Matti Ahinko

Clojuren viitteet tietokantayhteyden tukena

Tietotekniikan pro gradu -tutkielma

30. lokakuuta 2018

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Matti Ahinko

Yhteystiedot: matti.ahinko@gmail.com

Ohjaaja: Ville Tirronen

Työn nimi: Clojuren viitteet tietokantayhteyden tukena

Title in English: Clojure references as a support for database connection

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 66+1

Tiivistelmä: Tämä tutkielma tarkastelee, voisiko Clojuren viitteitä käyttää tietokantayhteyksien tukena. Tutkielmassa avataan, millainen Clojure on ohjelmointikielenä sekä miten sen tietorakenteet ja viitteet toimivat. Teoksen teknisessä osuudessa toteutetaan yksinkertainen kirjasto, jolla kokeillaan Clojuren Atom-viitteen soveltuvuutta tietokantayhteyden tukena. Kirjaston suorituskykyä ja ohjelmakoodin vaatimia muutoksia verrataan suoraan tietokantayhteyteen. Toteutettu kirjasto ja sen toimintaperiaate osoittautuu mielestäni epäkäytännölliseksi. Ohjelmakoodin yksinkertaistamiseen se toimii, mutta sen käyttö ohjelmassa ole muistinkäytön tai luotettavuuden kannalta järkevää. Tutkielma tarjoaa kuitenkin tiiviin ja kattavan paketin taustatietoa Clojuresta ohjelmointikielenä. Lisäksi teos tarjoaa syventävää tietoa viitteistä ja kirjaston toteuttamisesta Clojurelle.

Avainsanat: Clojure, Viitteet, Tietokannat, Tiedon tallennus, Funktionaalinen ohjelmointi, Tietorakenteet, Atom, Kirjasto

Abstract: This Master's thesis studies if references in Clojure programming language can be used for improving, simplifying, and generally supporting database connections. The thesis will also reveal what Clojure is like and how its data structures and references function. In the extensive technical part of the thesis a library for using references as database connection will be developed. The library's performance and usage will also be studied. The developed library ended up being quite impractical. It does work for simplifying code but practical,

stable, and reliable usage is not that beneficial in its current state. In the end this thesis provides compact but comprehensive information about Clojure and advanced information about references and creating a library for Clojure.

Keywords: Clojure, References, Databases, Data storage, Functional programming, Data structures, Atom, Library

Termiluettelo

Dead lock	Tilanne ohjelmakoodissa, jossa suoritus odottaa toista tai toisia suorituksia, jotka odottavat keskenään toisiaan. Tällöin suoritus jää niin sanotusti umpikujaan.
Higher order function	Korkeamman kertaluvun funktio. Esimerkiksi funktio, joka määrittelee ja palauttaa uuden funktion.
Infix	Notaatio, jossa operaattori kirjoitetaan operandien väliin. Prefixin vastakohta.
JVM	Java Virtual Machine on abstraktio määritelmä ja ajonaikainen virtuaalikone, jolla voi suorittaa Java-tavukoodia.
Kanava	Asynkroninen kommunikaatioväline rinnakkaisohjelmoinnissa. Englanniksi Channel.
Keyword	Clojuren symboli, jonka osoittama arvo on avainsana itse.
Makro	Käännösaikana tulkattava funktio, joka generoi ohjelmakoodia.
Nimiavaruus	Symboleja yhteen sitova määre, joka rajaa esimerkiksi muuttujien ja funktioiden näkyvyysaluetta. Rajauksella tehdään ohjelman osista keskenään yhteensopivia välttämällä nimien yhteentörmäyksiä. Englanniksi namespace.
POC	Englanniksi Proof of Concept tarkoittaa sovelluksen tai järjestelmän ensimmäistä versiota, jolla kokeillaan sovelluksen idean toiminta ja jonka pohjalta usein päätetään, toteutetaanko projekti.
Polymorfismi	Monimuotoisuus, joka mahdollistaa saman ohjelmakoodin osien uudelleenkäytön. Polymorfismi on tekniikka, jossa toteutetaan yhtenäinen rajapinta erilaisille ja erityyppisille arvoille.
Prefix	Notaatio, jossa operaattori kirjoitetaan ennen operandeja. Infixin vastakohta.
REPL	Read Eval Print Loop on ohjelmointiympäristö tai interaktiivinen tulkki, jolla voi suorittaa ohjelmakoodia.

Regex	Säännölliset lausekkeet.
Skeema	Tiedon muodon esittämistä kuvaava malli.
Thread safe	Rinnakkaisohjelmoinnissa säieturvallinen kommunikaatio, jossa esimerkiksi umpikujatilanteita ei synny.

Taulukot

Sisältö

1 Johdanto

"Simplicity is the ultimate sophistication"

- Leonardo da Vinci

Clojure on ohjelmointikielenä kiehtova. Se on samalla yksinkertainen, elegantti ja kuitenkin monipuolinen. Koodin uudelleenkäytettävyys on tärkeää Clojuressa. Tietorakenteet ja koodi itsessään pohjautuvat muutamaan eri funktioon ja tietorakenteeseen sekä näiden lukuisiin yhdistelmiin. Usein ohjelmakoodia kirjoittaessa huomaa, miten asiat voikin ilmaista yksinkertaisesti.

Olen käyttänyt Clojurea nyt useamman vuoden ajan. Koska kieli on vielä suhteellisen nuori ja sen käyttäjäkunta ei ole vielä kovin suuri, ei kielestä ole julkaistu suomenkielistä aineistoa liiaksi asti. Haluankin antaa oman panokseni tämän ongelman korjaamiseksi. Aiheesta olisi paljon helpompi kirjoittaa englanniksi, koska kaikkea kielen termistöä ei ole suomennettu sekä suurin osa lähteistä on englanniksi. Tästä huolimatta ajattelin, että annan kontribuutioni suomalaiseen Clojure-yhteisön kehittämiseen ja kenties omalla työlläni lisään näkyvyyttä kielelle kotimaassamme. Onhan Suomessa väkilukuun nähden kenties eniten clojuristeja maailmassa.

Palataksemme tutkielman aiheeseen. Miksi tietokantayhteyksiä ja tiedon pysyvää tallentamista pitäisi parantaa Clojuressa? Usein tietokantayhteyden rakentamista ohjelmaan ei voi kuvailla yksinkertaiseksi. Tietokantakyselyt tapahtuvat muusta ohjelmakoodista poikkeavalla ohjelmointikielellä ja erilaisella syntaksilla. Tietorakenteet ovat tietokannassa erilaiset kuin ohjelmakoodissa, joten näiden välille tarvitaan väistämättäkin muunnoksia. Muunnoksien vuoksi saatetaan joutua tekemään kompromisseja. Pahimmassa tapauksessa tietokanta rajoittaa ohjelman toiminnallisuuksia ja sen ajoympäristöä.

Clojure-ohjelmointikielessä data on keskiössä. Tiedon säilyttämistä varten kieleen on kehitetty tavallisten tietorakenteiden tueksi viitteet. Clojuren viitteet ovat monikäyttöisiä apuvälineitä niin tietorakenteeseen viittaamiseen kuin niiden jakamiseen esimerkiksi koodilohkojen

tai säikeiden kesken. Vaikka viitteitä ei ole montaa erilaista, löytyy silti jokaiseen tilanteeseen sopiva.

Voisiko näiden viitteiden ominaisuuksia kenties hyödyntää tiedon pysyvässä tallentamisessa? Saavutetaanko tällä mahdollisesti parempi ohjelman suorituskyky? Entä onko ohjelmakoodi yksinkertaisempi tai luettavampi, jos tiedon persistentti tallennus tapahtuu automaattisesti suoraan ohjelman tietorakenteita käyttämällä? Millaiseen ohjelmaan tällainen ratkaisu sopii?

Luku 2 on katsaus Clojure-ohjelmointikielestä. Siinä tutustutaan kielen syntyyn ja teknisiin ratkaisuihin. Luvussa syvennytään myös Clojuren rinnakkaisohjelmoinnin ratkaisuihin.

Luku 3 johdattaa lukijan viitteisiin. Luvussa selviää, mitä ne ovat ja mitä niillä voi tehdä. Luvussa luodaan katsaus Clojuren viitteisiin, niiden ominaisuuksiin ja käyttökohteisiin. Lisäksi tehdään vertailu Clojuressa olevista viitteistä. Clojuren lisäksi tutustutaan myös muiden kielten viitteisiin ja viitteen kaltaisiin ratkaisuihin.

Luvussa 4 sukelletaan tietorakenteisiin ja tietokantoihin. Siinä tehdään luotaus Clojuren tietotyyppeihin, tietorakenteiden abstraktioihin ja tietorakenteisiin. Luku antaa kuvan siitä, miten dataa hallitaan Clojure-ohjelmointikielessä. Lisäksi luvussa tehdään katsaus erilaisiin tietokantatyyppeihin. Lopuksi tutustutaan Clojuren 1.9.0-version kenties tärkeimpään ominaisuuteen, muun muassa datan validointiin käytettävään, clojure.spec-kirjastoon.

Luvussa 5 tehdään katsaus tutkielmassa toteutettavan kirjaston suunnitteluun. Siinä selviää, mitkä ovat kirjaston tavoitteet. Luvussa selvitetään myös kirjaston hyödyt ja haasteet. Lopuksi kerrotaan kehityksessä ja testauksessa käytettävät työkalut sekä kuinka kirjastoa tullaan testaamaan.

Luku 6 paneutuu kirjaston toteutukseen. Aluksi valitaan kolme vertailuprojektia koekaniineiksi kirjaston käytännön testaamiseen. Projektien avulla selvitetään, mitä muutoksia koodiin tarvitaan kirjaston käyttämiseksi. Näin saadaan arvokasta tietoa, helpottaako kirjasto kenties ohjelmoijan työtä. Luvussa käydään läpi myös kirjaston kehitysprosessi aina projektin luomisesta kirjaston toteutukseen asti. Lisäksi pohditaan, kuinka kehitettävästä kirjastosta

saataisiin parempi optimoimalla. Luvussa käydään läpi myös suorituskykytestien toteuttaminen ja kirjaston lisääminen vertailuprojekteihin.

Luvussa 7 käydään läpi suorituskykytestien tulokset. Näin selviää, kuinka kirjasto pärjää verrattuna suoraan tietokantayhteyteen. Lisäksi tehdään katsaus, mitä vaikutuksia kirjastolla on vertailuprojektien koodiin, kehitykseen ja niiden toimintaan. Luvussa selvitetään kehitettävän kirjaston hyödyt, haasteet ja käyttökohteet. Lisäksi pohditaan, miten kirjastoa voitaisiin parantaa.

Luvussa 8 käydään läpi johtopäätökset.

Luku 9 on kokonaisuudessaan yhteenveto tutkielman sisällöstä. Se tiivistää työn sisällön, rakenteen ja kuinka eri osat tukevat toisiaan. Luvussa palautetaan mieleen tutkimuskysymys, tutkielman tekninen osuus ja tulokset.

2 Clojure

2.1 Clojure ja sen historia

Vielä 1950-luvun lopulla siisti tulostus ja kätevä notaatio eivät olleet tärkeitä ohjelmoinnissa. Rajoitteena oli muun muassa laitteiden muisti. Rajoituksista huolimatta John McCarthy aloitti kehittämään ohjelmointikieltä, joka on ihmisellekin luettavaa. Lopulta tästä ajatuksesta syntyi Lisp. Kielen kehitys aloitettiin lopulta vuonna 1958. (McCarthy 1978)

McCarthy kehitti Lisp-ohjelmointikielen, jotta ohjelmakoodi voidaan esittää luettavina lauseina, jotka määrittää ohjelman toimintalogiikan. McCarthyn mielestä juurikin listamallinen lauseiden esittäminen ja tiedon prosessointi (list processing) vaikutti tehtävään sopivalta. Niin sanottu prefix-notaatio oli luonnollinen listamaisen ohjelmakoodin kanssa. Notaatio yksinkertaistaa laskutoimituksien ohjelmoinnin, kuten vähennyslaskut, sievennykset, integraatiot ja differentiaalilaskut. (McCarthy 1978)

Clojure, joka on yksi Lisp-kielen murteista, julkaistiin ensimmäisen kerran syksyllä 2007 ja ensimmäinen vakaa versio julkaistiin toukokuussa 2009 (VanderHart ja Sierra 2010, Luku 1). Clojuren on kehittänyt Rich Hickey (Pratley 2016, Johdanto). Hickey suunnitteli kieltä kaksi vuotta, ennen kuin aloitti varsinaisen kielen toteutuksen (McDonnell 2017, Luku 1). Clojureen on kerätty hyvät osat muista kielistä ja jätetty pois niiden huonot valinnat toteutuksessa (McDonnell 2017, Luku 1). Hickey kehitti kielen ensisijaisesti erityisen käytännölliseksi ohjelmointikieleksi (Rathore 2011, Luku 1.3.7). Tätä tukee JVM:n valinta alustaksi, koska se mahdollistaa laajan Java-kirjastovalikoiman käytön Clojuressa (Rathore 2011, Luku 1.3.7).

Clojure nojaa kolmeen kulmakiveen: Lisp-ideologia, funktio-ohjelmoinnin paradigma ja JVM. Clojure ei ole ensimmäinen JVM:n päälle kehitetty Lisp-kieli. Esimerkiksi JScheme, Kawa ja ABCL ovat aiempia yrityksiä, jotka syystä tai toisesta eivät koskaan saavuttaneet suosiota. (Rathore 2011, Luvut 1.1.1, 1.3)

2.2 JVM

Java on yksi suosituimmista ohjelmointikielistä. Sen suosiosta voidaan mahdollisesti kiittää Java Virtual Machinea (JVM). JVM on, kuten nimestäkin voi päätellä, virtuaalikoneympäristö. Se on saatavilla useille eri alustoille, joten JVM mahdollistaa saman ohjelmakoodin käyttämisen alustasta riippumatta ("Write once, run everywhere"-periaate). Sovellus käännetään tavukoodiksi, joka voidaan suorittaa niillä alustoilla, joille JVM on saatavilla. Ympäristönä JVM tarjoaa myös turvallisen hiekkalaatikon sekä sovelluksen tehokkuutta parantavia optimointeja. (Li, White ja Singer 2013)

Nimestä huolimatta Java ei ole ainoa kieli, jota voidaan kääntää JVM:n ymmärtäväksi tavukoodiksi. Usein valitaankin Javan sijaan toinen kieli, joka tarjoaa ominaisuuksia, mitä Javasta ei löydy. Asetelma mahdollistaa myös Java-yhteensopivan koodin kirjoittamisen valitulla kielellä. Tämä tarkoittaa muun muassa laajan Java-kirjastovalikoiman, automaattisen muistinhallinnan ja optimointien hyödyntämisen. Usein nämä muut kielet käyttävät puhdasta Javaa taustalla. JVM on selvästi Java-optimoitu, joten paras mahdollinen hyöty ympäristön tehokkuudesta saavutetaan Java-ohjelmointikielellä. Myös Clojure hyödyntää tätä. (Li, White ja Singer 2013)

2.3 Rinnakkaisohjelmointi ja core.async-kirjasto

Vaikka moniytimisten suorittimien idea ei ole uusi, sen merkitys on kasvanut viime aikoina selvästi. Aiemmin rinnakkaisuuden ja yhtäaikaisuuden tekniikat on voitu jättää pienemmälle huomiolle ja hyödyntää alati nopeutuvia suorittimia paremman suorituskyvyn saavuttamiseksi. Tämä kehityssuunta on kuitenkin hidastumassa, joten on aika suunnata katseet ohjelmointikielen ratkaisuihin. (Fogus ym. 2014, s. xxxviii)

Mitä eroa on ohjelmoinnissa yhtäaikaisuudella (engl. concurrency) ja rinnakkaisuudella (engl. parallelism)? Yhtäaikaisuus viittaa erillisten tehtävien suoritukseen suurin piirtein samaan aikaan. Jokaisella tehtävällä voi olla yhteiset jaetut resurssit, mutta ne eivät välttämättä suorita toisiinsa liittyviä tehtäviä. Usein yhtäaikaisten tehtävien lopputulokset vaikuttavat toisiin tehtäviin. Rinnakkaisuus taas viittaa tehtävän jakamiseen pienempiin osiin, joista jokainen suoritetaan samaan aikaan. Tyypillisesti rinnakkaisilla tehtävillä on yhteinen päämää-

rä. Jokainen tehtävä toimii itsenäisesti, eikä niiden lopputulokset vaikuta toisiinsa. (Fogus ym. 2014, Luku 10)

Clojure on suunniteltu monimutkaisten ongelmien ratkaisemiseen lisäämättä ohjelmakoodin monimutkaisuutta. Myös rinnakkaisohjelmoinnin toteutuksessa on pyritty tähän. Clojuren suunnittelussa on otettu opiksi esimerkiksi olio-ohjelmoinnin rinnakkaisohjelmoinnin haasteista. (Fogus ym. 2014, Luku 1.1.1 ja 1.4)

Core.async on kirjasto Clojurelle, joka on tarkoitettu asynkroniseen ja rinnakkaiseen kommunikaatioon kanavien (channel) kautta. Kanavat ovat tärkeä työkalu nopeaan datan käsittelyyn. Ne mahdollistavat itsenäisten säikeiden keskinäisen kommunikaation. (Meier 2015, Luku 6)

Core.async-kirjaston kanava luodaan oletuksena yhden arvon kokoisella puskurilla. Puskurin koko voidaan määrittää valinnaisella parametrilla, joka annetaan kanavaa luotaessa. Kanava ottaa vastaan minkä tahansa arvon, paitsi *nil*. Arvo nil on erikoistapaus ja indikoi suljettua kanavaa. Kanavaa voidaan käyttää sekä asynkronisesti että synkronisesti. Asynkroniset kutsut sijoitetaan *go*-lohkon sisälle. Go-lohkot ja niiden sisäisen silmukan (loop) sisältävä laajennus eivät ole sidottu säikeeseen, vaan ne ovat todella kevyitä prosesseja. Aiemmin mainitut synkroniset kutsut tunnistaa kahdesta huutomerkistä funktion nimen lopussa kun taas asynkronisessa kutsussa huutomerkkejä on vain yksi. (Meier 2015, Luku 6)

Meier (2015, Luku 6) ja Higginbotham (2015, Luku 11) esittelee *core.async*-kirjaston tärkeimmät funktiot ja niiden käyttötarkoitukset. Nämä kuvataan tiivistetysti taulukossa 1.

