Iida Sallinen

Verkkokaupan tietokantamallin valinta

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

6.11.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Iida Sallinen Verkkokaupan tietokantamallin valinta 47 sivua + 2 liitettä 6.11.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Lehtori Vesa Ollikainen

Insinöörityön tavoitteena oli selvittää NoSQL-tietokantojen toiminnallisuutta sekä vertailla NoSQL- ja SQL-tietokantamalleja toisiinsa. Tämän lisäksi haluttiin selvittää alustavasti, kuinka NoSQL-tietokanta tulisi toimimaan suorituskykynsä puolesta verkkokaupan yksinkertaistetun skeeman kanssa, ja katsoa, kuinka se vertautuu relaatiokannan suorituskykyyn.

Teoriaosuudessa käytiin ensin läpi relaatiomallia ja relaatiokantojen toimintaa. Olemassa olevasta verkkokauppasovelluksesta valittiin neljä SQL-kyselyä lähempään tarkasteluun käytännön osuutta varten. Tämän jälkeen käytiin läpi NoSQL-kantojen perusteet ja niiden toiminta sekä NoSQL-kantojen eri tyypit ja tarkasteltiin paremmin kolmea eri kantaa, jotka olivat ehdolla relaatiokannan tilalle. Lopulta käytettäväksi NoSQL-kannaksi valittiin MongoDB.

Käytännön osuudessa muodostettiin MongoDB-kanta ja siihen halutut SQL:n tauluja vastaavat kokoelmat. Tässä vaiheessa tauluihin ja kokoelmiin lisättiin testidata, jonka jälkeen muodostettiin halutut neljä NoSQL-kyselyä. Niiden suorituskykyä verrattiin vastaaviin SQL-kyselyihin, ja vertailun perusteella muodostettiin kuvaajia. Lopuksi käytiin läpi NoSQL-kantojen tietoturvaongelmia ja mahdollisia ratkaisuja niihin ongelmiin.

Insinöörityössä osoitettiin, että MongoDB voi suorituskykynsä puolesta tarjota vaihtoehdon relaatiokannalle käytössä olevassa verkkokaupassa. Seuraavana vaiheena MongoDB-kannan käyttöönotossa on kuormituskykymittausten suorittaminen sekä esiin tulleiden ongelmien ratkominen.

Avainsanat	Tietokanta, NoSQL, MongoDB, SQL, relaatiomalli, relaatiotietokanta

Author Title Number of Pages Date	Iida Sallinen Choosing a database model for an online store application 47 pages + 2 appendices 6 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Vesa Ollikainen, Senior Lecturer

The purpose of the thesis was to examine the functionality of NoSQL-databases and compare NoSQL-databases to relational databases. In addition to this, it was neccessary to find out how a NoSQL-database could perform with the already existing online store application and see how it compares to the performance of the relational database.

In the theory part of the thesis the relational model and relational database were first looked at. Four SQL-queries, used in the online store application, were chosen for further inspection and use in the work part of the thesis. After the queries it was time to look at what exactly is a NoSQL-database and how does it work. And after the basics were clear, the different database types, and three spesific databases, were looked. at In the end, MongoDB was chosen as the database to use.

In the work part of the thesis a MongoDB database and the needed collections were made. After this, data for testing purposes was added in both databases and four queries, mirroring the SQL-queries, were formed. The query times were then compared to each other and tables were formed from the comparisons. The last thing was to look at security problems in NoSQL-databases and the possible solutions to these problems.

The result of the thesis was that a MongoDB database can offer an alternative to the relational database. The next stage is to perform load testing to the MongoDB database and solve the concerns that came up during the process.

Keywords	Database, NoSQL, MongoDB, SQL, relational model, relational database

Sisällys

Lyhenteet

1 Johdanto 1

2 Relaatiotietokannat 2

2.1 Mikä on relaatiotietokanta? 2
2.1.1 Relaatiokantojen historiaa 2
2.1.2 Relaatiomalli 3
2.1.3 Rakenne 4
2.1.4 Relaatiokannan hyvät ja huonot puolet 5
2.2 Muutamia tarpeellisia asioita 5
2.2.1 SQL 5
2.2.2 ACID 6
2.2.3 High Availability 8

3 Relaatiotietokannan rakenne ja tarkasteltavat lauseet 8

3.1 Kannan rakenne ja syy lauseiden valintaan 8
3.2 Valitut lauseet 10

4 NoSQL-tietokannat 12

4.1 Käyttötarkoitukset 12
4.2 Mikä on NoSQL-tietokanta? 13
4.3 Hyvät ja huonot puolet 14
4.4 NoSQL- ja SQL-kantojen eroavaisuudet 16
4.5 NoSQL-tietokantatyypit 17
4.6 Eri NoSQL-kantoja ja niiden erityispiirteitä 19
4.6.1 MongoDB 19
4.6.2 RavenDB 20
4.6.3 Redis 22
4.6.4 Muita esimerkkejä 23
4.6.5 Kantojen vertailua 24

5 Toteutuksen valmistelua 24

5.1 Miksi NoSQL? 24

5.2 Valittujen lauseiden NoSQL-esitys 25

5.3 NoSQL-tietokannan valinta 26

6 Toteutus 26

6.1 Alustavat toimenpiteet 26

6.2 NoSQL-mallin toteutus 28

6.2.1 Kannan rakenne 28

6.2.2 Kyselyt 30

6.3 Suorituskykyvertailu 32

6.4 Indeksointi 35

6.5 Tietoturva 37

7 Tulokset 39

7.1 Lopputulokset 39

7.2 Ongelmat 40

7.3 Esiin tulleet kysymykset 41

7.4 Jatkokehitys 42

8 Yhteenveto 43

Lähteet 45

Liitteet
Liite 1. SQL-vedokset
Liite 2. Suorituskykymittausten tulokset

Lyhenteet

ACID	Atomicity, Consistency, Isolation, Durability. Transaktioiden ominaisuuksiin liittyvä lyhenne: atomisuus, eheys, eristyneisyys, pysyvyys.
DBMS	Database Management System. Tietokantojen tiedon manipulointiin käytettävä ohjelmisto.
CPU	Central Processing Unit. Suoritin.
GUI	Graphical User Interface. Ohjelmiston hallinnan apuna käytettävä graafinen työkalu.
HA	High Availability. Tietokanta, joka on kestävä ja suunniteltu toimimaan pitkiä aikoja.
JSON	JavaScript Object Notation. JavaScript-pohjainen kieli, jota käytetään web-sovelluksissa.
NoSQL	Not(-only) Structured Language. Tietokantamalli, joka ei käytä SQL-kieltä eikä noudata relaatiomallia.
RDBMS	Relational Database Management System. Relaatiotietokantojen manipulointiin käytettävä ohjelmisto.
SQL	Structured Query Language. Relaatiokantojen hallintaan ja muokkaamiseen käytettävä kieli.
TLS/SSL	Transport Layer Security/Secure Sockets Layer. TLS tarjoaa yhteyksien välistä eheyttä ja yksityisyyttä, eli on kryptaukseen tarkoitettu protokolla. SSL on sen vanhentunut versio.
XML	Extensible Markup Language. Käytetään datan esittämiseen ja siirtämiseen web-sovelluksissa.

1 Johdanto

Työn tavoitteena on vertailla eri tietokantamalleja (erityisesti SQL- ja NoSQL-tietokantamalleja) ja selvittää niiden toteuttamista verkkokaupan käyttöön. Lisäksi halutaan tietää, kuinka eri tietokantamallien käyttö eroaa toisistaan ja mallintaa, kuinka relaatiokantaa käyttävä verkkokauppa olisi mahdollista toteuttaa NoSQL-kannalla. Käytännössä halutaan selvittää, onko mahdollista ottaa käyttöön NoSQL-tietokanta, pitäen samalla kuitenkin sovellukseen aiheutuvat muutokset mahdollisimman pieninä. Insinöörityön aihe syntyi henkilökohtaisesta tarpeesta syventää tietoja eri tietokantamalleista.

Verkkokauppa on työssä valittu mallinnuksen kohteeksi siksi, että verkkokaupat ovat hyvin yleisiä ja siten myös tärkeitä. Yleisyyden, ja osittain myös muiden tekijöiden vuoksi ne ovat hyvin herkkiä suorituskykyongelmille, joita tässäkin työssä tullaan tarkastelemaan. Niitä mallintaessa ja tehdessä on myös hyvin tärkeää ottaa huomioon tietoturva-asiat, sillä verkkokaupat käsittelevät usein herkkäluonteista dataa (henkilötiedot, luottokorttitiedot jne.).

Relaatiotietokanta on tietokantamallina erittäin laajasti käytetty. Se on perinteisesti hyvin tuettu (esim. Oraclen tukipalvelut), ja tarjoaa hyvä tietoturvan sekä muita aikojen saatossa kehiteltyjä toimintoja. Sen huonoja puolia on kuitenkin jäykkä rakenne ja huono skaalautuvuus, joihin NoSQL-tietokantatuotteet tarjoavat taas ratkaisuja.

NoSQL-tietokannat ovat uudempi avoimen lähdekoodin ratkaisu. Tämä tekee kyseisistä tietokannoista helpommin käytettäviä (budjetin ei tarvitse olla iso), mutta tuo myös omat ongelmansa: siinä missä perinteisille relaatiotietokannoille löytyy etenkin business-versioissa omat tukihenkilöt (esim. Oraclen ratkaisu), NoSQL-kannat ovat, avoimen lähdekoodin tietokantoja, eli tuki on vähäistä tai olematonta, ja usein kannasta löytyvä dokumentaatio voi olla hieman huteraa. NoSQL-kannat ovat kuitenkin erittäin

suosittu kantaratkaisu osittain juuri ilmaisuuden vuoksi ja osittain siksi, että ne tarjoavat laajemmin eri malleja tallentaa ja näyttää tietoa.

Työssä on toteutettu laaja katsaus eri tietokantoihin, erityisesti NoSQL-kantoihin, niiden toimintaan ja parhaimpaan käyttöön. Tämän jälkeen on valittu sopiva NoSQL-kanta, jota on mallinnettu ja sen käyttöön on muodostettu kyselyitä, jotta nähdään, kuinka sitä voi mahdollisesti käyttää verkkokaupan tietokantana. Kyselyiden avulla on vertailtu relaatio- ja NoSQL-kannan suorituskykyjä. Käytännössä halutaan siis selvittää, onko olemassa olevan verkkokaupan kantana järkevää käyttää NoSQL-kantaa.

Iso osa alan kirjallisuudesta on olemassa käytännössä ainoastaan englanniksi. Tämä tarkoittaa sitä, että suomeksi kirjoittaminen ja asioiden saattaminen ymmärrettävään tilaan voi olla hankalaa. Työssä on tämän vuoksi tehty niin, että mikäli sopivaa, jo käytössä olevaa termiä ei ole löytynyt, englanninkieliset termit on käännetty itse suomeksi. Työssä käytetään sanoja tietokanta ja kanta tarkoittamaan samaa asiaa. Samoin relaatiotietokanta ja relaatiokanta tarkoittavat samaa tässä tapauksessa.

