokumentdatabaser

Lan Anh Vu

Masteroppgave, våren 2017

Skjemaekstraksjon fra MongoDBs dokumentdatabaser

Lan Anh Vu

Forord

Denne masteroppgaven er skrevet som en del av masterprogrammet Informatikk: Programmering og Nettverk ved Universitetet i Oslo.

Jeg vil takke veilederen min Ellen Munthe-Kaas, som har støttet meg gjennom arbeidet med denne oppgaven. I tillegg til å være veilederen min har Ellen vært en stor inspirasjon for min interesse innenfor databasefeltet.

Jeg vil takke familie og venner for støtten, og spesielt foreldrene mine som har vist sin tålmodighet og omsorg under hele denne perioden. En spesiell takk til Joakim Kristiansen for korrekturlesing av oppgaven.

Til slutt vil jeg takke min lille Pika for alle timene som *rubber duck*.

Universitetet i Oslo, April 2017
Lan Anh Vu

Sammendrag

I denne oppgaven undersøker vi metoder for å ekstrahere skjemaer fra dokumentdatabaser i MongoDB. Dokumentdatabaser har vokst frem som et populært alternativ til relasjonsdatabaser, og har fått stor medvind de siste årene. Vi skal se på problematikk rundt valget av dokumentdatabaser og MongoDB som databaseteknologi, og se på grunner til å migrere til et relasjonssystem.

Oppgaven ser på hvordan vi kan trekke ut skjemainformasjon på et konseptuelt nivå, som kan brukes videre i forarbeidet til en databasemigrering. Vi bruker *Object-Role Model (ORM)* som formalisme for dette. Vi har designet algoritmer som kan brukes til å automatisere skjemaekstraksjoner. Algoritmene er implementert i Java og testet på datasett hentet fra MongoDB.

Vi demonstrerer at det er mulig å trekke ut et skjema fra en skjemaløs dokumentdatabase under noen antakelser på hvordan dokumentene er formet. Vi diskuterer hvilke metoder som lar seg automatisere, og for hvilke metoder automatisering er mer utfordrende.

Innhold

1 Introduksjon **1**
1.1 Problemstillinger og oppgavens innhold 1
1.2 Motivasjon 3
1.3 Relatert arbeid 6
1.4 Kapitteloversikt 7

2 Object-role modeling (ORM) **9**
2.1 Introduksjon 9
2.2 Begreper og verdityper 9
2.3 Setningstyper og roller 10
2.4 Skranker 11
2.4.1 Entydighetsskranker 11
2.4.2 Påkrevde roller 13
2.5 Realiseringsalgoritmen 14

3 Dokumentdatabaser **15**
3.1 Struktur 15
3.2 Bruksområder 17
3.3 Implisitte skjemaer 17
3.4 Søk 18
3.5 MongoDB 18
3.5.1 Nøkkelegenskaper 19
3.5.2 Design av datamodeller 20

4 Når MongoDB blir problematisk **23**
4.1 Bakgrunn 23
4.2 Design og fleksibilitet 24
4.3 Et system for ansattrapporter i et selskap 24
4.4 Et systemt over navn og bosetninger 28
4.4.1 Funksjonalitet og krav 28
4.4.2 Utvidelse med opplysninger om kommuner 30

4.5 Problematikk og diskusjon 34

5 Fra MongoDB til ORM **37**

5.1 Antakelser 37
5.2 Metoder 38
5.3 Direkte kartlegging 39
5.3.1 MongoDBs zip-eksempel 48
5.3.2 Problematikk 50
5.4 Statisk analyse av spørringer 51
5.4.1 Idé, struktur og prinsipper 51
5.4.2 MongoDBs zips-eksempel 52
5.4.3 Problematikk ved statisk analyse av spørringer 57
5.5 Dynamisk analyse av data 57
5.5.1 Videre antakelser 59
5.5.2 Problematikk rundt dynamisk analyse av data 60

6 Algoritmer **63**

6.1 Forutsetning 63
6.2 Direkte kartlegging 64
6.2.1 Rekursjon for kartlegging av embedding 65
6.3 Statisk analyse av spørringer 68
6.4 Dynamisk analyse av data 70

7 Implementasjon og testing **73**

7.1 Preprosessering og java-struktur 73
7.2 Implementasjon av direkte kartlegging 74
7.2.1 Testing av MongoDBs *zips* 76
7.2.2 Testing av MongoDBs *restaurants* 77
7.3 Implementasjon av statisk analyse 81
7.3.1 Testing av MongoDB *zips* 81
7.4 Implementasjon av dynamisk analyse 83
7.4.1 Testing av MongoDBs *zips* 87
7.4.2 Testing av MongoDBs *restaurants* 87

8 Konklusjoner og videre arbeid **89**

8.1 Konklusjoner 89
8.2 Videre arbeid 90

Bibliografi **93**

Figurer

2.1 ORM - Eksempler på begreper 10
2.2 ORM - Eksempler på verdityper 10
2.3 ORM - Eksempel på en binær faktatype 11
2.4 ORM - Eksempel på en bro 11
2.5 ORM - Eksempel på en mange-til-én bro mellom A og B 12
2.6 ORM - Eksempel på en én-til-én bro mellom A og B 12
2.7 ORM - Eksempel på en mange-til-mange faktatype mellom A og B 13
2.8 ORM - Eksempel på en ekstern entydighetsskranke 13
2.9 ORM - Eksempel på en påkrevd rolle 14

3.1 Trestrukturen for dokumentet i eksempel 3.1 16
3.2 Dokumentdatabase, samlinger og dokumenter 17

4.1 Trestrukturen for eksempel 4.3 26
4.2 ORM-modell med utgangspunktet i eksempel 4.5 35
4.3 ORM-modell fra figur 4.2 utvidet med kommuner 35

5.1 Alternativ 1. ORM-modell etter direkte kartlegging av et persondokument. 41
5.2 Alternativ 2. ORM-modell etter direkte kartlegging av et persondokument med en entydighetsskranke over rollen til phone. . 41
5.3 ORM-modellen opprettet fra tabell 5.3 43
5.4 ORM-modellen opprettet fra tabell 5.3 med *address* som verditype i begrepet *Address* 43
5.5 ORM-modellen opprettet fra tabell 5.5 45
5.6 ORM-modellen for *Address* med preferert referanse 45
5.7 ORM-modellen etter direkte kartlegging av adresse 46
5.8 ORM-modellen etter direkte kartlegging personer og hobbyer . . 47
5.9 ORM etter direkte kartlegging av MongoDBs zips-eksempel . . 49
5.10 loc modellert basert på domenekunnskap 49
5.11 Eksempel på aggregering med fire steg. 52
5.12 MongoDBs pipeline med fire steg. 53

5.13 ORM-modell etter direkte kartlegging og analyse av spørringen i eksempel 5.6 54
5.14 ORM-modell etter direkte kartlegging og analyse av spørringen i eksempel 5.7 55
5.15 ORM-modell av stat og by etter analyse av spørringen i eksempel 5.7 56

6.1 ORM-modell av eksempel 6.1 generert fra direkte kartlegging . . 68
6.2 Strukturen på hvordan et aggregat skal se ut i et kodemiljø . . . 69

7.1 Datastrukturen i java 73
7.2 ORM-modellen fra direkte kartlegging av restaurants 80

Tabeller

3.1 *Terminologi i relasjonsdatabaser sammenliknet med MongoDB* . 18

5.1 *Direkte kartlegging av en dokumentsamling til en ORM-modell. Setningstyper skal genereres fra de ulike tilfellene av x, y og z* . 39
5.2 *Direkte kartlegging av samlingen person* 41
5.3 *Direkte kartlegging av samlingen person med embedding av adresser* 42
5.4 *Et eksempel på hvordan data vil bli lagret i databasen uten videre kartlegging av indre dokumenter.* 44
5.5 *Direkte kartlegging av embeddet adresseobjekter* 44
5.6 *Direkte kartlegging av MongoDBs zip-eksempel* 48
5.7 *Mangelfull data ved en dynamisk analyse* 60

6.1 *Key-value-struktur av eksempel 6.1* 63

Kapittel 1

Introduksjon

Vi lever i en verden der store deler av livene våre har blitt digitalisert, og bruker flere timer på verdenveven daglig. Vi bruker sosiale medier for å holde kontakten med venner, handle online, se på filmer, søke etter informasjon, og mange flere ting. Det produseres enorme mengder data for hver minste bevegelse vi gjør, og det finnes mange ulike teknologier som brukes til å oppbevare dataene våre.

NoSQL[1] [1] har de siste årene vokst frem som et svært populært alternativ til relasjonsdatabaser. NoSQL gir andre muligheter enn den tradisjonelle relasjonsmodellen, og er spesielt egnet i situasjoner hvor det kreves stor grad av datadistribusjon, er behov for skalering, og der databaseskjemaet er i konstant utvikling i tråd med hvilke data vi har behov for å lagre, og hvordan vi har behov for å lagre dem. Noen av databasene som går under begrepet NoSQL er grafdatabaser, *key-value*-databaser, *wide-column*-databaser og dokumentdatabaser [2].

1.1 Problemstillinger og oppgavens innhold

I et utviklingsperspektiv handler det om å velge riktig teknologi for ulike formål, og det er ikke alltid en enkel oppgave. Ofte velges teknologi ut ifra trend, og ikke tilstrekkelig basert på en grundig undersøkelse av hva som er egnet til formålet. I denne oppgaven skal vi se på valget av *dokumentdatabaser* som databaseteknologi, situasjoner der valget var et feilgrep, og hvordan vi isåfall kan legge til rette for migrasjon til andre typer databaser. Som del av dette ønsker vi å trekke ut et skjema fra en dokumentdatabase, fordi det trengs et eksplisitt skjema dersom vi skal migrere til en relasjonsdatabase

[1] Not only SQL

eller andre skjemaorienterte databaser. Det kan være ønskelig at skjema-ekstraksjonen formuleres i *ORM*[2], som er en formalisme for konseptuelle modeller. Vi trenger en konseptuell modell fordi vi ønsker å se på semantikken og konseptene bak databasen, og undersøke om vi manuelt kan rette opp konseptuelle problemer som forberedelse til databasemigrasjon.

Dokumentdatabaser. Dokumentdatabaser [1] er blant de mest populære NoSQL-databasene på markedet. Dokumentene formaliseres i språk som JSON og XML, og et dokument er en forekomst(*record*) av beslektet data i en dokumentsamling. Vi skal se nærmere på MongoDB [3], som er en dokumentdatabase som ikke stiller krav til et skjema på forhånd. Det betyr at databasen kan være skjemaløs eller med et fleksibelt skjema, som vil si at skjemaet kan endres dynamisk ved behov. Skjemaløse modeller og fleksible skjemaer kan være en fordel i mange ulike sammenhenger, som for eksempel ved e-handel, logging, i sammenhenger med hurtig vekst i data, eller distribuering av data. Dette kommer av at vi kan lagre store mengder data uten å ta hensyn til skjemaet, og blant annet er det ikke behov for å oversette typer, eller transformere data for å passe inn i et skjema. Hvert dokument bør lagres som en uavhengig enhet, og egner seg derfor til distribuering fordi relasjoner mellom data ikke fordeles på tvers av tjenere.

ORM. *Object-role-model (ORM)* [4] er en formalisme for å lage modeller på et konseptuelt nivå. Fordelen med å lage konseptuelle modeller, er å få en bedre forståelse av konsepter og semantikken bak data, og unngå misforståelser fordi modellen kan forstås av ikke-tekniske brukere. Det brukes ofte i sammenheng med relasjonsdatabaser, og er nyttig for å utvikle gode datamodeller for databasen. Det er essensielt å ha en god modell i utgangspunktet fordi det i praksis ikke alltid lar seg gjøre å rette opp feil i designproblemer i databasen på en god måte når databasen er i produksjon.

Problemstillinger. Denne oppgaven tar for seg følgende problemstillinger.

- **Hvordan kan vi utføre skjemaekstraksjon fra en skjemaløs dokumentdatabase?** Med skjemaekstraksjon mener vi å trekke ut relasjoner mellom data for å undersøke underliggende konsepter. Vi har utforsket tre metoder for å trekke ut skjema. *Direkte kartlegging* går ut på å trekke ut samlingsnavn og feltnavn, og løfte dem til et konseptuelt nivå. I tillegg kan vi trekke ut relasjoner mellom samlingsnavn og feltene. Den andre metoden har vi kalt *statisk analyse* av spørringer, som går ut på å analysere spørringer tilknyttet en do-

[2]Object-Role Model

kumentsamling, for å utforske om grupperinger i aggregater kan gi informasjon om hvordan data er relatert. Den siste metoden, *dynamisk analyse* av data, går ut på å analysere datamaterialet i en dokumentsamling for å undersøke funksjonelle avhengigheter. Nærvær av funksjonelle avhengigheter kan gi en indikasjon på denormalisert data, og fravær av antatte funksjonelle avhengigheter kan gi en indikasjon på feil i datamaterialet.

Vi ønsker å formalisere skjemaekstraksjonen i ORM for å få et bilde av konseptene bak en dokumentdatabase. Ved å bruke ORM kan vi enklere forstå domenet, og deretter rette opp i konseptuelle feil og designproblemer før data migreres fra MongoDB til en annen databaseteknologi.

- **I hvilken grad kan vi automatisere prosessen?**

  I denne oppgaven har vi skrevet et Java-program som leser inn en dokumentsamling fra en JSON-fil og utfører direkte kartlegging og dynamiske analyse. Statisk analyse var mer utfordrende å automatisere, ettersom oppbygningen av spørringer kan variere i stor grad, og vi fant ikke noe programbibliotek for å parsere og preprosessere spørringene.

- **Hvilke begrensninger må settes i prosessen?**

  For å kunne utføre skjemaekstraksjoner har vi gjort antakelser på hvordan en dokumentsamling kan se ut. Vi antar at formen på dokumentene i en samling ikke har for store variasjoner, og at dokumentdatabasen har et underliggende implisitt skjema. I tillegg har vi satt noen heuristiske begrensning på dynamiske analyser, fordi den generelle algoritmen har eksponentiell vekst.

1.2 Motivasjon

I dette delkapittelet skal vi se nærmere på hvordan vi kom frem til problemstillingene som er hovedfokus i denne oppgaven. Det opprinnelige utgangspunktet for våre undersøkelser var om det var mulig å integrere NoSQL-databaser i den semantiske veven. For å forklare dette nærmere, skal vi først si noe om hva NoSQL og den semantiske veven er. Deretter beskriver vi arbeidet med å snevre inn problemstillingene.

NoSQL. NoSQL har fått kraftig medvind de siste årene, men nettsiden *db-engines*[3] over populære databaser viser at relasjonsdatabaser fremdeles troner på toppen. Blant topp 10 finnes det kun tre databaseteknologier som går under begrepet NoSQL, og den øverste av dem er MongoDB på 5. plass[4].

NoSQL er et felles begrep for databaser som ikke baserer seg på relasjonsmodellen. I artikkelen *The relational model is dead, SQL is dead, and I don't feel so good myself* [5] skrevet av *Paolo Atzeni* med flere, legges det frem hvorfor NoSQL ikke klarer å få et større fotfeste enn først antatt. Det ble spekulert i at NoSQL ville utkonkurrere relasjonsmodellen, fordi mange mener at relasjonsmodellen er gammeldags og utdatert. Det poengteres i artikkelen at mange store systemer er bygget med relasjonsmodellen i bunnen, og ikke har noen planer om å bytte til NoSQL. Det ligger store investeringer i relasjonsdatabaser, fordi de fortsatt fungerer effektivt.

Den semantiske veven. Den semantiske veven [6] er en utvidelse av verdensveven, og har informasjonsdeling som mål uten begrensninger av applikasjon og nettsider. Det betyr at data må kunne leses og forstås av maskiner. Mennesker har ingen problemer med å forstå at hvis vi snakker om byen Oslo i Norge, så vil vi forstå det som en by med navn Oslo i landet Norge. På Wikipedia finnes det et oppslag om byen Oslo i Norge, og vi har lett for å forstå at vi snakker om det samme konseptet som i oppslaget om Oslo by i *Store norske leksikon (SNL)*. Den semantiske veven har som mål å gi maskiner samme forståelse gjennom universelle definisjoner av konsepter, og universelle definisjoner av relasjoner mellom konseptene.

Idéen bak oppgaven. Utgangspunktet for denne oppgaven var en forholdsvis løs idé om å undersøke ulike NoSQL databaseteknologier, og om det var mulig å trekke ut skjema til bruk i den semantiske veven. Idéen var å se på grafdatabaser, wide-column-databaser og dokumentdatabaser, mer spesifikt, *Neo4j* [7], *HBase* [8] og *MongoDB*. Vi ønsket å trekke ut modellene til bruk i den semantiske veven for å kunne representere data universelt, uten å være bundet til en gitt databaseteknologi.

Vi startet med å undersøke MongoDB og dokumentdatabaser, og oppdaget at det allerede finnes teknologier som gjør MongoDB kompatibel med den semantiske veven kalt *MongoGraph* [9], en utvidelse til grafdatabasen *Allegrograph* [10]. JSON-objektene i MongoDB blir oversatt til *tripler* [11], og lagt inn i en *triplestore*, der semantiske teknologier kan brukes. Triplene består av et *subjekt*, et *predikat*, og et *objekt*. Et dokument blir oversatt til et subjekt, feltene blir oversatt til predikater, og feltverdiene til objekter.

[3] https://db-engines.com/en/ranking
[4] Tall fra april 2017

Et dokument kan bestå av mange tripler.

Mongograph genererer tripler for å kunne bruke SPARQL, som er spørrespråket i den semantiske veven. JSON-objektene blir skrevet om til et annet format. Problemet er imidlertid at for å kunne utnytte fordelene med den semantiske veven, må en legge inn semantikk og regler manuelt, fordi triplene kun reflekterer syntaksen i dokumentene og ikke modellen [11]. Vi var ute etter mer informasjon om selve skjemaet og konseptene til dokumentdatabasene i MongoDB, og gjorde flere undersøkelser som gikk på skjemaekstraksjoner. Vi kom over interessante søkeresultater[5], men ingen som ga svar på det vi lette etter, som var semantikk og konsepter i et skjema. Vi så etter sammenhenger og relasjoner mellom data, mens søkeresultatene gav oss svar på hvilke datatyper som fantes, og hvordan dokumentene var *nestet*.

Vi kom over mange diskusjoner på nettet som gikk på problematikk rundt MongoDB, men ingen gode løsninger for å utføre en databasemigrering. Vi valgte derfor å fokusere på dokumentdatabaser, mer spesifikt MongoDB, og gikk bort fra videre undersøkelser rundt Neo4j og HBase.

Arbeidet for å spisse inn problemsstillingen. I forbindelse med arbeidet med å finne relevant materiale om MongoDB, kom vi over beskrivelser av problemer brukere opplevde med MongoDB. Det mest interessante av dem var et innlegg kalt *Why you should never use MongoDB* [12], som handlet om et konkret eksempel der dokumentdatabaser var et feilgrep. I tillegg florerte det med poster og kommentarer på forskjellige sider om hvordan mange hadde feilvurdert dokumentdatabaser, og heller ville brukt relasjonsdatabaser og SQL om de kunne ha designet databasen forfra.

Vi gjorde dermed videre undersøkelser på om det var mulig å migrere fra MongoDB til en relasjonsdatabase, og fant ut at det ikke fantes en offisiell metode for dette. Løsningene som dukket opp gikk ut på å designe databasen forfra basert på domenekunnskap, og deretter eksportere data fra MongoDB til relasjonsdatabasen. Vi fant ut at vi ville undersøke om det fantes bedre og mer generelle metoder for skjemaekstraksjon fra dokumentdatabaser, som kan brukes i en migreringsprosess.

Vi endret fokuset fra den semantiske veven til relasjonsdatabaser fordi søkeresultatene vi fikk for det meste handlet om overgangen fra MongoDB-databaser til relasjonsdatabaser. Ved å ekstrahere skjema til en konseptuell modell i ORM kan en likevel bruke modellen til andre typer databaseteknologier og den semantiske veven.

[5] Blogginnlegg og diskusjonssider

1.3 Relatert arbeid

Det finnes ikke mange artikler som tar for seg skjemaekstraksjon fra dokumentdatabaser fordi feltet fortsatt er relativt nytt. Videre er MongoDB fremdeles i stor endring, og det kommer stadig nye versjoner med ny funksjonalitet. Det finnes likevel mange bloggposter og diskusjoner i fora som tar for seg problematikken rundt databasemigrering fra MongoDB til MySQL eller PostgreSQL.

Switching Data Stores [13] er et arbeid presentert av systemutvikler *Sarah Mei* under *All you base conference 2013*, en databasekonferanse for webutviklere. Hun legger frem et prosjekt kalt *Diaspora*, som er et distribuert sosialt nettverk bygget i Ruby, med MongoDB som databaseteknologi. Presentasjonen handler om hvorfor de måtte migrere fra MongoDB til MySQL, og hvordan dette ble gjort basert på domenekunnskap.

Goodbye MongoDB, Hello PostgreSQL [14] handler om hvordan *Olery Developer Portal* ble nødt til å migrere fra MongoDB til PostgresSQL fordi de fikk problemer med det skjemaløse designet i MongoDB. De mente at MongoDB lovet dem enklere håndtering av databasen fordi de ikke måtte bestemme noe skjema, mens i realiteten ble ansvaret flyttet til utviklere og applikasjonene.

Ettersom det i liten grad finnes vitenskaplige artikler rundt temaet, består litteraturlisten i stor grad av materiale funnet på nettsider, som bloggposter, manualer, tekniske rapporter, og presentasjoner fra ulike konferanser.

1.4 Kapitteloversikt

- *Kapittel 1 - Introduksjon:* Det første kapittelet gir en introduksjon til oppgaven, med motivasjon og problemstillinger. Relatert arbeid er også inkludert i dette kapittelet.

- *Kapittel 2 - Object-role-modeling:* Kapittel 2 er et bakgrunnskapittel om *Object-role-modeling* som brukes til å presentere skjemaekstraksjoner på et konseptuelt nivå.

- *Kapittel 3 - Dokumentdatabaser:* Kapittel 3 er et bakgrunnskapittel om dokumentdatabaser. Bakgrunnsstoff om MongoDB er inkludert i dette kapittelet.