2.4 STM, MVCC ja transaktiot

Clojuressa ei käytetä lukkoja rinnakkaisohjelmoinnissa. Koska kehityksessä ei tarvita lukkoja tai ohjelmakohtaisia lukkoskeemoja, ei ohjelmakoodiin voi syntyä umpikujia (engl. deadlock). Clojuren käyttämän Software Transaction Memoryn (STM) vuoksi esimerkiksi viitteiden monitorointia ei tarvita ja säikeen herätys -ongelmia ei voi syntyä. Clojuren STM käyttää Multiversion Concurrency Control (MVCC) -menetelmää tilannekuvan eristämisen

Funktio	Käyttötarkoitus	Ominaisuudet
chan	Kanavan luonti	Valinnainen parametri määrittelee puskurin koon
>!	Datan laittaminen kanavaan	Asynkroninen, käytetään go-lohkon sisällä
>!!	Datan laittaminen kanavaan	Synkroninen
<!	Datan ottaminen kanavasta	Asynkroninen, käytetään go-lohkon sisällä
<!!	Datan ottaminen kanavasta	Synkroninen
alts!	Ensimmäisen datan ottaminen usean kanavan joukosta	Asynkroninen, käytetään go-lohkon sisällä
alts!!	Ensimmäisen datan ottaminen usean kanavan joukosta	Synkroninen
close!	Kanavan sulkeminen	

Taulukko 1. Core.async-kirjaston tärkeimmät funktiot

varmistamiseksi. Eristäminen tässä tapauksessa tarkoittaa sitä, että jokainen transaktio saa oman näkymän viitteidensä datasta jota se käyttää. (Fogus ym. 2014, Luku 10.1.3)

STM ei ole kuitenkaan hopealuoti vaan myös sen käytössä on omat ongelmansa. Esimerkiksi jos jokin transaktio käyttää viitteen arvoa oman käyttäytymisen sääntelyyn, mutta ei kirjoita kyseisen viitteen arvoon. Jos samaan aikaan toinen transaktio kirjoittaakin saman viitteen arvoon, syntyy niin kutsuttu kirjoitusvääristymä (engl. write skew). Clojure käyttää ongelman ratkaisemiseen *ensure*-funktiota. Toinen mahdollinen ongelmatilanne on reaaliaikainen lukko (engl. live lock). Se tapahtuu, kun transaktiot toistuvasti uudelleenkäynnistävät toisiaan. Clojuressa tilannetta hallitaan rajoittamalla transaktioiden uudelleenkäynnistyksien määrää sekä priorisoimalla vanhempia transaktioita uudempien edelle. (Fogus ym. 2014, Luku 10.1.5)

Clojuressa on rajoitteita, mitä transaktiossa tulisi tehdä. Kaikki ulkoisten liitäntöjen operaatiot (I/O) ovat kiellettyjä, koska transaktioita saatetaan joutua käynnistämään uudelleen montakin kertaa. Myös luokan ilmentymien muokkausta tulisi välttää, koska aina ei voida varmistua siitä, että jokaisella mutaatiokerralla olisi aina sama lopputulos. Näiden rajoituksien lisäksi tulisi pitää transaktiot mahdollisimman tiiviinä ja tehokkaina. (Fogus ym. 2014, Luku 10.1.6)

3 Viitteet

Jos jokin asia tapahtuu kokonaan tai ei tapahdu ollenkaan, kutsutaan sitä atomiseksi. Esimerkiksi monet tietokantatransaktiot ja joidenkin Unix-käyttöjärjestelmien tiedosto-operaatiot ovat juuri tällaisia. Ominaisuus tuo turvaa rinnakkaisohjelmointiin. Atomisten operaatioiden kanssa ei tarvitse esimerkiksi murehtia, mitä muut säikeet tekevät. (Kaihlavirta 2017, Luku 7, s.154-155)

3.1 Ratkaisuja muissa kielissä

Rust-ohjelmointikielen rinnakkaisohjelmoinnissa jaetun resurssin käyttö turvataan *mutexilla*. Se varmistaa, että tiettyä osaa koodista suoritetaan ainoastaan yhdessä säikeessä kerrallaan. Mutex lukitsee arvon käytön ajaksi. Mutex ei kuitenkaan toimi suoraan useasta eri säikeestä. Reference counted container on tarkoitettu Mutexin rinnalle arvojen jakamiseen useaan paikkaan. Tavallinen reference counted ei suoraan toimi, koska se ei ole säieturvallinen. Tätä tilannetta varten Rustissa on *Atomic RC*. Mutex-lukituksilla on kuitenkin iso vaikutus suorituskykyyn, joten sen käyttö hidastaa ohjelman suoritusta merkittävästi. (Kaihlavirta 2017, Luku 7, s.153-158)

Javassa on atomisen viitteen luokka nimeltä *AtomicReference*. Se tarjoaa säieturvallisen tavan muokata ja käyttää ennaltamäärättyä oliota. AtomicReference ei ole kuitenkaan ainoa Javan atomisista viitteistä. Kielessä on eri käyttötarkoituksiin viitteitä, kuten atomiset taulukko (AtomicReferenceArray) ja luokan kentän päivittäjä (AtomicReferenceFieldUpdater). (Flanagan 2005, Luku 16)

Scala-ohjelmointikielessä on tavoitteena puhtaat, eli ilman sivuvaikutuksia olevat funktiot. Kielen filosofian mukaan tämä ei kuitenkaan tarkoita, että kaikki ohjelmakoodissa olisi täysin muuttumatonta. Funktion sisällä arvojen muuttaminen luetaan vielä puhtaaksi, kunhan muutokset eivät näy ulkomaailmalle. Säieturvallisen ja säikeiden väliseen kommunikointiin on käytössä Javan AtomicReference. (Paul ja Bjarnason 2014, Luvut 14, 14.1 ja 7.4.4)

Elixir-ohjelmointikielessä on useita tapoja viestien ja operaatioiden lähettämiseen ja vastaanottamiseen eri solmujen välillä. Näistä helppokäyttöisiä ja yksinkertaisia abstraktioita ovat *Task* eli tehtävä ja *Agent* eli agentti. Yksinkertaistetusti Task on tausta-ajona suoritettava funktio. Suoritettavan funktion paluuarvo voidaan pyytää estämättä muuta ohjelman suoritusta. Tehtävää voidaan erikseen myös valvoa. Agentit taas ovat tausta-ajoprosesseja, jotka pystyvät säilyttämään tilaa. Tilan voi jakaa eri prosessien tai solmujen kesken. Tilaa pystyy myös päivittämään. (Thomas 2018, Luku 22)

3.2 Muita viitteen kaltaisia ratkaisuja

IVar on tunnettu deterministinen mekanismi rinnakkaisohjelmointiin. IVar on muuttumaton ja sen arvon voi asettaa vain kerran. Sen luku on suoritusta estävä. Tyhjän IVarin lukeminen estää suorituksen, kunnes sen arvo on asetettu. Nimen alkukirjain I on lyhennys englanninkielen sanasta immutable eli muuttumaton. *LVar* on IVarin yleistys. Se puolestaan sallii arvon kirjoituksen useamman kerran. Rajoituksena kirjoituksille on arvon monotoninen kasvu annetun hilan ehdoilla. (Kuper 2013, Luku 2.2)

Prokopec ym. (2013) esittää suunnitelman dataflow-tyyppisen kokoelman abstraktiosta nimeltä *FlowPool*. Se on suunnattu rinnakkaisohjelmointiin. FlowPoolin taustalla on tehokas suoritusta estämätön tietorakenne, joten esimerkiksi lukkoja ei tässä ratkaisussa tarvita. Perinteisen iteroinnin tai datavirran (engl. stream) sijaan FlowPooleja voidaan rakentaa korkeamman kertaluvun funktioilla (engl. higher order function), kuten *foreach* ja *aggregate*. Tämän ominaisuuden avulla suoritus tapahtuu asynkronisesti kun FlowPoolin elementit tietovirtakaaviossa (engl. dataflow graph) tulevat saataville. Rakentamiseen pohjautuva lähestymistapa mahdollistaa myös roskien keruun tietorakenteen osille, joita ei enää tarvita. (Prokopec ym. 2013)

3.3 Viitteet Clojuressa

Clojuressa on käytettävissä neljä viitetyyppiä: *ref*, *agent*, *atom* ja *var*. Mitä viitteet sitten oikein ovat? Emerick, Carper ja Grand (2012, Luku 4) kuvaa viitteitä eräänlaisina laatikoina, jotka sisältävät arvon. Tätä arvoa pystytään muokkaaman funktioilla, jotka ovat erilaisia jo-

kaiselle viitetyypille. Jokaisella viitteellä on aina arvo, ainakin nil (Emerick, Carper ja Grand 2012, Luku 4).

Huolimatta Clojuren viitteiden erilaisista muokkaus- ja päivitysfunktioista on kaikilla viitetyypeillä arvon noutaminen (dereference) samanlainen. Kaikkien viitteiden kanssa voidaan käyttää joko *deref*-funktiota tai @-lukijasyntaksia. Näistä jälkimmäistä suositaan lähes aina. Ainoastaan korkeamman kertaluvun funktiossa käytetään *deref*-funktiota[1]. Viitteen arvoa noudettaessa funktio ja lukija palauttavat viitteen sen hetkisen tilannekuvan. Arvoa ei kopioida. Jos käytetään Clojuren muuttumattomia tietorakenteita ja tietotyyppejä, on paluuarvo koskematon. *deref*-funktio ei koskaan estä suoritusta, vaikka viitteen semantiikka muuttuu tai viitettä operoidaan toisesta säikeestä. *deref* ei myöskään koskaan häiritse tai keskeytä mahdollista toista operaatiota. Funktion pystyy halutessaan estämään esimerkiksi *delay*-, *promise*- tai *future*-operaatioilla. Esimerkiksi joskus halutaan varmistua, että viitteelle on asetettu arvo ennen sen noutamista. (Emerick, Carper ja Grand 2012, Luku 4)

Jos viitteen arvon muutosta halutaan valvoa, voidaan käyttää *add-watch* -funktiota. Funktiolle annetaan yhtenä parametrina funktio, jota kutsutaan, kun viitteen arvo muuttuu. (Fogus ym. 2014, Luku 8.3)

3.4 Clojuren Atom-viitteen ominaisuuksia

Clojuressa *Atom* on synkroninen kuten *ref* ja koordinoimaton kuten *agent*. Atomia käytetään usein vertaa-ja-muuta-tyyppisissä operaatioissa. Atom soveltuu tilanteeseen, jossa viitteen uusi arvo liittyy vanhaan arvoon. Esimerkiksi uusi luku lasketaan vanhasta arvosta. Atomin päivitys tapahtuu paikallisesti suorittavassa säikeessä. Jos toinen säie ehtii päivittää arvon, yritetään päivitystä uudelleen. Uudelleenyritys tapahtuu silmukassa eikä *Software Transactional Memoryssa*. Atom on säieturvallinen viite. (Fogus ym. 2014, Luku 10.4, 10.4.1)

Atomin arvon päivitysoperaatio estää suorittavaa säiettä, kunnes päivitys on valmis. Päivitysoperaatio on eristetty, joten synkronointi esimerkiksi muiden säikeiden tai atomien välillä ei onnistu. Atomin arvon muokkaus tapahtuu nimestäkin päätellen atomisesti, joten sen arvo on aina konsistentti kaikille säikeille. (Emerick, Carper ja Grand 2012, Luku 4)

1. Korkeamman kertaluvun funktion kanssa ei lukijasyntaksia ole mahdollista käyttää

Kuinka Atom-viitettä käytetään? Ilmentymä luodaan *atom*-funktiolla. Atomin arvoa muokataan *compare-and-set!*- ja *swap!*-funktioilla. Jälkimmäiselle funktiolle annetaan parametrina funktio, jolla viitteen arvoa muokataan. Koska arvon päivityksen epäonnistuessa päivitystä yritetään uudelleen, tulee päivitysfunktion olla puhdas, eli ilman sivuvaikutuksia. Onnistuessaan operaatiossa *swap!*-funktio palauttaa uuden, atomin päivitetyn arvon. Sisäisesti *swap!*-funktio käyttää *compare-and-set!*-funktiota arvon päivittämiseen. Funktio palauttaa ainoastaan totuusarvon, joka kertoo, onnistuiko päivitys vai ei. Kuten nimestäkin voi päätellä, *compare-and-set!*-funktio vertailee atomin uutta ja vanhaa arvoa. Vertailu ei tapahdu semanttisesti arvoja vertailemalla vaan se perustuu identtisyyteen[2]. Viitteen arvo asetetaan uudelleen *reset!*-funktiolla. Lisää arvon päivittämisen uudelleenyrittämisestä kerrotaan seuraavassa luvussa. (Emerick, Carper ja Grand 2012, Luku 4)

3.5 Viitteiden vertailu

Emerick, Carper ja Grand (2012, Luku 4) kirjoittaa, että Clojuren viitteille on yhteistä arvon noutamisen funktion ja lukijan lisäksi seuraavat ominaisuudet:

- Mahdollisuus asettaa metatietoja.
- Mahdollisuus tilan muutoksien ilmoittamiselle.
- Mahdollisuus arvon validointiin.

Joillakin viitteillä on myös erityispiirteitä. Atom- ja ref-viitteen käyttämä uudelleenyrittämisen mahdollisuus on yksi näistä. Tämä tarkoittaa sitä, että esimerkiksi arvon päivittäminen on spekulatiivinen operaatio. Päivitystä saatetaan joutua yrittämään uudelleen. Ref-viitettä taas ohjaa Clojuren Software Transactional Memory. Se varmistaa tilan yhtenäisyyden ohjelman elinkaaren ajan transaktioiden avulla. Var-viitteitä lukuun ottamatta kaikki viitteet ovat jaettuja. Niiden tilan muutokset voidaan nähdä eri säikeissä. Vaikka usein eri viitteiden ominaisuudet ovatkin samanlaisia, on jokaiselle viitteelle oma ideaali käyttötapaus. (Fogus ym. 2014, Luku 10.1)

2. Clojuressa on *identical?*-funktio, jolla voidaan verrata kahden objektin identtisyyttä (Emerick, Carper ja Grand 2012, Luku 11)

Viitetyypin valintaa helpottamaan voidaan viitteiden rinnakkaiset operaatiot jakaa kahteen avainkonseptiin: koordinaatio ja synkronisaatio. Koordinoidussa operaatiossa kaikki toimijat tekevät yhteistyötä tai eivät ainakaan vaikuta toisiinsa. Toisaalta taas koordinoimattomat operaatiot ovat sellaisia, joissa toimijat eivät voi vaikuttaa toisiinsa. Niiden toimintaympäristöt ovat eristettyjä. Esimerkiksi useassa säikeessä eri tiedostoihin kirjoitus on eristetty. Synkronisissa operaatioissa kutsujan säikeen suoritus odottaa, estää tai nukkuu, kunnes sillä on yksinoikeus annettuun kontekstiin. Asynkroniset operaatiot taas ovat sellaisia, joita voi käynnistää tai ajastaa ilman että ne estävät kutsujan säiettä. (Emerick, Carper ja Grand 2012, Luku 4)

Jakamalla rinnakkaiset operaatiot aiemmin mainittuun kahteen konseptiin, jotka ovat koordinaatio ja synkronisaatio, pystytään kuvaamaan suurin osa rinnakkaisista operaatioista. Clojuren viitetyypit on suunniteltu kattamaan kyseisten konseptien permutaatiot. Näitä permutaatioita käyttäen viitteiden, paitsi var-viitteen, sopivuus käyttökohteeseen on kätevästi luokiteltavissa. Var-viitetyypit toimivat ainoastaan yhdessä säikeessä (thread-local), joten niitä ei koordinaatio tai synkronisaatio koske. (Emerick, Carper ja Grand 2012, Luku 4)

Luokittelu on havainnollistettu taulukossa 2.