2 Relaatiotietokannat

Tässä luvussa käydään läpi relaatiokantojen rakennetta ja toimintaa. Tämän lisäksi aliluvussa 2.2 tarkastellaan muutamia tietokantojen kannalta tärkeitä käsitteitä.

2.1 Mikä on relaatiotietokanta?

2.1.1 Relaatiokantojen historiaa

Tietokannat eri muodoissa ovat olleet käytössä 1960-luvulta lähtien. Tietokoneet yleistyivät tänä aikana, ja useat yksityiset firmat siirtyivät sähköiseen datan

säilömiseen. Seuraavalla vuosikymmenellä kehitettiin kaksi eri systeemiä, jotka myöhemmin poikivat käytännössä kaikki nykyiset SQL-kannat. Suurin 1970-luvun edistys tietokannoissa oli kuitenkin Dr. E. F. Coddin julkaisema paperi, jossa hän kehitteli relaatiomallin (aliluku 2.1.2). Paperin myötä relaatiotietokannat alkoivat yleistyä.

1980-luvulla kehitettiin itse SQL-kieli, jota tänäkin päivänä käytetään erittäin laajasti tietokannoissa. Tällä vuosikymmenellä relaatiotietokannat saivat varsinaisesti tuulta alleen, ja niistä tulikin suosituin datansäilytysmalli. 1990-luvulla tietokantojen tarve ja käyttö lähti reippaaseen nousuun internetin yleistymisen myötä. Tällä vuosisadalla tietokantojen tarve on kasvanut entisestään, ja tulee todennäköisesti edelleen kasvamaan. (A Timeline of Database History 2017.)

Samoihin aikoihin, kun internet ja relaatiotietokannat lähtivät nousuun, markkinoille alkoi tulla NoSQL-tietokantoja. Ne kehitettiin lähinnä vastineeksi internetin kasvavalle nopeudentarpeelle; oli myös entistä tärkeämpää pystyä käsittelemään dataa, joka ei suoraan sopinut relaatiomallin tietokantoihin. NoSQL-kannat ovat kuitenkin yleistyneet vasta tällä vuosituhannella, ja ne jatkavatkin edelleen kasvuaan.

2.1.2 Relaatiomalli

Yllä on mainittu Codd ja hänen kehittämänsä relaatiomalli. Avataan asiaa hiukan.

Hakkarainen on kirjassaan *Oracle-tietokannan tehokas hallinta* kääntänyt/määritellyt relaatiomallin näin:

> Relaatiomalli koostuu:
>
> - Joukosta relaatioita (tauluja), joihin tieto tallennetaan

- Määritellyistä operaatioista, joiden avulla käyttäjät voivat muokata tietokannan tietoja ja rakenteita

- Tiedon eheyssäännöistä. (Hakkarainen 2011: 57.)

Tämä malli antaa pohjan relaatiotietokannoille; relaatiotietokanta on tietokanta, joka noudattaa relaatiomallia.

Codd kehitti tarkalleen ottaen 12 sääntöä, joista on ylempänä olevassa lainauksessa esitelty ainoastaan kolme keskeisintä. Tietokanta lasketaan periaatteessa relaatiotietokannaksi, jos se noudattaa kahta Coddin säännöistä. Muina sääntöinä voisi mainita esimerkiksi null-arvojen systemaattinen käsittely ja taattu käsiksi pääsy tietoon (Codd's 12 Rules 2018).

2.1.3 Rakenne

Relaatiotietokanta koostuu yhdestä tai useammasta taulusta, jotka noudattavat relaatiomallin sääntöjä yksin ja yhdessä. Taulu taas sisältää rivejä ja kolumneja, joihin tiedot on tallennettu. Jokaisella tietueella tulee olla oma avain, joilla ne voi tunnistaa. Tauluissa voi myös olla erilaisia rajoitteita ja liipaisimia, joilla säädellään tiedon muokkaamista ja tarkastelua. Myös indeksit ovat tärkeä osa etenkin suuria tietokantoja.

Skeema on taas suurempi kokonaisuus, joka käytännössä määrittelee tietokannan ja sen, kuinka data liittyy toisiinsa sekä joukon muuta tietoa. SQL-kannoissa skeema on pakko määrittää ennen kuin itse tietokanta edes tehdään. Ongelmaksi usein muodostuukin se, että jos kantaa halutaan jälkikäteen muuttaa, joudutaan mahdollisesti siirtämään koko kanta uuteen paikkaan, jotta uusi skeema saadaan toteutettua. (DBMS - Database Schemas 2018.)

On tärkeää pitää mielessä relaatiotietokannan rakenne, kun ruvetaan käymään läpi sen eroja NoSQL-kantojen kanssa. Rakenne on tärkeä myös, kun mietitään tietokannan muuntamista NoSQL-kannaksi, sillä kannat ovat todennäköisesti oleellisesti erilaisia, eikä läheskään kaikista löydy tarvittavia ja haluttuja toimintoja ja rakenteita.

2.1.4 Relaatiokannan hyvät ja huonot puolet

Relaatiotietokannoilla on monta hyvää puolta. Ensinnäkin ne ovat useimmiten hyvin tuotettuja ja vakaita kantoja, eli niitä on turvallista käyttää. Niiden tietoturva on usein myös aika tiukka. Yksi vahva puoli on myös SQL-kielen helppous ja laaja käyttö: kaikki relaatiokannat käyttävät sitä, jolloin kantaa vaihtaessa ei tule ongelmia kielen syntaksin kanssa.

Kannat ovat kuitenkin enimmäkseen maksullisia (poikkeuksena MySQL ja PostgreSQL), mikä voi rajoittaa etenkin pienemmän yrityksen toimintaa ja mahdollisuuksia. Myös aliluvussa 2.1.3 mainittu skeema ja sen jäykkyys voi olla huono puoli; kannan rakenteen pitää olla selvillä jo hyvin alkuvaiheessa, mikä ei sovi esimerkiksi ketterään kehitykseen kunnolla. Skaalautuvuudesta on tullut parin viimeisen vuosikymmenen aikana ongelma. Sosiaalisen median nousun myötä on yhä enemmän tarvetta tietokannalle, joka kykenee skaalautumaan isoihin tietomassoihin, mikä aiheuttaa ongelmia relaatiokannoissa. Enimmäkseen kyseessä on kuitenkin joukko vakaita ja runsaasti käytettyjä kantoja, jotka ova suosittuja hyvästä syystä.

2.2 Muutamia tarpeellisia asioita

2.2.1 SQL

SQL eli Structured Query Language on kieli, jota käytetään relaatiokantojen kanssa. Lähes kaikki relaatiotietokannat tukevat SQL:ää, eli se on hyvin hyödyllinen ohjelmoijan tai ylläpitäjän kantilta. SQL-kieltä käytetään kolmessa eri tarkoituksessa:

- DCL: Data Control Language. Tarkoittaa grant- ja revoke-käskyjä. Käytetään siis käyttäjien lupien hallitsemiseen.

- DDL: Data Definition Language. Eli siis create, alter, drop jne. käskyjä. Käytännössä taulun hallintaan tietokannan sisällä.

- DML: Data Manipulation Language. Tarkoittaa komentoja select, insert, delete jne. Käytetään tietueiden hallintaan taulun sisällä. (Structured Query Language (SQL) 2016.)

2.2.2 ACID

ACID on tietokantoihin liittyvät lyhenne. Lyhenne liittyy tietokannan käsittelyn ominaisuuksiin. (Hakkarainen 2011: 63.) Tässä luvussa käydään läpi, mitä lyhenne tarkoittaa ja avataan sitä esimerkein.

Ensimmäinen kirjain **A** tulee sanasta atomicity eli **atomisuus**. Tämä tarkoittaa sitä, että kun transaktio aloitetaan, se suoritetaan myös kokonaan loppuun. Ellei sitä jostain syystä voi suorittaa loppuun, koko transaktio jätetään suorittamatta. Otetaan esimerkiksi vaikka transaktio, jossa ensin vähennetään tililtä rahaa ja samalla lisätään rahaa toiselle tilille. Jos transaktion ensimmäinen osa epäonnistuu ja toinen onnistuu, kannassa on silloin epäjohdonmukaisuutta. Se voi aiheuttaa hyvinkin suuria ongelmia, etenkin jos kyseessä on vaikka raha tai kaupan inventaario.

Tästä päästäänkin kätevästi seuraavaan kirjaimeen. **C**, consistency, eli **eheys** tarkoittaa sitä, että kun tietokantaan tehdään muutoksia, se pysyy eheänä muutoksesta huolimatta. Jos siis tietokantaa esimerkiksi päivitetään, päivityksen tulee noudattaa tietokannan asettamia eheyssääntöjä (liipaisimet jne.), jotta se voidaan suorittaa. Mikäli päivitys voidaan suorittaa, tietokannan tila muuttuu, mutta pysyy siis edelleen eheänä.

Kolmas kirjain on **I**. Isolation, eli **eristyneisyys** tässä tapauksessa tarkoittaa sitä, että kun tietokantaan tehdään transaktiota, sen aiheuttamat muutokset eivät näy muille transaktioille ennen kuin se on vahvistettu (eli siis annettu commit-käsky). Tämä voi aiheuttaa ongelmia etenkin silloin, jos kaksi käyttäjää yrittää samanaikaisesti muokata jotain tietuetta. Eri tietokannoilla on olemassa eri ratkaisuja tähän ongelmaan: on esimerkiksi mahdollista lukita kyseinen tietue (tai tarvittaessa koko rivi) ensimmäiselle transaktiolle, ja vapauttaa lukitus, kun commit on tehty. Näin seuraava jonossa oleva muokkaaja ei voi samanaikaisesti tehdä muutoksia, mikä voisi rikkoa eheyttä, vaan joutuu odottamaan edellistä ja on vasta sen jälkeen vapaa tekemään omat muutoksensa.

Viimeisenä on **D**: durability, eli **pysyvyys**. Kun tietokantaan on suoritettu jokin tapahtuma, se on pysyvä. Eli jos tietokannalle vaikka tapahtuu jotain, minkä vuoksi se palaa tapahtumaa edeltävään aikaan, tai joku epähuomiossa poistaa tehdyt muutokset, tapahtumasta pitää löytyä jostain jälki, jonka avulla se voidaan palauttaa takaisin. Hakkaraisen (2011: 63) mukaan Oraclen tietokannat hoitavat pysyvyyden käyttämällä esimerkiksi erilaisia tapahtumalokeja.