- *Kapittel 4 - Når MongoDB blir problematisk:* Kapittel 4 gir et overblikk over problematikken rundt MongoDB og dokumentdatabaser. Det blir lagt frem to ulike eksempler som handler om problemer rundt datautvidelse og fleksibilitet.

- *Kapittel 5 - Fra MongoDB til ORM:* Kapittel 5 tar for seg ulike metoder for skjemaekstraksjon fra dokumentdatabaser. Vi beskriver tre ulike metoder: *direkte kartlegging*, *statisk analyse av spørringer* og *dynamisk analyse av data*. Skjemaekstraksjonene blir representert ved hjelp av ORM-modeller. Kapittelet tar for seg hvilke antakelser vi gjør om dokumentdatabasene for å sikre at skjemaekstraksjon kan gjennomføres, og hvilke begrensninger og problemer hver metode har.

- *Kapittel 6 - Algoritmer:* Kapittel 6 gir en oversikt over de algoritmene som vi har designet for automatisering av metodene i kapittel 5.

- *Kapittel 7 - Implementasjon og testing:* Kapittel 7 inneholder implementasjoner i Java, testing av ulike datasett, resultater og observasjoner.

- *Kapittel 8 - Konklusjoner og videre arbeid:* Det siste kapittelet er en oppsummering av oppgaven med konklusjoner, og en diskusjon om videre arbeid.

Kapittel 2

Object-role modeling (ORM)

Dette kapittelet gir en oversikt over modellerinsspråket *ORM*, konsepter, og terminologi som skal brukes videre i oppgaven.

2.1 Introduksjon

Object-role modeling (ORM) er en formalisme for å designe databasemodeller på et konseptuelt nivå [4]. Å designe databasemodeller på et konseptuelt nivå er nyttig ved at det brukes naturlig språk og konsepter som er enklere å forstå for *ikke-tekniske* brukere. Dette er for å forsikre korrekthet i databasedesignet ved å redusere misforståelser som ofte oppstår ved tekniske modeller. ORM-modeller kan ofte fange opp flere forretningsregler fordi modellen ikke er bundet til tekniske begrensninger, og kan dermed uttrykke flere konsepter ved å bruke naturlig språk [15].

ORM brukes til å beskrive et *domenefelt*, eller på engelsk, *universe of discourse (UoD)*. Det er viktig å ha en god forståelse for domenet, og å modellere det på en forstått og *ikke-tvetydig* måte [4]. ORM benytter *elementære* setninger eller fakta for å beskrive domenet i naturlig språk. I tilegg brukes det *roller* og *objekter* for å gi en konseptuell tilnærming til modellen.

2.2 Begreper og verdityper

Et *begrep*, på engelsk kalt *entity*, er en konkret eller abstrakt gjenstand som kan refereres til gjennom verdier som for eksempel navn, tall, eller andre verdier [16]. Begreper brukes til å modellere objekter på et konseptuelt nivå, som for eksempel personer, utstyr, eller mer abstrakte ting som datoer.

Figur 2.1 viser tre eksempler på begreper. Begreper kjennetegnes ved en heltrukken kantlinje.

Figur 2.1: *ORM - Eksempler på begreper*

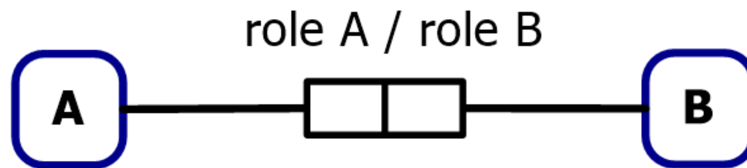
En *verditype*, også kalt *identifikator* eller *representasjon*, på engelsk kalt *value type*, er verdier tilknyttet et begrep. Verdier kan bestå av strenger og tall. Figur 2.2 viser tre eksempler på verdityper. Verdityper kjennetegnes ved en stiplet kantlinje.

Figur 2.2: *ORM - Eksempler på verdityper*

2.3 Setningstyper og roller

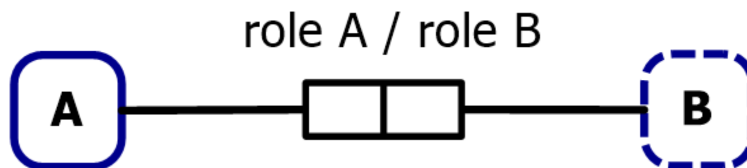
Setningstyper beskriver relasjoner og roller mellom begreper og verdityper. Det finnes to ulike setningstyper, *faktatyper* og *broer*. Faktatyper kan være *unære*, *binære*, *ternære* eller *n-ære* som vi gjerne kaller faktatyper som består av fire eller flere begreper. Alle faktatyper kan skrives om til *binære* faktatyper som beskriver relasjonen mellom to begreper.

En binær faktatype består av et *rollepar* med rollebeskrivelser. Rolleparene brukes til å modellere hvilke roller som spilles mellom to begreper, og rollebeskrivelsen formidler rollene i naturlig språk. Figur 2.3 viser en faktatype mellom begrep A og B. Begrep A har rollen *role A* til begrep B, og begrep B har rollen *role B* til begrep A. Boksene under rollebeskrivelsene kalles rollebokser og utgjør et rollepar. En ORM-modell krever at minst én rollebeskrivelse eksisterer, og det er ikke obligatorisk å ha med begge som vist i figuren.



Figur 2.3: *ORM - Eksempel på en binær faktatype*

En *bro* virker på samme måte som en faktatype, men en bro kan kun være *binær*. En bro beskriver relasjoner mellom et begrep og en verditype. Figur 2.4 viser et eksempel på en bro, og rollebeskrivelsene pleier å si noe om hvordan verdier i *B* reprsenterer *konseptet* i *A*. Et eksempel på en vanlig rollebeskrivelse er *med/for*, som for eksempel *Person*(begrep) ***med*** *navn*(verditype), og *navn* ***for*** *Person*.



Figur 2.4: *ORM - Eksempel på en bro*

2.4 Skranker

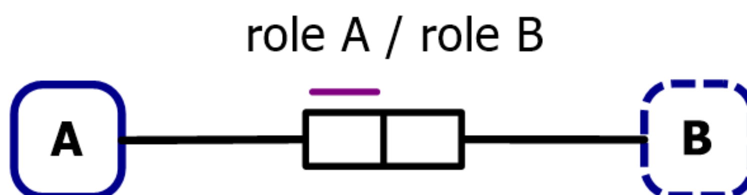
Skranker brukes til å beskrive integritetregler fra den virkelige verden som modelleres. Skranker setter begrensninger på hva som skal være lovlige instanser i domenet.

2.4.1 Entydighetsskranker

Entydighetsskranker brukes til å uttrykke unikhet i en rolle, eller i kombinasjoner av roller. Unikhet i en rolle betyr at for hver instans av konseptet som spiller rollen, så kan rollen kun spilles én gang. For eksempel kan en *person* kun ha ett personnummer, slik at konseptet *person* kun spiller rollen å ha et personnummer én gang [4]. Vi skiller mellom *interne* og *eksterne* entydighetsskranker [16].

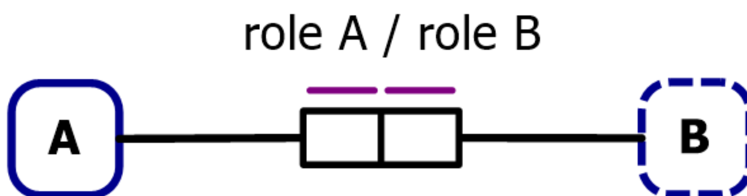
Intern entydighetsskranke

En intern entydighetsskranke ligger lokalt i en setningstype, og for å markere en entydighetsskranke settes en strek over rolleboksene. I binære setningstyper kan den settes over én av rolleboksene, over hver av rolleboksene, eller det kan settes en lengre strek over begge rolleboksene.



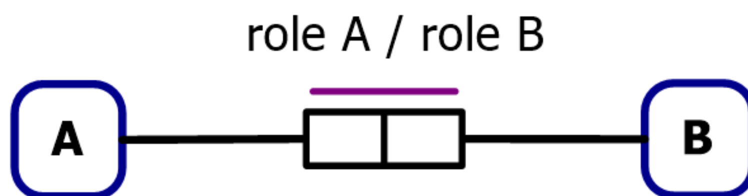
Figur 2.5: *ORM - Eksempel på en mange-til-én bro mellom A og B*

I figur 2.5 er det markert en entydighetsskranke over rollen til begrep A som betyr at en instans av A kun kan spille rollen *role A* én gang. Med *andre ord* betyr det at for hver A kan vi kun ha én verdi av B. Rollen til B står uten en entydighetsskranke som betyr at for den samme verdien av B kan vi ha forskjellige instanser av A. I broer forekommer entydighetsskranken nesten alltid over rollen til begrepet.



Figur 2.6: *ORM - Eksempel på en én-til-én bro mellom A og B*

I figur 2.6 er det markert entydighetsskranker over rollen til A og rollen til B som leses som et én-til-én forhold mellom A og B. Det betyr at for hver instans av A kan det kun finnes én verdi av B, og for hver verdi av B kan det kun finnes én instans av A. Én-til-én-setningstyper forekommer oftere i form av broer enn faktatyper. Årsaken er at det ikke finnes mange konsepter(begreper) fra virkeligheten med én-til-én forhold. Det forekommer oftere i relasjoner mellom konsept(begrep) og data(verdi), som foreksempel at en person kun kan ha ett personnummer, og et personnummer kun kan tilhøre én person.

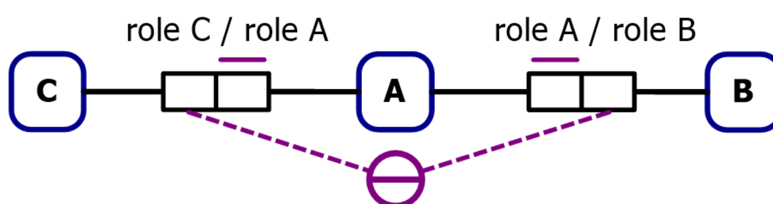


Figur 2.7: *ORM - Eksempel på en mange-til-mange faktatype mellom A og B*

I figur 2.7 er det markert en lang strek over rollene til A og B. Denne skranken markerer unikhet over kombinasjonen av rolleparet. Det betyr at for hver instans av A kan det finnes mange instanser av B så lenge instansene av B er ulike. Det samme gjelder den andre veien. Skranken uttrykker at samme kombinasjoner av A og B ikke kan repeteres. Mange-til-mange-forhold er vanlig å finne i form av faktatyper, og sjelden i form av broer.

Ekstern entydighetsskranke

Ekstern entydighetsskranke går på tvers av setningstyper, og uttrykker unikhet på tvers av setningstypene. I figur 2.8 er det markert en ekstern entydighetsskranke på rollen til C og rollen til B som skal gjelde i A. Det betyr at for alle instanser av A skal kombinasjonen av B og C være unike.

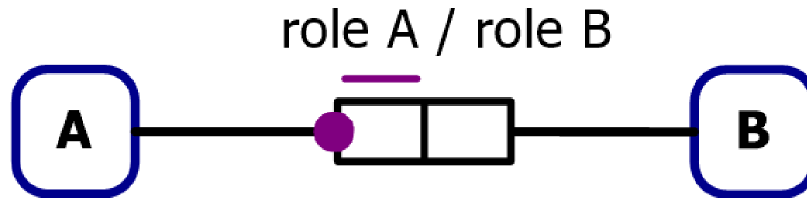


Figur 2.8: *ORM - Eksempel på en ekstern entydighetsskranke*

2.4.2 Påkrevde roller

Påkrevde roller brukes til å markere at en rolle må spilles av et begrep. Det markeres ved å sette en prikk på rolleboksen som vist i figur 2.9. Figuren

viser at begrepet A må spille rollen *role A* til B, som betyr at for hver instans av A må det finnes en instans av B.



Figur 2.9: *ORM - Eksempel på en påkrevd rolle*

2.5 Realiseringsalgoritmen

Vi kan bruke *realiseringsalgoritmen* [17] for å gruppere en ORM-modell til et relasjonsskjema. Realiseringsalgoritmen sier noe om hvordan vi skal trekke ut elementene i ORM-modellen, og hvordan elementene grupperes sammen til relasjoner. Det stilles et krav til at ORM-modellen må være refererbar, som betyr at alle begreper må ha en preferert referanse før den kan realiseres.

Undertrykte begreper

Som en del av realiseringen kan en velge å fjerne (undertrykke) *referansebegreper* fra relasjonsskjemaet. Et referansebegrep er et begrep som ikke spiller noen andre roller enn de som inngår i den prefererte referansen [18].

Kapittel 3

Dokumentdatabaser

En dokumentdatabase er designet for å lagre semistrukturert data, og formaliseres i språk som JSON og XML. Idéen bak dokumentdatabaser er å samle beslektet data i ett *dokument*, for å optimalisere uthenting av data. Fellestrekket med dokumentdatabaser er strukturen på dokumentene som kommer i form av nestede dokumenter, og JSON og XML er formalismer for å beskrive slike strukturer.

3.1 Struktur

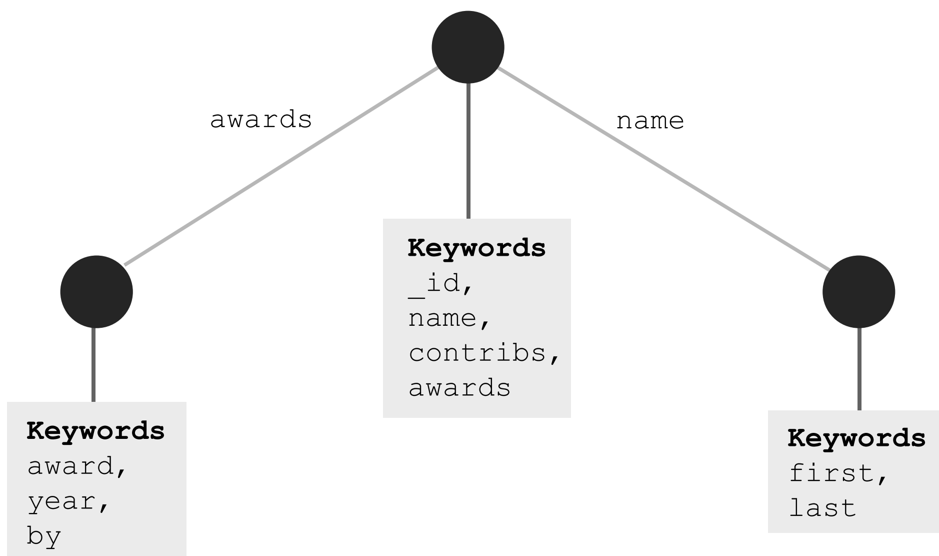
Dokumentene består av *nøkkel-verdi*-par som vist i eksempel 3.1 i form av JSON, og data er lagret i nestede hierarkier, gjerne en trestruktur som vist i figur 3.1. Trestrukturen viser at det opprettes en ny node for hvert dokument som er nestet. Beslektet data lagres sammen som en enhet og danner et *dokument*, og dokumentene lagres i en dokumentsamling (*collection*). En dokumentdatabase kan bestå av flere samlinger som vist i figur 3.2.

Dokumentene håndterer semistrukturert data som gjør det enklere å lagre data i forhold til relasjonsdatabaser. Dette er fordi en slipper å håndtere datatyper og skranker for å opprettholde dataintegritet. Nøkkelord kan variere fra dokument til dokument, selv om dokumentene befinner seg i den samme samlingen. Vi kaller dette for et *fleksibelt* skjema, der skjemaet kan endres dynamisk etter behov. Dokumentdatabaser kjennetegnes med *skjemaløse* modeller. Det stilles ikke krav til et definert og eksplisitt skjema, og det finnes ingen begrensning på hvordan dokumenter kan nestes, utenom begrensningene i XML og JSON. Et eksempel på en begrensning i XML og JSON er at det kun kan finnes én rotnode. En applikasjon som bruker en dokumentdatabase til grunn trenger dermed ikke å være avhengig av et

skjema for å kunne lagre data.

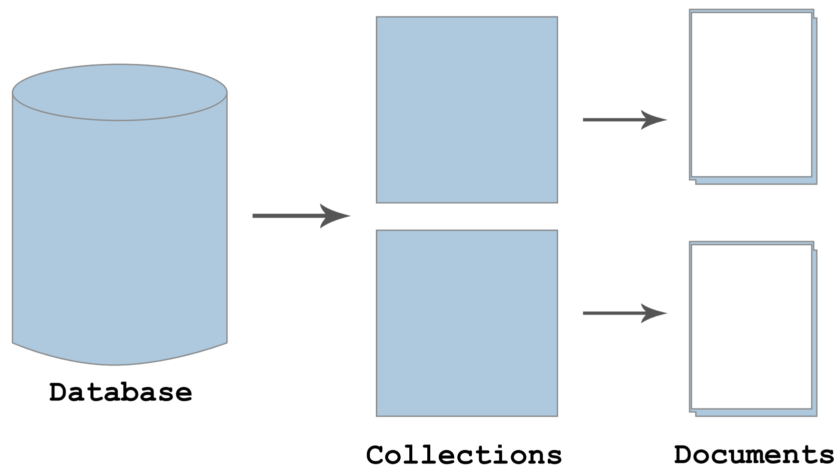
```
{
    "_id" : 1,
    "name" : { "first" : "John", "last" : "Backus" },
    "contribs" : [ "Fortran", "ALGOL" ],
    "awards" : [
        {
            "award" : "W.W. McDowell Award",
            "year" : 1967,
            "by" : "IEEE Computer Society"
        }
    ]
}
```

Eksempel 3.1: *JSON-dokument hentet fra [19]. JSON-dokumentet inneholder nøkkel-verdi-par der verdiene kan bestå av strenger, tall, lister og JSON-dokumenter.*



Figur 3.1: *Trestrukturen for dokumentet i eksempel 3.1*

.



Figur 3.2: *Dokumentdatabase, samlinger og dokumenter*

3.2 Bruksområder

De vanligste bruksområdene [20] for dokumentdatabaser er *event-logging*, nettsider med innholdsredigering, *web*-analyser, sanntidsanalyser, og e-handelssider. Fellestrekket med disse bruksområdene er store mengder data og behovet for hurtig dataaksess.

3.3 Implisitte skjemaer

Dokumentdatabaser tillater *fleksible* skjemaer, som medfører at applikasjonen må gjøre antakelser på innholdet i databasen. Dette kalles for et *implisitt* skjema [21]. Skjemaløse modeller betyr ikke nødvendigvis total uavhengighet fra skjema, fordi vi må ha en viss kunnskap om hvordan data er lagret for å kunne hente ut data, og traversere dokumenter.

Et implisitt skjema er kunnskapen vi trenger for å akesesse data fra en dokumentdatabase. Applikasjonen blir derfor nødt til å anta hvilke nøkkelord som assosieres med dokumentene, og den må ha en forståelse for dokumenthierarkiet for å kunne aksessere og hente ut data. I eksempel 3.1 må en applikasjon for eksempel vite om nøkkelordene *name*, *contribs* og *awards* for å hente ut informasjon om personer.

3.4 Søk

Data hentes ut ved hjelp av nøkkelverdier, også kalt *felter*. Nøkkelverdiene er ofte indeksert i databasen, og det skal være mulig å indeksere nøkkelverdier som ligger lengre ned i dokumenthierakiet. Databasen er dokumentorientert, og hele eller deler av dokumenter kan hentes ut.

3.5 MongoDB

MongoDB er den største og mest populære dokumentdatabasen som finnes på markedet i følge nettsiden *db-engines*[1]. MongoDB er en *open-source*-database gitt ut av MongoDB Inc, og lagrer data i form av JSON-lignende dokumenter kalt BSON[2], som kan håndtere *datatyper* som ikke er inkludert i ordinær JSON.

Alle dokumenter i MongoDB bruker feltet *_id* som primærreferanse. Dersom feltet *_id* ikke finnes i et dokument når det leses inn til databasen, blir det autogenerert av systemet.

Sammenliknet med relasjonsdatabaser tilsvarer en relasjon en *samling* i MongoDB, en rad tilsvarer et dokument, og en kolonne tilsvarer et felt. Se tabell 3.1 for terminologi. MongoDB støtter ikke *joins* på samme måte som en relasjonsdatabase, men ved hjelp av nestede dokumenter kan databasen aksessere beslektet data uten å måtte utføre *joins* som en ville ha gjort i en normalisert relasjonsmodell.

Relasjonsdatabase	MongoDB
Relasjon(Tabell)	Samling (Collection)
Rad	Dokument
Kolonne	Felt
Joins	Dokumentembedding, linking

Tabell 3.1: *Terminologi i relasjonsdatabaser sammenliknet med MongoDB*

[1] https://db-engines.com/en/ranking
[2] Binary JSON

3.5.1 Nøkkelegenskaper

Høy ytelse

Ved at systemet støtter *embedding*[3] som i denne sammenhengen betyr å lagre et dokument inni et annet, tilsvarende nesting av dokumenter, vil dette redusere antall skrive- og leseaktiviteter i databasen. Ved at hvert dokument blir lest som en uavhengig enhet kan databasen hente ut hele dokumenter der beslektet data hentes ut ved hjelp av kun én diskaksess.

Eksempel 3.2 viser et persondokument med et eksempel på *embedding*. Feltet *address* inneholder en liste med adressedokumenter, også kalt adresseobjekter, og vi kaller dette for *embedding* av adresser i et persondokument.

```
{
  "name": "Lana",
  "age": 23,
  "address": [
    {"street": "Eventyrveien 20", "city": "Oslo", "zip":
        0850},
    {"street": "Storgata 50", "city": "Oslo", "zip": 0730}
  ]
}
```

Eksempel 3.2: *Embedding*

Rikt spørrespråk

I tillegg til systemets vanlige *create*-, *read*-, *update*- og *delete*-operasjoner, også kjent som *CRUD*-operasjoner, gir MongoDB støtte for dataaggregering, tekstsøkt og noe de kaller for *Geospatial*-spørringer [22]. I denne oppgaven skal vi hovedsaklig se på *CRUD-operasjonene* og dataaggregering. Geospatial-spørringer er spørringer som relateres til geografiske lokasjoner og geometri, og er ikke er relevant for oppgaven.

Tilgjengelighet

Det tilbys støtte for replikering [23] av data som gir høy tilgjengelighet. Dette er nyttig ved distribuering der flere kopier av samme data lagres på forskjellige noder, slik at data lett blir tilgjengelig på flere enn én node.