	Koordinoitu	Koordinoimaton
Synkroninen	Ref	Atom
Asynkroninen		Agent

Taulukko 2. Viitteiden operaatiojako

4 Tietorakenteet ja tietokannat

4.1 Clojuren tietotyypit ja niiden syntaksi

Clojuressa on laaja kirjo erilaisia tietotyyppejä. Useimpien ohjelmointikielien tapaan löytyy myös Clojuresta skalaarityypit, kuten kokonaisluvut (integer), merkkijonot (string) ja liukuluvut (floating-point number). Clojuressa skalaarityypit jaetaan useaan eri kategoriaan, kuten kokonaislukuihin, liukulukuihin, rationaalilukuihin, symboleihin, avainsanoihin, merkkijonoihin, merkkeihin, totuusarvoihin ja regex-malleihin. (Fogus ym. 2014, Luku 2.1)

Clojuressa numeroiden syntaksi voi sisältää numerot nollasta yhdeksään, desimaalipisteen, plus- tai miinusmerkin. Lisäksi notaatiossa voi olla e-kirjain eksponentiaaliosoittimena, iso m-kirjain osoittamassa tarkkuutta tai iso n-kirjain esittämässä mielivaltaisen kokoista kokonaislukua. Esimerkkejä numeroiden syntaksista on esitelty taulukossa 3. Kymmenkantaisen desimaaliluvun lisäksi Clojure tukee myös oktaali- ja heksadesimaaleja. Primitiivityyppien tarkkuuden rajoitukset, lukuun ottamatta mielivaltaisella tarkkuudella olevat luvut, periytyvät alustalta. Alustat ovat tällä hetkellä JVM ja JavaScript. (Fogus ym. 2014, Luku 2.1.1)

Tietotyyppi	**Esimerkkejä**
Kokonaisluku	`1 +20 -500`
Liukuluku	`1.0 +20.5 -5. 2.4e8 6.9e-12 4e-14 9.99M 42N`
Rationaaliluvut	`-1/20 5/4`
Symboli	`some-value`
Avainsana	`:some-keyword`
Merkkijono	`"Example String"`
Merkki	`\a \B \u0042 \\ \u30DE`

Taulukko 3. Esimerkkejä Clojuren tietotyypeistä

Clojuressa kokonaislukujen syntaksi alkaa joko numerolla tai plus- tai miinusmerkillä jota seuraa ainoastaan numeroita (taulukko 3). Kokonaislukujen tarkkuutta rajoittaa ainoastaan käytössä oleva muisti (mielivaltainen tarkkuus). Kymmenkantaiset desimaaliluvut ovat Clojuressa JVM-alustalla yleensä joko Javan Long-tyyppisiä tai sen kokorajan ylittäessä Javan BigInt-tyyppisiä. (Fogus ym. 2014, Luku 2.1.2)

Liukuluvut ovat rationaalilukujen desimaalilaajennos. Kuten kokonaisluvut myös liukuluvut voivat olla mielivaltaisella tarkkuudella. Clojuressa liukuluvun syntaksi on numeroita desimaalimerkin tai eksponentiaalierottimen e kanssa (taulukko 3). Molemmissa notaatioissa valinnainen plus- tai miinusmerkki kertoo, onko luku positiivinen tai negatiivinen. (Fogus ym. 2014, Luvut 2.1.3 ja 2.1)

Clojuressa on kokonaislukujen ja liukulukujen tukena rationaaliluvut. Ne tarjoavat kompaktimman ja tarkemman esityksen luvusta kuin esimerkiksi liukuluvut. Kuten matematiikassa myös Clojuressa rationaaliluvut esitetään numeroilla ja jakomerkillä (taulukko 3). Rationaaliluvut sievennetään mahdollisuuksien mukaan automaattisesti. (Fogus ym. 2014, Luku 2.1.4)

Clojuressa symbolit (symbol) ovat objekteja, jotka esittävät asetettua arvoa. Kun symboli suoritetaan, palautuu symbolin osoittama arvo. Symboleja käytetäänkin pääasiassa viittaamaan funktion parametreihin, paikallisiin sekä globaaleihin muuttujiin ja Java-luokkiin. Esimerkki symbolin syntaksista esitellään taulukossa 3. (Fogus ym. 2014, Luku 2.1.6)

Avainsanat (keyword) ovat Clojuren symbolien kaltaisia, mutta niiden osoittama arvo on avainsana itse. Clojuren avainsanaa ei tule sekoittaa yleisesti ohjelmointikielissä käytettyyn avainsana-termiin. Avainsanan notaatio on kaksoispisteellä alkava merkkijono (taulukko 3). (Fogus ym. 2014, Luku 2.1.6)

Merkkijonot Clojuressa merkitään lainausmerkeillä, kuten monissa muissakin ohjelmointikielissä. Merkkijono voi koostua useasta eri rivistä. Yksittäisen merkin (character) notaatio on kenoviiva sekä merkki tai sitä vastaava unicode-tunniste. Esimerkki Clojuren merkkijonosta esitellään taulukossa 3. (Fogus ym. 2014, Luvut 2.1.7 ja 2.1.8)

4.2 Clojuren tietorakenteiden abstraktiot

Clojuressa abstraktiot menevät implementoinnin edelle, kun puhutaan tietorakenteista. Kaikki kielen tietorakenteet voidaan esittää seitsemällä abstraktiolla: kokoelma (collection), jono (sequence), assosiatiivinen (associative), indeksoitu (indexed), pino (stack), joukko (set) ja järjestetty (sorted). Koska monet tietorakenteiden funktiot, kuten *conj* ja *seq* (taulukko 4),

ovat polymorfisia tietorakenteiden tyyppien suhteen ja käyttävät abstraktioita, tukevat myös niitä käyttävät funktiot suoraan samoja tietorakenteita. Tästä esimerkkinä on *into*-funktio, joka on rakennettu *seq*- ja *conj*-funktioiden avulla. (Emerick, Carper ja Grand 2012, Luku 3)

Funktio	**Selite**
conj	Lisää arvon kokoelmaan
seq	Palauttaa jononäkymän kokoelmasta
into	Lisää arvot kokoelmasta toiseen
count	Palauttaa kokoelman koon
empty	Luo tyhjän kokoelman
first	Palauttaa kokoelman ensimmäisen alkion
rest	Palauttaa kokoelmasta muut alkiot, paitsi ensimmäisen
next	Palauttaa kokoelmasta seuraavan arvon, jos kokoelmassa on arvoja
assoc	Lisää arvon assosiatiiviseen kokoelmaan
dissoc	Poistaa arvon assosiatiivisesta kokoelmasta
get	Palauttaa arvon assosiatiivisesta kokoelmasta
contains?	Palauttaa totuusarvon siitä, löytyykö arvo kokoelmasta
nth	Palauttaa indeksin kohdalta arvon
pop	Palauttaa pinon ilman päällimmäistä arvoa
peek	Palauttaa pinon päällimmäisen arvon
disj	Poistaa joukosta arvon
rseq	Palauttaa jonon käänteisessä järjestyksessä
subseq	Palauttaa osan jonosta
rsubseq	Palauttaa osan jonosta käänteisessä järjestyksessä

Taulukko 4. Clojuren tietorakenteiden funktioita

Clojuressa on abstraktio *collection* eli kokoelma. Kaikki Clojuren tietorakenteet toteuttavat kokoelma-abstraktion. Kokoelmaan lisätään arvoja *conj*-funktiolla. Kokoelman koon voi kysyä *count*-funktiolla. Tyhjän kokoelman saa luotua *empty*-funktiolla. Kyseinen funktio palauttaa tyhjän tietorakenteen, joka on samaa tyyppiä kuin parametrina annettu tietorakenne. Edellä mainitut funktiot esitellään lyhyesti taulukossa 4. (Emerick, Carper ja Grand 2012, Luku 3)

Clojuren abstraktio *sequence*, eli jono, määrittelee tavan ottaa tietorakenteesta jononäkymä. Jononäkymä on tarkoitettu Clojuressa muun muassa tietorakenteen läpikäyntiin, mutta se ei ole iteraattori. Samaista abstraktiota käytetään myös lopputuloksena, jos käsitellään tietorakennetta. Jonoabstraktiotason toteuttavasta tietorakenteesta saa jononäkymän *seq*-funktiolla. Funktiot *first*, *rest* ja *next* ovat tarkoitettu tietorakenteen muokkaamiseen ja läpikäymiseen.

Erona rest- ja next-funktioilla on tyhjän ja yhden alkion kokoisen kokoelman käsittely. Edellä mainitut funktiot esitellään lyhyesti taulukossa 4. Kaikki Clojuren tietorakenteet toteuttavat jonon. Jononäkymän notaatio on sama kuin listalla (taulukko 5), mutta näitä kahta ei tule sekoittaa keskenään. (Emerick, Carper ja Grand 2012, Luku 3)

Abstraktio *associative*, eli assosiatiivinen, on avain-arvo-pareista koostuvien tietorakenteiden abstraktio. Funktiot *assoc*, *dissoc*, *get* ja *contains?* (taulukko 4) ovat tarkoitettu assosiatiivisen abstraktion käyttämiseen. (Emerick, Carper ja Grand 2012, Luku 3)

Clojuren abstraktio nimeltä *indexed*, eli indeksoitu, on niitä tietorakenteita varten, joissa arvot ovat tietyissä paikoissa. Arvojen paikat, eli indeksit, ovat nollasta n:nteen. Indeksoitu abstraktio tuo mukanaan *nth*-funktion (taulukko 4), jolla voidaan pyytää arvo tietystä paikasta tietorakennetta. (Emerick, Carper ja Grand 2012, Luku 3)

Abstraktio *stack*, eli pino, on nimensä mukaan pinojen abstraktio. Perinteisesti pinotietorakenteet toimivat LIFO-periaatteella (last-in, first-out), eli viimeiseksi lisätty arvo tulee ulos ensimmäisenä. Clojuressa ei ole suoraan pinotietorakennetta. Pinoabstraktion funktiot ovat *conj*, *pop* ja *peek* (taulukko 4). (Emerick, Carper ja Grand 2012, Luku 3)

Clojuren *set*, eli joukko, on abstraktio, jossa tietorakenteen arvot ovat itsensä avaimia. Joukon funktio on *disj* (taulukko 4), jolla poistetaan arvo tai arvot tietorakenteesta. (Emerick, Carper ja Grand 2012, Luku 3)

Viimeisenä esiteltävänä abstraktiona on *sorted* eli järjestetty. Tietorakenteet, jotka toteuttavat järjestetty-abstraktion, takaavat, että niiden arvot ovat aina järjestyksessä. Clojuren tietorakenteista ainoastaan mapit ja joukot tarjoavat järjestetty-abstraktion toteuttavan version. Funktiot *rseq*, *subseq* ja *rsubseq* (taulukko 4) ovat tarkoitettu järjestetty-abstraktion käyttämiseen. (Emerick, Carper ja Grand 2012, Luku 3)

4.3 Clojuren tietorakenteet

Tietorakenteet ovat muuttumattomia ja persistenttejä, kuten funktionaaliseen Clojure-ohjelmointiin kuuluukin. Clojuren yleisimmin käytetyt tietorakennetyypit ovat lista (list), vektori

(vector), joukko (set) ja map. Kaikkia näitä yhdistää jonoabstraktiotaso (sequence). (Emerick, Carper ja Grand 2012, Luku 3)

Lista, tietorakenne joka löytyy kaikista Lisp-kielistä, on Clojuren yksinkertaisin tietorakenne. Se koostuu yksittäin linkitetyistä arvoista. Jokainen solmu tietää oman etäisyytensä listan lopusta. Arvon etsiminen tapahtuu aina alusta arvoja läpi käymällä. Arvoja voi lisätä tai poistaa ainoastaan listan alusta. Listaa käytetään lähes yksinomaan koodilomakkeiden (code form) esittämiseen. Esimerkiksi funktioiden ja makrojen kutsut ovat listoja. Listaa voidaan käyttää pinona ja pinon funktiot toimivat listan kanssa. Erona pinoon on kuitenkin muistinkäyttö. Listaa ei tule käyttää kuitenkaan jonona (queue), koska arvojen lisäys ja poisto ei siihen sovellu. Lista ei ole tarkoitettu arvojen säilyttämiseen, jos arvoihin täytyy päästä käsiksi niiden indekseillä. Listaa ei tulisi käyttää silloinkaan, jos tarvitsee tarkistaa, löytyykö tietty arvo tietorakenteesta. Yleisesti ottaen listasta ei ole juuri etuja vektoriin verrattuna. Poikkeuksena muihin Lisp-kieliin verrattuna Clojuren lista on muuttumaton. (Fogus ym. 2014, Luvut 5.3, 5.3.1, 5.3.2 ja 5.3.3)

Arvon lisäys listaan tapahtuu *conj-* ja *cons*-funktioilla. *conj*-funktio on näistä suositeltu, koska se tekee lisäyksen tehokkaimmalla mahdollisella tavalla ja palauttaa aina listan. *cons* palauttaa jonon (Fogus ym. 2014, 5.3.1). Listan koko voidaan kysyä *count*-funktiolla. Operaation suoritusaika on aina vakio (Fogus ym. 2014, 5.3.1). Listasta saa otettua arvon *pop*-funktiolla ja loput listasta *rest*-funktiolla (Emerick, Carper ja Grand 2012, Luku 3). Lista luodaan *list*-funktiolla, jolle voi antaa minkä tahansa määrän arvoja (Emerick, Carper ja Grand 2012, Luku 3). *list?*-predikaattifunktio kertoo, onko jokin arvo lista (Emerick, Carper ja Grand 2012, Luku 3). Listan jononäkymän voi kysyä *seq*-funktiolla. Se on aina lista, toisin kuin muilla Clojuren tietorakenteilla. Listan esitystapa kuvataan taulukossa 5. (Emerick, Carper ja Grand 2012, Luku 3)

Tietorakenne	**Suomeksi**	**Syntaksi**
List	Lista	`(2 40 :arvo1 "Arvo 2")`
Vector	Vektori	`["Merkkijono" 20 :arvo]`
Set	Joukko	`#{100 :jokin-arvo}`
Map	Map	`{:avain "Arvo" :toinen-avain 50}`

Taulukko 5. Esimerkkejä Clojuren tietorakenteiden esitystavoista

Vektorit ovat Clojure-koodissa yleisimmin käytettyjä tietorakenteita arvojen tallentamiseen. Niissä on nolla tai useampi arvo indeksoituna ja järjestyksessä. Indeksiä käyttämällä vektorin kuhunkin arvoon pääsee käsiksi vakioajassa. Indeksien vuoksi liikkuminen eteen- ja taaksepäin on tehokasta. Vektori on samanlainen kuin useiden ohjelmointikielien taulukko. Vektori on näistä poiketen muuttumaton ja persistentti. Vektori on varsin monipuolinen tietorakenne ja sen muistin sekä prosessorin käyttö on tehokasta niin pienillä kuin suurillakin ilmentymillä. Suuriin listoihin verrattuna vektorin etuina ovat tehokas arvon lisäys ja poisto tietorakenteen lopusta, arvon ottaminen tai muokkaaminen tietorakenteen keskellä indeksiä käyttäen ja tietorakenteen läpikäyminen lopusta alkuun. (Fogus ym. 2014, Luvut 5.2 ja 5.2.2)

Vektoria käytetään ohjelman datan tallennuksen lisäksi funktion parametrien sekä *let*-makron liitoksen symbolien esittämiseen. Vektoria ei tule käyttää jos arvojen indeksit eivät voi olla järjestyksessä. Vektoriin ei voi lisätä arvoa jättämällä indeksiä välistä, vaan arvo voidaan lisätä ainoastaan vektorin loppuun. Vektoria ei voi käyttää jonona (queue), koska arvojen poistaminen ja lisääminen tietorakenteen alkuun ei ole tehokasta tai järkevää. Vektori ei myöskään sovellu käytettäväksi, jos tarvitaan tieto, onko jokin arvo tietorakenteessa. (Fogus ym. 2014, Luvut 5.2, 5.2.7)

Vektorin luonti tapahtuu joko *vec*-funktiolla jonosta, *vector*-funktiolla annetuista parametreistä tai hakasulkunotaatiolla (taulukko 5). Vektoriin lisätään arvoja toisesta jonosta *into*-funktiolla. Uuden vektorin ennaltamäärätyllä arvojen tyypillä luodaan *vector-of* -funktiolla. Funktiolle annetaan primitiivityyppi ja vektori yrittää muuntaa annetut arvot tähän tyyppiin. Arvoon tietyssä indeksissä päästään käsiksi *nth*-funktiolla. Myös *get*-funktio toimii, mutta silloin vektoria käytetään map-tietorakenteena. Sisäkkäisten vektorien arvo haetaan *get-in*-funktiolla. Myös vektorin käyttäminen funktiona toimii, kuten muidenkin Clojuren tietorakenteiden kanssa. Sopiva arvon ottamisen funktio riippuu tilanteesta. Arvojen määrä, eli vektorin koko, kysytään *count*-funktiolla. Arvoa muokataan *assoc*-funktiolla. Muuttamiseen kuluva aika on vakio, koska muokkaus luo uuden vektorin, joka on yhdistelmä vanhasta ja uudesta vektorin versiosta. Vektoria voi suurentaa *conj*-funktiolla. Tämä onnistuu myös *assoc*-funktiolla muokkaamalla vektorin ylimenevää indeksiä, joka on *assoc*-funktion poikkeustapaus. Lisäys ja poisto tapahtuu sisäisesti käsittelemällä vektoria pinona *conj* ja *pop* -funktioilla. Arvon voi ylikirjoittaa myös *replace*-funktiolla, joka käyttää sisäisesti *as-*

soc-funktiota. Sisäkkäisiä vektoreita, kuten map-tietorakenteitakin, voi muokata ja päivittää *assoc-in* ja *update-in* -funktioilla. Vektorista voi ottaa osan, niin kutsutun alivektorin, *subvec*-funktiolla. (Fogus ym. 2014, Luvut 5.2.1, 5.2.2, 5.2.3 ja 5.2.5)

Clojuressa on jonotietorakenne (queue). Jotta aiemmin esitelty jono (sequence) ei sekoitu seuraavaksi esiteltävään jonoon, kutsun tässä ja seuraavassa kappaleessa esiteltävää jonoa jonotietorakenteeksi. Clojuren muuttumaton jonotietorakenne toimii FIFO-periaatteella (first-in, first-out). Eli kokoelman vanhin arvo otetaan ensimmäisenä ulos. Jonotietorakenne koostuu sisäisesti kahdesta eri tietorakenteesta: jonosta, jossa on ensimmäiset arvot ja vektorista, jossa on loput arvot. Arvoja otetaan jonosta niin kauan, kuin niitä riittää. Sen jälkeen jälkimmäisestä vektorista otetaan jononäkymä ja käytetään sitä. Uudet arvot lisätään vektoriosan loppuun. Näin käytetään hyväksi molempien tietorakennetyyppien vahvuuksia. Usein ohjelmointikielissä käytetään käänteistä listaa jonotietorakenteena. On tärkeää huomata, että Clojuren jonotietorakenne on ainoastaan tietorakenne, ei työnkulkuun tarkoitettu mekanismi. (Fogus ym. 2014, Luvut 5.4 ja 5.4.2)

Clojuressa ei ole jonotietorakenteen luomiseen funktiota. Uusi jonotietorakenne luodaan käyttämällä tyhjää kokoelmaa: *clojure.lang.PersistentQueue/EMPTY*. Arvo lisätään kokoelman perälle *conj*-funktiolla. Arvo jonotietorakenteen alusta poistetaan *pop*-funktiolla. Ensimmäinen arvo palautetaan *peek*-funktiolla poistamatta sitä kokoelmasta. (Fogus ym. 2014, Luut 5.4, 5.4.1 ja 5.4.3)

Clojuressa joukko (set) toimii kuten matemaattinen joukko. Tavallisen joukon arvot eivät ole järjestyksessä. Clojuressa on myös järjestetty joukko, *sorted-set*. Jokaista arvoa joukossa voi olla vain yksi. Uutta joukon elementtiä lisätessä tarkistetaan arvon olemassaolo arvojen suoritustulosta vertaamalla. Esimerkiksi lista ja vektori, joilla on samat arvot, ovat joukon mielestä identtiset. (Fogus ym. 2014, Luvut 5.5, 5.5.1, 5.5.2)

Uusi joukko luodaan *hash-set*-funktiolla tai ristikkomerkin ja aaltosulkujen yhdistelmänotaatiolla (taulukko 5). Joukon voi luoda jonosta *set*-funktiolla (Emerick, Carper ja Grand 2012, Luku 3). Clojuressa joukko on sen jokaisen elementin funktio, joka palauttaa kyseisen arvon tai nil:n (Fogus ym. 2014, Luku 5.5.1). Arvon voi hakea myös *get*-funktiolla, jolloin voidaan määritellä myös oletusarvo, jos haettua arvoa ei joukosta löydy (Fogus ym. 2014,

Luku 5.5.1). Clojuressa on myös matemaattisten joukkojen funktioita, jotka löytyvät *clojure.set* nimiavaruudesta. Esimerkiksi *intersection*, *union* ja *difference* ovat tyypillisiä joukkojen kanssa käytettyjä funktiota (Fogus ym. 2014, Luvut 5.5.1, 5.5.4).