ACID on SQL-tietokantojen kannalta erittäin tärkeä lyhenne. Se tulee tärkeäksi etenkin, kun puhutaan kaupallisessa käytössä olevista tietokannoista. Jos pankin tietokanta ei noudata ACID-periaatteita, kyseinen pankki tuskin on pitkään pystyssä.

NoSQL-kannat sen sijaan lähes kategorisesti ovat kieltäytyneet ACID-periaatteen käytöstä. Poikkeuksiakin tähän on (kuten myöhemmin tulee ilmi), mutta aivan viime vuosiin asti ne eivät ole periaatteita SQL-kantojen tapaan noudattaneet. Tämä tietenkin tarkoittaa esimerkiksi sitä, että monisäie-ajo voi olla liian monimutkaista ja riskialtista NoSQL-kannoille, mikä taas rajoittaa toimivuutta. Mutta toisaalta periaatteet tuovat mukanaan tiettyä jäykkyyttä kannan toimintaan ja käyttöön, mikä voi myös olla haitallista. ACID-periaatteiden noudattamisen tärkeys onkin yksi asia, joka tulee ottaa huomioon tietokantaa valitessa.

2.2.3 High Availability

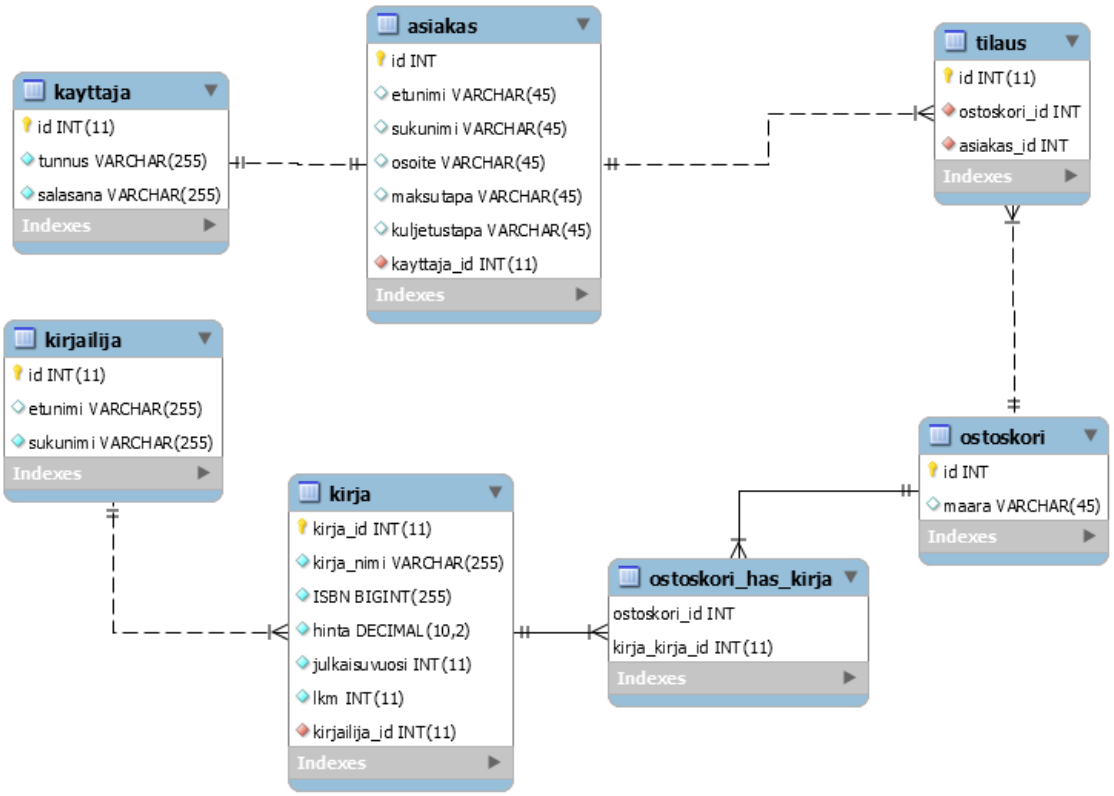
High Availability eli HA on termi, jota käytetään tietokannoista, jotka ovat kestäviä ja pystyvät toimimaan pitkiä aikoja ilman vikoja. Tämä on erityisen tärkeää tietokannoissa, koska niiden yhteydessä oleva sovellus on useimmiten riippuvainen kannan toiminnasta. HA on syytä ottaa huomioon jo kantaa suunnitellessa, koska jälkikäteen sovelluksien muokkaaminen voi olla hankalaa. (High Availability (HA) 2016.)

Relaatiotietokannoissa HA on useimmiten toteutettu yhdistelemällä useita eri teknologioita: RAID-levyt, Data Guard (Oracle) jne. NoSQL-kantojen joustavuus ja yleinen skaalautuvuus antaa paremmat mahdollisuudet HA:n toteutukseen. Kummassakin tyypissä paljon riippuu kuitenkin siitä, miten kannan arkkitehtuuri on toteutettu, eli palataan taas suunnitteluvaiheeseen. Relaatiokannoissa pitää tehdä valintoja asiaan liittyen jo aikaisessa vaiheessa, koska erilaiset HA:n mahdollistavat ratkaisut ovat yleensä maksullisia. NoSQL-kantojen kanssa tilanne on helpompi, sillä kuten luvussa 4 tullaan näkemään, ne ovat usein hyvin muokattavissa koko elinkaaren ajan, ja mikä toisinaan vielä tärkeämpää, ne ovat myös ilmaisia.

3 Relaatiotietokannan rakenne ja tarkasteltavat lauseet

3.1 Kannan rakenne ja syy lauseiden valintaan

Kuten johdannossa kerrottiin, tarkoituksena on katsoa, olisiko mahdollista siirtyä käyttämään relaatiokannan sijasta NoSQL-kantaa. Tässä työssä ei siirtymää tehdä, vaan keskitytään tarkastelemaan mahdollisia ratkaisuja ja niiden eri puolia. Tätä varten tulee tietää, mitä kannalta halutaan ja kuinka sitä on jo olemassa olevassa sovelluksessa käytetty. Kanta on tehty MySQL-tiedonhallintaohjelmistolla, eli kyseessä siis on relaatiomallin kanta. Liitteessä 1 on kyseisen kannan taulujen vedokset, eli sieltä näkyy, miten taulut on muodostettu ja miten ne ovat yhteydessä toisiinsa. Tämä on tärkeää tyypillisimpien SQL-lauseiden etsinnässä. Kannan rakenne näkyy myös kuvassa 1.



Kuva 1. SQL-kannan rakenne.

Kuten kuvasta näkyy, kayttaja-taulu on yhteydessä tilaukset-tauluun, kun taas kirja- ja kirjailija-taulut ovat yhteydessä toisiinsa (1-M-suhde). Kirjan ja kirjailijan suhde on määritelty niin, että yhdellä kirjailijalla voi olla monta kirjaa, mutta kirjalla voi olla vain yksi kirjailija. Samoin käyttäjällä voi olla useita tilauksia, mutta tilauksella vain yksi käyttäjä. Nämä päätökset tehtiin alunperin helpottamaan kantojen käsittelyä sovelluksessa, eikä niitä ole ole muutettu myöhemmin, mutta ne on otettava huomioon NoSQL-kannan kanssa. Keltaisella avaimella merkityt parametrit tauluissa ovat niiden avaimet, ja kirja-taulussa oleva punainen parametri kirjailija_id merkkaa vierasavainta, joka liittyy kirjailija-tauluun.

Kanta on siis kohtuullisen yksinkertainen, mikä sopii hyvin tähän työhön. Todellisessa, täysin toimivassa verkkokaupassa voi olla huomattavastikin monimutkaisempi kantarakenne. Tässä ei esimerkiksi käsitellä tarkemmin maksuvälineitä tai toimitustapoja, jotka esiintyvät tällä hetkellä ainoastaan asiakas-taulussa yhtenä kenttänä. Kantaa on todennäköisesti tarvetta laajentaa jossain vaiheessa ainakin edellä mainittujen seikkojen osalta, eli siis parametreille maksutapa ja kuljetustapa pitäisi lisätä omat taulut.. Uusien taulujen lisääminen ja käyttö vaikuttaa siihen, kuinka usein kanta tarvitaan ja kuinka monimutkaisia kyselyitä siihen täytyy suorittaa. Työssä jätetään tietoisesti tilaukset-taulu huomiotta, koska sen käyttö ei ole aivan niin runsasta kuin muiden taulujen, eikä se ole tällä hetkellä vielä täysin valmis sovelluksen kantilta katsottuna. Aloitusvaiheessa kannassa ei ole käytetty indeksointia, mutta sen käyttöä tarkastellaan luvussa 6.3.1.

Tarkoituksena on löytää verkkokaupassa toistuvasti käytetyt SQL-lauseet ja ottaa ne ylös. Niitä tarvitaan, kun tehdään NoSQL-mallinnusta ja vertailuja. Koska useimmin käytetyt lauseet sisältävät usein tärkeimmän tiedon (esimerkiksi asiakkaan tiedot tilausta varten), ne on järkevintä ottaa lähempään tarkasteluun. Niiden suorituskyky voi vaikuttaa hyvinkin suuresti verkkokaupan toimintaan; kukaan ei halua odotella hakutuloksiaan montaa minuuttia. Nämä kyselyt voivat sisältää myös hyvin arkaluontoista tietoa, jolloin tietoturvaan tulee panostaa. Myös tältä kantilta on pyritty valitsemaan muutama, mahdollisimman hyvin verkkokauppaa ja sen tietokantaa kuvaava SQL-lause.

3.2 Valitut lauseet

Tässä luvussa on neljä SQL-lausetta, joita käytetään verkkokaupassa runsaasti. Ne halutaan ottaa vertailuun ja mallintamiseen. Lauseita tarkastellessa täytyy huomata, että niissä on tietoturvan vuoksi käytetty valmisteltuja kyselyitä, mikä näkyy siitä, että annettujen parametrien edessä on kaksoispiste. Valmistelluissa kyselyissä SQL-lause annetaan koodissa ensin ja siihen syötetään vasta myöhemmin käyttäjän antama syöte: näin voidaan tehdä erilaisia varmistuksia ja siivota syötettä.

Ensimmäisessä SQL-kyselyssä (esimerkkikoodi 1) valitaan käyttäjän antaman syötteen perusteella näytettäväksi kahdesta eri taulusta (kirja ja kirjailija) tulokset, jotka vastaavat suunnilleen syötettä. Tämä tarkoittaa sitä, että syötteessä voi käyttää %-operaattoria, jota voi käyttää kun ei tiedetä kaikkia syötteeseen kuuluvia merkkejä (niin sanottu villi kortti).

```
SELECT ISBN, kirja_nimi, etunimi, sukunimi, julkaisuvuosi, hinta
FROM kirja INNER JOIN kirjailija
ON kirja.kirjailija_id=kirjailija.id
WHERE kirja_nimi LIKE :search OR sukunimi LIKE :search;
```

Esimerkkikoodi 1. Valitaan tietoja kahdesta eri taulusta.