[3] Embedding betyr innebygd og brukes ofte innenfor teknologi

Større grad av replikering gir høyere tilgjengelighet i data. Dette er blant egenskapene i MongoDB som gjør databasen attraktiv for distribuerte prosjekter.

Horisontal skalering

MongoDB gir god støtte for distribuering av data. Systemet benytter seg av *sharding*, som er en metode for å fordele data utover et nettverk av noder. En *shard* kan bekrives som en delmengde av data fra databasen, og lagres i en egen tjener for å fordele arbeidslasten. Ettersom MongoDB for det meste lagrer beslektet data i det samme dokumentet, vil distribuering være enklere i forhold til tradisjonelle relasjonsdatabaser, der mange av relasjonene er koblet gjennom fremmednøkler og jointabeller, og beslektet data gjerne ligger spredt over flere relasjoner.

3.5.2 Design av datamodeller

Embedding

Som nevnt er *embedding* én av nøkkelegenskapene i MongoDB. Strukturen tillater rask aksessering av data i bytte mot en *denormalisert* modell. Denne måten å designe databasen på er hensiktsmessig dersom responstid er viktig.

```
> db.contacts.find()

{
 "name": "John Doe",
 "address": [
    {
     "street": "Eventyrveien 1",
     "city": "Oslo",
     "zip": 0850
    }
 ],
 "phone": [ "123 45 678" , "987 65 432" ]
}
```

Eksempel 3.3: *Eksempel på et kontaktsystem*

Eksempel 3.3 viser hvordan vi kan gjøre et enkelt oppslag på en persons adresse og telefonnummer med kun én enkel spørring slik som vist i eksempel

3.4 som finner telefonnumre til personen som heter *John Doe*.

```
> db.contacts.find(
    {"name": "John Doe"},
    {"name": 1, "phone": 1}
)

----Results----
{
    "name": "John Doe",
    "phone": [ "123 45 678" , "987 65 432" ]

}
```

Eksempel 3.4: *Oppslag i telefonnumre som finner navn og numre*

Dokumentreferanser

Det er mulig å bruke referanser mellom dokumenter i tilfeller embedding ikke er ønskelig. Det gjøres ved å referere til *_id*-verdien i et annet dokument.

Normalisering

I utgangspunktet er det ikke vanlig å ha et fullstendig normalisert MongoDB-database, men det er likevel mulig med referanser mellom dokumenter. I følge MongoDB er en normalisert modell nyttig dersom

- Embedding fører til redundans, men ikke effektiviserer kjøretiden.
- Det trengs å representere komplekse *mange-til-mange*-relasjoner
- En trenger å modellere store hierarkiske datasett.

Disse punktene er nevnt under *Data Model Designs* [24].

Kapittel 4

Når MongoDB blir problematisk

4.1 Bakgrunn

Når en teknologi starter som en trend vil den potensielt få en stor brukergruppe der en andel av gruppen bruker teknologien på grunn av popularitet, i stedet for å vurdere teknologien som den beste løsningen. Dette kan gjerne sammenlignes med vanlige motetrender, trender innenfor interiør eller andre kategorier som berører oss i hverdagen. Vi ender ofte opp med ting vi nødvendigvis ikke trenger fordi vi ønsker å følge trender, som da alle skaffet seg en khalervase når vi egentlig ikke trengte en ny vase, fordi vi aldri kjøper blomster.

Sånn er det også innenfor teknologi. MongoDB har vokst frem som en svært populær NoSQL-database og er nummer fem på listen over *db-engines*[1] i skrivende stund. Det kan være flere årsaker til hvorfor akkurat MongoDB har blitt så populær de siste årene. Dokumentdatabaser nevnes på MongoDB-bloggen som et av de enkleste databasesystemene å starte med på grunn av den fleksible datamodellen, åpen kildekode, lave kostnader, og det er ikke behov for nøye planlegging av databasedesign på forhånd.

I et høyt rangert innlegg kalt *Why You Should Never Use MongoDB* [12] av systemutvikler *Sarah Mei*[2] legges det frem en situasjon der det ble valgt feil databaseteknologi. Hun legger frem konkrete eksempler på når MongoDB fungerte, og når det ble et problem, og hvilke tegn en bør være oppmerksom på som kan indikere ineffektiv bruk av dokumentdatabaser.

[1] Rangering sjekket april 2017
[2] http://www.sarahmei.com/

> *"Whether you're duplicating critical data (ugh), or using references and doing joins in your application code (double ugh), when you have links between documents, you've outgrown MongoDB"*
> — Sarah Mei, *Why you should never use MongoDB*

Eksempelet som blir lagt frem i innlegget handler om et sosialt nettverk med en MongoDB-database i bunnen. Hun poengterer at dokumentdatabaser kun kan brukes dersom vi har å gjøre med *ekte dokumenter*. Vi kan se for oss hva et dokument er, som for eksempel et dokument i *Microsoft Word* der vi har metadata som sier noe om innholdet, overskrifter, underoverskrifter, bilder og innholdstekst. Dokumentet har ikke relasjoner til andre dokumenter, og kan leses uavhengig. Det samme bør gjelde for dokumenter i MongoDB, men oppfyller vi dette kan dokumentene vokse seg større enn MongoDBs begrensning på 100 nestinger.

4.2 Design og fleksibilitet

Design av gode datamodeller er viktig for å kunne avlaste kompleksitet i applikasjonskoden ved at mye av arbeidet for å manipulere data kan bli gjort av databasen. Fleksibilitet i datamodeller fører til enklere drift og en bedre forutsetning for å utvide databasen.

Fleksibilitet i datamodellene oppnås ofte ved normaliserte databaser fordi en reduserer redundans i data. Mindre dobbeltlagring av data reduserer risikoen for feil i oppdateringer. Økt redundans i data i denormaliserte systemer fører til mer komplekse oppdateringer fordi databasen eller applikasjonen må sjekke om samme data har blitt oppdatert på alle områder. Dokumentdatabaser har ofte denormaliserte modeller, og vi kan risikere oppdateringsanomalier.

I de neste delkapitlene skal vi se på eksempeler der dokumentdatabaser blir problematisk i forhold til fleksibilitet og datautvidelse.

4.3 Et system for ansattrapporter i et selskap

Vi skal utvikle et system for et selskap som holder oversikten over ansatte i selskapet, ansattrapporter og hvem det reporteres til. Selskapet ønsker en oversikt over hvem som mottar hvilke rapporter fra hvem, og ønsker at systemet skal utvikles med MongoDB som databaseteknologi.

Vi kan starte med å lage en oversikt over ansatte som vist i eksempel 4.1. Hver ansatt har et ansattnr angitt med *_id*, og et navn. Videre skal det finnes en oversikt over rapporter som hver ansatt sender inn. I eksempel 4.2 er rapportene *nestet* inn i ansattdokumentet i form av *embedding*.

```
> db.ansatte.findOne()
{
  "_id": 1,
  "navn": "John Doe"
}
```

Eksempel 4.1: *Ansatte med ansattnr og navn*

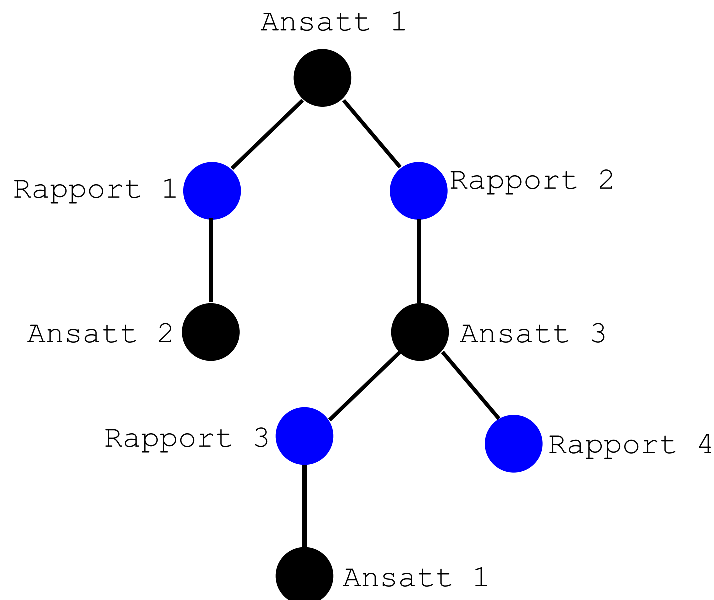
```
> db.ansatte.findOne()
{
  "_id": 1,
  "navn": "John Doe",
  "rapporter": [
    {
      "nr": 1,
      "kategori" : "salg"
    },
    {
      "nr": 2,
      "kategori" : "internt"
    }
  ]
}
```

Eksempel 4.2: *Ansatte med ansattnr og navn*

Hver rapport skal sendes inn til en annen ansatt i selskapet som skal godkjenne rapporten. Databasen trenger dermed å ha en oversikt over hvem som mottar hver rapport. Følger vi prinsippet om at hvert dokument skal være en uavhengig enhet kan vi fortsette å neste dokumentene som vist i eksempel 4.3. Eksempelet viser embedding av rapporter og ansatte. Det vil oppstå problember med denne metoden fordi systemet vil fortsette å neste rapporter og ansatte til den når dokumentbegrensningen på 100 nestinger. Se figur 4.1 for en illustrasjon av strukturen fra eksempel 4.3.

```
> db.ansatte.findOne()
{
  "_id": 1,
  "navn": "John Doe",
  "rapporter": [
    {
      "nr": 1,
      "kategori" : "salg",
      "rapportertTil":
       {
         "_id": 2,
         "navn": "Jane Doe",
         "rapporter": [...]
       }
    }
  ]
}
```

Eksempel 4.3: *Nesting av rapporter og hvem det rapporteres til*



Figur 4.1: *Trestrukturen for eksempel 4.3*

Figur 4.1 illustrerer en trestruktur av en situasjon der vi ønsker en grafstruktur. Måten vi bruker embedding på er ikke ideelt i dette tilfellet, og vi må prøve å løse problembet på en annet måte.

Hvis vi begrenser dokumentene til maksimalt to nestinger som vist i eksempel 4.4, har vi brukt *_id* til å referere til et annet ansattdokument. Vi har i tillegg valgt å denormalisere navn i feltetet *rapportertTil* fordi det ofte må slås opp et navn, og MongoDB støtter ikke join på samme måte som i SQL.

```
> db.ansatte.find()
{
  "_id": 1,
  "navn": "John Doe",
  "rapporter": [
    {
      "nr": 1,
      "kategori" : "salg",
      "rapportertTil": { "_id": 2, "navn": "Jane Doe" }
    }
  ]
},
{
  "_id": 2,
  "navn": "Jane Doe",
  "rapporter": [
    {
      "nr": 2,
      "kategori" : "internt",
      "rapportertTil": { "_id": 3, "navn": "J. Smith" }
    }
  ]
}
```

Eksempel 4.4: *Ansatte med maks to nestinger*

Systemet klarer nå å lage en oversikt over hvem som skriver rapporter og hvem som mottar rapportene. Selskapet ønsker videre å få en oversikt over hvor mange rapporter en ansatt skriver i forhold til hvor mange rapporter den ansatte mottar. Ettersom at MongoDB ikke støtter *joins* blir vi nødt til å gjøre to oppslag i databasen der den ene sjekker antall rapporter skrevet, og den andre traverserer gjennom alle dokumentene og sjekker om en ansatt forekommer i *rapportertTil*.

Selskapet ønsker etterhvert å utvide systemet med muligheten til å revidere en rapport, og det kan gjøres av ansatte som ikke skrev rapporten. En revidert versjon av en rapport skal også rapporteres til en annen ansatt i selskapet. Vi begynner å se problemet med fleksibiliteten i databasen. Hvordan skal vi utvide systemet med revisjon av rapporter? Det kan være hensiktsmessig å ha en egen dokumentsamling for rapportene istedet for å ha dem nestet inn i ansattdokumentene. Vi ender opp med en egen dokumentsamling for ansatte, og en egen for rapporter der samlingene har relasjoner mellom seg. Som *Sarah Mei* sier i sitt innlegg har vi nå vokst forbi hensikten og fordelene med MongoDB.

4.4 Et systemt over navn og bosetninger

Anta at vi i utgangspunktet har et system som holder informasjon om personer med fødselsnummer, navn og postnummer. Vi ønsker oss et oppdatert register over innbyggere i Norge og hvor de bor for å følge utviklingen i bosetninger på postnummernivå. Informasjon om nøyaktig adresse er dermed irrelevant. Systemet skal til enhver tid ha oppdatert opplysning om navn og postnummer, og det skal være mulig med navneendring og flytting.

```
> db.personer.findOne()
{
  "_id": 01019012345,
  "navn": "Ola Normann",
  "postnr": 1341
}
```

Eksempel 4.5: *Personer med postnumre*

Dokumentet i eksempel 4.5 er bare ett eksempel på et av mange dokumenter i samlingen *personer*. Fødselnummer er representert som *_id*, og postnummer med *postnr*.

4.4.1 Funksjonalitet og krav

Navnestatistikk

Systemet skal kunne finne en statistikk på hvor mange som heter hva. For enkelhets skyld skal systemet kun se på fulle navn, og ikke fornavn og etternavn separat. Vi kan telle ved å bruke MongoDBs *count*-funksjon.

```
> db.personer.count( { name: "Ola Nordmann" } )

115
```

Eksempel 4.6: *Finn antall personer med navnet Ola Normann*

Eksempel 4.6 viser hvordan *count*() teller antall personer som heter *Ola Normann* og returnerer kun antallet. Denne spørringen brukes til spesifikke navnsøk. Tallet som vises i eksempelet er bare et eksempel og ikke representativ for virkeligheten.

Alternativt vil systemet kunne vise topp ti navn i Norge etter antall personer. I denne spørringen kan vi bruke MongoDBs aggregatfunksjon.

```
> db.personer.aggregate( [
   { $group: { _id: "$navn", antallNavn: { $sum: 1 } } },
   { $sort: {antallNavn: -1} },
   { $limit: 10 }
] )
```

Eksempel 4.7: *Navneliste top 10*

Eksempelet i 4.7 grupperer først på *navn* og teller 1 per treff. Deretter sorteres hele listen synkende markert med -1, og til sist velges de 10 øverste treffene.

Oversikt over bosetning

For å ha en oversikt over hvor mange som bor på hvert postnummer må systemet kunne telle antall personer for hvert postnummer. Et enkelt aggregat vil kunne løse dette problemet.

```
> db.personer.aggregate( [
   { $group: { _id: "$postnr", antPers: { $sum: 1 } } },
   { $sort: {antPers: -1} }
] )
```

Eksempel 4.8: *Innbyggere per postnummer sortert etter befolkning*

Spørringen i eksempel 4.8 gjør mye av det samme som spørringen i eksempel 4.7, men i eksempel 4.8 grupperes *postnr*.

Systemet over navn og bosetninger er et relativt enkelt system som ikke gir mer informasjon enn navn og postnumre som kan aggregeres over. I tilfeller som dette vil MongoDB tilby en rask og enkel løsning ettersom det ikke er behov for komplekse strukturer eller avanserte relasjoner mellom data.

4.4.2 Utvidelse med opplysninger om kommuner

Vi ønsker å utvide systemet med informasjon om kommuner, og ønsker å se på folketall og antall postnumre i hver kommune. Det kan være flere måter å utvide en dokumentdatabase på, avhengig av hvor mye data som skal legges til, og relasjoner mellom eksisterende- og nye data.

Om kommuner ønsker vi å utvide med kommunenavn og kommunenummer. Uavhengig av det forrige systemet ønsker vi dokumenter om kommuner som vist i eksempel 4.9.

```
> db.kommuner.findOne()
{
  "kommunenr": 301,
  "kommunenavn": "Oslo"
}
```

Eksempel 4.9: *Et eksempel på et dokument over kommuner i Norge*

Dersom vi vil integrere dokumentene som vist i eksempel 4.9 inn i databasen definert under dette delkapittelet, kan det gjøres på flere ulike måter avhengig av hvilken funksjonalitet som forventes av applikasjonen.

Metode 1: Slå sammen dokumentene til en denormalisert modell

Dersom systemet har behov for å utføre raske spørringer vil det være aktuelt å lagre data i et felles dokument, der *kommunenr* og *kommunenavn* legges inn som ekstra felter i persondokumentene. Fordelen med en slik modell er at systemet unngår å måtte opprette relasjoner mellom dokumenter, tilsvarende en join i relasjonsmodellen, for å utføre spørringer over innbyggere i kommuner.

```
> db.personer.findOne()
{
  "_id": 01019012345,
  "navn": "Ola Normann",
  "postnr": 0021,
  "kommunenr": 301,
  "kommunenavn": "Oslo"
}
```

Eksempel 4.10: *Personer, postnumre og kommuner*

Eksempel 4.10 viser hvordan to nye felter er lagt inn i et persondokument. Problemer som kan oppstå ved denne metoden:

- Systemet må gå gjennom hvert personobjekt i databasen og legge til to nye felter.
- Det er ingen kontroll over at det registreres riktig postnr i henhold til kommune. Det betyr at systemet tillater at det samme postnummeret kan tilhøre forskjellige kommuner. Dette kan føre til oppdateringsanomalier.
- Dersom det meldes flytting, blir systemet nødt til å oppdatere opp til tre felter. Feltene som oppdateres vil være *postnr*, *kommunenr* og *kommunenavn*.
- Kommuner kan kun aksesseres gjennom personobjektene, og dersom vi ønsker å utvide systemet med ny funksjonalitet for kommunene, kan vi ikke hente ut kommuner uavhengig av personer.

Fordelen vil likevel være at spørringene blir enklere å skrive. Det vil alltid være enklere å gjøre en spørring over ett dokument av gangen enn på tvers av dokumenter med tanken på joins. Eksempelvis vil spørringen som teller antall innbyggere i hver kommune, ligne på aggregatene som vist tidligere i kapittelet, og utvidelsen har ikke hatt en innvirkning på eksisterende funksjonalitet nevnt under delkapittel 4.4.1.

```
> db.personer.aggregate( [
    { $group: { _id: "$kommunenr", knavn: "$kommunenavn",
        antallPers: { $sum: 1 } } },
    { $sort: {antallPers: -1} }
] )

-------------Resultater---------------
{ _id: "Oslo", antallPers: 658390 }
.
.
{ _id: "Utsira", antallPers: 200 }
```

Eksempel 4.11: *Antall innbyggere i hver kommune*

Eksempel 4.11 viser aggregatet for å telle antall innbyggere i hver kommune. Resultatet fra spørringen gir en liste med dokumenter bestående av kommunenr, kommunenavn og antall personer for hver kommune sortert etter flest innbyggere.

Metode 2: La kommune stå som et eget konsept

I dette systemet er det naturlig å tenke at kommune skal stå som et eget konsept, og at kommunene ikke må aksesseres gjennom personobjektene. Fordelen med dette er å kunne ha tilgang til kommunene uavhengig av personer, og det gjør det enklere å utvide kommunene med mer data senere. Vurdering om en mulig utvidelse av systemet senere bør tas hensyn til når databasen designes. Vi kan lage egne kommunedokumenter ved å liste alle postnumre tilhørende en kommune. Vi kan dermed lage en sammenheng mellom postnumrene listet i kommuner, og postnumrene fra persondokumentene dersom vi har behov for å hente ut data om både personer og bosetning.

```
> db.kommuner.findOne()
{
  "kommunenr": 301,
  "kommunenavn": "Oslo",
  "postnumre": [0139, 0150, 0151...]
}
```

Eksempel 4.12: *Kommuner i Norge med embedding av postnumre*

Eksempel 4.12 viser hvordan feltet *postnumre* lagrer en liste over alle postnumrene i en kommune. For å sjekke hvilken kommune en person bor i må det utføres et søk gjennom postnummerlistene i hver kommune, i tillegg til et oppslag av personer. Det betyr at for hver person hentes det ut et postnummer som skal sjekkes mot hver kommune. Dette blir en spørring som utføres på applikasjonsnivå ettersom MongoDB ikke gir støtte for *joins*, og betyr at det er applikasjonskoden som utfører joins mellom data, som fører til mer kompleksitet i applikasjonskoden.

Metode 3: Bruke referanser mellom personer og postkoder

Et annet alternativ blir å gjøre postkoder til et eget konsept med informasjon om kommuner. På den måten skiller vi personer fra lokasjoner, og kan bruke referanser til å koble personer og postkoder sammen gjennom objektsreferanser.

```
> db.postnumre.findOne()
{
  "_id": <ObjectID1>,
  "postnr": 0150,
  "kommunenr: 301,
  "kommunenavn": "Oslo"
}
> db.personer.findOne()
{
  "_id":  01019012345,
  "navn:" "Ola Normann",
  "postnr": <ObjectID1>
}
```

Eksempel 4.13: *Postnumre i Norge med kommuneinformasjon*

Eksempel 4.13 viser hvordan dokumentene henger sammen ved at feltet *postnr* i et persondokument refererer til feltet *_id* i et postnummerdokument. Denne metoden har mange likhetstrekk med *metode 2*, men i dette tilfellet vil vi få et raskere søk fordi vi kun skal søke etter et enkelt objekt, i stedet for å gjøre et søk gjennom lister. Ulempen med denne metoden vil likevel være mer bruk av lagringsplass siden vi får redundans i informasjonen om kommuner. En kommune som Oslo har flere hundre postnumre, og informasjon om Oslo vil dermed bli lagret mange ganger.

4.5 Problematikk og diskusjon

I utgangspunktet bør alle beslektede data lagres i det samme dokumentet i et forsøk på å unngå *joins*, fordi det ikke støttes i MongoDB. Som en konsekvens av dette får vi ofte denormalisert data, og mer komplekse strukturer i dokumentene. MongoDB legger dette frem som en anbefaling, men nevner også at det finnes situasjoner der dette ikke er fornuftig [25]. Løsningen som legges frem er en rikere struktur med bruk av *referanser* som vist i *metode 2* og *3*, men med større belastning i applikasjonens programkode for å manipulere data. For å kunne få samme resultat som i den første metoden blir applikasjonen nødt til å utføre to spørringer, én for hver av samlingene, og deretter manipulere og slå dem sammen på applikasjonssiden.

Spørsmålet er om vi ønsker å flytte arbeid fra databasesiden og inn i applikasjonssiden? Dette forutsetter at de som utvikler og vedlikeholder systemet må gjøre seg godt kjent med både databasen og applikasjonen for å forstå nøyaktig hvordan data kan manipuleres, og det kan variere i stor grad mellom ulike applikasjoner hvordan dette gjøres. Relasjonsdatabaser og SQL stiller dermed sterkere på dette området, der databasen har en rikere struktur og et sterkere spørrespråk.