Map koostuu avain-arvo-pareista. Map on hyvä tietotyyppi esimerkiksi sovelluksen kehitysvaiheessa, kun ei vielä tiedetä, millaista tietotyyppiä tarvitaan. Kun suunnitelma selkenee, voidaan sopiva tyyppi määrittää defrecord-makrolla. Mapin käyttökohteita ovat muun muassa yhteenvedot, indeksit tai käännöstaulukot. (Emerick, Carper ja Grand 2012, Luku 3)

Kolme yleisintä map-tyyppiä ovat *hash map*, *sorted map* ja *array map*. *Hash map* luodaan *hash-map*-funktiolla. Sen avaimet ovat heterogeenisiä. Avaimet voivat olla mitä tahansa tyyppiä ja jokaisella avain-arvo-parilla ne voivat olla erit. *Hash map* ei takaa, että sen merkinnät ovat järjestyksessä. Siihen tarkoitukseen on *sorted map*, joka luodaan *sorted-map*-funktiolla. Oletuksena *sorted map* on avaimen mukaan järjestyksessä *compare*-funktion avulla vertailtuna. Omalla vertailufunktiolla varustettu map luodaan *sorted-map-by*-funktiolla. Järjestämisen vuoksi *sorted mapin* avaimet eivät yleisesti ottaen ole heterogeenisiä. Avainta käytetään sen itsensä arvon mukaan. Eli kokonaisluku 1 on sama kuin reaaliluku 1.0, kun niitä vertaillaan. *Sorted mapissa* voidaan hypätä tehokkaasti haluttuun kohtaan tietorakennetta ja liikkua siitä eteen- ja taaksepäin. Kolmantena map-tyyppinä on *array map*. Sitä käytetään, kun avain-arvo-parien lisäys pitää pysyä järjestyksessä. *Array map* ei ole tehokas suurissa tietorakenteissa. Jos halutaan tietää, onko jokin arvo mahdollisesti map-tyyppiä, voidaan testaus tehdä *map?*-predikaattifunktiolla. Map-tietorakenteen esitystapa kuvataan taulukossa 5. (Fogus ym. 2014, Luvut 5.6.1, 5.6.2, 5.6.3 ja 5.7.1)

4.4 Clojuren record

Aiemmin esitelty map-tietotyyppi on kätevä ja helppo käyttää. Tällä dynaamisella tietotyypillä on kuitenkin kääntöpuoli. Toisinaan tarvitaan tieto siitä, millainen tietorakenne oikeasti on. Tätä varten Clojuressa on *record*. Recordeja esitellään *defrecord*-formilla, joka luo Java-luokan määritellyllä konstruktorilla. Määritelty luokka tuodaan käyttäjän sen hetkiseen nimiavaruuteen (engl. namespace). (Fogus ym. 2014, Luku 9.3.1)

Recordin määrittelyä voidaan käyttää dokumentoinnin apuna. Sillä kuvataan, millainen määriteltävä map oikeasti on ja millaiset sen arvot tulisi olla. Tämän lisäksi recordin määrittely parantaa ohjelman suorituskykyä. Esimääritelty olio luodaan nopeammin, se varaa vähemmän muistia ja avaimella arvon etsiminen on nopeampaa kuin vastaavalla mapilla. Record pystyy tallentamaan myös primitiiviarvoja, jotka varaavat selvästi vähemmän muistia kuin luokka-versiot kyseisistä arvoista (engl. boxed primitives). (Fogus ym. 2014, Luku 9.3.1)

Vaikka recordit käyttäytyvät ja vaikuttavat samanlaisilta kuin tavalliset mapit, on niissä kuitenkin joitain eroja. Recordeja ei nimittäin voida käyttää funktioina. Tämän lisäksi vertailuoperaatioissa record ei ole koskaan vastaava, kuin map, vaikka näillä olisi tismalleen samat avain-arvo-parit. (Fogus ym. 2014, Luku 9.3.1)

4.5 Protokollat Clojuressa

Protokollat ovat kokoelma funktiomääritelmiä (engl. function signature). Jokaisella funktiolla on vähintään yksi parametri ja sille annetaan kollektiivinen nimi. Tämä ensimmäinen parametri viittaa kohteena olevaan olioon. Luokan, joka toteuttaa määritellyn protokollan, tulisi sisältää jokaisen protokollassa määritellyn funktion. Protokollat vastaavatkin käyttökohteena osittain Javan rajapintoja (engl. interface) ja puhtaita virtuaalisia luokkia C++:ssa. Clojuren protokollat toimivat dynaamisesti, poiketen mainituista verrokeista. Molemmat Javan rajapinnat ja C++:n luokat vaativat, että toteuttavan luokan kaikki rajapinnat tai luokat, joita se toteuttaa, tulee olla esitelty määrittelyvaiheessa. Clojuren protokollien avulla voidaan esittely tehdä päinvastaisessa järjestyksessä. Protokolla ja sen toteuttaminen voidaan määritellä myöhemmin. Clojuren polymorfismi pohjautuukin näihin protokolla-funktioihin eikä luokkiin. (Fogus ym. 2014, Luku 9.3.2)

Protokollan toteutus määritellään *extend*-formeilla, joihin kuuluvat *extend*, *extend-type* ja *extend-protocol*. Kaikki kolme tekevät loppujen lopuksi saman asian, mutta *extend-type*- ja *extend-protocol*-makroja käytetään, kun määritetään annetulle tyypille useampi funktio. Protokollan toteutus voidaan määritellä luokalle, rajapinnalle, recordille ja jopa nil-tyypille. Viimeisestä ominaisuudesta on hyötyä virheenkäsittelyn toteuttamisessa. (Fogus ym. 2014, Luku 9.3.2)

4.6 Clojuren tietokantayhteydet

Javalla ja relaatiotietokannoilla on pitkä yhteinen historia. Tietokantayhteyksiä varten Javaan on kehitetty JDBC (Java Database Connectivity). Koska Clojure on kehitetty JVM:n päällä toimivaksi, voidaan sen kanssa hyödyntää JDBC:a ja sen monipuolisia ominaisuuksia. Cloju-ressa on kirjasto *clojure.java.jdbc*, joka toimii kevyenä kerroksena Clojuren ja JDBC:n välillä. Tämän lisäksi Clojurelle löytyy useita eri kirjastoja, joilla tietokantayhteyksiä voi hallita. Jos valikoimasta ei sopivaa löydy, niin aina on mahdollisuus käyttää suoraan Javan kirjastoja. Vaikka valitaankin tietokantayhteyden hallitsemiseen jokin Clojuren tai Javan kirjasto, on niidenkin pohjalla aina JDBC. Sille löytyy suoraan satojen eri tietokantojen ajureita. Lähes kaikki tietokantatyypit on tuettu. (Emerick, Carper ja Grand 2012, Luku 14)

Clojuressa JDBC:n avulla tietokannan käyttäminen on luonnollista. Taulut ja niiden sarakkeet ovat avainsanoja, tietokannan hallintaan käytetään map-tietorakenteita, data laitetaan tietokantaan map-tietorakenteena sekä kyselyt palauttavat jonon map-tietorakenteita. Kyselyiden tulokset, esimerkiksi *with-query-results*-makroa käytettäessä, ovat laiskoja. Tämän ansiosta voidaan hakea suuriakin datamääriä kerralla niitä kuitenkaan lataamatta kokonaan muistiin. (Emerick, Carper ja Grand 2012, Luku 14)

Relaatiotietokannat eivät ole ainoa vaihtoehto datan tallentamiseen Clojure-ohjelmointikieltä käytettäessä. Tätä vaihtoehtoista joukkoa kutsutaankin yleisesti ei-relaatio- eli *NoSQL*-tietokannoiksi. Nykyisin NoSQL-tietokantojen valikoima on laaja. Tämä joukko voidaan jakaa tiedon tallennustavan suhteen kolmeen eri ryhmään: avain-arvo-parit, kolumnit ja dokumentit. (Emerick, Carper ja Grand 2012, Luku 15)

4.7 NoSQL ja relaatiotietokannat

Relaatiotietokannat ovat käytännössä hallinneet markkinoita jo yli neljäkymmentä vuotta. Niiden toimintaperiaate, joka koostuu atomisuudesta, johdonmukaisuudesta, eristyksestä ja kestävyydestä, on sopinut lähes kaikkeen datan tallennukseen. Nykypäivänä kuitenkin skaalautuvat web-sovellukset ovat synnyttäneet tarpeen, johon vastaavat NoSQL-tietokannat. Aluksi nimensä mukaisesti NoSQL-tietokannat eivät tukeneet ollenkaan relaatioita, mutta nyky-

ään NoSQL-lyhenteen sanotaan olevan “Not Only SQL”, eli mahdollisuus relaatioille jätetään toteutukseen. (Lourenço ym. 2015)

Valinta relaatio- ja NoSQL-tietokannan välillä on haastava. Usein ei ole selvää, kumpi näistä kahdesta olisi sopiva tiedon tallentamiseen kehitettävässä sovelluksessa. NoSQL-tietokannan valinta mahdollistaa skaalautuvuuden, joka saavutetaan suorituskyvyn kustannuksella. NoSQL-tietokannat ovat kasvavassa määrin löytämässä tiensä myös yritystason sovelluksiin. Kirjallisuutta ja tutkimustuloksia niistä on kuitenkin vielä huonosti saatavilla, joten NoSQL-tietokantoja pidetään kehitysasteella olevana ratkaisuna. (Lourenço ym. 2015)

4.8 Datomic

Clojuren kehittäjätiimin luomus Datomic on uudenlainen tietokanta. Se on kehitetty pilvipalveluita silmällä pitäen. Datomic on suunniteltu helpoksi ja joustavaksi käyttää. Datomicin toimintaperiaate keskittyy muuttumattoman datan, arvokeskeisyyden ja joustavuuden ympärille. Data on Datomicin, kuten Clojurenkin, keskipisteessä. Datomicia ei voi suoranaisesti luokitella yhdellä tietokantatyypillä. Se sisältää piirteitä niin NoSQL-, hajautetuista, loogisista ja graafitietokannoista sekä transaktio- ja analyysisäilöistä (engl. transaction store ja analytics store). Vaikka Datomic on saanut inspiraationsa useista eri lähteistä, on sen perusperiaate ja tavoite olla suoraan vaihtoehto ja korvike relaatiotietokannoille. Datomic on suljetun lähdekoodin kaupallinen kirjasto, mutta siitä on saatavilla myös ilmainen rajoitettu versio. (Pratley 2016, Luku 6)

Datomicin kyselyt muodostetaan logiikkaohjelmointikieli Datalogia käyttäen. Datan kanssa vuorovaikutus tapahtuu kokonaisuuskuvauksia (engl. entity maps) käyttämällä. Tieto kirjoitetaan transaktioina, jotka muodostuvat tiettyjen sääntöjen alaisista tietorakenteista. Datomic tallentaa tietoa kokoelmina, joita yhdistää indeksit. Poiketen relaatiotietokannoista Datomicissa indeksit sisältävät kaiken tiedon ja ne ovatkin ainoa esitysmalli kaikelle järjestelmään tallennetulle datalle. Tämä mahdollistaa nopean etsimisen malleilla (engl. pattern) sekä joustavan ja tehokkaan kyselyjärjestelmän. Datomic tallentaa tiedon neljään indeksiin sekä transaktiologiin. Jokainen indeksi on järjestetty ja tallennettu puumallin esitysmuotoon, kukin eri järjestykseen. Datomicissa on myös välimuistitus, mikä sopii hyvin muuttumattoman

datan käsittelyyn, koska dataa ei tarvitse koskaan mitätöidä. Datomic koostuukin neljästä kerroksesta: luku, kirjoitus, säilytys ja välimuisti. (Pratley 2016, Luku 6)

Datomicissa on helppokäyttöinen, joustava ja tietomallia hyvin kuvaava skeema-järjestelmä. Näitä skeemoja käytetään sovelluksen tallennettavan tietorakenteen määrittämiseen. Datomicissa yksittäiseen kokonaisuuteen (engl. entity) viitataan tunnistemääritteillä (engl. identity attributes). Tunnistemääritteet ovat tarkoitettu ulkoisiksi avaimiksi yksittäisille kokonaisuuksille. Datomicissa kokonaisuuksien liittäminen yhteen, kuten monesta moneen -suhteet (engl. many-to-many), on toteutettu viitteillä kokonaisuuksien välillä sekä niin sanotulla vanhempi-komponentti-suhteilla. Relaatiotietokannoissa käytettyjä liittämislausekkeita (engl. join) tai erillisiä tauluja ei tarvita. (Pratley 2016, Luku 6)

4.9 Clojure.spec

Staattisesti tyypitetyissä kielissä, kuten Javassa, ohjelmakoodi rakentuu määrätyistä tietorakenteista. Se helpottaa selvästi ohjelmakoodin lukua. Dynaamisesti tyypitetyillä kielillä, kuten Clojure, ei ole tätä etua. (Miller, Halloway ja Bedra 2018, Luku 5)

Clojuren versio 1.9 toi mukanaan *clojure.spec*-kirjaston. Clojure.spec mahdollistaa funktioiden ja datan rakenteen kuvaamisen. Näitä kutsutaan englanniksi nimellä “specs”. Näiden specien, tai suomeksi määritteiden, kanssa itse ohjelmakoodi ei ole tavallisesta poikkeava. Sen sijaan määritteet tarjoavat ohjelmakoodin rinnalla tarkan kuvauksen siitä, millainen jokin tietorakenne on, millaiset ovat funktion parametrit tai mitä funktio palauttaa. Määritteet eivät kerro pelkästään siitä, mitä tyyppiä jokin yksittäinen arvo on tai mikä on tietorakenteen arvojen tyyppi. Nämä määritteet voivat kuvata esimerkiksi millainen jokin arvo voi olla tai mitä se ei ole. Esimerkiksi arvon määrite voi olla parillinen kokonaisluku, joka on alle 100 tai se voi olla vektori, joka sisältää merkkijonoja tai nil-arvoja. Määritteet eivät toimi pelkästään ohjelmakoodin dokumentaationa. Ne mahdollistavat datan validoinnin, esimerkkidatan generoinnin ja automaattisesti generoitavat testit. Määritteet luetaan ja suoritetaan ajonaikaisesti. (Miller, Halloway ja Bedra 2018, Luku 5)

Tavalliset yksikkötestit perustuvat kehittäjän tuottamiin esimerkkisyötteisiin ja arvojen vertailuun. Generoiva testaus taas tuottaa tuhansia satunnaisia syötteitä ja kutsuu testattavaa

funktiota syötteillä sekä lopuksi varmistaa, että lopputulos on määrätynlainen. Clojure.spec-kirjasto pystyy suorittamaan generoivia testejä, jotka perustuvat funktion määritteisiin. Testit ajetaan *test.check*-kirjaston avulla. Jos generoivissa testeissä tapahtuu virhe, käytettävä *test.check*-kirjasto rajaa testitapaukset mahdollisimman yksinkertaiseen syötesarjaan, jolla virhe tapahtuu. Tämän lisäksi käyttäjälle annetaan generoitujen arvojen siemen (engl. seed), jonka avulla hän voi tarvittaessa toistaa testit. Arvojen automaattisen generoinnin lisäksi voidaan käyttää kustomoitua generaattoria. Tämä ominaisuus tulee tarpeeseen, jos automaattinen generaattori ei ole riittävä tai se tuottaa epäsopivia arvoja. (Miller, Halloway ja Bedra 2018, Luku 5)

5 Clojuren viitteet tietokantayhteytenä

5.1 Tavoitteet

Olen huomannut, että usein sovelluskehitysprojektien niin sanotun POC (Proof of Concept) -vaiheen tiedon tallentamisratkaisu jätetään kokonaan pois tai siihen käytetään paljon aikaa. Esimerkiksi relaatiotietokantojen tapauksessa tietokantarakenne tulee miettiä jossain määrin etukäteen ja luoda sopivaksi. Tietokantarakenteen muutokset hallitaan usein manuaalisesti luotavilla migraatioilla. Tämän lisäksi kehittäjän tulee luoda tietokantakutsut etukäteen ja ylläpitää niitä sovelluskehityksen edetessä.

Itse olen Clojure-projekteissa työskennellessä pitänyt esimerkiksi Atomin helppokäyttöisyydestä, kun tarvitaan synkronista tiedonsiirtoa eri säikeiden välillä tai sovelluksen tilan tallentamista. Tavoitteenani on kehittää mahdollisimman helppokäyttöinen tietokantayhteys POC-vaiheen sovelluskehitykseen. Atomisen viitteen, kuten Clojuren Atomin, toimintamalli vaikuttaisi sopivan tarkoitukseen varsin hyvin, kuten luvusta 3.5 käy ilmi. Olen todennut sen helppokäyttöiseksi ja nopeaksi ottaa käyttöön.