Esimerkkikoodissa 2 halutaan vain näyttää kaikki kahden taulun rivit käyttäjälle. Tämä näyttää siis kaikki verkkokaupan tuotteet yhdessä näkymässä.

```
SELECT ISBN, kirja_nimi, etunimi, sukunimi, julkaisuvuosi, hinta
FROM kirja INNER JOIN kirjailija
ON kirja.kirjailija_id=kirjailija.id;
```

Esimerkkikoodi 2. Valitaan kaikki kahden taulun rivit.

Seuraavassa olevassa SQL-lauseessa (esimerkkikoodi 3) valitaan käyttäjä taulusta nimeltä kayttaja. Tämä tehdään tekemällä haku käyttäjän syöttämällä tunnuksella. Jälleen kerran on käytetty valmisteltua kyselyä.

```
SELECT * FROM kayttaja WHERE tunnus= :tunnus;
```

Esimerkkikoodi 3. Valitaan käyttäjä taulusta.

Esimerkkikoodissa 4 halutaan lisätä uusi käyttäjä kayttaja-tauluun. Tässä on jälleen kerran käytetty valmisteltua kyselyä, jotta vältytään mahdollisilta SQL-injektioilta. Tämä lause ei ole käytössä kovin usein, mutta se on siitä huolimatta mielenkiintoinen tutkailtava, koska esimerkkikoodit 1-3 ovat käytännössä vain hyvin yksinkertaisia hakuja.

```
INSERT INTO kayttaja (id, tunnus, salasana) VALUES
(0, :tunnus, :pwd);
```

Esimerkkikoodi 4. Lisätään uusi käyttäjä.

Nämä SQL-lauseet ovat tyypillisiä verkkokauppasovelluksessa käytettäviä lauseita. Muitakin tarvitaan, kuten voi päätetllä kuvassa 1 nähtävästä kannan rakenteesta, mutta näillä saadaan jo jonkinlainen käsitys yleisistä vasteajoista. Näissä SQL-lauseissa on jonkin verran asioita, jotka tulee NoSQL-mallinnusta tehtäessä ratkaista. Esimerkiksi join-operaattorin ja valmisteltujen kyselyiden käyttö. Näistä ja muusta puhutaan tarkemmin aliluvussa 5.2.

4 NoSQL-tietokannat

4.1 Käyttötarkoitukset

Tietokannan valinta on aina monivaiheinen ja jopa monimutkainen prosessi. Siinä tulee ottaa monia eri asioita huomioon; jo ennen kantatyypin valitsemista on syytä käydä omat kriteerit tarkasti läpi. Tarvitsenko joustavuutta rakenteeseen? Kuinka iso kanta

tulee olemaan? Haluanko valmiin paketin vai olenko valmis tekemään itse ison osan ja yhdistelemään tarvittavat loppuosat?

Mikäli kyseessä on suuria tietomääriä sisältävä tietokanta, jonka kuitenkin halutaan toimivan hyvin verkkosovelluksen kanssa, NoSQL-kantaa on todellakin syytä miettiä. (Why NoSQL Database? 2018.) Aina on myös mahdollista käyttää relaatiokannan rinnalla NoSQL-kantaa, jos sovellus ja vaatimukset siitä hyötyvät. Esimerkkiratkaisuna voisi olla sovellus, jossa käyttäjätunnukset säilytetään relaatiotietokannassa (parempi tietoturva ja hallinta) ja kaikki muu on NoSQL-kannassa (joustavampi, parempi suorituskyky). Näin on mahdollista saada kummankin kantatyypin hyvät ominaisuudet sovelluksen parhaan toiminnan takaamiseksi.

Tässä luvussa tullaan käymään läpi, mikä oikeastaan on NoSQL-kanta. Katsotaan myös sen rakennetta ja eri tyyppejä sekä sitä, miten NoSQL-kanta eroaa SQL-kannasta. Käydään myös läpi kolme eri kantaa, joista sitten valitaan verkkokauppaan sopivin.

4.2 Mikä on NoSQL-tietokanta?

Tässä vaiheessa on hyvä käydä läpi, mikä NoSQL-tietokanta oikeastaan on. NoSQL- eli Not(-only) Structured Query Language on DBMS (Database Management System), joka ei noudata relaatiomallia. Osa nykyaikaisista NoSQL-kannoista voi noudattaa esimerkiksi yhtä relaatiosäännöistä, mutta ne eivät silti ole relaatiomallin mukaisia. (NoSQL Databases Explained 2018.) Tunnusomaista NoSQL-kannoille on myös niiden suhtautuminen ACID-sääntöihin (katso aliluku 2.2.2 säännöistä); käytännössä kaikki ovat kiertäneet ne kaukaa.

Perustavanlaatuisin ero relaatiomallin kantoihin on se, miten tietoa säilytetään ja tarkastellaan. Coddin relaatiomalli kertoo, että relaatiomallissa on relaatioita (tauluja), joissa tieto on. Tämä ei NoSQL-kannoissa pidä ollenkaan (tai vain osittain) paikkaansa.

NoSQL-kannoissa tieto voidaan säilyttää usealla eri tavalla (mm. dokumentteina), joita käydään tarkemmin läpi luvussa 4.5. Kantojen välillä on myös useita muita eroja (mm. skaalautuvuus, skeemat). Näitä eroja käydään myös läpi tarkemmin alempana.

NoSQL-kannat käyttävät tietojen käsittelyyn ja ylläpitoon JSON:ia. JSON eli JavaScript Object Notation on nimensä mukaisesti JavaScript-pohjainen kieli, jota käytetään enimmäkseen verkkosovelluksissa. Se soveltuu XML:ää paremmin tiedonsiirtoon, koska se on tekstipohjainen, nopeampi ja vie vähemmän CPU:ta. Tärkein JSON:in ominaisuus on kuitenkin sen mahdollisuus siirtää dataa epäsopivien järjestelmien välillä ilman, että se tarvitsee ylimääräisiä ohjelmistoja siirron toteuttamiseksi sujuvasti. Tätä käytettäessä on otettava tietoturva erityisen hyvin huomioon, sillä JSON on haavoittuvainen JavaScript-injektioille. (JavaScript Object Notation (JSON) 2016.) Tässä yhteydessä JSON on esillä, koska useat NoSQL-tietokannat käyttävät sitä tiedon siirtämiseen ja tallentamiseen esimerkiksi dokumentteina.

4.3 Hyvät ja huonot puolet

NoSQL-tietokanta on laaja käsite, joka sisältää suuren määrän eri laatuisia ja mallisia kantoja. Niillä on kuitenkin pääosin tiettyjä yhteisiä piirteitä, joita tässä käydään läpi.

Suurimmat NoSQL-kantojen hyödyt ovat skaalautuvuus ja mahdollisuus käsitellä suuria määriä nopeasti muuttuvaa dataa, mikä tekeekin niistä suosittuja etenkin verkkosovelluksissa. Siinä missä relaatiokannat skaalautuvat ainoastaan vertikaalisesti (niihin voi lisätä enemmän dataa), NoSQL-kannat skaalautuvat horisontaalisesti. (NoSQL Databases Explained 2018.) Tämä tarkoittaa sitä, että koska perinteisen kannan pitää mahtua kokonaan yhdelle palvelimelle, ainoa tapa lisätä siihen kapasiteettia on lisätä levyä tai vaihtaa isompaan palvelimeen. NoSQL-kannat sen sijaan mahdollistavat sen, että kanta jakautuu usealla eri palvelimelle; tästä käytetään termiä hajautus (sharding). Tämä mahdollistaa sen, että tietokanta pystyy käsittelemään todella suuria tietomääriä ilman, että sovellus edes huomaa koko asiaa.

Osa NoSQL-kannoista tukee myös automaattista hajautusta, jossa tämä prosessi tapahtuu automaattisesti ilman, että ylläpitäjän tarvitsee tehdä asialle mitään.

Seuraavana suurena etuna NoSQL-kannoilla on joustava skeema. Kuten aliluvussa 2.1.3 kerrottiin, relaatiokannoissa skeeman tulee olla määriteltynä etukäteen, ja sitä ei voi muuttaa enää datan ollessa kannassa. Näin ei kuitenkaan ole NoSQL-kannoissa, vaan lähes kaikissa on joustava skeema (tai ei skeemaa ollenkaan), mitä pystyy muuttamaan sitä mukaa kun on tarve. (NoSQL Databases Explained 2018.) Tämä sopii erittäin hyvin ketterään kehitykseen, kun kehittäjä pystyy lisämään esimerkiksi kehityksen puolivälissä kokonaan uuden kolumnin tauluun. Tällöin ei tarvitse huolehtia kannan alasajosta tai migraatioista, kuten relaatiokantojen kanssa. Joissain NoSQL-kannoissa on myös mahdollista rakentaa sisäisiä sääntöjä ja muita rakenteita, jotka muuten pitäisi toteuttaa sovelluksen puolella.

NoSQL-kannat tukevat myös hyvin laajasti replikointia, ja osa tekee sen jopa automaattisesti. Tämä tarkoittaa sitä, että pääasiallisessa käytössä oleva tietokanta kopioidaan toiselle palvelimelle tai esimerkiksi pilveen. Näin pidetään huoli siitä, että mikäli kanta jostain syystä ei ole toimintakunnossa, on olemassa varakanta, joka voidaan siirtää käyttöön tilapäisesti. (What is NoSQL? 2016.) Relaatiokannoillakin on kyseisiä ratkaisuja, mutta ne vaativat usein hyvinkin työläitä ylimääräisiä skriptejä ja levytilaa, tai vaihtoehtoisesti maksullisia ratkaisuja.

Puutteitakin NoSQL-kannoista toki löytyy. Useat, etenkin isot firmat suosivat relaatiokantoja jo ihan senkin vuoksi, että ne tarjoavat usein laajat analysointi- ja raportointimahdollisuudet. Näillä pystytään seuraamaan tietokantojen tilaa lähes livenä, ja analysointityökalut auttavat usein etsimään piileviä vikoja sekä parantamaan suorituskykyä. Nämä ovat ylläpitäjän kannalta hyvin tärkeitä funktioita.

NoSQL-kannat toteuttavat ACID-periaatteita vaihtelevasti, kuten aliluvussa 2.2.2 mainittiin. Usein esimerkiksi pysyvyys on heikolla tolalla, kun kannassa ei ole minkäänlaisia lokeja tai mitään valvontaa ylipätään. Eheys on myös edelleen joillekin kannoille ongelmallinen asia: jos kannassa ei ole minkäänlaista skeemaa tai muita

sääntöjä, esimerkiksi päivitystilanteessa voi tiedolle voi epähuomiossa tapahtua jotain, mikä rikkoo kannan. Lähtökohtaisesti NoSQL-kantojen alkuvaiheessa ACID:ia ei ole tarkoituksella käytetty, joten sen sisällyttäminen on ollut hidasta. Isommat NoSQL-kannat, kuten MongoDB, kyllä nykyisin noudattavat ACID:ia edes osittain.