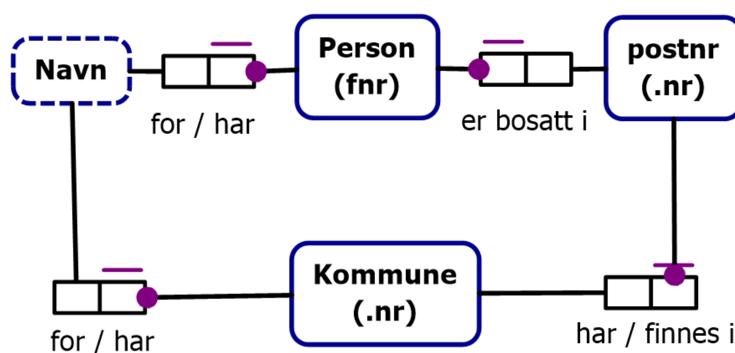
I en artikkel kalt *A Study of Normalization and Embedding in MongoDB* [26], blir det lagt frem en studie om normalisering av MongoDB, der resultatene viser at normalisering i MongoDB er langt mindre effektivt enn i SQL og relasjonsdatabaser. Artikkelen baserer seg på kjøretiden av ulike spørringer etter hvordan databasen er normalisert, men mangler å legge frem hvilke spørringer som er testet. Artikkelen er likevel et bidrag for å understreke at embedding av data gir en jevn kjøretid av de ulike spørringene, mens normalisering fører til mer inkonsistente kjøretider.

Ser vi på den samme problemstillingen i en relasjonsdatabase, ser vi for oss at utgangspunktet er den samme som i eksempel 4.5. Figur 4.2 viser den tilsvarende ORM-modellen.

Dersom vi har den samme problemstillingen med å utvide datamodellen med informasjon om kommuner, kan ORM brukes som et nyttig verktøy for å følge prinsippene i relasjonsmodellen. Løsningen blir å legge til *kommune* som et eget *begrep* der *kommunenr* identifiserer en kommune entydig. Det finnes flere kommuner i Norge med samme navn slik at *kommunenavn* vil stå som en *verditype* med en *en-til-mange* relasjon til *kommune*.

Det har blitt gjort en endring mellom figur 4.2 og figur 4.3, der verditypen *postnr* har blitt gjort om til et begrep for å unngå det som kalles for en

Figur 4.2: *ORM-modell med utgangspunktet i eksempel 4.5*



Figur 4.3: *ORM-modell fra figur 4.2 utvidet med kommuner*

synonym bro[3]. Ut ifra figur 4.3 kan vi lage et relasjonsskjema ved hjelp av *realiseringsalgoritmen*, og få en normalisert relasjonsdatabase.

Sammenlignet med MongoDB virker denne prosessen mindre avansert da det er enklere å følge designprinsippene for relasjonsmodellen. Dokumentdatabaser har friere tøyler, men fallhøyden blir større dersom en skulle velge feil metode når databasen utvides. Dette viser hvordan systemer med et behov for rikere datastrukturer får et problem når det kommer til utvidelse av data i MongoDB.

Det kan være en utfordring å vite hvilken databaseteknologi som er egnet til ulike prosjekter. Fordelen ved å velge dokumentdatabaser er at det er raskt å komme i gang, hurtig dataaksessering, selv med enorme datasett, og det er egnet for distribuering [27]. Når er det vi faktiske trenger disse egenskapene? Hvor ofte sitter vi med så store data at vi har behov for å distribuere dem, eller at en relasjonsdatabase ikke er rask nok? Ofte settes prosjekter i gang uten at det gjøres en grundig vurdering av databasevalget,

[3]En bro med entydighetsskranke over rollen til verditypen

og dokumentdatabaser velges ofte ut ifra at det ikke er like stort behov for å sette seg inn i, eller velge skjema. Med andre ord blir det brukt som en rask og enkel løsning. Når er feil databaseteknologi valgt, og hvordan oppdages dette?

Dersom dokumentdatabaser er valgt fra begynnelsen, vil det som regel være uproblematisk dersom applikasjonen ikke utvider seg med nye funksjoner, som har behov for å legge til nye datafelter. Med en gang dette skjer, er det en risiko for at datasettet vokser i ukontrollerte omgivelser, grunnet den skjemaløse modellen. Skulle det videre oppstå et behov for å holde en rik struktur i bunnen der det kan uføres komplekse spørringer, bør migring til en databaseteknologi som støtter dette vurderes.

Kapittel 5

Fra MongoDB til ORM

La oss anta at vi ønsker å migrere fra en MongoDB-database til en relasjonsdatabase. Vi vil trekke ut semantikken fra en dokumentsamling, og uttrykke et skjema ved hjelp av ORM. Dette gir økt fleksibilitet når modellen skal realiseres ved å dra nytte av semantikken uten å være bundet til syntaksen i en teknologi. Vi skal likevel rette fokuset mot en relasjonsdatabase som preferert slutteknologi. I dette kapittelet skal vi se på strategier, forutsetninger og noen ulike databaser og hvilke behov de har for å migrere.

5.1 Antakelser

Dokumentdatabaser kommer i mange ulike former, og vi skal prøve å gjøre rede for karakteristikken som må til for å kunne migrere. Dersom vi har en database med en samling av dokumenter, der dokumentene har forskjellige felter og varierer i stor grad i struktur, med andre ord en database som utnytter muligheten av et fleksibelt skjema, vil ikke migrering nødvendigvis være en fordel. På grunn av det strenge kravet til et skjema i relasjonsdatabaser, må vi anta at dokumentdatabasen følger et implisitt skjema, og vi må være i stand til å definere det eksplisitte skjemaet som benyttes i et relasjonsdatabasesystem. En enkel antakelse som sikrer dette er at alle dokumenter i en samling bør ha de samme feltene med samme *verdityper*[1]. I ORM og relasjonsdatabaser er det tillatt med *null*-verdier, og det må regnes med at ikke alle dokumentene inneholder nøyaktig de samme feltene, men store variasjoner vil gi en database med mange *null*-verdier. En viktig antakelse for å migrere fra en dokumentabase er at databasen i begrenset grad har et fleksibelt skjema, og at dokumentene ser tilnærmet like ut, selv

[1]Med type menes strenger, tall, lister eller dokumentobjekter

om små variasjoner kan forekomme.

Videre blir vi nødt til å gjøre noen antakelser for å kunne utføre en skjemaekstraksjon av en MongoDB-samling til ORM. Forutsetningen for en gyldig ORM-modell er at alle begreper må ha en preferert referanse, som i et relasjonsskjema vil bli satt som primærnøkkel, og vi antar derfra at det finnes en unik id for hvert dokument innad en samling. I MongoDB er dette ofte oppgitt som *_id*, eller *samlingsnavn_id*. MongoDB bruker *_id* som en standard preferert referanse, og dersom et dokument ikke inneholder feltet *_id*, vil databasen automatisk generere feltet med en objektreferanse. Vi får kun hentet ut objektreferansen dersom vi eksporterer dokumentene fra en MongoDB-database, men vi kan ikke forvente at dette feltet alltid eksisterer. Flere av MongoDBs eksempler kommer uten objektsreferanser.

En dokumentdatabase er bygget opp med utgangspunkt i en gitt mengde spørringer, fordi det ønskes et design som gjør spørringene effektive, som ved bruk av embedding. Dette gjør at en spørring kan ha tilgang til en mengde beslektet data som lagres i det samme dokumentet, der det ellers i et relasjonsskjema ville ha blitt delt opp i flere relasjoner for å unngå oppdateringsanomalier. Vi ønsker dermed å utføre en analyse av spørringene og se om det er mulig å finne informasjon om et skjema basert på en statisk analyse. Vi må derfor anta at vi har tilgang på typiske spørringer tilknyttet databasen.

5.2 Metoder

Vi skal se på tre forskjellige metoder for å finne informasjonen vi trenger for å trekke ut skjema fra en MongoDB-database til en ORM-modell. I *direkte kartlegging* undersøkes det hvilken informasjon som er mulig å trekke ut fra dokumenter i en samling. Dette er nok den vanligste metoden som legges til grunn ved migrering. Dokumentet skal analyseres etter struktur og oppbygning, og det vil omtrent fungere som en ren oversettelse. Antatte problemer rundt denne metoden er denormaliserte strukturer i dokumentdatabaser, som gjør at en direkte kartlegging vil gi et denormalisert relasjonsskjema. I denne oppgaven er målet et relasjonsskjema med høyest mulig normalform.

I en *statisk analyse* gjøres det en analyse av oppbygningen til en spørring. Dette kan gi informasjon som direkte kartlegging ikke gir fordi en spørring kan si mer om hvordan data er relatert. Det kan være å vurdere hva intensjonen til spørringen er, eller å se på hvordan data *grupperes* sammen i aggregater. Årsaken til at dette er relevant er fordi spørringer og oppbygningen av dokumenter henger sterkt sammen, med tanken på at det

for eksempel bygges en struktur som tar hensyn til at spørringen unngår å måtte bruke *joins*. Det tas hensyn til hvor raskt en spørring har tilgang til data.

Den siste metoden kalt *dynamisk analyse av data* kan kartlegge antatte funksjonelle avhengigheter. Dokumentdatabaser har ofte en denormalisert struktur, og funksjonelle avhengigheter kan indikere potensielle dekomposisjoner til høyere normalformer. Dette gjøres ved å kjøre gjennom data og finne ut om datamaterialet oppfører seg i henhold til en antatt funksjonell avhengighet. Det er mulig at dette søket må begrenses fordi algoritmen for å gjøre en slik analyse er eksponentiell. Det skal undersøkes hvilke begrensninger som skal velges, og hvorvidt dette er en nyttig metode i situasjoner med feil i data, eller at datasettet ikke er stort nok.

5.3 Direkte kartlegging

Vi ser først på *direkte kartlegging* som vil være å ta et dokument fra en samling og direkte skrive det om til en ORM-modell. En samling tilsvarer et begrep, feltene i et dokument tilsvarer et nytt begrep eller en verditype, og hva slags datatyper som finnes i verdifeltene til dokumentet kan indikere setningstypene. Det kan være problematisk å kartlegge rollerbeskrivelser og totale roller fordi en direkte kartlegging hverken kan finne naturlig språk for å beskrive hvordan data er beslektet, eller sjekke om det finnes forekomster av data for alle felter i alle dokumentene.

Dokumentdatabase		ORM
C er en samling	→	C er et begrep
D er et dokument i C	→	D har begreper/verdityper for C
Felt x i D består av et dokument	→	x er et begrep
Felt y i D består av en liste	→	y er et begrep
Felt z i D er en streng/et tall	→	z er en verditype

Tabell 5.1: *Direkte kartlegging av en dokumentsamling til en ORM-modell. Setningstyper skal genereres fra de ulike tilfellene av x, y og z*

Tabell 5.1 viser hvordan en samling C kan kartlegges til ORM. Vi er kun ute etter modell og semantikk fra oppbygningen av dokumenter, og det er derfor ikke interessant å se på forekomster av data. Tabell 5.1 beskriver hvordan det skal opprettes et nytt begrep eller en ny verditype i ulike situasjoner.

Vi antar at det finnes tre forskjellige tilfeller som vist med x, y og z fra tabellen.

Der x består av et dokument i form av embedding ønsker vi å opprette et nytt begrep fordi x kan oppfattes som et eget konsept som igjen kan inneholde nye verdityper eller relasjoner til andre begreper. Mellom C og x skal det opprettes en *faktatype* med en entydighetsskranke over rollen til C, fordi x kun består av ett dokument. Skranken over rollen til x forblir ukjent.

Der y består av en liste ønsker vi å opprette et nytt begrep fordi vi ønsker å beskrive et *mange*-forhold mellom C og y. Vi vet dermed at skranken over C vil være for mange forekomster som en liste indikerer, men skranken over rollen til y forblir ukjent. Dersom listen består av verdier hadde det vært et alternativ å opprette y som en verditype, men vi risikerer å få *synonyme* broer, eller *mange-til-mange*-broer.

Der z kun består av en verdi, som for eksempel en streng eller et tall, skal det kun opprettes verdityper. Det skal opprettes broer mellom C og z med en entydighetsskranke over rollen til C fordi feltet z kun består av én verdi. Skranken over rollen til z forblir ukjent.

Ved å anvende direkte kartlegging får vi en indikasjon på hvordan relasjonsskjemaet vil se ut. Vi kan begynne med å se på et enkelt eksempel der vi har et dokument som ikke inneholder lister eller embedding av dokumenter. Eksempel 5.1 viser et enkelt persondokument hentet fra samlingen *Person*. I tilfeller uten lister eller embedding av dokumenter forventes det kun ett begrep i modellen.

```
> db.person.findOne()
{
  "_id": "01019012345",
  "name" : "John Doe",
  "phone" : "12345678"
}
```

Eksempel 5.1: *Et persondokument som skal kartlegges til ORM*

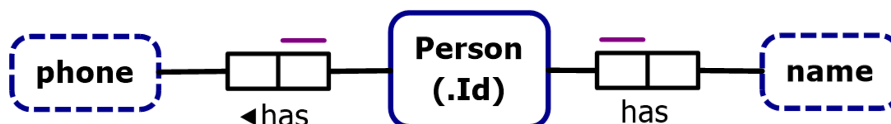
Følger vi stegene fra tabellen i 5.1 får vi tabell 5.2. I tillegg får vi tre nye broer med entydighetsskranker over alle rollene til *Person*. Direkte kartlegging kan ikke avdekke nøkler, men vi antar i dette tilfellet at *_id* holder

som en preferert referanse.

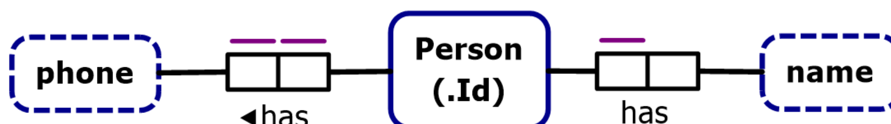
Dokumentdatabase		ORM
Person er en samling	→	Person er et begrep
Feltet *_id* består av en streng	→	_id er en verditype
Feltet *name* består av en streng	→	name er en verditype
Feltet *phone* består av en streng	→	phone er en verditype

Tabell 5.2: *Direkte kartlegging av samlingen person*

Med tabell 5.2 kan vi konstruere ORM-modellen som vist i figur 5.1. For enkelthets skyld oppretter vi rollenavn som *har/has* fordi det ikke vil være mulig å trekke ut naturlige rollenavn som beskriver forholdet mellom data. Med direkte kartlegging forblir entydighetsskranken over rollene til *phone* og *name* ukjent. Figur 5.1 viser et alternativ der vi antar hvilke skranker som skal gjelde for rollene til *phone* og *name*. Figur 5.2 viser et annet alternativ der det er satt en entydighetsskranke over rollen til *phone*.



Figur 5.1: *Alternativ 1. ORM-modell etter direkte kartlegging av et persondokument.*



Figur 5.2: *Alternativ 2. ORM-modell etter direkte kartlegging av et persondokument med en entydighetsskranke over rollen til phone.*

Ved direkte kartlegging er det ikke mulig å avdekke hvilket alternativ som er korrekt. Vi velger derfor å anta alternativer med færrest integritetsregler,

og senere legge til nye skranker etterhvert som de oppdages gjennom andre metoder. Vi velger derfor å anta at figur 5.1 gir det beste alternativet for videre analyse.

Vi kan se på et annet eksempel med dokumenter som støtter *embedding*. I eksempel 5.2 har vi det samme dokumentet som i eksempel 5.1 men utvidet med embedding av adressedokumenter. Dersom vi bruker de samme stegene fra tabell 5.1 får vi tabell 5.3.

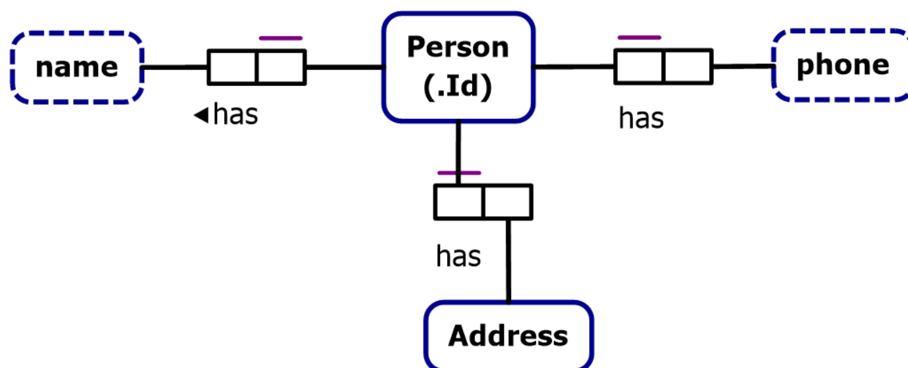
```
> db.person.findOne()
{
  "_id": "01019012345",
  "name" : "John Doe",
  "phone" : "12345678",
  "address" :
        {"street": "ABC 20", "city": "Oslo", "zip": 0850}

}
```

Eksempel 5.2: *Embedding av adresse*

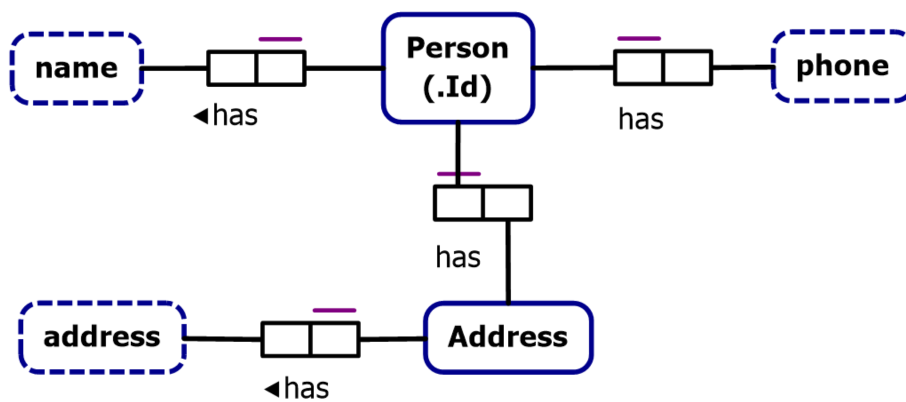
Dokumentdatabase		ORM
Person er en samling	→	Person er et begrep
_id består av en streng	→	_id er en verditype
name består av en streng	→	name er en verditype
phone består av en streng	→	phone er en verditype
address består av et dokument	→	address er et begrep

Tabell 5.3: *Direkte kartlegging av samlingen person med embedding av adresser*

Hva skjer med dokumentet i feltet *address*? Modellen i figur 5.3 viser et tomt adressebegrep. Det finnes flere måter å løse dette problemet på. Vi kan ignorere embeddingen ved at hele dokumentet i *address* lagres i relasjonsdatabasen som vist i tabell 5.4, men dette bryter kravet om atomære verdier. Dersom vi velger denne metoden må det opprettes en verditype i begrepet *address* vist i figur 5.4 som skal inneholde hele dokumentet som en streng.



Figur 5.3: *ORM-modellen opprettet fra tabell 5.3*



Figur 5.4: *ORM-modellen opprettet fra tabell 5.3 med address som verditype i begrepet Address*

id	name	phone	address

01019012345	John Doe	2345678	{
			street : ABC 20,
			city : Oslo
			zip: 0850
			}

Tabell 5.4: *Et eksempel på hvordan data vil bli lagret i databasen uten videre kartlegging av indre dokumenter.*

Problemet med modellen i figur 5.4 er at *Adress* forblir et undertrykt begrep, og at vi i utgangspunktet kunne ha opprettet en ny verditype for adresse i stedet for å gjøre adresser til et begrep. For å hente ut informasjonen om en adresse blir databasen eller applikasjonen nødt til å behandle tekststrengen som inneholder data om adresser. Det kan oppstå problemer med oppdateringer, som for eksempel der databasen skal endre en adresse.

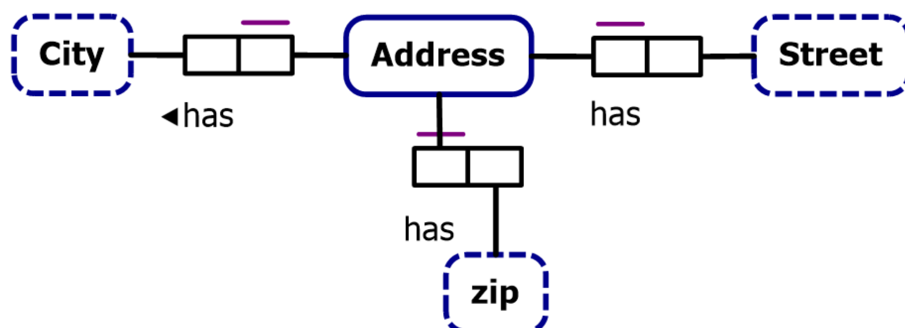
Alternativt kan vi kartlegge adressedokumentene. Utfører vi de samme stegene fra tabell 5.1 ønsker vi å se på *address* som et eget dokument. Tabellen i 5.5 viser hvordan en adresse direkte kartlegges til et begrep der feltene til et adressedokument hentes ut og kartlegges til nye verdityper med relasjoner til adressebegrepet. Det blir i tillegg opprettet broer med entydighetsskranke over alle rollene til *Address*. Modellen er illustrert i figur 5.5.

Dokumentdatabase		ORM
Address er en samling	→	Address er et begrep
street består av en streng	→	street er en verditype
city består av en streng	→	city er en verditype
zip består av et tall	→	zip er en verditype

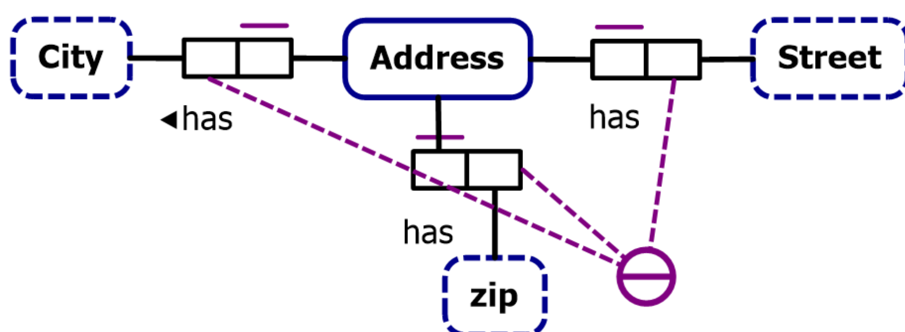
Tabell 5.5: *Direkte kartlegging av embeddet adresseobjekter*

Problemet nå er at adressebegrepet mangler en preferert referanse, og vi velger derfor å følge relasjonsmodellen ved å sette kombinasjonen av alle verditypene i adressebegrepet til en unik identifikator som vist i figur 5.6. I dette tilfellet er *Address* fremdeles et undertrykt begrep, men vi har klart å bryte opp data relatert til adresser og opprettholder kravet om atomisitet i relasjonsmodellen.