Tämän tutkielman tavoitteena onkin kehittää kirjasto, jonka käyttö on mahdollisimman yksinkertaista sekä käyttöönotto on nopeaa ja helppoa. Tietokannan rakenteen määritys sekä kehityksen edetessä sen päivitys tapahtuu täysin automaattisesti. Lisäksi käyttäjällä on mahdollisuus valita, säilytetäänkö historiadata, eli logi, vai pidetäänkö tallessa ainoastaan Atomin viimeisin tila. Kun arvo päivitetään Atomiin onnistuneesti, tallennetaan tilanne tietokantaan automaattisesti. Kun Atom alustetaan, esimerkiksi sovelluksen käynnistyessä, ladataan Atomin viimeinen tila tietokannasta.

5.2 Mahdolliset hyödyt ja haasteet

Clojuren viitteiden, kuten Atomin, helppokäyttöisyyden hyödyntäminen lienee suurin etu verrattuna perinteiseen tietokantayhteyden rakentamiseen sovelluskehitysprojektissa. Tämä pienentää oppimiskäyrää, koska tietokantayhteys toimii kuten Clojuren tietorakenne. Kirjaston käyttö on tällöin mahdollisimman intuitiivinen. Mahdollista aiempaa osaamista Atomin

käytöstä voi suoraan hyödyntää kirjaston kanssa. Nämä nopeuttavat sovelluskehitysprojektin ensimmäisen vaiheen käynnistymistä sekä sen etenemistä. Myös kehittäjän aikaa säästynee runsaasti, kun kirjasto hallitsee kehittäjän puolesta tietokantamääritykset ja -migraatiot sekä kyselylausekkeet.

Tämän tutkielman puitteissa kehitetty kirjasto ei todennäköisesti sovi tuotantokäyttöön. Huolimatta toteutettavista optimoinneista, ei sen tehokkuus tule vastaamaan suoraa tietokantayhteyttä. Taustalla toimivan tietokannan suorituskykyä ei pystytä täysin hyödyntämään, koska tallennettavat tietorakenteet luodaan dynaamisesti. Myös tietynlaisen tiedon hakeminen kyselyllä tai datan suodattaminen voidaan suorittaa ainoastaan ohjelmallisesti sovelluksen muistissa ja käytettävän viitteen rajojen puitteissa, koska Atom ei ole suunniteltu niin, että tietoa haetaan kyselyillä. Taustatietokantaa käytetään pääasiassa tiedon pysyvään tallennukseen. Myös muistinkäyttö on suurempaa, koska sovelluksen tila säilytetään tietokannan lisäksi myös muistissa.

5.3 Käytettävät työkalut

Kehitysympäristönä käytetään Clojuren 1.9.0 versiota ja käännös- ja riippuvuuksienhallintatyökaluna *Leiningenin* 2.8.1 versiota. Olen todennut, että käytössä Leiningen on varsin oiva työkalu Clojuren kääntämiseen sekä riippuvuuksien hallintaan. Leiningenillä voidaan esimerkiksi luoda uusi projekti, ajaa testejä tai vaikkapa suorittaa haluttuja komentoja. Se lataa myös projektin riippuvuudet automaattisesti. Yksikkötesteissä käytetään Clojuren mukana tulevaa *clojure.test*-kirjastoa ja Clojuren spec-kirjaston 0.1.143 alfa-versiota. Tietokantakutsujen ja kehitettävän kirjaston suorituskykyeroa mitataan Clojuren core-kirjaston *time*-makron muunnoksella. Normaali *time*-makro tulostaa kuluneen ajan millisekunteina. Testejä varten luodaan oma muunnelma, joka tulostamisen sijaan palauttaa kuluneen ajan.

Vertailuprojektien muutoksia tarkastellaan sekä koodirivien määrän erolla että koodin toteutuksen yksinkertaisuudella. Rivien laskentaan käytetään *Cloc*-ohjelman versionumeroa 1.72.

Suorituskykytestit suoritetaan Dell XPS 15 9560 kannettavalla tietokoneella, jossa on Intel Core i7-7700HQ suoritin ja 16Gb keskusmuistia. Käyttöjärjestelmä on 64 bittinen Fedora 28

kernelin versiolla 4.17.2-200.fc28.x86_64. Testit ajetaan virtalähde kytkettynä, joten suorittimen energiansäästöt eivät ole päällä.

5.4 Kirjaston testaus

Kirjaston funktioille kirjoitetaan Clojuren spec-määritteet. Näin saadaan luotua dokumentaatio ja hyödynnettyä ajonaikaista tarkistusta. Myös automaattiset generoidut testit olisi mahdollista luoda, mutta tämän tutkielman puitteissa se ei ole mahdollista, joten jätetään se jatkokehitysvaiheeseen. Määritteiden lisäksi luodaan tavalliset yksikkötestit.

Testejä varten tarvitaan erillinen testitietokanta, jotta tuotantodata ei häiriinny testeistä. Suorituskykytestausta varten luodaan sarja operaatioita, jotka suoritetaan kehitettävän kirjaston tietorakennetta hyödyntäen sekä suoralla tietokantayhteydellä. Näissä testitapaukset ovat samankaltaisten arvojen tallennusta tietokantaan. Suorituskykytesteissä funktion suoritusajat kellotetaan ja tuloksien vertailusta tulostetaan yhteenveto.

6 Kirjaston toteutus

6.1 Vertailuprojektien valinta

Ennen kirjaston kehityksen aloittamista, etsitään vertailuprojektit. Projektien avulla tutkitaan kirjaston käyttöä, suorituskykyä ja vaikutuksia koodipohjaan. Valitaan kolme erilaista Clojure-sovellusprojektia, joissa käytetään relaatiotietokantaa tilan tallentamiseen. Valintaprosessissa käytiin läpi noin 1500 avoimen lähdekoodin projektia Github-verkkosivustolla. Ihannetilanteessa valituissa projekteissa olisi kattavat yksikkötestit ja hyvä dokumentaatio. Näin kehitettävän kirjaston lisääminen ja tietokantayhteyden korvaaminen olisi helppoa. Testattuja ja kriteerit täyttäviä projekteja ei kuitenkaan löytynyt, kuin yksi. Valitut projektit esitellään lyhyesti taulukossa 6.

Projektit valitaan niin, että niiden kehityksen prototyyppivaiheessa voisi olla järkevää käyttää kehitettävää kirjastoa. Esimerkiksi runsaasti dataa tallentavan tai analysoivan sovelluksen tilan tallennukseen tuskin olisi edes prototyyppivaiheessa järkevää käyttää viitettä. Sen hyödyt jäisivät auttamatta minimaalisiksi, koska suorituskyky ja tiedon hallittavuus ovat tärkeät osat sovelluksen toimintaa.

Projektien muokkausta varten luodaan jokaisesta projektista haarauma (engl. fork) Github-käyttäjätilille. Näin säilytetään tutkielmassa tehdyt muutokset versiohallinnassa vaikka alkuperäinen projekti jostain syystä poistetaan. Myös vertaus alkuperäiseen toteutukseen on helpompaa, koska git-versiohallinta ja Github tarjoavat tähän valmiit työkalut.

Projekti	Kuvaus	Kehittäjän tunnus
Prototyping blog engine	Yksinkertainen blog-moottori, joka hyödyntää Clojuren Compojure- ja ring-kirjastoja.	Keto256 (2018)
address-book	Esimerkkisovellus Clojuren ring ja Compojure -kirjastojen käyttämiseen.	JarrodCTaylor (2018)
guestbook	Luminus-nimisen mikrosovelluskehyksen esimerkkiprojekti.	Yogthos ja Paulrd (2018)

Taulukko 6. Vertailuprojektit

6.2 Projektin luonti

Uuden projektin luomiseen käytetään Leiningeniä. Oletuksena projekti luodaan kirjasto-sapluunaa käyttäen. Leiningen lataa sapluunan, luo projektin ja päivittää tarvittavat arvot tiedostoihin ja kansiorakenteisiin. Nimiavaruudeksi tulee projektin nimi. Projektin nimeksi valitaan *clj-wiite*. Nimen alkuosa viittaa siihen, että kyseessä on Clojure-kirjasto. Leiningenin antama tuloste uuden projektin luomisesta on listauksessa 6.1 ja sen projektirakenne listauksessa 6.2.

```
$ lein new clj-wiite
Generating a project called clj-wiite based on the 'default' template.
The default template is intended for library projects, not applications.
To see other templates (app, plugin, etc), try `lein help new`.
```

Listaus 6.1. Projektin luonti Leiningenillä

```
-rw-r--r--  1   772 Apr 25 20:50 CHANGELOG.md
drwxr-xr-x  2  4096 Apr 25 20:50 doc
-rw-r--r--  1    99 Apr 25 20:50 .gitignore
-rw-r--r--  1   122 Apr 25 20:50 .hgignore
-rw-r--r--  1 11219 Apr 25 20:50 LICENSE
-rw-r--r--  1   269 Apr 25 20:50 project.clj
-rw-r--r--  1   235 Apr 25 20:50 README.md
drwxr-xr-x  2  4096 Apr 25 20:50 resources
drwxr-xr-x  3  4096 Apr 25 20:50 src
drwxr-xr-x  3  4096 Apr 25 20:50 test
```

Listaus 6.2. Sapluunan luoma projekti

Ensimmäiseksi on syytä päivittää Clojuren versio. 25.4.2018 ladatussa sapluunassa on käytetty vielä Clojuren 1.8.0 versiota. Projektin riippuvuudet, kuten Clojuren versio, on määritelty *project.clj*-tiedostossa. Samaisessa tiedostossa on määritelty myös muut projektin tiedot. Muutetaan versioksi 1.9.0, joka on viimeisin vakaa Clojuren versio. Clojure.spec-kirjasto vaatii sen toimiakseen. Samalla päivitetään projektin versio, kuvaus, verkkosivun osoite sekä lisenssi. Kirjaston sapluunassa on oletusarvona Eclipse Public -lisenssi, jonka alla myös Clojure on julkaistu. Vaihdetaan se MIT lisenssiksi, koska kyseinen lisenssi on sallivampi ja mielestäni selkeämpi. Päivitetyt projektimääritykset näkyvät listauksessa 6.3.

```
(defproject clj-wiite "0.1.0"
  :description "Clojure library for using database backed Atom"
  :url "https://github.com/stormaaja/clj-wiite"
  :license {:name "MIT"
            :url "https://opensource.org/licenses/MIT"}
  :dependencies [[org.clojure/clojure "1.9.0"]])
```

Listaus 6.3. Projektin project.clj-tiedosto

Seuraavaksi tyhjennetään esimerkkifunktion leiningenin luomasta *core.clj*-kooditiedostosta. Se sijaitsee *src*-koodikansiossa kirjaston nimiavaruuden mukaan nimetyssä alikansiossa. Kuten tiedoston nimestäkin voi päätellä, kyseiseen tiedostoon lisätään kirjaston ydinfunktiot. Tässä vaiheessa ohitetaan vielä generoidut *test* ja *resources* -kansiot sekä *CHANGELOG.md*-tiedoston, koska näitä ei vielä projektin alkuvaiheessa tarvita. Projektin dokumentaatio sijaitsee *doc*-kansiossa. Sapluuna on luonut *intro.md*-tiedoston, josta poistetaan esimerkkirivi. Nyt projekti on valmis kehityksen aloittamiseen.

6.3 Tiedon tallennus

Luodaan ensimmäiseksi tiedon tallennukseen protokolla (listaus 6.4). Protokolla helpottaa tulevaa kehitystä ja testausta. Sen avulla voidaan toteuttaa erilaisia rajapintoja tallennukseen. Näistä esimerkkinä ovat tutkielmassakin toteutetut relaatiotietokanta- tai tiedostotallennusrajapinnat. Protokollan avulla voidaan tallennuskohde vaihtaa vaikuttamatta muun sovelluksen toimintaan. Nimetään protokolla *Storeksi*. Luodaan Storelle kaksi funktiota, *write-state!* ja *load-state*. Funktiossa *write-state!* huutomerkki on konventio, joka kertoo, että funktio muuttaa ohjelman tilaa. Kuten luvussa 4.5 kerrotaan, on protokollan funktion ensimmäinen parametri aina ilmentymä itse, eli tässä tapauksessa store. Luodaan lopuksi funktio *store?*, jolla voi testata, toteuttaako annettu ilmentymä Store-protokollan. Tätä funktiota voidaan käyttää myös spec-määritteiden kanssa.

Virheenkäsittelyä voisi tässä vaiheessa myös suunnitella, mutta esimerkiksi prototyyppisovelluksissa mielestäni ohjelmointikielen oletuspoikkeukset riittävät hyvin. Tuotantokäyttöön suunnatussa kirjastossa toteuttaisin yhtenevät, esimerkiksi Storen määrittelemät poikkeukset tai muun yhtenäisen virheenkäsittelyn. Muuten jokainen Store saattaa tuottaa erilaisia poikkeuksia riippuen niiden lopputallennuspaikasta.

```clojure
(ns clj-wiite.store)

(defprotocol Store
  (write-state! [store state] "Write state to store")
  (load-state [store] "Load state from store"))

(defn
  ^{:doc "Function for checking if given instance implements Store protocol"
    :added "0.1.0"}
  store? [x]
  (satisfies? Store x))
```

Listaus 6.4. Store-protokolla store.clj

Luodaan ensimmäiseksi tiedostoon tallentava Store (listaus 6.5). Nimetään se *FileStoreksi*. FileStore on tässä tapauksessa record ja se toteuttaa Store-protokollan, kuten luvussa 4.4 esitellään. Näin FileStorea voidaan käyttää Storen esittelemillä funktioilla. Luodaan ensimmäiseksi funktio, jolla voidaan helposti testata, onko tiedosto olemassa. Annetaan sille nimeksi *file-exists?* ja merkitään se privaattifunktioksi. Kysymysmerkki funktion nimessä on konventio, jolla ilmaistaan, että funktio palauttaa totuusarvon. Privaattifunktio luodaan *defn-*-makrolla. Seuraavaksi luodaan record, joka toteuttaa Store-protokollan. Kirjoitetaan Store-protokollan *write-state!*- sekä *load-state*-funktiot. Käytetään tiedoston lukuun ja kirjoitukseen Clojuren funktioita *spit* ja *slurp*. Lisätään tilan latausvaiheeseen tiedoston olemassaolon tarkistus aiemmin luotua *file-exists?*-funktiota hyödyntäen. Kuten aiemmin on kuvattu, jätetään virheenkäsittely tiedoston olemassaolon tarkistukseen. Lisätään lopuksi *file-store*-funktio helpottamaan recordin käyttöä. Funktiolla voidaan luoda file-store antamalla tiedostonnimi. Lisätään lopuksi funktio *file-store?*, jolla voidaan testata, onko ilmentymä FileStore. Tätä funktiota voidaan käyttää myös spec-määritteiden kanssa.

Seuraavaksi lisätään *specit* eli määritteet funktioille. Tässä vaiheessa määritteet toimivat dokumentaation apuna. Määritteissä kerrotaan, mitä funktiot ottavat sisään ja mitä ne palauttavat. Kuten Miller, Halloway ja Bedra (2018, Luku 5) kirjoittavat, funktion parametrit määritellään *args*-avainsanalla ja paluuarvo *ret*-avainsanalla. Parametrit yhdistetään vastaamaan tyyppiään *cat*-makrolla.

```clojure
(ns clj-wiite.file-store
  (:require [clojure.spec.alpha :as s]
            [clojure.edn :as edn]
            [clojure.java.io :as io]
            [clj-wiite.store :refer [Store]]))

(s/fdef file-exists?
        :args (s/cat :path string?)
        :ret boolean?)

(defn- file-exists? [path]
  (.exists (io/as-file path)))

(defn
  ^{:doc "Function for checking if given instance is FileStore"
    :added "0.1.0"}
  file-store? [x]
  (instance? FileStore x))

(defrecord
  ^{:doc "Store for keeping state in a given file.
          File doesn't have to exist. If it does, it will be overwritten.
          If file does not exist, load-state will return nil."
    :added "0.1.0"}
  FileStore [file]
  Store
  (write-state! [store state]
    (spit (:file store) (str state)))
  (load-state [store]
    (when (file-exists? (:file store))
      (let [v (slurp (:file store))]
        (edn/read-string v)))))

(s/fdef file-store
        :args (s/cat :file string?)
        :ret #(instance? FileStore %))

(defn
  ^{:doc "Store for keeping state in a given file.
          File doesn't have to exist. If it does, it will be overwritten."
    :added "0.1.0"}
  file-store [file]
  (FileStore. file))
```

Listaus 6.5. FileStore file_store.clj

Seuraavaksi luodaan yksikkötestit FileStorelle (listaus 6.6). Luodaan tilan kirjoitukseen ja lukuun perinteiset yksikkötestit. Koska kyseessä on tiedostojärjestelmää käyttävät, eli tilaa muuttavat funktiot, on tässä vaiheessa helpompi testata sovellusta yksikkötesteillä. Testejä varten luodaan yksi satunnaisen tiedoston generointifunktio, jolla voidaan molemmissa testitapauksissa luoda väliaikainen tiedosto järjestelmän väliaikaiskansioon. Kirjoitetaan lisäksi makro *with-store*, jolla luodaan väliaikaistiedosto ja store sekä poistetaan käytetty tiedosto testin lopuksi. Makron avulla vältytään turhalta toistolta.