NoSQL-kantojen kanssa täytyy myös suhtautua tarkemmin tietoturvaan. Etenkin verkkosovelluksissa tietoturva on isossa osassa: niissä käsitellään nykyään paljon maksutietoja ja myös muuta luottamuksellista tietoa, jonka joutuminen vääriin käsiin on vaarallista. Tiedonsiirto NoSQL-kannoissa tapahtuu pääasiallisesti JSON:lla, joka on jokseenkin haavoittuvainen injektioille, kuten aiemmin on mainittu. Tämä johtaa siihen, että kehittäjän on usein keksittävä ylimääräisiä turvakeinoja, jotta kantaan ei pääse vaarallista tietoa. Myös kannan replikointi ja säilytys usealla eri palvelimella tuo omat tietoturvahaasteensa. Mitä useammassa paikassa tietoa on, sitä todennäköisimmin joku ulkopuolinen pääsee siihen kiinni. NoSQL-kannat harvoin tarjoavat suoraan mitään mahdollisuutta kryptata tietoa, ainakaan samalla tasolla kuin relaatiotietokannat.

Kaiken kaikkiaan ylläpitäjän ja kehittäjän on siis syytä perehtyä etukäteen tarkkaan eri vaihtoehtoihin ja NoSQL-kantojen käyttöön yleensäkin.

4.4 NoSQL- ja SQL-kantojen eroavaisuudet

Tässä luvussa katsotaan ikäänkuin yhteenvetona, mitä eroa NoSQL- ja SQL-kannoilla varsinaisesti on.

Taulukko 1. NoSQL- ja SQL-kantojen eroavaisuudet.

Ominaisuudet	**SQL**	**NoSQL**
Kieli	SQL	JSON (enimmäkseen)
ACID	Kyllä	Riippuu kannasta
Tiedon tallennus	Taulu	Runsaasti ratkaisuja

Skeema	Jäykkä	Joustava/olematon
Transaktiot	Kyllä	Riippuu kannasta
HA	Kyllä	Kyllä

Taulukkoon 1 on kasattu SQL- ja NoSQL-kantojen yleisimpiä eroja. Taulukko ei missään nimessä ole kaikenkattava jo pelkästään NoSQL-kantojen laajuuden vuoksi, mutta antaa jonkinlaista kuvaa yleisistä eroista ja asioista, jotka tulee kantaa valitessa ottaa huomioon.

Kuten aiemmin on mainittu, kannat skaalautuvat eri tavoilla. Relaatiokantaa on vaikeampi skaalata muuten kuin lisäämällä laitteistoa tai tekemällä muita isoja muutoksia, kun taas NoSQL-kannat skaalautuvat pääsääntöisesti hyvin pienellä vaivalla. Aliluvussa 4.6 käydään vielä tarkemmin läpi erilaisia NoSQL-kantoja, joiden ominaisuudet voivat poiketa taulukossa olevista ominaisuuksista. Tässä on kuitenkin hyvä yleiskatsaus tärkeimmistä/suurimmista eroista mallien välillä.

4.5 NoSQL-tietokantatyypit

NoSQL-tietokannat voidaan jakaa neljään eri tyyppiin, jotka käydään seuraavissa luvuissa läpi. Täytyy kuitenkin muistaa, että koska kyseessä on hyvin suuri lajitelma eri tietokantoja, tyyppijako ei ole selvärajainen. Osa kannoista voi hyvinkin käyttää muutaman eri tyypin piirteitä, sillä niiden pointtina on olla mahdollisimman toimivia ja tehokkaita eikä vain pysyä omassa karsinassaan.

Ensimmäisenä on **dokumenttitietokanta** (document store). Dokumentilla tarkoitetaan tässä tapauksessa useimmiten XML- tai JSON-lohkoa, joka sisältää datan. Dokumentti sisältää datan tyypin kuvauksen ja itse arvon. Jokainen dokumentti voi olla samanlainen, tai kaikki voivat olla erilaisia. Kuten tapana on, skeema on tässäkin tapauksessa joustava, joten uusien datatyyppien lisääminen dokumenttikantaan on helppoa: lisätään vain uusi dokumentti. (Document Databases Explained 2018.) Koska

jokainen dokumentti on oma yksikkönsä, niiden tallentaminen ja koko kannan jakaminen eri servereille on yksinkertaista (Document Database 2017).

Dokumentit voidaan kasata yhteen, jolloin niille voi suorittaa kyselyitä helpommin. Dokumenttikanta voi olla myös key-value-kanta (kts. seuraavana), jolloin sen rakenne ja kyselyiden suorittaminen on hieman erilainen. Useilla dokumenttikannoilla on hyvin voimakas querty engine, eli kyselyiden käsittelyyn tarkoitettu moottori, ja ne käyttävät indeksointia tehokkaasti. Tämä tekee niistä hyvin suosittuja ja suositeltavia, jos työskennellään isojen datamäärien kanssa. (Document Databases Explained 2018.)

Toisena voidaan mainita **avain-arvo-tietotakanta** (key-value store). Tässä tieto tallennetaan array-taulukoista tutussa muodossa, eli jokaiselle datalla on avain (key), johon liittyy arvo (value). Kuten arrayssa, yksi avain voi sisältää toisen arrayn tai listan tarvittaessa. Avain-arvo-kannoilla ei tarvitse olla skeemaa ollenkaan, ja niihin tallennettava tieto voi olla käytännössä missä muodossa tahansa. Tämä tekee kyseisestä kantatyypistä kaikkein joustavimman, koska sovellus voi täysin määritellä kaiken dataan liittyvän. (Key Value Databases Explained 2017.)

Avain-arvo-kannan suorituskyky on todella suuri, sillä se ei vaadi esimerkiksi join-operaatioita, koska kaikki tieto saadaan avaimen avulla. Tämä tosin tarkoittaa sitä, että avaimen valitsemisessa tulee olla tarkkana, koska se tosiaan on ainoa keino päästää dataan käsiksi; jos halutaan käyttää useita eri vaihtoehtoja haussa, tämä ei onnistu. Esimerkkinä käyttäjä haluaa etsiä kirjaa kirjan tai kirjailijan nimellä, mutta avaimena toimiikin ISBN; tämä haku ei tule toimimaan. Mutta jos kyseessä on vaikka kanta käyttäjistä, ja avaimena on käyttäjätunnus, tämä kantatyyppi on erityisen hyvä valinta.

Sarakeperhetietokanta (wide-column store) on myös käytössä. Siinä on nimensä mukaisesti sarakkeita, mutta myös rivejä ja tauluja, eli se äkkiseltään katsottuna vaikuttaa relaatiokantojen mukaiselta. Erona on kuitenkin se, että sarakkeita ei määritellä skeemassa, vaan ne ovat jokaiselle riville omat. Tätä voi ajatella ikäänkuin avain-arvo-kantana. Jokainen sarake siis muodostuu avaimesta ja siihen liittyvästä arvosta.

Sarakeperhe on myös tärkeä termi: jokainen rivi siinä vastaa yhtä saraketta, jolla on avain ja tarvittava määrä sarakkeita tiedon sisältämiseen. Sarakkeita ei tarvitse etukäteen määrittää, mutta sarakeperhe tulee määritellä ennen käyttöönottoa, koska sillä on merkitystä datan tallennuksen kannalta.

Sarakeperhekannan idea käytännössä siis on säilöä avain-arvo-pareja tauluun siten, että jokaista avainta vastaa sarake, jossa tieto sijaitsee. Kuten avain-arvo-kannan kanssa, avaimen arvo voi olla myös toinen avain-arvo-pari. Tämä ratkaisu, kuten aiemmatkin, on hyvin skaalautuva, käyttää indeksejä tarvittaessa ja soveltuu erittäin hyvin suurien datamäärien tallentamiseen ja hallintaan. (Guide to Wide-Column Store 2017.)

Viimeisenä on **verkkotietokanta** (graph store). Se on yksi nousevia tähtiä tietokantojen alalla. Verkkotietokannat koostuvat noodeista, jotka sisältävät avain-arvo-pareja ja suhteista, jotka yhdistävät noodeja ja joilla on suunta. Suhteet kertovat, miten noodit yhdistyvät, ja voivat myös sisältää tietoa.

Tätä tyyppiä käytetään, kun halutaan saada selville ja tallentaa monimutkaisia suhteita sisältäviä tietokantoja. Tätä kantatyyppiä käytettiin esimerkiksi Panaman papereiden selvittelyssä. Eli kun avainten sisältämät arvot liittyvät toisiinsa, on syytä harkita tämän kannan käyttöä. Verkkotietokannant käyttävät enimmäkseen SQL:n kaltaisia kieliä, esim. Crypto ja Gremlin. (Getting Started With Graph Databases 2018.)

Kuten voidaan nähdä, NoSQL-kantatyypit ovat hyvin erilaisia. Tämä tarkoittaa sitä, että lähes joka tarpeeseen löytyy kanta. Noin yhteenvetona voidaan esittää seuraava erottelu: verkkokaupan ostoskoriin kannattaa käyttää avain-arvo-kantaa, tuoteinfoa sisältävään kantaan dokumenttikantaa ja kun halutaan selvittää, kuinka joku liikkuu pisteestä 1 pisteeseen 2, käytetään verkkotietokantaa. Seuraavassa aliluvussa esitellään kolme eri NoSQL-kantaa, jotka ovat kaikki omalta osaltaan erilaisia ja suosittuja.

4.6 Eri NoSQL-kantoja ja niiden erityispiirteitä

4.6.1 MongoDB

MongoDB (What is MongoDB? 2018) on vuonna 2009 käyttöön tullut avoimen lähdekoodin NoSQL-tietokanta. Tätä kirjoittaessa viimeisin versio on 4.0.1, joka on ilmestynyt kesäkuussa 2018. Kyseessä on dokumenttitietokanta, eli se tallentaa datan tässä tapauksessa JSON-dokumenttien kaltaisina BSON-dokumentteina, jotka ovat binäärisiä JSON:eja, eli käytännössä se mahdollistaa suuremman lajin eri datatyyppejä.

MongoDB:stä löytyy kaikki NoSQL-kannan hyvät puolet: se skaalautuu hajautusperiaatteiden mukaan automaattisesti, sen skeema on joustava ja suorituskyky on hyvä. HA on hajautuksen takia myös olemassa, ja se tukee myös indeksointia. (What is MongoDB? 2018.)