Figur 5.6 er konstruert uten totale roller selv om dette hører med i en



Figur 5.5: *ORM-modellen opprettet fra tabell 5.5*



Figur 5.6: *ORM-modellen for Address med preferert referanse*

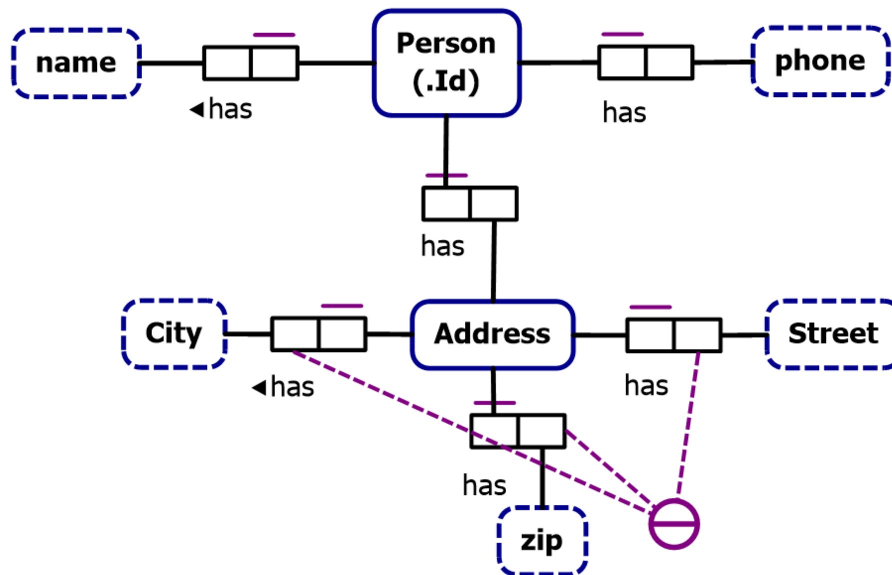
preferert referanse. Det må diskuteres hvorvidt adressedokumentet i dette tilfellet er representativ for alle adressedokumentene i samlingen. Vi kan ikke se bort ifra at dokumentene kan ha ulike felter, eller at det ikke finnes null-verdier. Det betyr at vi blir nødt til å gjøre en del antakelser, som i utgangspunktet ikke er karakteristisk for en dokumentdatabase, for å kunne få til en direkte kartlegging.

I mange tilfeller vil dokumentene ha flere nivåer av embedding, og vi blir nødt til å repetere de samme stegene for kartlegging helt til vi når det dypeste laget i dokumentet. Det finnes ingen grense for hvor dypt dette kan være utenom størrelsesbegrensningen til et dokument. Nedenfor ser vi realiseringen av både *Person* og *Address* inkludert fremmednøkkelen mellom relasjonene. Vi ønsker ikke å undertrykke *Address* fordi vi enkelt kan legge til flere verdier dersom vi ønsker å utvide begrepet. Dette gir rom for økt

fleksibilitet.

```
Person(id, name, phone, street, city, zip)
Address(street, city, zip)
Fremmednøkler: Person(street, city, zip) → Address
```

Figuren i 5.7 viser nå hvordan vi kan få en rikere struktur i databasen ved å kartlegge embeddede dokumenter. *Address* står som et eget konsept der vi lett har tilgang på informasjon om en adresse, og står med et bedre utgangspunkt for å utvide adressebegrepet ytterligere.



Figur 5.7: *ORM-modellen etter direkte kartlegging av adresse*

Et annet problem som oppstår ved direkte kartlegging er embedding av lister som også er et brudd på atomære verdier, og det finnes flere måter å håndtere dette på. Listen kan lagres i databasen som den er, eller så kan den brytes opp for å få relasjonen mest mulig korrekt etter relasjonsmodellen ved å kun lagre atomære verdier.

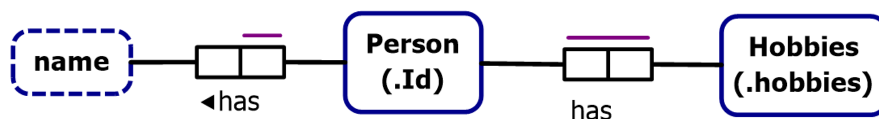
En liste kan indikere at et dokument kan ha mange verdier for det samme feltet. Lister kan inneholde verdier, objekter eller lister av lister. La oss se på eksempel 5.3 som viser et persondokument med en liste av hobbyer.

```
> db.persons.findOne()
{
  "_id": "123",
  "name": "John Doe",
  "hobbies": ["Music","Football","Tennis"]
}
```

Eksempel 5.3: *Personer med en liste av hobbyer*

Gitt kartleggingstegene fra tidligere skal felter som inneholder lister kartlegges til et nytt begrep, hovedsaklig for å unngå synonyme broer. Dersom listen igjen består av dokumenter må det gjøres til et begrep fordi vi ønsker å se på alle dokumenter som egne konsepter. Igjen står vi med samme den problemstillingen som tidligere. Vi kan gjøre det enkelt ved å lagre hele lister i databasen vi migrerer til, eller så kan vi følge relasjonsmodellen ved å dele opp data ytterligere. De fleste relasjonsdatabaser gir støtte for lister, men vi ønsker å følge relasjonsmodellen i størst mulig grad for å dra nytte av funksjonelle avhengigheter og normalformene.

Ved lister som inneholder strenger eller tall kan vi konstruere modellen i figur 5.8. Det opprettes et nytt begrep av feltet som inneholder en liste, og deretter opprettes det en verditype i det nye begrepet for å kunne lagre listeverdiene. Vi kan anta at verditypen kan brukes som en unik identifikator i dette tilfellet. Deretter opprettes det en *mange-til-mange*-faktatype fordi vi kjenner til rollen til *Person*. Skranken over rollen til *Hobbies* er fremdeles ukjent, og vi antar alternativet med færrest integritetsregler.



Figur 5.8: *ORM-modellen etter direkte kartlegging personer og hobbyer*

Ved lister som inneholder dokumenter kan vi anvende samme metode som vi gjorde i eksempelet over, og deretter anvende metoden for å kartlegge indre dokumenter.

5.3.1 MongoDBs zip-eksempel

Det finnes en testdatabase gitt ut av MongoDB som inneholder data over *zip codes* i USA. Dokumentet i eksempel 5.4 viser et eksempeldokument fra samlingen *zips*.

```
> db.zips.findOne()
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [-74.016323, 40.710537]
}
```

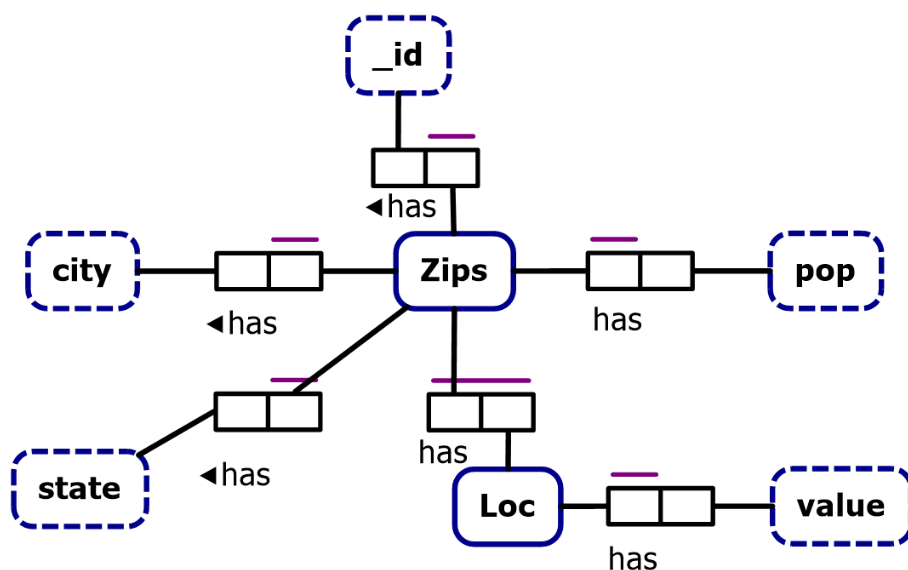
Eksempel 5.4: *MongoDBs zip-eksempel hentet fra dokumentasjonen*

Anvender vi direkte kartlegging får vi resultatet i tabell 5.6. I tillegg genereres alle tilhørende setningstyper og rollepar basert på antakelsene om hvilke entydighetsskranker vi skal sette i de ulike situasjonene.

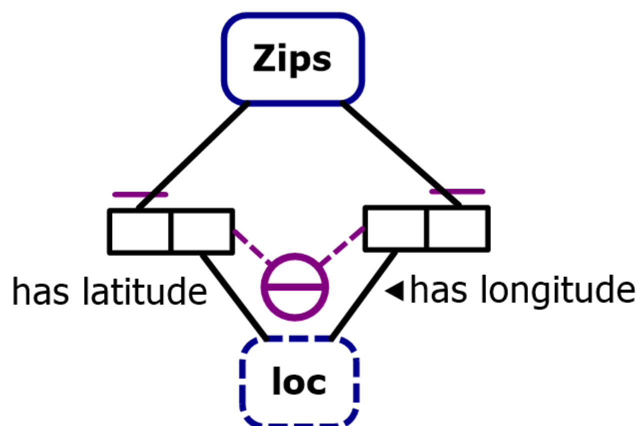
Dokumentdatabase		ORM
Zips er en samling	→	Zips er et begrep
state består av en streng	→	state er en verditype
city består av en streng	→	city er en verditype
pop består av et tall	→	pop er en verditype
loc består av en liste	→	loc er et begrep

Tabell 5.6: *Direkte kartlegging av MongoDBs zip-eksempel*

Listen i *loc* består av koordinatverdier gitt som breddegrad og lengdegrad, og i direkte kartlegging vil dette bli tolket som en liste med tallverdier. Det fører til at vi kommer til å miste semantikken bak hva som er en breddegrad og hva som er en lengdegrad i databasen. I figur 5.9 vises ORM-modellen vi får fra en direkte kartlegging, og figur 5.10 viser hvordan vi i praksis hadde modellert *loc* basert på domenekunnskap.



Figur 5.9: *ORM etter direkte kartlegging av MongoDBs zips-eksempel*



Figur 5.10: *loc modellert basert på domenekunnskap*

5.3.2 Problematikk

Hva om vi har flatere strukturer i dokumentene? Dette forekommer svært ofte i dokumentdatabaser. Dersom vi ser på *person-adresse*-eksempelet fra eksempel 5.2 igjen med en flatere struktur som vist i eksempel 5.5, har vi ingen måte vi kan kartlegge *street*, *city* og *zip* som et eget adressekonsept. Dette fører til at vi mister fleksibiliteten i databasen fordi vi ikke kan legge til ny adressedata uavhengig av personer. Dette er et av hovedproblemene med direkte kartlegging og vi ender opp med en relasjon som vist under. I tillegg kan vi anta at det finnes en funksjonell avhengighet mellom *zip* og *city*, gitt som $zip \rightarrow city$ som direkte kartlegging ikke klarer å fange opp.

```
Person(_id, name, phone, street, city, zip)
```

```
> db.person.findOne()
{
  "_id": "01019012345",
  "name" : "John Dove",
  "phone" : "12345678",
  "street": "ABC 20",
  "city": "Oslo",
  "zip": 0850
}
```

Eksempel 5.5: *Personer med adresser*

Et annet problem ved direkte kartlegging er hvilke dokumenter vi velger å kartlegge. Vi kan ikke anta at dokumentene vi velger å kartlegge gir et skjema som holder for hele samlingen, fordi dokumentene kan variere i struktur og felter. Vi må enten gjøre en preprosessering for å finne- eller generere et dokument som inneholder alle feltene som finnes i samlingen. En annen mulighet er å kartlegge alle dokumentene, og deretter slå sammen ORM-modellene.

Videre i kapittelet skal vi se på metoder som undersøker om vi kan finne metainformasjon som direkte kartlegging ikke dekker. For eksempel kan det være interessant å finne ut om rollene til *phone* og *name* fra eksempel 5.1 skal være unike, om *person-hobby*-scenariet fra eksempel 5.3 vil gi en *mange-til-én*-faktatype i stedet for en *mange-til-mange*-faktatype, eller om vi klarer å kartlegge funksjonelle avhengigheter.

5.4 Statisk analyse av spørringer

Med *statisk analyse* av spørringer menes en analyse av oppbygging og innhold av spørringer uten å måtte kjøre spørringene. Ved hjelp av en slik analyse kan vi anta hvordan data henger sammen, og vi kan vite en del om hvilke antakelser systemet må gjøre for å kjøre spørringene. Med andre ord kan vi få en del informasjon om det *implisitte skjemaet*[2].

Dersom vi har en samling av data må vi anta at vi har et sett med spørringer som har vært med på å definere strukturen til dokumentene og databasen. En dokumentdatabase blir ikke konstruert tilfeldig. Den konstrueres ut ifra hva som forventes av data, og ut ifra spørringene som skal hente informasjon fra databasen. I mange tilfeller blir dokumentene konstruert slik at spørringene blir så enkle som mulig å skrive, og det hender at en god og oversiktlig struktur blir nedprioritert. Dette er ofte årsaken til denormalisering der effektive spørringer og responstid har en høyere prioritet en selve strukturen på dokumentene.

Ved å se på spørringene kan vi prøve å forstå hva hensikten med spørringen er, og hvordan den lager sammenhenger mellom data. Vi skal prøve å utforske om dette kan bidra til å kartlegge metainformasjon som kan brukes til migreringen. Ettersom spørringer kan variere i stor grad vil det være en utfordring å automatisere prosessen.

I dette delkapittelet skal vi hovedsaklig se på aggregatfunksjoner. De vanlige CRUD-operasjonene kan være nyttig å undersøke, men enkle CRUD-operasjoner vil ikke gi annen informasjon enn feltnavn og hvilke data som finnes i databasen. MongoDB tilbyr i tillegg *MapReduce*-operasjoner, som grupperer data til nyttige aggregater. Vi skal derfor hovedsaklig fokusere på aggregatspørringer der vi skal undersøke om vi kan trekke ut semantikk fra måten data grupperes, og om det finnes måter å undersøke hvordan data er relatert.

5.4.1 Idé, struktur og prinsipper

For å kunne trekke semantikk ut av en spørring må vi gjøre rede for hva vi skal se etter i spørringene. Idéen er å undersøke om spørringer kan gi en indikasjon på relasjoner mellom data ved grupperinger. I aggregater finnes det grupperinger av data der vi kan anta at data som grupperes ikke kan være unike i samlingen, med mindre spørringen kaller på *$unwind*, *$lookup*, *$sample*, eller andre funksjoner som kan gi duplikater av nøkler i forkant

[2]Hvilke antakelser applikasjonen har for innholdet i databasen

av en gruppering. Det er ingen grunn til å gruppere på unike verdier fordi det kun vil lage grupper av data tilsvarende dokumentene i samlingen, slik at hvert dokument danner én gruppe. Vår første antakelse er at spørringene fra en samling er fornuftig skrevet uten unødvendig kompleksitet. Ut ifra antakelsen om at alle felter det grupperes på ikke kan være unike, vil vi kunne kartlegge informasjon om datasettet som *direkte kartlegging* ikke fanger opp.

Et aggregat er bygget opp ved å liste hvordan data skal behandles gjennom en *pipeline* som vist i figur 5.11. Vi kan se på dette som en liste av steg, der hvert steg kan inneholde ulike funksjoner som *$project*, *$match*, *$group*, og en rekke andre funksjoner. Listen er ordnet, og det første steget behandler data fra den fulle databasen. Deretter behandles hvert steg med data fra det forrige steget, og det genereres midlertidige dokumenter i MongoDBs *pipeline* som vist i figur 5.12. Vi skal undersøke i hvilke steg det finnes en *$group*-funksjon, og for hver av stegene undersøke hvordan data er gruppert basert på det forrige stegets resultater.

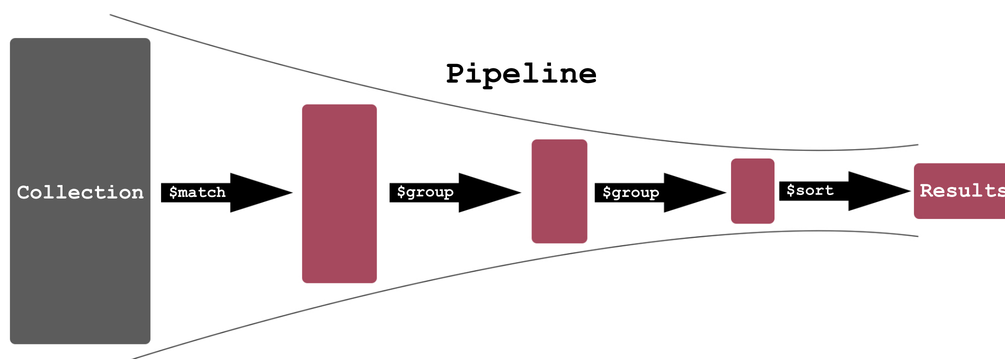
```
db.example.aggregate( [
    { $match: ...},
    { $group: ... },      1,..,n steps
    { $group: ... },
    { $sort: ...}
] )
```

Figur 5.11: *Eksempel på aggregering med fire steg.*

5.4.2 MongoDBs zips-eksempel

Vi kan se på et eksempel gitt ut av MongoDB som er en database over alle postkoder i USA [28]. De legger frem dette eksempelet for å illustrere aggregering med MongoDB.

Eksempel 5.4 fra forrige delkapittel viser et dokument i samlingen *zips*. MongoDB presiserer at alle dokumentene følger den samme modellen med feltene *_id*, *city*, *state*, *pop*, og *loc*. Vi kan anta at dette er det *implisitte skjemaet*, og at det ikke er nødvendig å ta hensyn til avvikende dokumenter.



Figur 5.12: *MongoDBs pipeline med fire steg.*

Ved å se på spørringene til samlingen skal vi finne ut om det er mulig å avdekke ny metainformasjon som direkte kartlegging ikke dekker.

Aggregat 1: Populasjonen i hver stat

Det første eksempelet fra [28] vises i eksempel 5.6, og er en spørring som finner alle stater med en befolkning over 10 millioner. Spørringen starter med en gruppering på *state*. Dette betyr at for hver stat vil vi få informasjon om *totalPop* som er summen av *pop* for hver stat, og videre at en stat repeteres flere ganger i datasettet som impliserer at det må finnes flere postkoder i én stat.

Fra direkte kartlegging kunne vi vite at en postkode bare kunne ha én stat, mens forholdet mellom stat til postkode forble ukjent. Ved hjelp av denne spørringen kan vi anta at det foreligger en *mange-til-én*-relasjon mellom postkoder og stat.

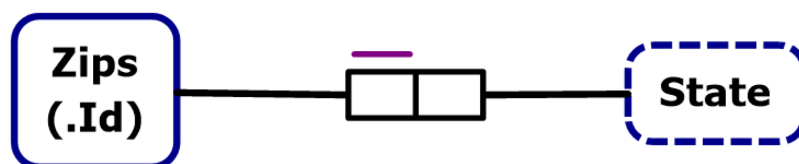
```
> db.zips.aggregate( [
    { $group:{ _id: "$state", totalPop: { $sum: "$pop"}}},
    { $match:{ totalPop: { $gte: 10*1000*1000 }}} )
```

Eksempel 5.6: *Populasjonen i hver stat over 10 millioner*

I spørringen i eksempel 5.6 er det ikke interessant å se på *$match* da den ikke gir informasjon om data, men er delen av spørringen som setter en betingelse

på hvilke resultater som er ønsket fra spørringen. Dette tilsvarer **WHERE** eller **HAVING** i SQL. Så langt er det kun *$group* som er interessant for analysen.

Figuren i 5.13 viser hvordan relasjonen mellom *zips* og *state* er representert som en *mange-til-én*-faktatype. Det eneste figuren mangler for å være en gyldig og komplett ORM-modell er passende rollebeskrivelser.



Figur 5.13: *ORM-modell etter direkte kartlegging og analyse av spørringen i eksempel 5.6*

I eksempelet over er det viktig å trekke informasjon ut av *$group*-funksjonen. Som nevnt tidligere må det antas at det kun blir gjort fornuftige grupperinger i spørringene som analyseres. For eksempel vil det ikke være hensiktsmessig å gruppere på *_id* da dette allerede er unikt for hvert dokument og det ikke kan lages nye grupper av dokumentene.

Aggregat 2: Gjennomsnittlig bypopulasjon i hver stat

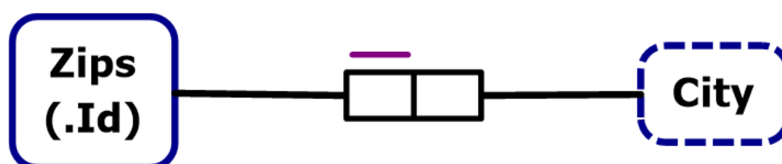
Den neste spørringen vi skal se på returnerer det gjennomsnittlige innbyggertallet for byene i en stat, og i denne spørringen grupperes det på mer enn ett felt. Det lages grupper på alle unike kombinasjoner av *state* og *city*, og nedenfor vises stegene i spørringen fra eksempel 5.7. I første steg grupperes dokumentene på kombinasjonen av *state* og *city*. I det neste steget summeres *pop* for hver *zip*, og et midlertidig resultat lagres i en *pipeline*. Deretter grupperes *state* på nytt for å regne ut gjennomsnittet for hver by gitt en stat.

```
$group: state, city , $sum: pop
                ↓
   $group: state ,$avg: pop
```

```
> db.zips.aggregate( [
    { $group:
      {
        _id: { state: "$state", city: "$city" },
        pop: { $sum: "$pop" }
      }
    },
    { $group:
      {
        _id: "$_id.state",
        avgCityPop: { $avg: "$pop" }
      }
    }
])
```

Eksempel 5.7: *Gjennomsnittlig bypopulasjon i hver stat*

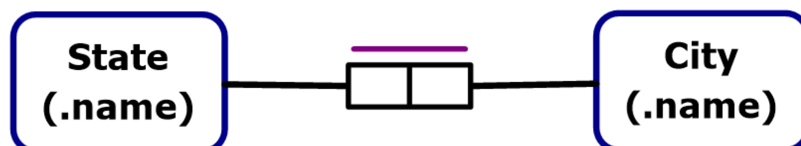
Ved at det grupperes på både stat og by indikerer at ingen av feltene er unike i samlingen *Zips*. Dersom for eksempel *city* hadde vært unik i *Zips*, hadde grupperingen blitt unødvendig da vi bare hadde fått ett dokument per gruppe. Ut ifra antakelsen om nødvendige grupperinger kan vi konkludere med at det finnes flere zipkoder i en by som vist i figur 5.14.