Kirjoitusfunktion testauksessa varmistetaan, että tiedosto todellakin tallentuu tiedostojärjestelmään. Aluksi varmistetaan, että tiedostoa ei ole olemassa. Tarkistuksen jälkeen kirjoitetaan esimerkkidata ja varmistetaan, että luotu tiedosto löytyy tiedostojärjestelmästä. Lukufunktio testataan kirjoittamalla kaksi erilaista tilaa tiedostojärjestelmään ja varmistamalla, että kirjoituksen jälkeen kirjoitettu tila on sama kuin kirjoitettava tila. Kahdella eri tilalla varmistutaan siitä, että tila tosiaankin päivittyy tiedostoon.

```
(ns clj-wiite.file-store-test
  (:require [clojure.test :refer :all]
            [clj-wiite.file-store :refer :all]
            [clj-wiite.store :refer :all]
            [clojure.java.io :as io]))

(def example-data {:some "String" :num 2 })

(defn random-file []
  (io/file (System/getProperty "java.io.tmpdir")
           (str (java.util.UUID/randomUUID))))

(defmacro with-store [file store & body]
  `(let [~file (random-file)
         ~store (file-store (.getAbsolutePath ~file))]
     (when (.exists ~file) (io/delete-file ~file))
     (do ~@body)
     (when (.exists ~file) (io/delete-file ~file))))

(deftest file-store-write-state
  (testing "Writing store state"
    (with-store file store
      (is (not (.exists file)))
```

```
24        (write-state! store example-data)
25        (is (.exists file)))))
26
27  (deftest file-store-load-state
28    (testing "Loading store state"
29      (with-store file store
30        (is (nil? (load-state store)))
31        (write-state! store example-data)
32        (is (= (load-state store) example-data))
33        (let [changed-state (assoc example-data :some "String II" :num 3)]
34          (write-state! store changed-state)
35          (is (= (load-state store) changed-state))))))
```

Listaus 6.6. FileStoren yksikkötestit file_store_test.clj

6.4 Persistentin tietorakenteen toteutus

Yksinkertaisin ratkaisu persistentin tietorakenteen toteutukseen on luoda Clojuren Atom ja lisätä sille valvonta-funktio. Luvussa 3.3 esitellään Atom ja sen valvontafunktion käyttö. Tämän ratkaisun ongelma on se, että käyttäjä voi poistaa valvontafunktion, jolloin kirjasto ei enää toimi oikein. Valvontafunktion poisto voidaan estää luomalla oma tietorakenne, joka toteuttaa Atomin vaatimat rajapinnat. Tällöin se toimii kuten Clojuren Atom, ja estää *remove-watch*-funktiossa sisäänrakennetun valvontafunktion poiston. Atomia käytettäessä tärkeimmät rajapinnat lienevät IAtom, IDeref, IReset, ISwap ja IWatchable.

Yksinkertaisessa ratkaisussa käyttäjä ei voi asettaa alkutilaa tai määrittää, ladataanko alkutila storesta vai ei. Jos nämä mahdollisuudet lisätään tietorakenteelle, tulisi valvontafunktio luoda esimerkiksi korkeamman kertaluvun funktiona tai säilyttää funktioon viittaus, jotta sitä voidaan kutsua alkutilaa asetettaessa. Tällöin myös alkutilan lataus pitäisi olla mahdollista ohittaa, jotta tietokannassa oleva tila ei ylikirjoita tietorakenteen tilaa.

Tämän tutkielman puitteissa ei ole kuitenkaan olennaista, vaikka käyttäjä voi poistaa valvontafunktion, joten toteutetaan tietorakenne yksinkertaisella tavalla. Valitaan tietorakenteen nimeksi *Watom*. Se muistuttaa atomista, mutta mukana on viittaus toteutettavan kirjaston nimeen. Luodaan aluksi funktio, joka alustaa atomin, lataa storesta alkutilan sekä asettaa

valvontafunktion. Valvontafunktio on yksinkertaisessa mallissa lambda-funktio. Valvontafunktiolle tulee antaa avain, jolla funktio tunnistetaan. Määritetään se erikseen, jotta arvoa voidaan muuttaa tarvittaessa helposti. Kirjoitetaan lopuksi funktiolle spec-määritteet ja dokumentaatio. Toteutus esitellään listauksessa 6.7

```
(ns clj-wiite.watom
  (:require [clj-wiite.store :refer :all]
            [clojure.spec.alpha :as s])
  (:import [clj-wiite.store.Store]))

(def watom-key :watom)

(s/def ::store store?)

(s/fdef create-watom
        :args (s/cat :store ::store)
        :ret #(instance? clojure.lang.Atom %))

(defn
  ^{:doc "Create Clojure Atom with watcher for writing state to store.
          Initial state will be loaded from store as well."
    :added "0.1.0"}
  create-watom [store]
  (let [a (atom (load-state store))]
    (add-watch
      a watom-key
      (fn [k a old-state new-state]
        (write-state! store new-state)))
    a))
```

Listaus 6.7. Atomin kapsulointi watom.clj

6.5 Tietokantayhteyden luominen

Aluksi lisätään tarvittavat riippuvuudet projektiin. Tietokantayhteyksissä käytetään Clojuren JDBC-kirjastoa ja ensimmäisessä versiossa PostgreSQL-tietokannan versiota 10. Kirjaston ensimmäisessä yksinkertaisessa toteutuksessa ei tulla hyödyntämään esimerkiksi relaatiotietokannan etuja, joten tallennetaan tieto suoraan JSON-muodossa (JavaScript Object No-

tation). Tämä ratkaisu ei vaadi erityistä käsittelyä konversiota lukuun ottamatta, joten sen toteutus on nopea ja riittävä tämän tutkielman puitteissa. Tarvitaankin datan konvertointi EDN-notaatiosta (Extensible Data Notation) JSON-notaatioon. Tähän käytetään *data.json*-kirjastoa. Lisätään riippuvuudet *project.clj*-tiedostoon.

Luodaan seuraavaksi aputiedosto PostgreSQL-yhteydelle 6.8. Koska JDBC:n PostgreSQL-ajuri ei tue Clojuren tietorakenteiden suoraa konvertointia JSON-formaattiin, lisätään ne laajentamalla JDBC:n protokollia. Luodaan aluksi funktio, joka konvertoi arvon JSON-muotoon ja asettaa sen PostgreSQL-ajurin käyttämälle PGobjectille. Seuraavaksi laajennetaan JDBC:n ISQLValue-rajapintaa. Annetaan protokollalle Clojuren map- ja vector-tietotyyppien rajapinnat ja määritellään konversio käyttämällä aiemmin luotua funktiota.

Seuraavaksi lisätään kyselytuloksen konvertointi, jos arvo on JSON-formaatissa. Tämä onnistuu laajentamalla JDBC:n IResultSetReadColumn protokollaa ja määrittämällä sille konversio, kun kolumni on PGobject-tyyppiä ja sen arvon tyyppi on JSON. Konversio tapahtuu toteuttamalla *result-set-read-column*-funktion, jossa tyypin tarkastus tapahtuu ja mahdollinen konversio toteutetaan.

```
(ns clj-wiite.postgresql
  (:require [clojure.java.jdbc :as jdbc]
            [clojure.data.json :as json])
  (:import [org.postgresql.util PGobject]))

(extend-protocol jdbc/IResultSetReadColumn
  PGobject
  (result-set-read-column [pgobj metadata idx]
    (let [type (.getType pgobj)
          value (.getValue pgobj)]
      (if (= type "json")
        (json/read-str value :key-fn keyword)
        value))))

(defn value-to-json-pgobject [value]
  (doto (PGobject.)
    (.setType "json")
    (.setValue (json/write-str value))))

(extend-protocol jdbc/ISQLValue
```

```
21  clojure.lang.IPersistentMap
22  (sql-value [value] (value-to-json-pgobject value))
23
24  clojure.lang.IPersistentVector
25  (sql-value [value] (value-to-json-pgobject value)))
```

Listaus 6.8. Datan konvertointi postgresql.clj

6.6 Tietokantayhteyden hyödyntäminen

Kun tietokanta on valmis datan konvertointiin, luodaan store, joka tallentaa annetun tilan tietokantaan (listaus 6.9). Ensimmäiseksi lisätään *use*-avainsanalla tieto, että käytetään aiemmin luotuja protokollan laajennuksia. Seuraavaksi lisätään kirjastot spec-määritteiden luomista, EDN-tyyppisen tilan konvertointia ja tietokantayhteyden määrittämistä varten. Lisätään viittaus myös Store-protokollaan, jonka nyt luotava tietokanta-store tulee toteuttamaan.

Seuraavaksi luodaan tietokantakyselyt sisältävät funktiot. Ensimmäiseksi on hyvä toteuttaa tarvittavan taulun olemassaolon tarkistusfunktio. Tämä tarkistus tehdään, kun storen ilmentymä luodaan. Koska tutkielmassa käytetty JDBC:n versio 0.7.6 ei tue suoraan taulun olemassaolon tarkistusta, luodaan funktio, joka tekee tyhjän vastauksen palauttavan kyselyn. Nyt voidaan tarkistaa, heittääkö kyselyfunktio poikkeuksen. Varmistetaan *some?*-funktiolla, onko palautuva arvo mahdollisesti jotain. Poikkeus pitää kuitenkin tarkistaa, jotta otetaan kiinni ainoastaan tarvittava poikkeus. PostgreSQL:n ajuri heittää yleisen *PSQLException*-poikkeuksen, jolla on *.getSQLState*-funktio. Kyseisen funktion paluuarvo tarkistetaan. Jos arvo ei ole “taulu ei ole olemassa”-tilaa vastaava, heitetään poikkeus eteenpäin.

Taulun luonti tapahtuu storen ilmentymää luotaessa. Lisätään luontia varten funktio. JDBC-kirjastossa on funktio *create-table-ddl*, joka muodostaa SQL-yhteensopivan lausekkeen taulun luomista varten annetusta DDL-lausekkeesta (engl. Data definition language). Lauseke muodostetaan vektorista vektoreita, joissa määritellään sarakkeen nimi ja tyyppi pareina. Määritetään tilan *state*-sarake JSON-tyyppiseksi. Tilan säästämiseksi voitaisiin käyttää JSONB, eli binääriformaattia, mutta tämän tutkimuksen kannalta se ei ole olennaista. Välitetään *create-table-ddl*-funktion paluuarvo eteenpäin JDBC:n *db-do-commands*-funktiolle.

Seuraavaksi luodaan tilan hallintaan funktiot. JDBC-kirjastossa tietokantakyselyt välitetään tietokantaan *query*-funktiolla. Funktiolle annetaan yhteysmääritteiden lisäksi lausekkeet ja parametrit vektorissa ja valinnaiset määritteet map-tietorakenteessa. Määritteissä voidaan antaa esimerkiksi funktio, jota rivin kohdalla kutsutaan. Näin saadaan kysely palauttamaan ainoastaan haluttu arvo. Kyselyfunktio palauttaa aina kokoelman rivejä, vaikka niitä olisi vain yksi, joten otetaan paluuarvosta ensimmäinen rivi Clojuren *first*-funktiolla. Tilan tallentamiseen käytetään JDBC-kirjaston *insert!*-funktiota. Sille annetaan yhteysmääritteiden lisäksi taulu ja arvo. Kapsuloidaan tallennettava tila JSON-tietorakenteen sisään, koska muuten kirjasto ei tukisi muita tietotyyppejä kuin EDN-tietorakenteita ja kokoelmia. Kapsuloinnille olisi vaihtoehtona arvon tyypin tarkistus ja hajauttaminen sen mukaan esimerkiksi eri tauluihin, mutta se jääköön tutkielman ulkopuolelle jatkokehitykseen. Kirjaston jatkokehitystä käsitellään tarkemmin alaluvussa 7.6 ja optimointia alaluvussa 6.8.

Kun apufunktiot ovat valmiina, luodaan *DBStore* record ja ilmentymän luomista varten apufunktio. Koska DBStore toteuttaa Store-protokollan, tulee sille luoda *write-state!* ja *load-state* -funktiot. Käytännössä nämä funktiot ovat ainoastaan aiemmin tilanhallintaan luotujen funktioiden kutsuja, jotka käyttävät recordille annettua yhteyden määritystä. Ilmentymän luomisen apufunktioon lisätään taulun luonti tarvittaessa.

Luodaan tässä vaiheessa vielä *config*-apuluokka. Se lukee automaattisesti ympäristökohtaiset asetukset, kuten tietokantayhteyden määritykset, EDN-tiedostoista. Tätä hyödynnetään jo seuraavaksi luotavissa integraatiotesteissä.

Lopuksi luodaan spec-määritteet ja integraatiotestit DBStorelle. Yhteysmäärityksen spec luodaan JDBC:n PostgreSQL-ajurin dokumentaation mukaan. Pakollisiksi arvoiksi lisätään tietokannan tyyppi, nimi ja osoite. Valinnaisiksi asetetaan käyttäjänimi, salasana ja salauksen määritykset. Ilmentymän luomisen apufunktion spec-määritteessä kuvataan sekä yhteysmäärityksen spec ja paluuarvona DBStore:n ilmentymä. Integraatiotesteissä luodaan aluksi apumakro, jolla voidaan testin aluksi luoda storen ilmentymä ja tyhjentää tietokantataulu testien jälkeen. Sen jälkeen kirjoitetaan testit tilan lukemiseen ja tallentamiseen.

Ensimmäisessä tietokantayhteyden versiossa esimerkiksi tilaa yksilöivää tietoa ei syötetä, joten kirjastoa käyttävä sovellus voi lisätä vain yhden viitteen jokaista käytettyä tietokantaa

kohden, jos halutaan välttyä epäkoherentin tilan ongelmilta. Ongelman korjaisi esimerkiksi tilan yksilöivä avainsana tai taulun määritys yhteysasetuksissa. Jätetään tämän toteutus kuitenkin jatkokehitysvaiheeseen, koska se ei ole tutkielman kannalta olennainen ominaisuus.

Kehitysvaiheessa huomattiin, että JDBC-kirjaston ja DDL-lausekkeiden käyttäminen saa aikaan virheilmoitusten kapsulointiongelman. Jos virhe tapahtuu tietokantakyselyssä, viittaa poikkeus esimerkiksi SQL-lausekkeeseen eikä alkuperäiseen DDL-lausekkeeseen, jolloin virhe saattaa olla vaikea paikantaa.

```
(ns clj-wiite.db-store
  (:use clj-wiite.postgresql)
  (:require [clojure.spec.alpha :as s]
            [clj-wiite.store :refer :all]
            [clojure.java.jdbc :as jdbc])
  (:import [org.postgresql.util PSQLException]))

(def ^:private table-not-exists-state "42P01")

(s/def ::conn (s/keys :req [::dbtype ::dbname ::host]
                      :opt [::user ::password ::ssl ::sslfactory]))

(defn- table-exists? [conn]
  (some?
    (try
      (jdbc/query
        conn
        ["SELECT state FROM wiite_states ORDER BY id DESC LIMIT 0"])
      (catch PSQLException e
        (when-not (= (.getSQLState e) table-not-exists-state)
          (throw e))))))

(defn- create-table! [conn]
  (jdbc/db-do-commands
    conn
    (jdbc/create-table-ddl
      :wiite_states
      [[:id "serial"]
       [:created_at "timestamp with time zone" "default now()"]
       [:state "json"]])))

```

```
(defn- select-latest-state [conn]
  (first
    (jdbc/query
     conn
     ["SELECT state FROM wiite_states ORDER BY id DESC LIMIT 1"]
     {:row-fn #(get-in % [:state :value])})))

(defn- insert-latest-state! [conn state]
  (jdbc/insert!
    conn
    :wiite_states
    {:state {:value state}}))

(defrecord
  ^{:doc "Store for keeping state in database.
          Currently only PostgreSQL-database is fully supported"
    :added "0.1.0"}
  DBStore [conn]
  Store
  (write-state! [store state]
    (insert-latest-state! (:conn store) state))
  (load-state [store]
    (select-latest-state (:conn store))))

(s/fdef db-store
        :args (s/cat :conn ::conn)
        :ret #(instance? DBStore %))

(defn
  ^{:doc "Store for keeping state in database.
          Tables needed are being created if they don't exist.
          Currently only PostgreSQL-database is fully supported
          See: https://github.com/clojure/java.jdbc"
    :added "0.1.0"}
  db-store [conn]
  {:pre [(s/valid? ::conn conn)]}
  (when-not (table-exists? conn) (create-table! conn))
  (DBStore. conn))
```

Listaus 6.9. Tietokantayhteys db_store.clj

6.7 Tietokantayhteys ja tietorakenne

Kun sopiva viitettä hyödyntävä tietorakenne ja tietokantayhteys on rakennettu, olisi edessä näiden yhdistäminen. Käytön yksinkertaistamiseksi luodaan tätä varten funktio kirjaston *core.clj*-tiedostoon (listaus 6.10). Funktiolle annetaan tietokantayhteyden määritykset parametrina. Funktio luo DBStore-ilmentymän ja antaa tämän viitteelle parametrina. Funktio palauttaa Clojuren atomin, joka lukee ja kirjoittaa tilansa tietokantaan.

```
1  (ns clj-wiite.core
2    (:require [clj-wiite.watom :refer [create-watom]]
3              [clj-wiite.db-store :refer [db-store]]))
4
5  (defn
6    ^{:added "0.1.0"
7      :doc "Create Clojure Atom with watch with two operations:
8            - Changed Atom state is being written to DB store
9            - Initial state will be loaded from DB store
10          DB store will be initialized with parameters given."}
11   watom [conn]
12   (create-watom (db-store conn)))
```

Listaus 6.10. Luodaan viite core.clj

6.8 Optimointi

Kirjaston ensimmäinen versio on varsin alkeellinen ja suorituskykyyn ei ole panostettu lainkaan. Tämän tutkielman puitteissa ei kannata optimointia tehdä, mutta käyn seuraavaksi läpi ajatuksia, joita ensimmäisen version kehityksessä tuli esille.

Tutkielmassa kehitetty versio nojaa vahvasti Atom-viitteeseen. Toteutus pohjautuu täysin sen toimintaan ja tallentaa tiedon tietokantaan kuuntelijan avulla. Tämän vuoksi tieto on tallennettu turhaan sekä muistiin että tietokantaan. Kirjastoa kannattaisi kehittää niin, että muistissa pidetään vain osa tietorakenteen datasta niin sanottuna välimuistina. Useimmin tarvittavat arvot olisivat näin nopeasti käytettävissä, mutta harvemmin käytetyt eivät varaisi muistia. Tämä vaatisi tietorakenteen muuttamista niin, että se automaattisesti jakautuu pienempiin osiin ja pitää yllä tietoa siitä, milloin mitäkin osaa on käytetty.