Kanta noudattaa ACID-periaatteita yksittäisellä dokumenttitasolla, mikä on yleistä dokumenttikannoissa, mutta vasta versiosta 4.0 eteenpäin periaatteet ovat tulleet mukaan myös kokoelmatasolla. Se käyttää JavaScriptiä dokumenttien muodostamiseen ja tiedon lisäämiseen, mutta esimerkiksi PHP:tä on myös täysin mahdollista käyttää sovelluksen puolella, kuten monia muitakin kieliä.

Kanta tarjoaa myös monia eri vaihtoehtoja tietoturvallisuuden alueella: TLS/SSL, eri autentikaatiomallit, tiedon kryptaaminen eri vaiheissa jne. MongoDB ottaa myös huomioon samanaikaisesti tapahtuvat transaktiot käyttämällä lukkoja relaatiokantojen tapaan. Viimeisin versio tarjoaa myös rajoitetusti eri valvontamahdollisuuksia, jotka yleensä NoSQL-kannoista puuttuvat.

MongoDB:n voi ottaa käyttöön muutamalla eri tavalla. Helpoin on ottaa valmiiksi pilvipalvelussa hostattu Atlas, mutta sen voi myös asentaa paikallisesti omille

palvelimille. Jälkimäisessä tapauksessa on mahdollista ottaa käyttöön myös tiedon kompressointi, mikä vähentää käytettävää levytilaa.

Kaiken kaikkiaan MongoDB on hyvä valinta kannaksi, mikä näkyy myös tilastoissa: se on yksi suosituimpia NoSQL-kantoja tällä hetkellä. Se tarjoaa erinomaisen suorityskyvyn ja helpon käytön, ja etenkin tällaisessa pienemmässä sovelluskäytössä sen tarjoamat suojausominaisuudet ja kaikki muut ominaisuudet vaikuttavat riittäviltä.

4.6.2 RavenDB

RavenDB (RavenDB 2018) on noin kymmenen vuotta vanha avoimen lähdekoodin NoSQL-dokumenttitietokanta. Sen tällä hetkellä viimeisin versio on 4.0. RavenDB on poikkeaa muutamilla tavoilla ”perinteisestä” NoSQL-kannasta. Ensinnäkin se on alusta asti noudattanut ACID-periaatteita. Toiseksi se sopii hyvin käytettäväksi jo olemassa olevan SQL-kannan kanssa, eli siis se voidaan lisätä päälle ikään kuin kerrokseksi.

Tietokanta käyttää omaa RQL-kieltä (Raven Query Language), joka vastaa hyvin pitkälti SQL-kieltä, eli se on helppo ottaa käyttöön, jos SQL on tuttu. RQL käsittelee myös kaikki kyselyt (DDL jne.) SQL-kyselyiden tavoin. Kuten MongoDB, RavenDB mahdollistaa myös indeksoinnin ja tarjoaa hyvät HA-mahdollisuudet.

MongoDB:stä eroten RavenDB tarjoaa myös eri mahdollisuuksia valvontaan ja analysointiin. Myös klusterointi on tuettu. (RavenDB 2018.) Skeema on kuitenkin edelleen joustava, ja hajautus on toiminnassa. Kanta tukee useiden eri kielien käyttöä, joten sen käyttöönotto eri sovelluksissa ei ole mikään ongelma. Kaikki yleisimmät kielet ainakin on tuettu (JS, PHP), ja kuten MongoDB:lle, myös Ravenille löytyy kolmansien osapuolien laatimia skriptejä kielivalikoiman lisäämiseksi.

RavenDB tarjoaa kolme eri tasoa, ja erillisen ainoastaan kehitykseen tarkoitetun tason, joissa kaikissa on vähän eri tavalla ominaisuuksia. Ensimmäinen taso on ilmainen, ja se tarjoaa kaikki perusasiat; klusterioinnin (rajoitettu määrä), sertifikaatit, rajaton määrä

kantoja, kryptaaminen. Seuraavat kaksi tasoa ovat maksullisia ja tarjoavat enemmän vaihtoehtoja: mm. rajattoman klusteroinnin, service-level agreementin, paremmat replikaatio- ja varmistusmahdollisuudet. Developer tarjoaa osia maksullisista versioista, mutta ei nimensä mukaisesti ole käytettävissä tuotantokannoissa. Se on kuitenkin hyvä ratkaisu, jos haluaa testata kantaa ennen käyttöönottoa. (RavenDB Product 2018.)

Kaiken kaikkiaan RavenDB on hyvä vaihtoehto, jos on tottunut käyttämään relaatiokantoja, ja haluaa samankaltaisen kokemuksen edelleen, mutta kuitenkin NoSQL:n tarjoamat edut. Parempien versioiden maksullisuus on kuitenkin pieni miinus, mutta toisaalta suuremmassa tuotantokäytössä se tulee todennäköisesti silti halvemmaksi kuin perinteinen relaatiokantaratkaisu.

4.6.3 Redis

Redis (Redis FAQ 2018) on ollut olemassa vuodesta 2009. Kirjoitushetkellä viimeisin versio on 4.0.10. Se on avoimen lähdekoodin avain-arvo-tietokanta. Redis käyttää JSON:ia tiedon siirtoon. Kuten aiemmin mainitut kannat, Redis mahdollistaa myös klusteroinnin (Redis Cluster) ja tarjoaa HA:n (Redis Sentinel).

Redis poikkeaa joiltain osin tavanomaisesta avain-arvo-kannasta. Se käyttää termiä in-memory data structure store. Redisin FAQ:n (2018) mukaan tämä tarkoittaa sitä, että sillä voi tallentaa monimutkaisempia datatyyppejä, ja se on muistissa ja levyllä oleva tietokanta. Tämä tarkoittaa sitä, että luku- ja kirjoitusoperaatiot ovat nopeita. Toisaalta taas datan määrä on rajoitettu, mikä voi rajoittaa käyttöä suurten datamäärien kanssa.

Monimutkaisten datatyyppien tallentamisella tarkoitetaan sitä, että kun normaalissa avain-arvo-kannassa käytetään string-tyyppiä, niin avaimen kuin arvon tallentamiseen, Redisissä arvo voi olla muunlainen. Käytännössä siis puhutaan listoista, seteistä, bitmapeista ja hasheista (An introduction to Redis data types and Abstractions 2018). Kanta tukee transaktioita ja on yksisäikeinen, jonka ansiosta se noudattaa myös ACID-periaatteita.

Redis on pääasiassa tarkoitettu Linux-alustoille, mutta Windows-plugin on myös olemassa. Kanta tukee laajasti useita eri kieliä, ja sille on mahdollista myös ladata kolmansien osapuolien tekemiä skriptejä, joilla kielivalikoimaa voi laajentaa. Kuten RavenDB:n tapauksessa tavallinen, ilmainen versio tarjoaa kohtuullisen laajan kattauksen, mutta rahalla saa lisää.

Koska Redis tosiaan on avain-arvo-kanta, se ei sinällään välttämättä sovellu tähän työhön. Siitä saa arvoja ainoastaan tietämällä juuri oikean avaimen, mikä aiheuttaa rajoitteita, jos halutaan käyttää useampia hakuehtoja kyselyitä suorittaessa.

4.6.4 Muita esimerkkejä

Nämä kolme edelle käytyä kantaa ovat vain pieni osa tarjolla olevista. Ne otettiin tarkasteluun siksi, että erilaisuuksistaan huolimatta ne olivat parhaat ehdokkaat käytännön osion toteutusta varten. Muita, hyvinkin suosittuja kantoja kuitenkin löytyy. Näistä voidaan mainita esimerkiksi seuraavat:

- Cassandra ja HBase; sarakeperhekanta
- DynamoDB ja Riak; avain-arvo-kanta
- CouchDB; dokumenttikanta
- Neo4j; verkkotietokanta.

Kaikki edellä mainitut kannat ovat puhtaita NoSQL-kantoja, ja ahkerassa käytössä ympäri maailmaa. On olemassa myös useita pienempiä kantoja, mutta niiden ominaisuudet ja tuki eivät välttämättä riitä tuotantokäytössä olevan sovelluksen kantoihin, joten ne jätetään tässä huomiotta.

Puhtaiden NoSQL-kantojen lisäksi on myös muutamia sekamalleja. Ne voivat esimerkiksi olla relaatiokantoja, joissa on NoSQL-piirteitä. Yleistä niille on se, että ne noudattavat hyvin pitkälle relaatiomallia ja käyttävät SQL-kieltä, mutta tarjoavat silti joitain NoSQL-piirteitä. Tunnetuimmat näistä ovat MariaDB ja PostgreSQL. Ne ovat hyviä vaihtoehtoja, jos halutaan helposti saada kummankin mallin parhaat puolet, ja ne ovatkin kohtuullisen suosittuja.

4.6.5 Kantojen vertailua

Aliluvuissa 4.6.1-4.6.3 on esitelty tarkemmin kolme eri NoSQL-kantaa, joista valitaan työtä varten yksi. Tässä vaiheessa on vielä hyvä käydä läpi ja vertailla keskenään nämä kolme tarkempaan käsittelyyn otettua tietokantaa.

MongoDB ja RavenDB ovat dokumenttikantoja, kun taas Redis on avain-arvo-kanta. Tämä pitää ottaa huomioon, kun mietitään, miten tieto halutaan tallentaa ja saada nähtäville.

RavenDB käyttää kielenä SQL:n kaltaista kieltä, kun taas kaksi muuta käyttävät JSON:ia. Raven on myös ainoa, joka noudattaa täysin ACID-periaatteita, ja tarjoaa myös maksullisissa versioissa käyttäjätukea, joka on usein tärkeää ylläpitäjän kannalta.

MongoDB taas on sen verran suosittu, että siitä löytyy hyvin tietoa eri lähteistä, kuten keskustelupalstoilta, mistä on hyötyä pikkuongelmissa. Vaikka se ei täysin noudata ACID:ia, tärkeimmät ominaisuudet ovat kuitenkin mukana.

5 Toteutuksen valmistelua

5.1 Miksi NoSQL?

Syy NoSQL-kannan käyttöönotolle tässä työssä on pääasiallisesti mielenkiinto oppia käyttämään uutta kantatyyppiä. Verkkokauppa on tällä hetkellä sovelluksena vielä hyvin pieni ja relaatiokanta soveltuu siihen toistaiseksi vallan mainiosti. Kuitenkin tulevaisuudessa tarkoituksena olisi laajentaa sovellusta, mikä tarkoittaa myös kannan muutoksia, etenkin jos halutaan pitää suorituskykyä yllä. Verkkokaupoissa nopeus on tärkeää, kukaan ei halua odottaa pitkiä aikoja hakutuloksiaan. Jo pelkästään tämän vuoksi on syytä etsiä etukäteen vaihtoehtoja nykyiselle ratkaisulle.

Tarkoituksena on siis ottaa selvää yhden NoSQL-kannan toiminnasta ja käytöstä, ja pohtia samalla, olisiko se mahdollista ottaa käyttöön myös verkkokauppasovelluksessa.