Figur 5.14: *ORM-modell etter direkte kartlegging og analyse av spørringen i eksempel 5.7*

Det kan være vanskelig å si noe om det finnes en sammenheng mellom en stat og en by, og det er derfor viktig å undersøke hvorfor de er gruppert sammen. I utgangspunktet ville vi ha tenkt at en by kun kan forekomme i én stat, mens en stat kan ha mange byer. Problemet med datasettet er at det ikke finnes en egen unik identifikator for en by, og mange byer heter det samme. Spørringen antar at det ikke finnes byer med samme navn gitt en

stat, og identifiserer en by gjennom navnet på byen og staten byen ligger i. Vi kan anta et *mange-til-mange*-forhold mellom by og stat der et bynavn kan gå igjen i mange stater, og en stat kan ha mange byer.



Figur 5.15: *ORM-modell av stat og by etter analyse av spørringen i eksempel 5.7*

I figur 5.15 er *state* og *city* laget som begreper for å illustrere en *mange-til-mange*-faktatype. Problemet med denne modellen er mange byer kan hete det samme, og det blir ikke korrekt å sette *name* som en unikt representasjon for city. Vi ønsker derfor ikke ha denne modellen med i analysen, og det er nødvendigvis ikke relevant å vite hvilken relasjon det er mellom *state* og *city*. Vi ønsker derfor å forholde oss til figur 5.13 og 5.14.

I det neste steget i spørringen grupperes det på stat fra den forrige utregningen. En slik gruppering gir en indikasjon på at det finnes mange *pop* i hver stat regnet ut fra steget før. Da må det også finnes mange byer i hver stat fordi en ny gruppe basert på stater må ha mange byer ved at steget før lagde grupper av unike kombinasjoner av stat og by. Dersom det bare hadde fantes én by i hver stat ville det ikke vært fornuftig å gruppere videre på en stat. Dette betyr at det ikke kan finnes en funksjonell avhengighet mellom stat og by.

Det vi kan se fra dette eksempelet er at det ikke gir mening å gruppere på felter som er funksjonelt avhengig av hverandre. Hvis vi antar en funksjonell avhengighet fra stat til by gitt som `state` $\rightarrow$ `city` , som betyr at *city* er funksjonelt av *state*, så impliserer det at det kun hadde vært fornuftig å gruppere på *state* i utgangspunktet, og ikke kombinasjoner av *state* og *city*. Årsaken til dette er at for hver gruppe basert på kun *state* kunne vi ha funnet den ene verdien av *city* som hadde vært gyldig gitt en stat. Det å generere grupper av unike kombinasjoner hadde dermed vært unødvendig. Det samme gjelder den andre veien dersom vi antar den funksjonelle avhengigheten `city` $\rightarrow$ `state`, som i virkeligheten hadde vært en fornuftig integritetsregel basert på domenekunnskap.

Vi kan dermed anta at for hver kombinasjon det grupperes på i en aggregering må forholdet mellom data være *mange-til-mange*. I tillegg kan vi også regne med at det ikke kan finnes funksjonelle avhengigheter mellom data det grupperes på.

5.4.3 Problematikk ved statisk analyse av spørringer

Denne oppgaven er begrenset til å kun se på én type spørring. Det finnes mange andre typer spørringer som ikke har blitt undersøkt. Årsaken til at aggregater er valgt er fordi grupperinger kan gi oss mer informasjon om hvordan data er relatert, i motsetning til for eksempel *insert*- eller *delete*-spørringer som gir informasjon om data som settes inn eller slettes. Det kan likevel ikke forventes at en dokumentdatabase har aggregatspørringer nok til å avdekke alle relasjoner mellom data.

Analysen kan være nyttig dersom det finnes aggregatspørringer, fordi statiske analyser ikke er bundet størrelsen på datasettet. En statisk analyse kartlegger delvis samme informasjon som ved en *dynamisk analyse*, og bør kun bruke dersom det ikke er mulig å utføre en dynamisk analyse.

5.5 Dynamisk analyse av data

Dynamisk analyse går ut på å kjøre gjennom data i en samling for teste datamaterialet. Vi kan kartlegge hvilke felter, eller det vi omtaler her som *attributter*, som utgjør nøkler, og vi kan teste funksjonelle avhengigheter. Vi kan bruke algoritmer som ser på forekomst av data, og hvordan de henger sammen. Den vil kun klare å gi en indikasjon på integritetsregler som vi antar kan holde. Misvisende analyse kan komme av tilfeldigheter, feil i datasettet eller at datasettet ikke er tilstrekkelig nok.

Idéen bak en dynamisk analyse er å kartlegge denormalisert data, som er årsaken til hvorfor vi ønsker å finne funksjonelle avhengigheter. Analysen kan bli en svært omfattende prosess dersom vi har store datasett, men som vi regner med vil gi mer korrekte resultater enn mindre datasett.

For å generere funksjonelle avhengigheter skal vi generere *vestresider* og *høyresider* hver for seg. En venstreside kan bestå av ulike delmenger av samtlige attributter, bortsett fra den tommemengden, og delmengden som består av alle attributtene. Dersom vi har n attributter kan vi få delmengdene gitt ved $\mathcal{P}(n)$, som er potensmengden av n. Høyresiden skal bestå av attributtene som ikke er med i venstresiden.

I det verste tilfellet blir vi nødt til å analysere $|\mathcal{P}(n)| - 2$ attributter fordi vi fjerner den tomme mengden, og mengden som består av samtlige attributter. Dersom det finnes n attributter vil det finnes $2^n - 2$ kombinasjoner av attributter, der det gjøres et søk over tilhørende data. Dette gir et søk som vil utføre

$$(2^n - 2) \times M \times N, \text{ antall iterasjoner.}$$
$$M = \text{summen av antall attributter ikke med i venstresidene.}$$
$$N = \text{antall dokumenter i samlingen.}$$

Det er derfor viktig å finne måter å redusere antall iterasjoner på for å optimalisere søket. Under har vi listet opp noen punkter som kan bidra til å redusere antall iterasjoner.

- Fjerne alle kandidatnøkler fra søket. Vi er kun ute etter funksjonelle avhengigheter som bryter BCNF.
- Anta at det finnes brudd på en funksjonell avhengighet, og bryte søket etter funn av brudd.
- Kun teste for minimale funksjonelle avhengigheter. Dersom vi vet at $X \to A$ holder, så vet vi at $XY \to A$ også holder.

Eksempel. Gitt et datasett S med $30,000$ dokumenter og feltende $a, b, c\ og\ d$ i hvert dokument. Det må antas at hvert dokument inneholder nøyaktig de samme feltene, det vi omtaler som attributter. Det antas også at a er unikt i S og kan regnes som en nøkkel.

Det første som skal gjøres er å fjerne a fra settet og vi sitter igjen med b, c, d før vi skal generere potensmengden som utgjør venstresidene. Vi kan fjerne a fordi vi ikke er interessert i funksjonelle avhengigheter på BCNF, og alle sett som inneholder minst én nøkkel er en supernøkkel som oppfyller kravet for BCNF.

Størrelsen på potensmengden blir da $2^3 = 8$, som genererer $\emptyset$, $\{b\}$, $\{c\}$, $\{d\}$, $\{b, c\}$, $\{b, d\}$, $\{c, d\}$ og $\{b, c, d\}$. Da kan $\emptyset$ og $\{b, c, d\}$ fjernes fordi vi ikke kan generere funksjonelle avhengigheter av dem. Deretter må i verste fall alle kombinasjoner testes. Eksempler på kombinasjoner er $b \to c$, $b \to d$, $bc \to d$, $c \to b$, osv...

Regnestykket for antall iterasjoner blir dermed $(2^3 - 2) \times 9 \times 30,000 = 1,620,000$, der 9 er summen av alle iterasjoner med attributter som ikke

var med i delmengden for hver iterasjon. Dette vil potensielt bli en omfattende prosess dersom vi har mange attributter, da kompleksiteten til potensmengden vokser eksponentielt.

5.5.1 Videre antakelser

Ut ifra eksempelet blir vi nødt til å gjøre videre antakelser om datasettet for å redusere søket. Dersom vi har en denormalisert database med mange attributter og rader kan vi risikere at søket aldri terminerer i praksis. Dersom vi har en relasjon med attributtene *a,b,c,d,e,f,g,h* kan vi gjøre antakelser om at vi ikke er interessert i funksjonelle avhengigheter eller nøkler som består av fler enn tre attributter på venstre siden. For eksempel er vi ikke interessert i den funksjonelle avhengigheten $abcde \rightarrow g$ eller at $cdeg$ er en kandidatnøkkel.

Årsaken til at dette er en rimelig antakelse er at det kan være vanskelig å opprettholde funksjonelle avhengigheter lavere enn BCNF med venstresider bestående av mange attributter, som i vår antakelse overstiger tre attributter. Dette gir økt risiko for feil i databasen, og en omfattende dynamisk analyse vil være bortkastet. Det er heller ikke vanlig at nøkler består av mange attributter i praksis, men heller erstattes med et *id*-attributt.

Vi kan nå begrense den eksponentielle veksten med å kun generere delmengder opp til kardinalitet tre, slik at ingen delmengder overstiger tre attributter. Dette begrenser antall iterasjoner betraktelig fordi vi for store verdier av n, der n er antall attributter i samling, kan kutte store deler av potensmengden.

Eksempel 1. Anta at vi har en samling med 7 attributter, som vil si at $n = 7$. Potensmengden inneholder $2^7 = 128$ ulike delmengder, men dersom vi kan kutte alle delmengdene med kardinalitet større enn 3 og den tomme mengden, står vi igjen med 63 delmengder i stedet for 128. Vi kan regne ut antall delmengder med formelen under.

$$Delmengder = n + \binom{n}{2} + \binom{n}{3}$$
$$\downarrow$$
$$\text{som gir } 7 + \frac{7!}{2!(7-2)!} + \frac{7!}{3!(7-3)!} = 63.$$

Eksempel 2. Anta at $n = 20$. Potensmengden av $2^{20} = 1,048,576$. Hvis vi kun genererer delmengder med 3 eller færre attributter får vi

$$20 + \frac{20!}{2!(20-2)!} + \frac{20!}{3!(20-3)!} = 1350.$$

Så for store verdier av n viser denne antakelsen seg å være mer effektiv i forhold til n, da antall iterasjoner ikke lengre har en eksponentiell vekst.

5.5.2 Problematikk rundt dynamisk analyse av data

Av de tre metodene vi har lagt frem ser dynamisk analyse av data ut til å være den tyngste prosessen å gjennomføre, da den er avhengig av størrelsen på dokumentene og størrelsen på samlingen. Å avgjøre om en dynamisk analyse er hensiktsmessig for migreringsprosessen kan være utfordrende, da den kan være tidkrevende, og en er ikke sikret mot avvikende resultater. Vi kan risikere å bruke tid på å ikke få noen resultater, eller å få feil resultater.

Årsaken til feil kan komme av flere grunner. Det kan ligge i at datasettet ikke er stort nok for en slik analyse, og vi kan ende opp med *falske* funksjonelle avhengigheter. Med *falske* funksjonelle avhengigheter menes de funksjonelle avhengighetene som oppstår gjennom analysen på grunn av mangelfull data der vi ut ifra domenekunnskap ikke ville ha tatt med slike avhengigheter.

id	firstname	lastname
1	John	Doe
2	Jake	Smith
3	Sarah	White
4	John	Doe

Tabell 5.7: *Mangelfull data ved en dynamisk analyse*

La oss ta for oss et eksempel med tabell 5.7 som utgangspunkt. Tabellen illustrerer et utsnitt av data hvor personer med samme fornavn tilfeldigvis har samme etternavn, men det skal være mulig å kunne lagre det samme fornavnet med forskjellige etternavn. En dynamisk analyse vil gi ut den funksjonelle avhengigheten *firstname* $\rightarrow$ *lastname*, fordi for alle fornavn har vi samme etternavn. Dette er rene tilfeldigheter som kan påvirke resultatet vårt negativt.

Når vet vi om et datasett er stort nok til å redusere tilfeldighetene og gi mer pålitelige resultater? Vi kan aldri være 100% sikret mot det, og en må vurdere slike faktorer basert på kjennskap til datasettet.

Et annet problem er når vi har feil i datasettet, og som konsekvens ikke får ut de funksjonelle avhengighetene vi ønsker. Det er viktig å ta hensyn til dette problemet fordi det i utgangspunktet kan være vanskelig å oppdatere denormalisert data, og sjansen er stor for feil i oppdateringer.

Det kan virke som om det er flere ulemper ved denne metoden enn fordeler. Dersom vi ikke bruker metoden vil vi i verste fall ende opp med denormalisert data. Hvis vi benytttter oss av metoden kan vi ende opp med falske funksjonelle avhengigheter, og som en konsekvens få uriktige dekomposisjoner.

Vi skal i det neste kapittelet se på algoritmer og idéer til hvordan disse metodene kan implementeres for å automatisere prosessen i størst mulig grad.

Kapittel 6

Algoritmer

I dette kapittelet skal vi se på algoritmer og idéer som kan brukes til å automatisere metodene fra forrige kapittel. Vi skal også ta for oss hvilke begrensninger som bør tas hensyn til.

6.1 Forutsetning

Før vi studerer algoritmer, må vi stille krav til hvilke datastrukturer vi jobber med. Oppgaven tar utgangspunkt i et Java-miljø, og datastrukturer som finnes i Java.

Vi kan lagre data fra en dokumentsamling i form av en *nøkkel-verdi*-struktur som vist i tabell 6.1, basert på dokumenteksempelet i eksempel 6.1 over filmer. Strukturen lagrer hele dokumentsamlingen som strenger og tar i utgangpunktet ikke hensyn til embedding eller lister. Indre dokumenter og lister blir i første omgang lagret som strengverdier.

Key	Value
_id	1
title	The Movie
genre	[Thriller", "Horror"]
directors	[embedded-document, embedded-document]

Tabell 6.1: *Key-value-struktur av eksempel 6.1*

Ved hjelp av denne måten å strukturere data på kan vi lett ha tilgang til databaseelementer gjennom et kodemiljø uten å miste semantikken bak dokumentene i samlingen. For hvert dokument har vi et sett med *nøkkel-*

verdi-forekomster tilsvarende *felt-verdi*-par på første nivå i dokumentene. Algoritmene vi skal studere tar utgangpunktet i denne strukturen.

6.2 Direkte kartlegging

Vi skal begynne med å se på en algoritme som kan generere ORM-modellen gitt et dokument i en samling. I eksempel 6.1 har vi et dokument fra en filmdatabase.

I det første steget skal algoritmen generere et begrep ut ifra navnet på dokumentsamlingen. Vi kan anta at filen som leses inn heter det samme som navnet på samlingen, som i eksempel 6.1 er *film*. Videre skal algoritmen traversere gjennom hvert felt i form av en nøkkelverdi og sjekke hva slags verdier som finnes for hver nøkkel. I dokumentet i eksempel 6.1 kan vi enkelt traversere gjennom *_id, title, genre* og *directors* fordi vi har en *nøkkel-verdi*-struktur for felter og data på det ytterste laget i dokumentet. Vi ser at de indre dokumentene i *directors* er lagret som en streng i tabell 6.1, men vi vil ha behov for å traversere feltene i disse dokumentene ytterligere for å kartlegge embedding.

```
> db.film.findOne()
{
    "_id" : 1,
    "title" : "The Movie",
    "genre" : ["Thriller", "Horror"],
    "directors" : [
                    {
                        "id" : 001,
                        "firstname" : "John",
                        "lastname" : "Doe"
                    },
                    {
                        "_id" : 002,
                        "firstname" : "Jane",
                        "lastname" : "Doe"
                    }
                ],
}
```

Eksempel 6.1: *Et dokument fra en filmsamling*

Algoritmen må ta hensyn til embedding og lister, men vi vet ikke hvor dype dokumentene er. Det kan være en idé å bruke rekursjon for å håndtere dokumenter med vilkårlig dybde.

6.2.1 Rekursjon for kartlegging av embedding

Anta at vi har dokumentene lagret i form av nøkkel-verdi-par som vist i tabell 6.1. Vi plukker ut et dokument som vi antar inneholder alle feltene som finnes i samlingen, og som derfor kan betraktes som representativt for alle dokumentene i samlingen. Dette dokumentet kan vi finne ved hjelp av en preprosessering av databasen som nevnt i forrige kapittel. Det første algoritmen skal gjøre, er å danne et begrep av samlingsnavnet, som vi i dette tilfellet kaller for *Begrep 1* . Deretter skal det traverseres gjennom feltene i dokumentet, og for hvert felt som inneholder en verdi i form av en streng eller et tall, skal algoritmen generere verdityper tilhørende begrepet som nevnt over. For hver av verditypene skal det genereres en *én-til-mange*-bro fra verditypen til begrepet.

Dersom et felt inneholder et indre dokument, det vi kaller for *embedding*, skal algoritmen danne et nytt begrep av feltnavnet, som vi i dette tilfellet kaller *Begrep 2*, og deretter generere en *én-til-mange*-faktatype fra *Begrep 2* til *Begrep 1*. Rekursivt skal algoritmen utføre den samme prosessen for det indre dokumentet som beskrevet i det første avsnittet, og fortsette å kartlegge nye indre dokumenter med faktatyper til det forrige begrepet. Basistilfellet vil være å analysere et dokument som ikke inneholder nye indre dokumenter.

Dersom et felt fra det ytterste laget i dokumentet inneholder en liste skal algoritmen danne et nytt begrep av feltet, som vi i dette tilfellet kaller *Begrep 3*, og deretter skal algoritmen generere en *mange-til-mange*-faktatype mellom *Begrep 3* og *Begrep 1*. Algoritmen må så avgjøre hva slags verdier som finnes i listen, og vi antar at listen enten skal bestå av indre dokumenter, eller vanlige strenger eller tall. Det må også antas at det ikke kan være kombinasjoner av dem. Basistilfellet er når en liste består av kun strenger eller tall, og algoritmen vil da generere en ny verditype tilhørende listebegrepet med en *én-til-mange*-bro mellom verditypen og begrepet. Dersom listen består av indre dokumenter, skal rekursjonen velge et dokument i listen og utføre direkte kartlegging som nevnt i avsnittet over.

I algoritme 6.1 tar funksjonen inn et begrep og et dokument som parameter og forutsetter at programmet har definert et nytt begrep i forkant. Algoritmen genererer nye begreper hver gang et felt er en liste eller et doku-

Algoritme 6.1 Direkte kartlegging av et dokument

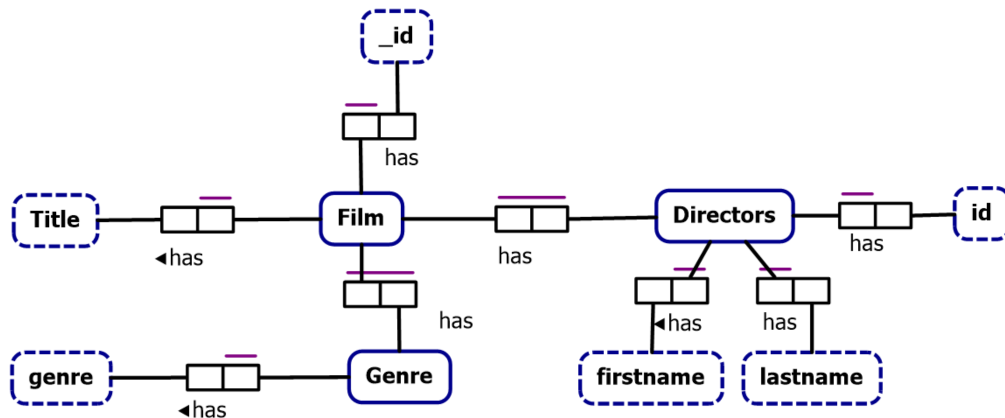
```
 1: function DirectMapping(currentEntity, Document<key, value>)
 2:     for key in Document do
 3:         value ←GetValue(Document,key)
 4:         if value is embedded document then
 5:             newEntity ← CreateEntity(key)
 6:             CreateFactType(currentEntity, newEntity, 1, n)
 7:             DirectMapping(newEntity, value)      ▷ recursive function
 8:         else if value is embedded list then
 9:             newEntity ← CreateEntity(key)
10:             CreateFactType(currentEntity, newEntity, m, n)
11:             if value contains embedded document then
12:                 firstDocInList ← GetDoc(value)
13:                 DirectMapping(newEntity, firstDocInList)
14:             else
15:                 newValueType ← CreateValueType(key)
16:                 CreateRefType(newEntity, newValueType, 1, n)
17:         else
18:             newValueType ← CreateValueType(key)
19:             CreateRefType(currentEntity, newValueType, 1, n)
```

ment med *newEntity*, og lager den tilhørende faktatypen med funksjonskallet *AddFactType* som tar i mot *nåværende begrep*, *nytt begrep*, *skranken til nåværende begrep* og *skranken til det nye begrepet*. Deretter kalles metoden rekursivt så lenge algoritmen oppdager nye indre dokumenter. Tilsvarende for funksjonskallet *AddRefType*. Funksjonene *AddEntity* og *AddValueType* tar inn en nøkkel som parameter og returnerer et nytt begrep eller verditype med tilsvarende navn.