Ensimmäinen versio tallentaa tiedon JSON-muodossa yhteen sarakkeeseen tietokannassa. Tämä ei ole tilan käytön eikä suorituskyvyn kannalta optimaalinen ratkaisu. Jos tila muuttuu, kirjoitetaan koko tietorakenteen data tietokantaan. Ensimmäinen vaihe olisi muuttaa tallennettava data käyttämään JSONB-formaattia. JSONB on binäärimuotoista dataa, joten se vaatii vähemmän tilaa tallennettaessa. Lopullinen ratkaisu olisi kuitenkin hajottaa tietorakenne niin, että jokainen sen arvo tallennetaan erikseen tietokantaan primitiivimuodossa. Tätä varten tarvitaan mekanismi, jolla yksittäiset arvot saadaan yhdistettyä toisiinsa kokonaiseksi tietorakenteeksi. Tämä ratkaisu mahdollistaisi myös ainoastaan muuttuneen osan tallennuksen. Lisäksi arvojen haku olisi nopeampaa.

Yllämainitut optimointiehdotukset vaativat Atom-viitteen hylkäämisen ja täysin oman tietorakenteen toteutuksen. Oma tietorakenne toteuttaisi Atomin vastaavat rajapinnat, jolloin sitä käytettäisiin samalla tavalla kuin Atomia. Tämä mahdollistaa oman toimintalogiikan toteutuksen. Ratkaisu vaatisi todennäköisesti matalamman tason tietorakenteen toteuttamista ja abstraktioiden hyödyntämistä, joihin saa apua luvusta 4.

Myös tietokantayhteyttä voisi parantaa. Tietokantayhteyden optimoinnissa esimerkiksi indeksointi mahdollistaa tiedon nopean hakemisen. Indeksoinnista on merkittävä hyöty vasta, kun tallennettava tieto hajautetaan pienempiin osiin.

6.9 Suorituskykytestit

Suorituskykytestejä varten luodaan kaksi erilaista käyttötilannetta. Molemmissa tilanteissa verrataan toteutettua tietorakennetta ja suoraa tietokantayhteyttä. Testeissä esimerkiksi arvojen generointi jätetään ajan mittaamisen ulkopuolelle. Ainoastaan kirjoittamiseen ja lukemiseen kuluva aika mitataan. Mittaustulokset esitellään luvussa 7.1.

Testitapauksia varten kehitetään yksinkertainen ja kevyt Clojure-sovellus. Sovellus määritellään tallentamaan tulokset tiedostoon. Jokainen testitapaus suoritetaan samaa tietokantaa käyttäen, jotta ympäristön vaikutus saadaan minimoitua. Testeissä ei vertailla dynaamisuuden tuomaa etua, koska se ei mielestäni sovi luonteeltaan relaatiotietokantojen malliin. Dynaamisuuden tuomaa etua vertaillaan luvussa 7.2. Testejä varten luodaan muunnelma Cloju-

ren core-kirjaston *time*-makrosta, kuten luvussa 5.3 mainitaan. Jokaisen testin alussa tietokantaan luodaan taulu testattavan tietorakenteen skeemalla. Testin lopuksi taulu poistetaan.

Ensimmäisessä testitapauksessa kirjoitetaan ja luetaan ennaltamääritelty tietorakenne tietokantaan tuhat kertaa ja tallennetaan kulunut aika. Lopuksi lasketaan keskiarvo ja mediaani kuluneista ajoista ja vertaillaan niitä ratkaisujen kesken. Tietorakenteen arvot muuttuvat testien aikana satunnaisesti, mutta pysyvät tyypiltään samoina. Arvot koostuvat merkkijonosta, kokonaisluvusta, liukuluvusta ja totuusarvosta. Samaa tietorakennetta käytetään molempien tallennustapojen testaamiseen. Tämä testitapaus on suoralle tietokantayhteydelle kirjoituksen osalta edullinen, koska tietokantaan luodaan määriteltyä tietorakennetta vastaava taulu. Testillä saadaan hyvää tietoa, kuinka paljon aikaa uhrataan kirjaston näkökulmasta pahimmassa tapauksessa, kun tila kirjoitetaan tietokantaan. Tilan lukemisen suhteen puolestaan kirjastolla on etu, koska kehitetty versio säilyttää tilan muistissa.

Toisessa testitapauksessa etsitään arvoa annetuilla parametreillä tietorakenteesta ja tietokannasta. Ennen testien suorittamista generoidaan ja annetaan Watomille viidensadan kappaleen kokoelma satunnaisia arvoja. Arvot koostuvat merkkijonosta, kokonaisluvusta, liukuluvusta ja totuusarvosta. Vastaavat arvot tallennetaan myös vertailutietokantaan. Seuraavaksi etsitään useaan kertaan erilaisia arvoja erilaisilla hakuehdoilla sekä tietorakenteesta että tietokannasta yhteensä tuhat kertaa. Testitapauksessa käytetään seuraavia hakuehtoja:

- Ensimmäinen annetun rajan ylittävä arvo
- Kaikki annetun rajan ylittävät arvot
- Ensimmäinen annetun arvon suuruinen arvo

Lopuksi lasketaan keskiarvo, mediaani ja keskihajonta operaatioihin kuluvista ajoista ja vertaillaan jokaiseen operaatioon kuluvan ajan eroa tietokanta- ja viiteratkaisun välillä. Tässäkään testissä arvojen tyypit eivät muutu. Testitapauksessa kirjaston ensimmäisen version muistin käytön heikkous tulee esille, koska koko kokoelma pidetään jatkuvasti muistissa. Kokoelman pitäminen muistissa tuo esille viiteratkaisua hyödyntävän nopeusedun. Lukuoperaatiot muistista ovat auttamatta nopeampia kuin tietokannasta.

Kolmannessa testitapauksessa kirjoitetaan kirjaston tietorakenteeseen ja tietokantaan saman kokoiset kokoelmat satunnaisia arvoja. Arvoja kirjoitetaan 500 kappaletta. Testi toistetaan

tuhat kertaa. Jokainen arvo lisätään yksitellen. Testissä tarkastellaan muistin käyttöä alku- ja lopputilanteessa. Testiä varten luodaan funktio, joka kutsuu aluksi roskienkerääjää ja laskee sitten vapaan keskusmuistin käytön.

6.10 Vertailuprojektit ja kirjasto

Tässä luvussa korvataan vertailuprojekteissa suora tietokantayhteys tutkielmassa kehitetyn kirjaston ratkaisulla.

Aloitetaan Compojure-esimerkkiprojektista (JarrodCTaylor 2018). Kyseessä on osoitekirjasovellus. Projektiin on lisätty yksitoista kappaletta yksikkötestejä, mikä helpottaa tietokantayhteyden korvaamista. Ensimmäiseksi päivitetään projektiin sen riippuvuuksien ja Clojuren versio. Esimerkkiprojekti on jo sen verran vanha, että riippuvuudet eivät ole yhteensopivia kehitetyn kirjaston riippuvuuksien kanssa. Päivityksien jälkeen lisätään kehitetty kirjasto projektin riippuvuuksiin.

Seuraavaksi lisätään globaali watom *address_book_routes.clj*-tiedostoon. Watomin tietokantayhteyden määritykset lisätään projektin *profiles.clj*-tiedostoon, josta ne annetaan watomille tätä luotaessa. Lisätään vielä apufunktio, joka asettaa watomin arvoksi vektorin, jos se on nil. Näin saadaan ensimmäinen tila sovellukselle, jos sitä ei ole vielä asetettu. Funktiota kutsutaan sovelluksen käynnistymisvaiheessa ja se korvaa tietokantataulujen luonnit.

Sovelluksen tilaa muokataan ainoastaan *address_book_routes.clj*-tiedostossa. Tietokantakutsut voidaan korvata suoraan watomin tilan päivityksellä, lukuun ottamatta entiteetin tunnisteella arvon poistoa. Poisto-operaatiota varten lisätään pieni apufunktio, jolla voidaan etsiä arvoa tietyllä tunnisteella kokoelmasta. Projektin alkuperäinen toteutus pohjautuu juokseviin tunnisteisiin. Korvataan ratkaisu tarkistamalla suurin yhteystiedon tunniste ja antamalla seuraava vapaa tunniste uudelle yhteystiedolle. Tämä ei ole kuitenkaan kestävä ratkaisu, koska tunnisteita saatetaan tahattomasti kierrättää, jos loppupään yhteystietoja poistetaan. Kirjaston vertailussa tämä on kuitenkin riittävä toteutus. Lopuksi voidaan poistaa kokonaan tietokantakutsut ja niiden määritykset *models*-kansiosta.

Yksikkötesteissä, kuten sovelluskoodissakin, korvataan suorat tietokantakutsut watomin tilan päivityksellä.

Seuraavaksi otetaan työn alle *guestbook*-projekti (Yogthos ja Paulrd 2018). Ensimmäiseksi päivitetään projektiin Clojure 1.9.0 versioon, jotta kehitetty kirjasto toimii projektin sovelluksessa. Seuraavaksi pystytetään testitietokanta ja käynnistetään sovellus. Projekti käyttää H2-relaatiotietokantaa.

Lisätään seuraavaksi riippuvuus kehitettyyn kirjastoon ja korvataan tietokantakutsut. Sovelluksen tietokantakutsut sijaitsevat kaikki *routes/home.clj*-tiedostossa. Lisätään taas globaali watom ja korvataan tietokantakutsut watomin tilan päivityksillä. Kuten aiemminkin, kirjoitetaan ohjelmaan funktio, joka määrittää watomin ensimmäisen tilan, jos kyseessä on ensimmäinen käynnistyskerta. Jo toisen projektin kohdalla tulee selväksi, että kehitetyn kirjaston watomin luomisessa olisi hyvä olla mahdollisuus ensimmäisen tilan määrittämiseen. Silloin ei tarvitsisi projektissa apufunktiota tilan määritykseen. Kirjaston ensimmäisessä versiossa päivämääräolioita ei voi tallentaa tietorakenteeseen. Päivämääräolioita varten tarvitaan erillinen muunnos, mitä ei tämän tutkielman puitteissa toteuteta. Vertailuprojektissa joudutaan siis muuntamaan ja tallentamaan päivämäärät merkkijonoina.

Kolmantena vertailuprojektina on *Prototyping blog engine*, eli kokeiluihin suunniteltu blogmoottori. Kuten aiempienkin vertailuprojektien kanssa, päivitetään ensimmäiseksi projektin riippuvuudet uusiin versioihin. Seuraavaksi käynnistetään sovellus, jotta saadaan testattua sen toiminta.

Kun sovellus on testattu, korvataan tietokantakutsut watomin tilan päivityksillä. Tässä vertailuprojektissa se ei kuitenkaan ole ihan suoraviivaista, koska tietokantataulujen yhdistäminen on toteutettu koodissa eikä relaatioiden avulla tietokantakutsuissa. Tämän vuoksi koodia joudutaan yksinkertaistamaan, koska arvot voivat olla nyt samassa tietorakenteessa. Lisäksi kirjoitetaan yksi apufunktio, jolla haetaan arvo sen tunnisteen perusteella.

7 Toteutuksen analysointi

7.1 Suorituskykytestien tulokset

Ennaltamäärätyn arvon kirjoitusnopeuksissa kirjaston ja suoran tietokantayhteyden välillä ei suuria eroja esiintynyt (taulukko 7). Keskiarvot poikkesivat vain vajaa 0,2 ms. Keskihajonta oli Wiitteellä kirjoituksen osalta yli kaksi kertaa suurempi. Lukunopeuden suuri ero keskiarvossa johtunee siitä, että kirjaston viite säilyttää arvonsa keskusmuistissa.

	Wiite kirjoitus	**Tietokanta kirjoitus**	**Wiite luku**	**Tietokanta luku**
Keskiarvo	11,3087	11,1107	0,0049	7,9176
Mediaani	11,6360	11,6142	0,0055	8,3960
Maksimi	103,5012	24,4316	0,0189	38,4766
Minimi	5,3068	5,1338	0,0009	2,6917
Keskihajonta	4,0609	1,9053	0,0014	1,9264

Taulukko 7. Kirjoitus ja luku millisekunteina

Arvojen etsimisessä erot ovat selvät (taulukko 8). Muun muassa rajan ylittävän arvon etsimisessä tietokannan osalta kuluneen ajan keskiarvo on 443 kertaa suurempi ja keskihajontakin 50-kertainen. Jokaisessa testitapauksessa kirjaston suorituskyky on selvästi parempi verrattuna suoraan tietokantayhteyteen. Tämä selittynee sillä, että kirjasto säilyttää arvonsa keskusmuistissa. Arvojen etsimisen testit osoittavat, että useassa tapauksessa on tehokkaampaa tehdä esimerkiksi suodatus- tai etsimisoperaatiot keskusmuistissa olevasta tietorakenteessa kuin tietokantakyselyllä. Tämä ei tosin aina ole resurssien rajoitteiden tai suurien tietomäärien vuoksi mahdollista tai järkevää. Todennäköisesti tulokset olisivat olleet erilaisia, jos testeissä olisi käytetty huomattavasti suurempia tietomääriä.

Muistinkäyttöä testaavat suorituskykytestit epäonnistuivat täysin. Mittaaminen osoittautui hankalaksi ja epäluotettavaksi, joten järkeviä tuloksia ei saatu esiteltäväksi. Mittaamisessa käytettiin *java.lang.Runtime*-oliota, jolta kysyttiin vapaan muistin ja kokonaismuistin määrää testien alussa ja lopussa. Arvoja sekoittivat kenties ajon aikainen roskien keräys ja JVM:n optimoinnit.

		Keskiarvo	Mediaani	Maksimi	Minimi	Keskihajonta
Ensimmäinen rajan ylittävä arvo	Wiite	0,00665640	0,00549650	0,26156400	0,00360900	0,00971880
	Tietok.	2,95059000	2,83812650	5,99119200	2,43149300	0,49265670
Kaikki rajan ylittävät arvot	Wiite	0,00494260	0,00437500	0,22281500	0,00334500	0,00715010
	Tietok.	4,20697950	3,89873400	48,18229500	2,43062500	2,59544650
Ensimmäinen saman suuruinen arvo	Wiite	0,04961160	0,04488750	0,56370900	0,00736600	0,03301490
	Tietok.	2,99802170	2,86982850	9,90590600	2,43770900	0,53368770
Kaikki saman suuruiset arvot	Wiite	0,00486720	0,00439250	0,17111200	0,00310400	0,00556310
	Tietok.	2,97452880	2,83701850	23,01550600	2,40251400	0,85018420

Taulukko 8. Arvojen etsimiseen kulunut aika millisekunteina, kun vertaillaan suoraa tietokantayhteyttä ja tutkielmassa toteutettua kirjastoa

7.2 Vertailuprojektit kirjaston kanssa ja ilman

Kirjaston lisäyksen jälkeen kaikki luvussa 6.1 esitellyt vertailuprojektit toimivat kuten ennen muutosta. Yleisesti koodimuutokset eivät olleet kovin suuria. Suorituskyvyssä ei ole eroa, joskin projektien pienen koon ja datan pienen määrän vuoksi suorituskyvyn eron tulisi olla huomattava, ennen kuin se vaikuttaisi sovelluksen toimintaan. Toteutuksen alkuvaiheessa ilmeni heti, että kirjaston pienilläkin muutoksilla sen käytöstä saisi helpompaa ja tarvittava koodin määrä vähenisi entisestään. Esimerkiksi alkutilan määritys poistaisi alustusfunktion tarpeen.

Guestbook- (Yogthos ja Paulrd 2018) ja addressbook-projektissa (JarrodCTaylor 2018) koodin määrä väheni 1-2 % ja blogiprojektissa noin 14 % (taulukko 7.2). Jokaisessa projektissa SQL-koodi siivottiin kokonaan pois. Blogi-projektissa SQL-koodin siivouksen lisäksi poistettiin arvojen yhdistämisiä ohjelmakoodista, koska esimerkiksi yhden blogikirjoituksen tunnisteet (engl. tag) tallennettiin muutoksen jälkeen suoraan tietorakenteeseen eikä erilliseen tauluun. Toteutettu kirjasto vähentää koodin määrää keskimäärin SQL-lausekkeiden verran.

Toteutettu kirjasto vaikuttaa merkittävästi koodin ulkoasuun ja luettavuuteen. Mielestäni ohjelmakoodi yksinkertaistui kirjaston lisäyksen jälkeen. Tiedon tallennus ja lataus tapahtuu nyt käyttämällä viitettä, mikä on mielestäni yksinkertaisempaa kuin tietokantakyselyiden toteuttaminen. Myös usean eri ohjelmointikielen, kuten SQL ja Clojure, yhdistäminen usein heikentänee ohjelmakoodin luettavuutta.

	address-book		guestbook		Prototyping blog engine	
	Ennen	Jälkeen	Ennen	Jälkeen	Ennen	Jälkeen
Clojure	185	200	352	352	469	446
SQL	23	0	10	0	50	0
Yhteensä	208	200	362	352	519	446

Taulukko 9. Ohjelmakoodin rivimäärän muutos vertailuprojekteissa Wiite-kirjaston käyttöönoton jälkeen

7.3 Hyödyt ja haasteet

Kirjaston käyttö poistaa projektista tietokantasuunnittelun tarpeen kokonaan. Tavallisesti se on erittäin tärkeä osa sovelluksen suunnittelua ja usein se ohjaa sovelluskehityksen kulkua. Tietokantasuunnittelu ei yleensä ole prototyyppisovelluksessa olennaista. Esimerkiksi tietokantarakenteen muuttaminen jälkikäteen usein aiheuttaa lisätyötä kuten tietomallien päivityksiä. Tietokantamigraatioiden määritykset yleisesti prototyyppisovelluksessa taas tuntuvat mielestäni liioitelluilta, sillä oletuksella että prototyyppiversiota ei käytetä lopullisen version pohjana. Kirjaston käyttöönotto vaatii ainoastaan olemassa olevan tietokannan taustalleen. Kirjasto hoitaa tietokannan rakenteen luomisen.