5.2 Valittujen lauseiden NoSQL-esitys

SQL-lauseiden kääntäminen suoraan NoSQL-kannan käyttöön ei tietenkään ole aivan suoraviivaista. Ennen aloitusta on jo havaittavissa muutamia mahdollisia ongelmia ja pulmakohtia.

Luvussa 3.2 nähdään, että osassa lauseista on käytetty valmisteltuja kyselyitä, eli käyttäjän antamia parametreja ei ole suoraan otettu kyselyyn. Tämä on syytä tehdäkin tietoturvan vuoksi, mutta voi aiheuttaa tässä tapauksessa monimutkaisuuksia. Testaamisessa ei ole mikään ongelma pistää parametreja suoraan hakuun, mutta oikeassa sovelluksessa näin ei tietenkään voi toimia. Toisaalta taas tämä voi vaikuttaa myös suorituskykyyn, mitä tulee miettiä vertailuja tehdessä.

Toinen mahdollinen ongelma on se, että salasanat on kryptattu toteutuksessa jo ennen SQL-lauseeseen sijoittamista. Tämä on tehty PHP:n avulla. Testauksessa kryptauksesta ei tarvitse välittää, mutta toisaalta myöhemmin täytyy miettiä sen tekemistä NoSQL:ssä. Tietoturva on kuitenkin edelleen iso asia.

Kolmantena tulee miettiä, miten taulut saadaan olemaan yhteydessä toisiinsa NoSQL-toteutuksessa. Kuten aliluvussa 3.2 käy ilmi, kahdessa SQL-lauseessa on käytetty join-operaattoria kahden taulun välillä (kirja ja kirjailija). Tämä täytyy pystyä toteuttamaan jotenkin mallinnuksessa, tai vaihtoehtoisesti muokata lauseiden rakenteita sopimaan uuteen malliin. Tulee myös miettiä, miten käsitellään kirjailija-taulun rivit, joilla on useampi kirja taulussa kirja. Voiko esimerkiksi ottaa kirjailijan avaimeksi ja sitoa siihen useamman kirjan?

5.3 NoSQL-tietokannan valinta

Tässä vaiheessa pitäisi sitten valita NoSQL-kanta, jota käytetään mallinnuksessa. Luvussa 4.6 käytiin läpi kolme eri vaihtoehtoa, joista lopullinen valinta tehdään.

Kaikissa kannoissa on hyvät puolensa, ja niitä olisi mielenkiintoista testata. Mutta valinta osuu MongoDB-kantaan. Dokumenttikanta sopii todennäköisesti parhaiten tähän projektiin, kun ottaa huomioon miten ja millaista tietoa halutaan käsitellä. Se myös vaikuttaa hyvältä ja tarpeeksi helpolta aloittelevalle NoSQL-käyttäjälle. MongoDB:n ominaisuudet ovat tarpeeksi laajat isompaankin projektiin, vaikka jonkin verran jääkin ilmaisversiosta puuttumaan.

6 Toteutus

6.1 Alustavat toimenpiteet

Nyt kun NoSQL-kanta on valittu, on aika siirtyä miettimään kannan toimintaa tarkemmin. Ensin täytyy ladata tarvittavat tietokannat ja sen jälkeen syöttää niihin testidataa, jotta voidaan tehdä mittauksia. Sekä MySQL- että MongoDB-kannat ovat Community Edition -kantoja, eli siis ilmaisia. Ohjelmistot toimivat Windows-ympäristössä. Kumpikin kanta toimii ainoastaan yhdellä palvelimella, mikä vaikuttaa yleisesti suorituskykyyn.

MySQL-kannasta ja sen Graphical User Interfacesta (GUI), eli graafisesta käyttöliittymästä Workbenchistä on käytössä versio 8.0.12. Kannan GUI on monipuolinen ja kätevä (joskin hidas) käyttää, joten kaikki kyselyt ja mittaukset on suoritettu käyttäen sitä.

Relaatiokantaan syötettiin testidataa yksinkertaisella while-toistorakenteella, jossa id-arvoja lukuunottamatta kaikki arvot säilyivät identtisinä. Testidatan syöttämiseen meni noin 30 minuuttia per kanta, mikä on kohtuullisen pitkä aika. Id-arvoiksi otettiin aina while-toistorakenteen laskurin nykyinen arvo. Näin ne pysyivät erilaisina ja yksiselitteisinä. Taulujen sisältämä data on seuraavanlainen:

- kirja: 15 000 identtistä riviä + 23 uniikkia riviä testejä varten
- kirjailija: 15 000 identtistä riviä + 14 uniikkia riviä testejä varten
- kayttaja: 15 000 identtistä riviä + 7 uniikkia riviä testejä varten.

Jokaiseen tauluun on siis lisätty joitain uniikkeja rivejä, jotta suorituskykymittauksia on voitu tehdä tehokkaasti ja järkevästi.

MongoDB:stä on käytössä versio 4.0.1, ja sen graafisesta käyttöliittymästä Compassista versio 1.15.1. Nämä olivat käyttöönottohetkellä vakaimmat versiot. Kannan GUI on sen verran selkeästi vielä alkutekijöissään, että siitä ei hirveästi hyötyä työn tekemisessä ollut. Kaikki kyselyt on siis käytännössä suoritettu MongoDB:n shellin kautta, ja GUI:ta on käytetty lähinnä pikaisiin testauksiin ja datan tarkasteluun. Luvussa 6.2 käydään läpi tämän kannan testidata ja sen syöttäminen.

6.2 NoSQL-mallin toteutus

6.2.1 Kannan rakenne

Seuraavaksi pitäisi siis luoda haluttu MongoDB-tietokanta. Kuvassa 2 näkyy kannan rakenne ja sen sisältämät collectionit eli kokoelmat. Nämä vastaavat SQL-kantojen tauluja.

Kuva 2. MongoDB-kannan rakenne.

Kuten kuvasta nähdään, rakenne on hieman muuttunut aiemmasta. Seitsemän taulun sijasta käytössä on enää kolme kokoelmaa. Näistä kayttaja- ja tilaukset-kokoelmat ovat identtisiä kayttaja- ja tilaukset-taulujen kanssa, mutta kirja- ja kirjailija-taulut on yhdistetty yhdeksi kokoelmaksi. Kaikki parametrit ovat pysyneet samana, mutta niille ei ole enää mitään vaatimuksia, koska kannassa ei ole kiinteää skeemaa. Taulujen yhdistäminen on tehty ihan vain siitä syystä, että se yksinkertaistaa kannan rakennetta. MongoDB mahdollistaa kyllä join-kyselyitä vastaavat kyselyt, eli useamman kokoelman käyttökin olisi ollut mahdollista. Kuitenkin koska tässä tilanteessa kyseessä on testivaiheessa oleva kanta (ja sovellus), muutoksia on helppo vielä tehdä.

Taulujen yhdistäminen tuo jonkun verran toistoa, koska yhdellä kirjailijalla voi edelleen olla useampi kirja. Tällä voi olla jonkin verran vaikutusta suorituskykyyn, mutta ei kuitenkaan mitenkään häiritsevän paljoa. Myöskään tässä tapauksessa ei ole sen tarkemmin tarkasteltu tilaukset-kokoelmaa. Ja kuten relaatiokantatoteutukseenkin, tähän täytyy sovelluksen edetessä lisätä kokoelmia ja parametrikenttiä, mutta ne eivät juuri tämän työn aikana ole mitenkään ajankohtaisia. Niille tulee kuitenkin olemaan merkitystä kyselyiden monimutkaisuuden ja kyselyiden suorituskyvyn kannalta.

Tässä vaiheessa MongoDB-kantaan ei myöskään ole lisätty kuin ainoastaan tarpeelliset kokoelmat, eli se eroaa relaatiokannan skeemasta jonkun verran. Näiden kokoelmien teko olisi kuitenkin ollut ylimääräistä työtä tässä vaiheessa, sillä olisi täytynyt miettiä tiedon toisteisuutta ja muutoinkin hyvää tapaa toteuttaa vastaava rakenne. Relaatiokannan rakenteen voisi kopioida käytännössä suoraan MongoDB-kantaan, mutta silloin sillä olisi vaikutusta kannan toimintaan, eikä mitenkään positiivisesti. Kokonaan uuden, toimivan skeeman miettiminen toteutukselle, joka on teoriatasolla, vie enemmän aikaa kuin tämän työn puitteissa oli mahdollista.

Kokoelmia ja dokumentteja luodessa on hyvä ottaa huomioon, että MongoDB luo automaattisesti uniikin id:n ja tekee siitä myös indeksin. Tätä indeksia ei käytetä kyselyissä ellei käyttäjä niin määrittele, mutta se on silti olemassa. Kokoelmia ei myöskään ole pakko luoda erikseen: MongoDB luo uuden kokoelman insert-lausetta suorittaessa, mikäli se ei löydä vastaavaa jo valmiiksi. Kuvan 2 kokoelmista huomaa selkeästi myös sen, että parametreille ei ole annettu minkäänlaisia rajoituksia, eli

kokoelman sisältämä data voi olla aivan muuta kuin sen pitäisi ilman, että kukaan huomaa. MongoDB:ssä on mahdollista tehdä skeemavalidointia ja muita tiedon käsittelyyn liittyvää rajoittamista, mutta niistä keskustellaan lisää aliluvussa 6.4.

MongoDBn testidata on tehty niin, että kokoelmiin on syötetty for-toistorakenteessa tietty määrä tietoa. Kanta siis generoi id:n jokaiselle dokumentille, joten siitä ei tarvinnut huolehtia kuten SQL:n kanssa piti. Kaikki tieto id:tä lukuunottamatta on siis jälleen täysin identtistä, lukuunottamatta muutamaa kyselyitä varten tehtyä dokumenttia. Testidatan syöttämiseen MongoDB:ssä meni suunnilleen 30 sekuntia per kokoelma, mikä on huomattavasti vähemmän kuin MySQL:ssä, ja oli todella lupaavaa suorituskyvyn kantilta katsottuna. Testidata on siis seuraavanlaista:

- kirja: 15 000 identtistä riviä + 23 uniikkia riviä testejä varten

- kayttaja: 15 000 identtistä riviä + 7 uniikkia riviä testejä varten.

Eli kuten nähdään, testidata on kummassakin kannassa samanlaista ja sama määrä. Tämä tehdään siksi, jotta suorituskykyä voidaan vertailla kantojen välillä mahdollisimman hyvin.

6.2.2 Kyselyt

Nyt kun kanta ja kokoelmat on tehty, on aika siirtyä tiedon manipulointiin. Kokoelmissa on siis tässä vaiheessa aliluvussa 6.2.1 mainittu testidata, joka nyt pitäisi jotenkin saada näkyville. Tarkoituksena on saada kannasta samanlaista tietoa kuin SQL-lauseilla. Kuten aliluvussa 4.6.1 on mainittu, MongoDB käyttää JavaScriptiä tiedon manipulointiin, eli alla olevat koodiesimerkitkin ovat JS:ää. MongoDB:n sivuilta löytyvä dokumentaatio toimi erinomaisena apuna lauseita miettiessä ja testaillessa, etenkin aloitusvaiheessa.