La oss se på filmdokumentet i eksempel 6.1. Anvender vi algoritmen, vil vi få følgende steg i prosessen:

```
Forutsetning
Begrep: Film

Ytre del av dokumentet film

Verditype: _id
Bro: (Film, _id, 1, n)

Verditype: title
Bro: (Film, title, 1, n)

Begrep: Genre
Faktatype: (Film, Genre, m, n)

Verditype: genre
Bro: (Genre, genre, 1, n)

Begrep: Directors
Faktatype: (Film, Directors, m, n)

Embedded del av Directors

Verditype: id
Bro: (Directors, id, 1, n)

Verditype: firstname
Bro: (Directors, firstname, 1, n)

Verditype: lastname
Bro: (Directors, lastname, 1, n)
```

Av resultatet over kan vi konstruere ORM-modellen vist i figur 6.1. Problemet med modellen er at vi ikke kjenner den prefererte referansen i hvert begrep. Alle begrepene mangler altså en nøkkelverdi, og spørsmålet er dermed om vi kan anta om alle id-verdier kan holde som nøkler, og hva skal være den prefererte referansen i *Genre*? Med direkte kartlegging kan vi kun gjøre antakelser om nøkler, entydighetsskranker og totale roller. For å finne svar på dette er det mulig at en statisk- eller dynamisk analyse må til, eller vi kan gå inn å rette opp modellen manuelt basert på domenekunnskap.



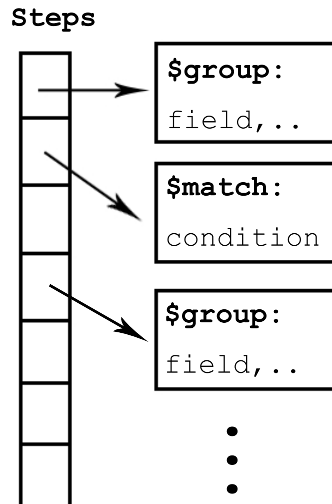
Figur 6.1: *ORM-modell av eksempel 6.1 generert fra direkte kartlegging*

Algoritmen er kun basert på antakelser nevnt i forrige kapittel og er ikke garantert å gi et riktig resultat. Det skal være mulig å modifisere algoritmen til å produsere forskjellige alternativer ved å endre parametrene for skrankene. Algoritmen tar likevel ikke hensyn til enkelttilfeller, som for eksempel ved at det genereres *mange-til-mange*-faktatyper ved noen tilfeller av lister, mens det genereres *én-til-mange*-faktatyper ved andre tilfeller. Det betyr dersom vi bestemmer oss for å generere *mange-til-mange*-faktatyper mellom begrepet av dokumentet som kartlegges og lister, så vil det gjelde i alle tilfeller mellom dokumenter og lister.

6.3 Statisk analyse av spørringer

Spørringer kan variere i stor grad, og vi må bestemme oss for hvilke typer spørringer som skal analyseres, for å kunne undersøke algoritmene for en analyse. Ved aggregater trenger vi en algoritme som traverserer gjennom hvert steg i spørringen og sjekker hvilke felter det utføres en gruppering på.

Dersom vi har en liste over alle stegene, og hvert steg består av en struktur som holder på informasjon om de forskjellige funksjonene i spørrespråket til MongoDB, kan vi hente ut alle feltene som blir listet i en *$group*-funksjonen. Se figur 6.2.



Figur 6.2: *Strukturen på hvordan et aggregat skal se ut i et kodemiljø*

For hvert steg i listen skal algoritmen sjekke om det er en et *$group*-steg, og deretter hente ut feltene som grupperes. Vi må sette en begrensning på hvilke funksjoner vi kan ha med i aggregatet som analyseres, fordi denne metoden kun fungerer der nøkler ikke dupliseres i samlingen, som f.eks ved en *$unwind* av lister. Feltene listet i en *$group*-funksjon kan ikke være unike i samlingen, og vi kan dermed utelukke entydighetsskranker for disse feltene. Algoritmen trenger også å vite hvilke felter som genereres fra hvert steg for å bruke dem i det neste steget.

Algoritme 6.2 Analyse av et aggregat

1: **function** AGGREGATEANALYZER(stageList)
2:     **for** each stage in *stageList* **do**
3:         **if** GETSTAGETYPE(*stage*) = $*group* **then**
4:             $fields \leftarrow$ GETFIELDS(*stage*)

Algoritme 6.2 er en enkel løkke som traverserer alle stegene, og henter ut feltene fra grupperinger med et kall på en funksjon kalt *GetFields*.

Denne algoritmen er begrenset til enkle aggregater, og vil ikke kunne brukes for andre typer spørringer som ikke involverer MongoDB *pipeline*, eller avanserte aggregater med funksjoner som dupliserer nøkler. Det antas i tillegg at det er gjort en del preprosessering for å få spørringene på samme datastruktur som illustrert i figur 6.2. I et videre arbeid hadde det derfor

vært nyttig å gjøre et dypere dykk i spørringer som tilbys i MongoDB. Vi ser at det ikke skal mye til før vi finner interessant informasjon om skjema fra et enkelt aggregat.

6.4 Dynamisk analyse av data

Før de funksjonelle avhengighetene skal kartlegges, bør alle nøkler som består av ett attributt fjernes fra mengden som skal generere venstresiden av en funksjonell avhengighet. Programmet må da kunne identifisere nøklene ut ifra datasettet. En enkel måte å avgjøre om et felt kun inneholder unike verdier, vil være å lagre verdiene i et sett og deretter sjekke om størrelsen på settet er det samme som antall dokumenter i samlingen.

Algoritme 6.3 Finne nøkler bestående av ett attributt

```
 1: function FindKeys(collection, fields)
 2:     s                                              ▷ Empty set buffer
 3:     uniqueKeys
 4:     numDocs ← CollectionSize(collection)
 5:     i ← 0
 6:     for each field in fields do
 7:         s ← ResetBuffer()
 8:         for numDocs do
 9:             if i ≠ SizeOf(s) then
10:                 break
11:             s ← s + GetFieldValueByDocId(i, field)
12:             i ← i + 1
13:         if SizeOf(s) = numDocs then
14:             uniqueKeys ← uniqueKeys + field        ▷ Add unique key
```

Algoritme 6.3 finner alle nøkler som kun består av ett attributt. Denne algoritmen er nødvendig for å optimalisere søket etter funksjonelle avhengigheter, ettersom vi kan utelukke alle venstresider som består av en supernøkkel fra søket. En venstreside utgjør en supernøkkel hvis minst ett av attributtene er en nøkkel.

Funksjonen tar imot en samling av dokumenter, og en liste av feltene som finnes i hvert dokument. Deretter oppretter den et sett som skal brukes som et buffer for å sjekke om alle verdiene i et felt er unike. Dersom traverseringen kommer over like verdier vil algoritmen avbryte søket for å optimalisere kjøretiden. Alle nøkler vil få et databuffer der lengden er lik antall dokumenter som finnes totalt i samlingen.

Videre skal vi se på en algoritme som undersøker funksjonelle avhengigheter. Idéen bak algoritmen er å iterere gjennom potensmengden av potensielle venstresider i en funksjonell avhengighet, og for hver av mengdene skal algoritmen teste oppførselen over datamateriale for hvert felt som ikke er med i venstresiden. Det betyr at algoritmen skal teste datamaterialet for brudd på en funksjonell avhengighet. Algoritme 6.4 viser hvordan vi kan bruke tre for-løkker for å traversere data, slik at vi kan sammenlikne data tilhørende *leftSide* med data tilhørende *rightSide*, der vi antar den funksjonelle avhengigheten *leftSide* → *rightSide*.

Algoritme 6.4 Finne funksjonelle avhengigheter

```
1: function FindFD(powerset, fields, collection)
2:     for each leftSide in powerset do
3:         rightSide ← fields − leftSide
4:         for each field in rightSide do
5:             for each doc in collection do
6:                 ...                                  ▷ Check FD-violations
```

For å generere potensmengden *powerset* som blir tatt inn som argument i funksjonen *FindKeys*, kan vi lage en algoritme som begrenser størrelsen på delmengdene. Dette kommer fra antakelsen i forrige kapittel om å begrense potensmengden til at hver delmengde ikke overstiger tre felter, for å unngå eksponentiell vekst i kjøretiden. Dersom vi begrenser potensmengden til å kun inneholde delmengder som ikke overstiger kardinalitet tre, kan vi bruke tre *for*-løkker.

Algoritme 6.5 Finne en begrenset potensmengde av feltene i samlingen

```
1: function LimitedPowerset(fields)
2:     n ← SizeOf(fields)
3:     pset                                        ▷ pset is a list of sets
4:     for i ← 0 to n do
5:         pset ← pset + fields[i]                 ▷ sets of cardinality 1
6:         for j ← i + 1 to n do
7:             pset ← pset + (fields[i], fields[j])   ▷ sets of cardinality 2
8:             for k ← j + 1 to n do
9:                 pset ← pset + (fields[i], fields[j], fields[k])
```

Algoritme 6.5 viser hvordan vi kan generere en begrenset potensmengde ved hjelp av tre *for*-løkker, og legge til hver mengde i *pset*. Hver av mengdene skal brukes til å generere potensielle venstresider i en funksjonell avhengighet. Funksjonen tar inn feltene som finnes i en samling som argument.

For å sørge for at minimale funksjonelle avhengigheter sjekkes først må *pset* sorteres etter størrelsen på delmengdene. Deretter kan vi sende med *pset* som et argument i algoritme 6.4, og begrense algoritmen til å kun sjekke minimale funksjonelle avhengigheter. Det betyr at dersom $X \to A$ er funnet, skal ikke algoritme 6.4 gjøre en test på $XY \to A$.

Videre trenger vi en algoritme for å teste datamaterialet. Vi kan for eksempel erstatte den innerste *for*-løkken i algoritme 6.4 med et nytt funksjonskall som skal teste om datamaterialet oppfører seg i henhold til den funksjonelle avhengigheten som testes.

Algoritme 6.6 Teste en funksjonell avhengighet mot datasettet

```
1: function CheckDocuments(set, field, collection)
2:     map                                          ▷ map is a key-value dictionary
3:     for each doc in collection do
4:         leftSide ← GetValuesByDoc(doc, set)                    ▷ (keys)
5:         rightSide ← GetValuesByDoc(doc, field)                 ▷ (values)
6:         if GetKeySet(map) contains leftSide then
7:             if GetMapValue(map, leftSide) ≠ rightSide then
8:                 break                                          ▷ FD-violation
9:         else map ← map + (leftSide, rightSide)
```

I algoritme 6.6 brukes *map* til å sjekke brudd på en funksjonell avhengighet. Brudd oppstår der vi har samme verdier i *leftSide*, men ulike verdier i *rightSide*. Dersom det finnes brudd skal testen avsluttes.

I det neste kapittelet skal vi implementere og teste algoritmene som er lagt frem i dette kapittelet.

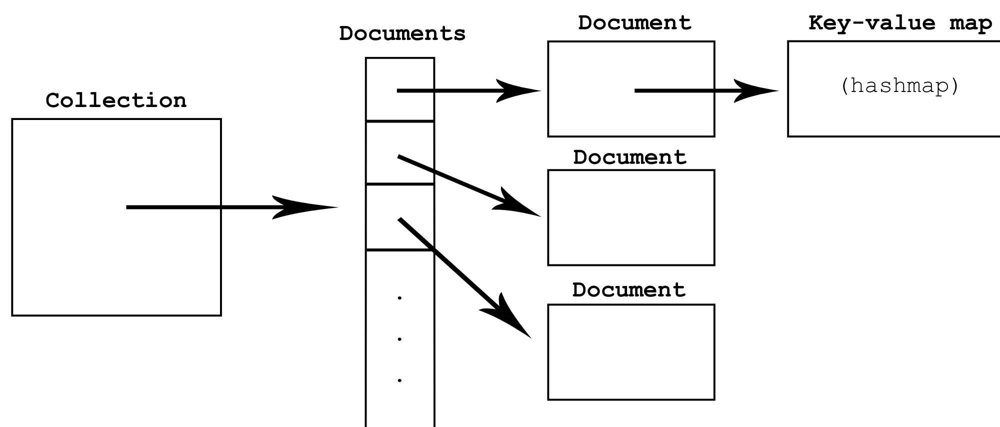
Kapittel 7

Implementasjon og testing

7.1 Preprosessering og java-struktur

Før vi implementerer algoritmene fra forrige kapittel, må vi gjøre en del preprosessering for å lese inn en dokumentsamling til et java-miljø. For å lese inn JSON-dokumenter kan vi bruke et eksternt java-bibliotek [29] som leser inn JSON-objekter til en HashMap.

Når en samling leses inn skal det opprettes et nytt samlingsobjekt i java, og for hvert dokument opprettes det et dokumentobjekt med en HashMap som lagrer data fra JSON. Figur 7.1 viser hvordan strukturen skal se ut. Vi har valgt å opprette dokumentobjekter slik at vi kan implementere hjelpemetoder for å aksessere dokumentdata på en ryddig og enkel måte.



Figur 7.1: *Datastrukturen i java*

For eksempel kan dokumentklassen bli implementert som vist i eksempel 7.1. Når et dokument opprettes sendes det med en HashMap med data fra et innlest dokument, og samlingsnavnet. Vi kan enkelt implementere metoder for å hente ut samlingsnavnet eller dokumentdata etter behov. Innlesningen vil kun opprette en HashMap for det ytterste laget i et dokument, og dersom et dokument inneholder embedding eller lister blir dette lagret som strenger.

```
public class Document{
    String collectionName;
    HashMap<String, String> doc;

    //Konstruktor
    public Document(HashMap<String, String> doc, String
        collectionName){
        this.doc = doc;
        this.collectionName = collectionName;
    }
}
```

Eksempel 7.1: *Dokumentklassen*

7.2 Implementasjon av direkte kartlegging

Implementasjonen av *direkte kartlegging* i eksempel 7.3 følger algoritme 6.1 fra forrige kapittel. Metoden forutsetter at det sendes med et samlingsnavn og en HashMap av et dokument i samlingen. For eksempel kan metoden kalles dersom klassen har tilgang til et dokument *doc* som vist i eksempel 7.2.

```
public DirectMapping(Document doc){
        recursiveMap(doc.getCollectionName(), doc.getMap());
}
```

Eksempel 7.2: *Kall på metoden recursiveMap()*

Eksempel 7.3 traverserer gjennom et dokument ved hjelp av et *keyset()*, som er alle feltene i et dokument, og sjekker verdier for lister og embedding. Implementasjonen antar at embedding alltid begynner med en krøllparentes, og lister alltid begynner med en hakeparentes. JSON-biblioteket blir brukt til å konvertere strengverdier til nye HashMap for å kartlegge indre do-

kumenter. Det blir i tillegg brukt hjelpemetoder for å skrive ut begreper, verdityper og setningstyper.

```
public void recursiveMap(String currentEntity,
                         HashMap<String, String> m){

       for(String key : m.keySet()){
           String value = m.get(key);
           char firstchar = value.charAt(0);

           //Felt bestar av et dokument
           if(firstchar == '{'){
               newEntity(key);
               newPredicate(currentEntity, "1", "n", key);

               //Rekursivt steg
               recursiveMap(key, toJson.jsonToMap(value));

           //Felt bestar av en liste
           }else if(firstchar == '['){
               String firstElem = jsonToArray(value).get(0);
               newEntity(key);
               newPredicate(currentEntity, "m", "n", key);

               if(firstElem.charAt(0)=='{'){
                   //Rekursivt steg
                   recursiveMap(key, toJson.jsonToMap(firstElem));

               }else{
                   newValueType(key+"_value");
                   newPredicate(key, "1", "1", key+"_value");

               }
           }else{
               newValueType(key);
               newPredicate(currentEntity, "1", "n", key);
           }
       }
   }
```

Eksempel 7.3: *Implementasjon av direkte kartlegging*

7.2.1 Testing av MongoDBs *zips*

Vi ønsker å teste implementasjonen på MongoDBs *zips*-eksempel gjengitt i eksempel 7.4. Vi velger å teste det første dokumentet i samlingen, fordi MongoDB sier på sin side at alle dokumentene har de samme feltene. Det er derfor ikke nødvendig å utføre en preprosessering av dokumentene for å finne et dokument som inneholder alle feltene som finnes i en samling.

```
> db.zips.findOne()
{
    "_id" : "01001",
    "city" : "AGAWAM",
    "loc" : [ -72.622739, 42.070206 ],
    "pop" : 15338,
    "state" : "MA"
}
```

Eksempel 7.4: *Et eksempel på et dokument i MongoDBs samling av zipcodes*

Resultater av MongoDBs *zips*-eskempel

Dersom vi kjører programmet der dokumentet i eksempel 7.4 kartlegges, får vi resultatet under.

```
/*--Direkte kartlegging--*/
Zips : new entity
pop : new value type
zips - 1:n - pop
Loc : new entity
zips - m:n - loc
loc_value : new value type
loc - 1:1 - loc_value
city : new value type
zips - 1:n - city
_id : new value type
zips - 1:n - _id
state : new value type
zips - 1:n - state
```

Observasjon av MongoDBs *zips*-eskempel

Vi får resultatet som er forventet av algoritmen. Dette eksempelet mangler embedding av dokumenter og vi får derfor ikke testet den rekursive delen av metoden. Vi har også valgt å anta noen av skrankene, men vi kan kjøre programmet på nytt der alle skrankene som er antatt er byttet ut med *unknown* som vist under.

```
/*--Direkte kartlegging--*/
Zips : new entity
pop : new value type
zips - 1:unknown - pop
Loc : new entity
zips - m:unknown - loc
loc_value : new value type
loc - 1:1 - loc_value
city : new value type
zips - 1:unknown - city
_id : new value type
zips - 1:unknown - _id
state : new value type
zips - 1:unknown - state
```

Programmet klarer å trekke ut begreper og verdityper fra et dokument, men vi ser at setningstypene er mangelfulle, og de fleste av dem mangler en skranke. Det ser likevel ut til at programmet fint klarer å kartlegge dokumenter uten embedding av indre dokumenter.

7.2.2 Testing av MongoDBs *restaurants*

Vi kan teste et nytt eksempel som inneholder embedding av dokumenter. MongoDB har et annet datasett med data over restauranter i New York [30]. Et eksempeldokument er vist i 7.5. Samlingen inneholder i overkant av 25,000 dokumenter, og inneholder embedding av *address* og embedding av dokumenter i lister. Vi kan dermed teste om den rekursive delen av metoden håndterer begge tilfellene av embedding.

```
> db.restaurants.findOne()
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
    {
        "date": { "$date": 1393804800000 },
        "grade": "A",
        "score": 2
    },
    {
        "date": { "$date": 1378857600000 },
        "grade": "A",
        "score": 6
    },
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

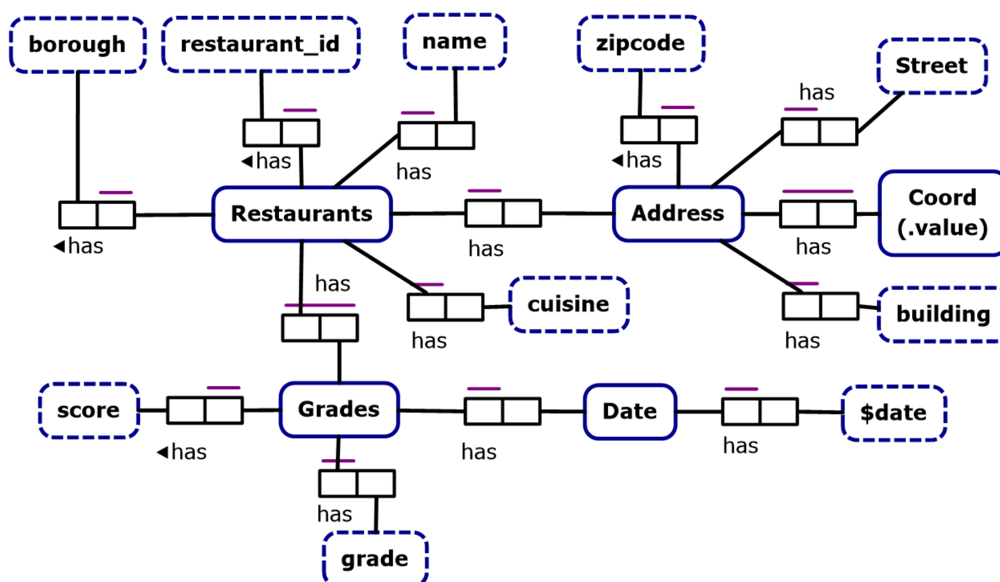
Eksempel 7.5: *Et eksempel på et dokument i MongoDBs samling av restaurants*

Algoritmen vil ikke ta hensyn til eksterne entydighetsskranker i indre dokumenter. Det forventes dermed et resultat uten eksterne skranker.

Resultater av MongoDBs *restaurants*-eksempel

Vi får resultatet på neste side ved å kjøre programmet med MongoDBs *restaurants*-samling med antakelser om skranker. For å gjøre resultatet mer leselig har vi konstruert en ORM-modell i figur 7.2 basert på resultatet.

```
/*--Direkte kartlegging--*/
Restaurants : new entity
Address : new entity
restaurants - 1:n - address
zipcode : new value type
address - 1:n - zipcode
Coord : new entity
address - m:n - coord
coord_value : new value type
coord - 1:1 - coord_value
street : new value type
address - 1:n - street
building : new value type
address - 1:n - building
restaurant_id : new value type
restaurants - 1:n - restaurant_id
name : new value type
restaurants - 1:n - name
cuisine : new value type
restaurants - 1:n - cuisine
borough : new value type
restaurants - 1:n - borough
Grades : new entity
restaurants - m:n - grades
Date : new entity
grades - 1:n - date
$date : new value type
date - 1:n - $date
score : new value type
grades - 1:n - score
grade : new value type
grades - 1:n - grade
```



Figur 7.2: *ORM-modellen fra direkte kartlegging av restaurants*

Observasjon av MongoDBs *restaurant*-eksempel

Ingen av begrepene bortsett fra *Coord* får en preferert referanse. I realiteten bør begrepet *Restaurants* ha *restaurant_id*, begrepet *Address* bør ha en ekstern entydighetsskranke mellom *zipcode*, *street* og *building*, og begrepet *Grades* bør ha en ekstern entydighetsskranke mellom *date*, *grade*, *score* og *Restaurant*, der faktatypen mellom *Restaurant* og *Grades* bør være en *én-til-mange*-faktatype.

Programmet vet heller ikke hvordan *Coord* bør kartlegges. Dette var også problematisk i eksempelet med *zips*, fordi koordinatverdier bør være en måte å identifisere et sted eller en adresse. Vi ser at direkte kartlegging er nyttig for kartlegge begreper og verdityper, og gir en indikasjon på hvordan data er relatert i databasen. Direkte kartlegging bør ikke være den eneste metoden for å migrere til et annet system. Testene viser at prosessen kan automatiseres til en viss grad, og deretter må vi enten gå videre til en annen metode, eller å rette opp modellen manuelt ved hjelp av domenekunnskap.

7.3 Implementasjon av statisk analyse

En statisk analyse av spørringer kan være nyttig dersom en full dynamisk analyse ikke er mulig på grunn av størrelsen på datasettet. I denne oppgaven må vi anta at det har blitt gjort en del preprosessering før implementasjonen av en statisk analyse, fordi spørringene kan variere, og å skrive et program som tar hensyn til alle typer spørringer blir for omfattende.

Vi begrenser programmet til å kun akseptere aggregater. Vi kan sette opp en java-klasse som vist i eksempel 7.6. Klassen har en *pipeline* som inneholder ulike aggregeringsfunksjoner som tilbys i MongoDB.