Tiedon tallentaminen ja lukeminen on helppoa kirjaston avulla. Ainoastaan atomin tilan muutos ja arvon noutaminen riittävät pysyväksi tiedon tallentamiseksi. Kirjaston viite on luonnollinen osa Clojure-ohjelmakoodia ja tiedon käsittelyyn ei tarvita erillisiä kyselyitä tai mahdollisesti muusta ohjelmakoodista poikkeavaa kyselykieltä.

Suurien tietomäärien tallentaminen ei ole kirjaston kanssa järkevää. Ohjelman käyttämä muistin määrä karkaa helposti hallinnasta, jos tallennettavat tietomäärät kasvavat suuriksi. Lisäksi suurien tietomäärien käsittely saattaa vaatia paljonkin tiedon käsittelyä ohjelmakoodissa. Vastaavat operaatiot ovat usein suhteellisen helppoja ja tehokkaita toteuttaa tietokantakutsuilla.

Kirjaston taustatietokannaksi relaatiotietokanta ei ole järkevä valinta. Sen tarjoamat hyödyt, kuten tiedon hajauttaminen ja relaatioilla yhdistäminen, tehokkaat kyselyt ja tiedon suodatus, jäävät lähes täysin käyttämättä. Jatkokehitystä varten esimerkiksi NoSQL-tuki olisi mieles-

täni hyvä valinta taustatietokannaksi. Dokumenttitietokantana NoSQL-tietokanta soveltuu luonteeltaan Clojuren tietorakenteiden tallentamiseen paremmin.

7.4 Erot vastaaviin työkaluihin

Tällä hetkellä on olemassa ainakin kolme erilaista julkisesti saatavilla olevaa harrastelijaprojektia, jotka tarjoavat samankaltaisia ominaisuuksia kuin tutkielmassa kehitetty kirjasto. Näiden lisäksi on olemassa kaupallinen tuote Datomic, jonka toimintaperiaate on etäisesti samanlainen kuin kehitetty kirjasto. Datomicista kerrotaan tarkemmin luvussa 4.8.

Lähin vaihtoehto kehitetylle kirjastolle harrasteprojekteista on *duratom*, jonka on kehittänyt Piliouras (2016). Se tukee tiedostoon ja PostgreSQL-tietokantaan tallentamisen lisäksi *Amazon S3* -palvelua. Joitakin eroja tutkielmassa toteutetun kirjaston ja duratomin välillä on. Esimerkiksi duratom ei tue historiatietoa, siinä on datan poistamisen tuki ja monipuolisempi viitteen määrittely.

Enduro-kirjastossa on toteutettu oma vastine Clojuren Atomille, joten kirjaston toteutus ei sovellu suoraan Atom-viitteen korvikkeeksi. Kirjaston viite ei toteuta Atomin rajapintoja ja sitä hallitaan omilla funktioilla. Kirjaston on kehittänyt Dipert (2012).

Durable-atom, jonka on kehittänyt Sloan (2016), on yksinkertaisesti tiedostovarmennettu Clojuren Atom.

7.5 Käyttökohteet

Tutkielmassa kehitetty kirjasto soveltuu hyvin pienten sovellusten ja prototyyppiversioiden tiedon tallennusratkaisuksi. Myös sovelluskehitysprojektissa alkuvaiheen tallennusratkaisuksi kirjaston viite sopii, jos käsiteltävät tietomäärät eivät ole suuria. Kirjaston käyttö nopeuttaa niin sanotun yksinkertaisimman toimivan ratkaisun (engl. Minimum Viable Product) saamista esittelykuntoon. Hyvin pohjustettu tiedontallennus ohjelmakoodissa mahdollistaa vaihdon esimerkiksi suoraan tietokantayhteyteen. Kirjaston viite soveltuu myös REPL-istunnon tiedon pysyvään tallentamiseen.

7.6 Jatkokehitys

Tutkielman puitteissa kehitetty ensimmäinen versio kirjastosta on varsin suppea ominaisuuksiltaan. Sen suorituskyky ei ole muistinkäytön suhteen hyvä. Lisäksi kirjaston vikasietoisuutta ei ole otettu huomioon. Käytössä viitteen tila ei välttämättä ole koherentti, jos tietokannassa olevaa dataa muokataan viitteen ulkopuolella. Tähän auttaisi kuuntelija, joka välittää tiedon tilan muutoksista tietokannasta viitteille.

Yleisesti kirjaston virheenkäsittelyyn tulisi panostaa. Esimerkiksi Clojuren Atomin periaate, jossa operaatio toteutetaan silmukassa kunnes se onnistuu, voisi toimia myös tietokantaan tallennuksen yhteydessä. Silmukan kanssa tulee olla tarkka virheiden ja poikkeusten käsittelyssä. Kun käsitellään vain virhetilanteet, joissa uudelleenyrityksellä on mahdollista tilanteesta palautua, vältetään ikuiset silmukat.

PostgreSQL-tietokanta ei ole mielestäni paras viitteen tilan tallennukseen, ellei tietorakennetta hajota pienempiin osiin ja hajauta sitä eri tauluihin. Jatkokehityksessä kannattaakin ottaa huomioon tiedon tallennuksen taustaratkaisu. Esimerkiksi NoSQL-tietokanta soveltuisi paremmin Clojuren tietorakenteiden tallentamiseen, koska tietoa käsitellään dokumentteina. WebSQL-tuki taas lisäisi käyttökohteita, koska silloin kirjasto soveltuisi myös websovellusten lokaalin tilan tallentamiseen.

Muistinkäyttö saataisiin kuriin siirtämällä tila muistista taustajärjestelmään. Tällöin kannattaisi kuitenkin lisätä jonkinlainen välimuistitus. Välimuisti lisäisi huomattavasti viitteen suorituskykyä verrattuna suoraan tietokantayhteyteen.

Tieto tallennetaan tietokantaan nykyisessä ratkaisussa synkronisesti. Tämä tarkoittaa sitä, että sovelluksen suoritus jatkuu vasta, kun tila on tallennettu tietokantaan. Tämä sopii hyvin Clojuren Atomin toimintaperiaatteisiin. Sen rinnalle voisi kuitenkin harkita tukea myös muille Clojuren viitteille, kuten agentille. Näin saavutettaisiin tarvittaessa asynkroninen tilan tallennus tietokantaan.

Kirjaston käyttöä helpottamaan ensimmäisen tilan asettaminen olisi hyvä ominaisuus. Kuten luvussa 7.2 mainitaan, vertailuprojektien muutoksissa alkutilan asettaminen tulisi selvästi tarpeeseen.

Kirjaston viite voisi tukea datan yksilöintiä tietokannassa. Näin sovelluksessa voisi olla useita viitteitä, joiden tila olisi tietokantaan tallennettuna. Yksilöinti voisi tapahtua esimerkiksi yksilöivällä avainsanalla.

8 Johtopäätökset

Tämän pro gradu -tutkielman tavoitteena oli tutkia, voidaanko Clojuren viitteitä ja niiden ominaisuuksia hyödyntää tietokantayhteyden tukena. Lisäksi tutkittiin, onko viitteillä vaikutusta sovelluksen suorituskykyyn ja ohjelmakoodin luettavuuteen. Tämän suomenkielisen tutkimuksen sivutuotteena tutkielmasta syntyi mahdollista opiskelu- ja opetusmateriaalia. Lisäksi tutkielman tavoitteena oli selvittää, millaista Clojure-ohjelmointikielessä on käyttää tietokantayhteyksiä ja kuinka nykytilannetta voitaisiin parantaa. Tutkimuksessa selvitettiin, voidaanko Clojuren viitteitä hyödyntää tietokantayhteyksien yksinkertaistamisessa. Tutkielman tavoitteena oli selvittää myös, onko viitteitä käyttämällä mahdollista parantaa ohjelmakoodia, missä tilanteissa viitteitä voisi käyttää ja onko niistä apua suorituskyvyn parantamisessa. Tutkimuksen yhteydessä luotiin kirjasto, jolla testataan käytännössä viitteiden hyödyntämistä tietokantayhteyksissä.

Tutkielman taustatutkimus antaa paljon hyödyllistä suomenkielistä materiaalia Clojure-ohjelmointikielestä. Materiaalia voi hyödyntää opiskelussa esimerkiksi syventävänä tietona, mutta tutkielma kokonaisuutena ei soveltune opetusmateriaaliksi. Tämä pro gradu -työ sisältää paljon materiaalia, joka on todennäköisesti opetustarkoituksessa epärelevanttia tai liian yksityiskohtaista. Kuitenkin esimerkiksi tietorakenteiden, viitteiden ja rinnakkaisohjelmoinnin katsausta voisi mielestäni hyödyntää osana Clojuren perusteiden opettelua.

Tämän tutkielman pihvi, eli viitteiden hyödyntäminen tietokantayhteyksissä, osoittautuu mielestäni osittain epäkäytännölliseksi. Koodin yksinkertaistamisen osalta se mielestäni toimii, mutta jään kaipaamaan tehokkaampaa ja ohjelmakoodiin luonnollisemmin sulautuvaa ratkaisua, joka kuitenkaan ei tee kompromisseja suorituskyvyn suhteen. Ratkaisun ei tulisi myöskään asettaa rajoituksia esimerkiksi tietorakenteiden koolle tai tyypille.

Prototyyppisovellusten tai pienten sisäiseen käyttöön tarkoitettujen lokaalia sovelluksen tilaa säilyttävien ohjelmien osana tutkielmassa toteutettu kirjasto voisi toimia hyvinkin. Tuotantokäyttöön en sitä kuitenkaan laittaisi. Lisäksi toteutus jäi huomattavasti suppeammaksi, mitä alunperin suunnittelin. Tutkielman käytännön osuuden kirjoitushetkellä selvisi myös, että on olemassa useita vastaavia työkaluja. Ennen toteutusosuutta näistä en löytänyt tietoa, koska

toteutustekniikka ja kirjaston laajuus eivät olleet vielä selvillä. Tutkielmaa hahmotellessa suunnittelin kirjaston toteutuksen olevan Clojuren tietorakenteiden kaltainen matalamman tason ratkaisu, jollaisen toteutukseen luku 4 antaa olennaista taustatietoa.

Tutkielmassa kehitetty kirjasto kaipaisi mielestäni runsaasti jatkokehitystä, ennen kuin sitä voisi käyttää edes kehityksen tukena tai sisäisessä käytössä. Joitain suunniteltuja ominaisuuksia, kuten valinnainen historiatiedon säilyttäminen ja alkutilan asettaminen, jäivät tutkielman puitteissa pois.

Tutkielman testaus jäi puutteelliseksi. Lukuoperaatioissa tietomäärät jäivät liian pieniksi. Tämän vuoksi kirjaston ja suoran tietokantayhteyden erot jäivät epäselviksi. Suuremmilla tietomäärillä lukunopeuden ero olisi tullut selvemmin esille ja vertailu suorituskyvyssä olisi saattanut olla helpompaa ja luotettavampaa. Muistin käyttöä vertailevat testit nekin epäonnistuivat ja olivat epäluotettavia. Lisäksi testeissä olisi voinut tutkia hakutulosten manipulointia, kuten järjestämistä, suodatusta ja yhdistämistä.

Kaiken kaikkiaan tutkielma tarjoaa tiiviin ja kattavan paketin taustatietoa Clojuresta ohjelmointikielenä. Lisäksi teos tarjoaa syventävää tietoa viitteistä ja kirjaston toteuttamisesta Clojurelle.

9 Yhteenveto

Tämä opinnäytetyö tutkii Clojure-ohjelmointikieltä, siinä käytettäviä viitteitä ja niiden hyödyntämistä tietokantayhteyksien kanssa. Tutkielmassa toteutetaan kirjasto, jolla voidaan korvata perinteinen tietokantayhteys Atom-viitteen kaltaisella ratkaisulla.

Luvussa 2 tehdään katsaus Clojureen ohjelmointikielenä. Luvussa esitellään kielen syntyä, teknisiä ratkaisuja ja raapaistaan Clojuren rinnakkaisohjelmointia.

Luku 3 esittelee viitteet. Siinä kerrotaan, mitä viitteet oikein ovat ja mitä niillä tehdään. Luku syventyy myös Clojuren omiin viitteisiin, niiden ominaisuuksiin ja käyttökohteisiin. Lisäksi luvussa tehdään kevyt vertailu Clojuren viitteistä.

Luvussa 4 keskitytään tietorakenteisiin ja tietokantoihin. Keskiössä ovat Clojuren tietotyypit sekä tietorakenteet ja niiden abstraktiot. Luvussa selviää, miten Clojuressa hallitaan tietoa. Mukana on myös katsaus Clojuren spec-kirjastoon. Lisäksi luku esittelee erilaisia tietokantatyyppejä.

Luku 5 esittelee tutkielmassa toteutettavan kirjaston suunnitelma. Luvussa käydään läpi kirjaston tavoitteet, hyödyt ja haasteet. Lisäksi esitellään testauksessa käytettävät työkalut ja työskentelytavat. Suunnitelmaan saadaan hyvä pohja aiemmista luvuista. Ne asettavat raamit kirjaston toteutuksen laajuudelle.

Luku 6 on kirjaston toteutusosuus. Luvun alussa valitaan vertailuprojektit, joissa kirjastoa käytetään. Tämän avulla selvitetään, miten kirjasto käytännössä toimii ja miten se sulautuu ohjelmakoodiin. Tämä auttaa selvittämään, helpottaako kirjaston käyttö ohjelmoijan työtä. Lisäksi käydään läpi optimointi-ideoita, suorituskykytestien toteutus ja kirjaston käyttö vertailuprojekteissa. Toteutuksen apuna toimii aiemmat kappaleet. Varsinkin jatkokehitysideat nojaavat vahvasti luvussa 4 käsiteltyihin aiheisiin.

Luku 7 käy läpi suorituskykytestien tulokset. Lisäksi tutkitaan, miten kirjasto pärjää suoran tietokantayhteyden rinnalla. Luvussa selviää, millaisia vaikutuksia kirjastolla on vertailuprojektien koodipohjissa, miten kirjasto vaikuttaa sovelluksen kehitykseen ja koko vertai-

luprojektin sovelluksen toimintaan. Luku avaa kirjaston hyötyjä, haittoja ja käyttökohteita. Luvussa pohditaan myös, kuinka kirjastoa voisi mahdollisesti parantaa.

Luvussa 8 esitellään tutkielman johtopäätökset.

Lähteet

Dipert, Alan. 2012. *enduro.* Viitattu 3. heinäkuuta 2018. `https://github.com/alandipert/enduro`.

Emerick, Chas, Brian Carper ja Christophe Grand. 2012. *Clojure Programming.* 632. O'Reilly Media, Inc. ISBN: 9781449394707.

Flanagan, David. 2005. *Java In A Nutshell, 5th Edition.* O'Reilly Media, Inc. ISBN: 0596007736.

Fogus, Michael, Chris Houser, William E Byrd, Daniel P Friedman ja Andrew Oswald. 2014. *The Joy of Clojure.* 2. painos. Manning Publications Co. ISBN: 9781617291418.

Higginbotham, Daniel. 2015. *Clojure for the Brave and True.* No Starch Press. ISBN: 978-1593275914.

JarrodCTaylor. 2018. *address-book.* `https://github.com/JarrodCTaylor/compojure-address-book`.

Kaihlavirta, Vesa. 2017. *Mastering Rust.* 354. Packt. ISBN: 9781785885303.

Keto256. 2018. *Prototyping blog engine.* `https://github.com/keto256/proto-blog`.

Kuper, Lindsey. 2013. "Thesis Proposal : Lattice-based Data Structures for Deterministic Parallel and Distributed Programming", numero September: 1–20.

Li, Wing Hang, David R. White ja Jeremy Singer. 2013. "JVM-hosted languages". Teoksessa *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools - PPPJ '13,* 101–112. ISBN: 9781450321112.

Lourenço, João Ricardo, Bruno Cabral, Paulo Carreiro, Marco Vieira ja Jorge Bernardino. 2015. "Choosing the right NoSQL database for the job: a quality attribute evaluation". *Journal of Big Data* 2 (1): 18.

McCarthy, John. 1978. "History of LISP". *ACM SIGPLAN Notices* 13 (February): 217–223.

McDonnell, Mark. 2017. *Quick Clojure: Effective Functional Programming.* 200. Apress. ISBN: 978-1-4842-2952-1.

Meier, Carin. 2015. *Living Clojure.* O'Reilly Media, Inc. ISBN: 9781491909041.

Miller, A, S Halloway ja A Bedra. 2018. *Programming Clojure.* Pragmatic Bookshelf. ISBN: 9781680505726.

Paul, Chiusano, ja Runar Bjarnason. 2014. *Functional Programming in Scala,* 1:1–320. Manning Publications Co. ISBN: 9788578110796.

Piliouras, Dimitrios. 2016. *Duratom.* Viitattu 3. heinäkuuta 2018. `https://github.com/jimpil/duratom`.

Pratley, Timothy. 2016. *Professional Clojure.* Wrox. ISBN: 978-1119267270.

Prokopec, Aleksandar, Heather Miller, Tobias Schlatter, Philipp Haller ja Martin Odersky. 2013. "FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction". *Languages and Compilers for Parallel Computing* 7760:158–173.

Rathore, Amit. 2011. *Clojure in Action - Elegant Applications on the JVM.* 432. Manning Publications Co. ISBN: 9781935182597.

Sloan, Stephen. 2016. *durable-atom.* Viitattu 3. heinäkuuta 2018. `https://github.com/polygloton/durable-atom`.

Thomas, Dave. 2018. *Programming Elixir.* Pragmatic Bookshelf. ISBN: 978-1-68050-299-2.

VanderHart, Luke, ja Stuart Sierra. 2010. *Practical Clojure.* 232. Apress. ISBN: 978-1-4302-7230-4.

Yogthos ja Paulrd. 2018. *guestbook.* `https://github.com/luminus-framework/examples/tree/master/guestbook`.

Liitteet