```
private class Aggregation{
	ArrayList<PipelineEntry> pipeline = new ArrayList<>();
}
```

Eksempel 7.6: *Aggregatklassen*

For eksempel kan en *PipelineEntry* se ut som eksempel 7.7. I dette tilfellet har vi valgt å begrense *PipelineEntry* til *$group*-funksjonen. Det hadde vært mulig å implementere flere konstruktører til klassen, slik at *PipelineEntry* kan ta imot andre funksjoner enn kun *$group*. Vi skal likevel kun implementere og teste dette eksempelet.

```
private class PipelineEntry{
    String [] group;

    PipelineEntry(String [] group){
        this.group = group;
    }
}
```

Eksempel 7.7: *PipelineEntry*

7.3.1 Testing av MongoDB *zips*

Hvis vi ser tilbake på en spørring som er gjengitt i eksempel 7.8, er vi ute etter å trekke ut *$group*-funksjonene, og legge disse inn i *pipeline*-strukturen fra eksempel 7.6. Eksempel 7.8 viser to aggregeringssteg, og hvert av disse stegene blir lagt inn som en *PipelineEntry*.

```
> db.zips.aggregate( [
 { $group: {_id: { state: "$state", city: "$city" },
    pop: { $sum: "$pop" } }
 },
 { $group: { _id: "$_id.state",
    avgCityPop: { $avg: "$pop" } }
 }
] )
```

Eksempel 7.8: *Gjennomsnittlig bypopulasjon i hver stat*

Vi kan legg inn *$group*-stegene som vist i eksempel 7.9, og deretter kan vi iterere gjennom *pipeline*-strukturen og hente ut feltene som grupperes.

```
Aggregation(){
    pipeline.add(new PipelineEntry(new String[]{"$city","$state"}));
    pipeline.add(new PipelineEntry(new String[]{"$state"}));
}
```

Eksempel 7.9: *Aggregeringssteg i en pipeline*

Resultater

En iterasjon gjennom *pipeline*-strukturen gir resultatet under. Dersom det finnes mer enn ett felt i en *group*-liste, skriver programmet ut at det ikke kan finnes en funksjonell avhengighet mellom disse feltene.

```
/*----Query Analyzer---*/

/*--- 1. pipe entry ---*/
No functional dependency between $city, $state,
$city is not unique in zips
$state is not unique in zips

/*--- 2. pipe entry ---*/
$state is not unique in zips
```

Observasjon

Ut ifra resultatet kan vi vite at entydighetsskranker ikke vil holde for rollene til *$city* og *$state* i *zips* fordi feltene ikke kan være unike i samlingen. En statisk analyse kan være mangelfull dersom det ikke finnes spørringer som dekker de andre feltene som finnes i databasen.

7.4 Implementasjon av dynamisk analyse

Vi kan bruke dynamisk analyse på to ulike måter. Vi kan bruke analysen til å oppdage mulige funksjonelle avhengigheter i databasen, eller så kan det brukes til å indikere om en antatt funksjonell avhengighet holder. Sistnevnte er hva det bør brukes til fordi vi ikke kan stole på datamaterialet dersom vi ønsker å oppdage nye funksjonelle avhengigheter. Analysen vil være nyttig dersom vi vet noe om domenet, men ønsker å sjekke om datamaterialet oppfører seg i tråd med integritetsreglene vi mener bør holde, eller om det finnes avvik i databasen.

Vi må uansett gjøre en full analyse over data, og vi skal implementere algoritmene under dynamisk analyse fra forrige kapittel. Programmet vil kun fungere for samlinger der alle dokumentene har nøyaktig de samme feltene, og tar ikke hensyn til manglende felter i dokumenter.

Det opprettes en egen klasse for *dynamisk analyse* som vist i eksempel 7.10, og klassen vil ha tilgang til hele dokumentsamlingen.

```
public class DynamicAnalyzer {
    DBCollection dbc;
    ArrayList<Document> docs;
    List<String> fields;
}
```

Eksempel 7.10: *Klassen for dynamisk analyse*

Før vi setter i gang med å generere funksjonelle avhengigheter, må vi fjerne alle enkle felter som utgjør nøkler. Det kan gjøres ved hjelp av metoden implementert i eksempel 7.11. Vi kan også ved hjelp av denne metoden vite hvilke felter som ikke er unike, som gjør at vi får kartlagt flere setningtyper.

```
public Set<String> findKeys(){
    Set<String> UniqueKeys = new HashSet<>();
    Set<String> dataset; //Bufferset

    //Itererer gjennom all data for hvert attributt
    for(String a: fields){
        dataset = new HashSet<String>();

        for(int i = 0; i<docs.size(); i++){
            //Optimalisering ved a sjekke nokkelkrav
            if(i != dataset.size()) break;
            dataset.add(docs.get(i).getValuesByKey(a));
        }

        //Sjekker om alle verdiene er unike
        if(dataset.size() == docs.size()) UniqueKeys.add(a);
    }
    return UniqueKeys;
}
```

Eksempel 7.11: *Metoden for å finne nøkler som består av ett felt*

For å generere venstresiden av en funksjonell avhengighet, må vi implementere algoritmen som genererer potensmengden av feltene som ikke utgjør nøkler, som vi i dette tilfellet begrenser til mengder av kardinalitet tre eller mindre. Algoritmen er implementert i eksempel 7.12, og det er også implementert en *comparator* for å sortere listen etter kardinalitet.

```
public ArrayList<List<String>> limitedPowerSet(){
        ArrayList<List<String>> pset = new ArrayList<>();

        for(int i=0; i<fields.size(); i++){
            pset.add(Arrays.asList(fields.get(i)));

            for(int j = i+1; j<fields.size(); j++){
                pset.add(Arrays.asList(fields.get(i), fields.get(j)));

                for(int k=j+1; k<fields.size(); k++){
                    pset.add(Arrays.asList(fields.get(i),
                        fields.get(j), fields.get(k)));
                }
            }
        }

        Collections.sort(pset, new Comparator<List<String>> () {...});
        return pset;
    }
```

Eksempel 7.12: *Metoden for generere venstresider*

```
public boolean documentCheck(List<String> set, String field){
        //Tmp hashmap for a teste data
        HashMap<String, String> mapFD = new HashMap<>();
        boolean trueFD = true;

        for(Document d: dbc.getDocuments()){
            String key = d.getValuesByNKeys(set);
            String value = d.getValuesByKey(field);

            if(mapFD.containsKey(key)) {
                if(!mapFD.get(key).equals(value)) {
                    trueFD = false;
                    break;
                }
            }else mapFD.put(key,value);
        }
        return trueFD;
    }
```

Eksempel 7.13: *Metode som tester venstresider mot høyresider i en FD*

Vi trenger også en metode for å sjekke vestresider av funksjonelle avhengigheter mot høyresider for å undersøke om det finnes brudd på den funksjonelle avhengigheten som testes. Metoden er implementert i eksempel 7.13, og tar imot *set* som tilsvarer en venstreside, og *field* som tilsvarer en høyreside som argument. Metoden returnerer *false* dersom det finnes brudd på den funksjonelle avhengigheten som testes.

Til slutt kan vi implementere den fulle algoritmen som vist i eksempel 7.14. Den optimaliserer søket ved et kall på metoden *checkMinimalFD()* som tester om det allerede finnes en minimal funksjonell avhengighet.

```
public Map<List<String>, List<String>> findFDs(
    ArrayList <List<String>> lpset){

    Map<List<String>, List<String>> retrFDs = new HashMap<>();

    for(List<String> set: lpset){

        //Felter som ikke er med i venstresiden
        List<String> notInSet = new ArrayList<>(fields);
        notInSet.removeAll(set);

        for(String field: notInSet){
            if(checkMinimalFd(set, retrFDs, field)) continue;

            //Itererer gjennon alle dokumentene med documentCheck
            if(documentCheck(set, field)){

                //Sjekker om det finnes en FD med lik venstreside
                if (!retrFDs.containsKey(set))
                    retrFDs.put(set, Arrays.asList(field));
                else {
                    //Oppdaterer eksiterende verdier
                    List<String> tmpFields = new
                        ArrayList<>(retrFDs.get(set));
                    tmpFields.add(field);
                    retrFDs.put(set, tmpFields);
                }
            }
        }
    }
    return retrFDs;
}
```

Eksempel 7.14: *Metode for å finne funksjonelle avhengigheter*

7.4.1 Testing av MongoDBs *zips*

Vi ønsker å teste data for nøkler og funksjonelle avhengigheter. Vi skal kun teste felter som ikke inneholder lister, og hvis vi ser tilbake på eksempel 7.4 skal ikke *loc* være involvert i søket.

Ut ifra domenekunnskap kan vi anta at den funksjonelle avhengigheten $city \rightarrow state$ bør holde, fordi en by ikke kan finnes i flere stater.

Resultater

```
/*--Dynamisk analyse av data--*/
Keys:[_id]
```

Observasjon

Analysen gav ikke ut resultater på nye funksjonelle avhengigheter. Vi kan bekrefte at $city \rightarrow state$ ikke holder i datasettet, og årsaken er at *city* ikke kan identifiseres som et eget bykonsept. Dersom vi har byen *SPRINGFIELD*, så vet vi ikke eksakt hvilken by vi refererer til, fordi det finnes mange byer i USA med navnet *SPRINGFIELD*.

Fra resultatet kan vi se at *_id* holder som en nøkkel i samlingen, og vi kan dermed sette *_id* som den prefererte referansen i *Zips*. Vi kan likevel ikke stole hundre prosent på at programmet gir ut riktige nøkler hver gang, men vi velger å stole på resultatet i dette tilfellet fordi samlingen inneholder over 25,000 dokumenter, og det virker ikke tilfeldig at alle *_id*-verdiene er unike.

7.4.2 Testing av MongoDBs *restaurants*

Vi kan teste *restaurants*-samlingen som vist i eksempel 7.5 for nøkler og den funksjonelle avhengigheten $address \rightarrow borough$. Ut ifra domenekunnskap antar vi at en adresse kun kan ligge i ett distrikt eller bydel.

Resultater

```
/*--Dynamisk analyse av data--*/
Keys:[restaurant_id]
[address, name] -> [borough]
[address, cuisine] -> [borough]
```

Observasjon

Vi kan bekrefte at *restaurant_id* holder som en nøkkel, og vi setter *restaurant_id* som den prefererte referansen i *Restaurants*. De funksjonelle avhengighetene er ikke nøyaktig det vi forventet av analysen. Det kan se ut til at det finnes data som bryter *address* → *borough*, men kombinert med enten *name*, eller *cuisine*, holder antakelsen.

Ved slike resultater er det grunn til å mistenke feil i datasettet. Etter en manuell undersøkelse viser det seg å være fire avvik i databasen, og blant tre av avvikene er verdiene i *borough* lagt inn med strengen *missing*. Der det ikke eksisterer data har MongoDB valgt å legge inn feltet med *missing* som verdi, istedet for å benytte et fleksibelt skjema og fjerne *borough* fra dokumentet. I det siste avviket er det lagt inn feil *borough* i den ene adressen som vist i eksempel 7.15. Med et raskt søk i Google, finnes adressen i *Bronx* og ikke i *Manhattan*.

```
{
  "address" : {
    "building" : "223",
    "coord" : [ -73.9054854, 40.8794154 ],
    "street" : "West  231 Street", "zipcode" : "10463"
  },
  "borough" : "Manhattan"
}
{   "address" : {
    "building" : "223",
    "coord" : [ -73.9054854, 40.8794154 ],
    "street" : "West  231 Street", "zipcode" : "10463"
  }, "
  borough" : "Bronx"
}
```

Eksempel 7.15: *Avvik: Ulike borough for samme address*

Vi kan bruke denne informasjonen til være oppmerksom på at feil i datamaterialet bør rettes opp i forkant av en migreringsprosess. Ved hjelp av testene har vi oppdaget at programmet ikke tar hensyn til strenger som plassholdere for manglende data, og i et videre arbeid hadde det vært interessant å få programmet til å håndtere små avvik.

Koden er lastet opp på github [31], og tar inn en JSON-fil som argument.

Kapittel 8

Konklusjoner og videre arbeid

Målet med oppgaven var å undersøke om vi kunne ekstrahere skjema fra skjemaløse dokumentdatabaser i MongoDB. De resulterende skjemaene ble representert på et konseptuelt nivå ved hjelp av ORM. Vi brukte ORM fordi vi ønsket å undersøke relasjoner mellom data i et skjema uten å måtte ta hensyn til implementasjonsdetaljer som datatyper, noe som må tas med i et mer teknisk relasjonsskjema. Vi så på tre forskjellige metoder, *direkte kartlegging*, *statisk analyse av spørringer*, og *dynamisk analyse av data*. Vi gjorde rede for algoritmer for å automatisere prosessen, og implementerte et program i Java for å teste ulike datasett.

8.1 Konklusjoner

For å kunne utføre skjemaekstraksjon måtte vi gjøre en rekke antakelser om dokumentene og datasettet. I *direkte kartlegging* gikk dette ut på å anta at dokumentene så tilnærmet like ut, og hadde et underliggende implisitt skjema. Vi antok deretter at vi utførte direkte kartlegging på et dokument som inneholdt alle feltene, og som var representativt for alle dokumentene i databasen. Vi klarte å kartlegge konsepter fra et dokument, men analysen var noe mangelfull ved at ikke alle relasjonene mellom konseptene lot seg kartlegge. Vi endte opp med å anta hvilke skranker som skulle gjelde over ulike roller i ORM-modellen. Det var heller ikke mulig å kartlegge påkrevde roller, men vi fikk likevel en indikasjon på hvordan skjemaet ville se ut. Direkte kartlegging ble en effektiv metode å identifisere begreper og verdityper i en ORM-modell.

For å kartlegge flere relasjoner introduserte vi en metode kalt *statisk analyse av spørringer*. Denne metoden gikk ut på å analysere aggregatspørringer

for grupperinger, der vi antok at felter som grupperes ikke består av unike verdier. Begrunnelsen ligger i at vi antar at det ikke er nødvendig med gruppering dersom verdiene er unike. Vi fant ut at metoden bør brukes dersom det ikke er mulig å utføre analyser over hele datasettet. En statisk analyse er uavhengig av størrelsen på datasettet, men den forutsetter at det finnes spørringer å analysere.

Den tredje metoden kalt *dynamisk analyse av data* gikk ut på å analysere om datamaterialet oppførte seg etter antatte funksjonelle avhengigheter. I tillegg brukte vi dynamisk analyse til å kartlegge mulige nøkler. Vi brukte metoden for å se etter feil i data etter at vi hadde gjort oss noen formeninger om hvilke funksjonelle avhengigheter som burde holde basert på domenekunnskap.

Etter å ha testet metodene kunne vi få et overblikk over skjemaekstraksjonen, og deretter bruke ORM til å videreutvikle konseptene vi kartla fra dokumentdatabasen. Vi fikk til å automatisere direkte kartlegging og dynamisk analyse av data, men den statiske analysen var mer problematisk fordi vi ikke fant et bibliotek for å parsere spørringene til Java-programmet. Ettersom spørringene kunne variere i stor grad, ble det for omfattende å lage en slik parseringsalgoritme selv.

8.2 Videre arbeid

- **Direkte kartlegging.** Det kan det være interessant å undersøke om det er mulig å kartlegge alle dokumentene i en samling, og deretter slå sammen ORM-modellene. I denne oppgaven antar vi at det finnes minst ett dokument som inneholder alle feltene i en samling, og vi antar at vi utfører en preprosessering for å finne dette dokumentet. Dersom det i realiteten ikke finnes et slikt dokument, kan det tenkes at en direkte kartlegging på flere dokumenter samlingen vil være hensiktsmessig. Det vil gi et mer korrekt resultat enn om vi må begrense analysen til antakelser om at dokumentene har samme form.

- **Statisk analyse av spørringer.** Det hadde vært interessant å se på andre typer spørringer enn kun grupperinger i aggregater. Programmet for å utføre en analyse kan bli bedre implementert ved å programmere selve preprosesseringen. Det må finnes ut av hvordan ulike spørringer kan parseres til et Java-miljø.

- **Dynamisk analyse av data.** Ut ifra resultatet av testene vi utførte, viste det seg at vi fikk små avvik i datamaterialet. Det kunne ha vært

interessant å prøve å ta hensyn til at små avvik kan forekomme i store datasett. Dette kan være feil i data som gir brudd på funksjonelle avhengigheter som vi mener bør holde i datasettet. Programmet bør kunne gi informasjon om at en funksjonell avhengighet kan holde selv om det finnes brudd i datamaterialet.

- **Utføre en ekte migreringsprosess**. For å undersøke hvor anvendbare våre metoder er i en migreringsprosess, hadde det vært interessant å utføre en ekte databasemigrering fra MongoDB til en relasjonsdatabase.

Bibliografi

[1] Pramod J. Sadalage og Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

[2] Dan McCreary. Making sense of NoSQL. `http://macc.foxia.com/files/macc/files/macc_mccreary.pdf`, 2014. Presentation.

[3] MongoDB. Introduction to MongoDB. `https://docs.mongodb.com/manual/introduction/`. [Online; accessed 03-Jan-2017].

[4] Terry Halpin. Object Role Modeling: An overview. Technical report, Visio Corporation, 1997–1998.

[5] Paolo Atzeni med flere. The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, 42(2):416–421, June 2013.

[6] Sean B. Palmer. The semantic web: An introduction. `http://infomesh.net/2001/swintro/`. [Online; accessed 28-Apr-2017].

[7] Justin J. Miller. Graph database applications and concepts with neo4j, 2013. Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA.

[8] The Apache Software Foundation. Apache HBase. `https://hbase.apache.org/`.

[9] Franz Inc. AllegroGraph - interfacing with MongoDB. `http://franz.com/agraph/support/documentation/current/mongo-interface.html`. [Online: acessed 26.04.17].

[10] Franz Inc. AllegroGraph. `http://allegrograph.com/allegrograph/`. [Online: acessed 26.04.17].

[11] Srini Penchikala. MongoGraph brings semantic web features to MongoDB developers. https://www.infoq.com/news/2011/12/mongograph-qa. [Online: acessed 26.04.17].

[12] Sarah Mei. Why you should never use MongoDB. http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/. [Online; accessed 09-Mar-2017].

[13] Sarah Mei. Switching data stores. https://speakerdeck.com/sarahmei/switching-data-stores-a-postmodern-comedy, 2013. Presentation.

[14] Olery Developer Portal. Goodbye MongoDB, Hello PostgreSQL. http://developer.olery.com/blog/goodbye-mongodb-hello-postgresql/. [Online; accessed 18-Apr-2017].

[15] Terry Halpin. Object Role Modeling - the official site for conceptual data modeling. http://www.orm.net/. [Online; accessed 13-Apr-2017].

[16] Terry Halpin og Tony Morgan. *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers, Elsevier Inc., Burlington, Massachusetts, USA, 2nd edition, 2008.

[17] Ellen Munthe-Kaas. Realiseringsalgoritmen. http://www.uio.no/studier/emner/matnat/ifi/INF1300/h16/forelesningsmateriale/realiseringsalgoritmen.pdf, 2016. Forelesningsmateriale fra INF1300.

[18] Ellen Munthe-Kaas. Undertrykking av begreper. http://www.uio.no/studier/emner/matnat/ifi/INF1300/h16/forelesningsmateriale/underbegreper.pdf, 2016. Forelesningsmateriale fra INF1300.

[19] MongoDB. JSON and BSON. https://www.mongodb.com/json-and-bson/. [Online; accessed 07-Mar-2017].

[20] CUNY school of professional studies. Introduction to document databases. https://www.youtube.com/watch?v=Nh6Y7DgZDrg. Video - [Online; accessed 11-Apr-2017].

[21] Martin Fowler. Schemaless data structures. https://martinfowler.com/articles/schemaless/, 2013. Presentation.

[22] MongoDB. Find restaurants with geospatial queries. https://docs.mongodb.com/manual/tutorial/geospatial-tutorial/. [Online; accessed 03-Jan-2017].

[23] MongoDB. Replications. https://docs.mongodb.com/manual/replication/. [Online; accessed 03-Jan-2017].

[24] MongoDB. Data model design. https://docs.mongodb.com/manual/core/data-model-design/. [Online; accessed 11-Jan-2017].

[25] MongoDB. Database references. https://docs.mongodb.com/manual/reference/database-references/. [Online; accessed 19-Jan-2017].

[26] Anuradha Kanade med flere. A study of normalization and embedding in MongoDB. *Advance Computing Conference (IACC), IEEE International*, 2014.

[27] MongoDB. Top 5 considerations when evaluating NoSQL databases. *A MongoDB White Paper*, November 2016.

[28] MongoDB. Aggregation with the zip code data set. https://docs.mongodb.com/v3.2/tutorial/aggregation-zip-code-data-set/. [Online; accessed 03-Jan-2017].

[29] Package org.json. https://stleary.github.io/JSON-java/. [Online; accessed 21-Feb-2017].

[30] MongoDB. Import example dataset. https://docs.mongodb.com/getting-started/shell/import-data/. [Online; accessed 21-Feb-2017].

[31] Lan Anh Vu. Kildekode for implementasjonen av skjemaekstraksjon. https://github.com/lanavu/Masteroppgave.