Lauseet ovat tässä samassa järjestyksessä kuin SQL-lauseet aiemmin, eli kaksi ensimmäistä on tuotteita koskevia lauseita ja kaksi jälkimmäistä käsittelee käyttäjiä. Kyselyissä merkinnällä <*parametri*> viitataan muuttuviin parametrien arvoihin, joita tietoa manipuloidessa on käytetty. Tässä vaiheessa ei ole vielä selvää, kuinka valmistellut kyselyt mahdollisesti voisi MongoDB:ssä toteuttaa. Sitä mietitään vasta myöhemmin.

Ensimmäinen kysely (esimerkkikoodi 5) hakee kirja-kokoelmasta tietueet, joissa kirjan nimi tai kirjailijan sukunimi vastaa annettua parametriä. Kysely käy läpi koko kokoelman, etsii parametrejä vastaavat dokumentit ja tulostaa ne näkyviin. Varsinainen *find()*-kysely koostuu tässä tapauksessa kahdesta osasta: ensin etsitään *$or*-operaattoria käyttäen halutut dokumentit ja sen jälkeen määritellään, mitä tietoa dokumenteista näytetään käyttäjälle. Jos parametrin arvona on 1, kyseinen tieto näytetään. Parametrin *_id* arvona on 0, koska sitä ei ole tarpeellista näyttää käyttäjälle, mutta MongoDB printtaa sen näkyviin oletuksena. Vastaavia tarkennuksia tiedon näyttämisessä on käytetty myös seuraavassa kyselyssä (esimerkkikoodi 6).

```
db.kirja.find({$or:[{ kirja_nimi: "<parametri>"},
{sukunimi:"<parametri>"}]},{ISBN:1, kirja_nimi:1, etunimi:1, sukunimi:1,
julkaisuvuosi:1, hinta:1, _id: 0})
```

Esimerkkikoodi 5. Valitaan tietoja kokoelmasta.

Tämä koodi, esimerkkikoodi 6, hyvin yksinkertaisesti vain hakee kaikki kokoelmassa kirja olevat dokumentit ja tulostaa ne näkyviin.

```
db.kirja.find({}, {ISBN:1, kirja_nimi:1, etunimi:1, sukunimi:1,
julkaisuvuosi:1, hinta:1, _id: 0})
```

Esimerkkikoodi 6. Valitaan kokoelman kaikki tiedot.

Koodi 7 etsii kayttaja-kokoelmasta käyttäjän, jonka tunnus vastaa sille syötettyä parametriä. Tässä voisi käyttää myös *findOne()*-komentoa, joka olettaa, että on olemassa ainoastaan yksi hakua vastaava dokumentti. Sen käyttö tuo joitain rajoitteita, mutta se voi olla hyödyllinen, jos halutaan keventää kyselyitä ympäröivää koodia sovelluksen puolella.

```
db.kayttaja.find({tunnus:"<parametri>"})
```

Esimerkkikoodi 7. Valitaan käyttäjä kokoelmasta.

Viimeinen oikea kysely, esimerkkikoodi 8, lisää uuden käyttäjän kayttaja-kokoelmaan. Jälleen kerran voisi käyttää myös vaihtoehtoista komentoa, *insertOne()*, joka lisää ainoastaan yhden dokumentin.

```
db.kayttaja.insert({tunnus: "<parametri>", pwd:"<parametri>"})
```

Esimerkkikoodi 8. Lisätään uusi käyttäjä kokoelmaan.

Edellä olevissa kyselyissä ei ole käytetty *LIKE*-operaattoria, kuten SQL-kyselyissä aliluvussa 3.2. Tämä johtuu siitä, että MongoDB:ssä ei ole suoraa vastaavuutta sille. Sen sijaan voi kuitenkin käyttää *$regex*-operaattoria. Tällöin kyselyn syntaksi muuttuu hieman. Esimerkiksi koodi 7 muuttuisi alla olevaan muotoon.

Kuten koodista 9 nähdään, kysely ei muutu mitenkään olennaisesti, joskin ylimääräiset sulut ja hipsut voivat aiheuttaa vaikeuksia tekijälle.

```
db.kayttaja.find({tunnus:"{$regex:"<parametri>"}"})
```

Esimerkkikoodi 9. Valitaan käyttäjä kokoelmasta käyttäen $regex-operaattoria.

Kyselyiden tuottaminen ja käyttäminen ei kuitenkaan ole mitenkään vaikeaa, vaikka niihin voikin olla SQL:n jälkeen hieman vaikea tottua. MongoDB ja sen käyttämä JS ovat kuitenkin hyvin joustavia ja sopivat laajasti eri kantatarpeisiin. Kuten aiemmin tuli mainittua, MongoDB:n dokumentaatio on todella laaja ja sitä kannattaa käyttää hyödyksi.

6.3 Suorituskykyvertailu

Kyselyiden ollessa selvillä tulee aika ruveta testailemaan niitä. Testit on toteutettu niin, että jokaista kyselyä (koodit 1-8) on ajettu 20 kertaa peräkkäin vaihtelevilla parametreillä, ja jokaisen ajon aika on otettu ylös. Tätä varten testidatassa oli uniikkeja arvoja, jotta voidaan nähdä, kuinka kauan muutaman tietueen etsiminen kestää.

Kyselyiden ajat on saatu MySQL:stä suorittamalla ne Workbenchissä, ja katsomalla jokaisen ajon jälkeen Output Pane-ikkunasta Duration-välilehti. MongoDB:n ajat on otettu ylös käyttämällä erityistä komentoa, koska kyselyt täytyi suorittaa shellissä, kuten luvussa 6.1 mainittiin. Komento oli *explain("executionStats")*, joka liitetään tavallisen kyselyn perään. Se tarjoaa runsaasti tietoa kyselyn suorittamisesta, mutta tässä kohdassa ainoastaan parametrin *executionTimeMillis* arvo oli tärkeä. Tässä käydään läpi suorituskykymittaukset taulukoiden avulla, joissa käytetyt tarkat ajat mittauksille löytyvät liitteestä 2. Jokaisessa taulukossa on ylärivillä koodit, joita kyseisissä mittauksissa on käytetty.

Ensimmäisenä on kyselyiden 1 ja 5 vertailu. Nämä kyselyt hakivat siis tietoa kirjoista/kirjailijoista annettujen parametrien avulla.

Taulukko 2. Esimerkkikoodien 1 ja 5 suoritusaikamittausten tulokset.

`SELECT ISBN, kirja_nimi, etunimi, sukunimi, julkaisuvuosi, hinta FROM kirja INNER JOIN kirjailija ON kirja.kirjailija_id=kirjailija.id WHERE kirja_nimi LIKE :search OR sukunimi LIKE :search;`			`db.kirja.find({$or:[{ kirja_nimi: "<parametri>"}, {sukunimi:"<parametri>"}]}, {ISBN:1, kirja_nimi:1, etunimi:1, sukunimi:1, julkaisuvuosi:1, hinta:1, _id: 0})`		
MySQL			**MongoDB**		
Keskiarvo	Nopein	Hitain	Keskiarvo	Nopein	Hitain
47,9 ms	10,6 ms	78 ms	19,05 ms	18 ms	25 ms

Taulukossa 2 näkyy tärkeimmät tiedot koodien 1 ja 5 suoritusaikamittausten tuloksista. Kuten aiemmin tuli mainittua, kyselyt on suoritettu 20 kertaa. Taulukossa näkyy

jokaisen koodin nopein ja hitain aika, sekä keskiarvo. Kaikki arvot ilmoitetaan millisekunteina (ms). Kuten selkeästi nähdään, MongoDB suoriutuu kyselyistä enimmäkseen huomattavan paljon nopeammin kuin SQL: MongoDB:n keskiarvo on 20 ms:n paikkeilla, kun taas MySQL:n keskiarvo on suunnilleen 50 ms:n kohdilla ja lähenee jopa 80 ms:ää kerran. Osa kyselyiden parametreistä on käyttänyt hyväksi *%*- ja *$regex*-operaattoreita, joiden voisi olettaa vaikuttavan kyselyn käsittelyyn ja tulosten etsimiseen. MongoDB:ssä tämä ei kuitenkaan aiheuttanut mitään merkittävia muutoksia, mutta SQL-kyselyiden kesto laski, ja nopein arvo onkin juuri tästä setistä.

Taulukko 3. Esimerkkikoodien 2 ja 6 suoritusaikamittausten tulokset.

`SELECT ISBN, kirja_nimi, etunimi, sukunimi, julkaisuvuosi, hinta FROM kirja INNER JOIN kirjailija ON kirja.kirjailija_id=kirjailija.id;`			`db.kirja.find({}, {ISBN:1, kirja_nimi:1, etunimi:1, sukunimi:1, julkaisuvuosi:1, hinta:1, _id: 0})`		
MySQL			**MongoDB**		
Keskiarvo	Nopein	Hitain	Keskiarvo	Nopein	Hitain
69,11 ms	63,6 ms	84,8 ms	17,75 ms	17 ms	24 ms

Taulukossa 3 näkyy kyselyiden 2 ja 6 mittausten tuloksia. Näissä kyselyissä haettiin ja tulostettiin koko kannan/kokoelman tietueet. Kuten jälleen nähdään, MongoDB:n suoritusajat pysyvät tasaisesti n. 20 ms:n paikkeilla. SQL:n ajat taas ovat 70 ms:n paikkeilla, ja nousevat jopa lähes 90 ms:ään kerran. MongoDB vaikuttaisi siis taas toimivan tehokkaammin ja nopeammin kuin SQL.

Taulukko 4. Esimerkkikoodien 3 ja 7 suoritusaikamittausten tulokset.

`SELECT * FROM kayttaja WHERE tunnus= :tunnus;`			`db.kayttaja.find({tunnus: "<parametri>"})`		
MySQL			**MongoDB**		
Keskiarvo	Nopein	Hitain	Keskiarvo	Nopein	Hitain
8,76 ms	7,4 ms	11,5 ms	10 ms	10 ms	10 ms

Taulukossa 4 on koodien 3 ja 7 suoritusajat. Näillä kyselyillä etsittiin siis kayttaja-taulusta/kokoelmasta yhtä tiettyä käyttäjää tunnuksen perusteella. Kuten nähdään, ajat ovat aika pitkälle samat MySQL- ja MongoDB-kyselyissä. Aiempien perusteella voisi olettaa, että MongoDB suoriutuisi kyselyistä huomattavasti nopeammin, mutta niin ei

kuitenkaan ole. Tämä voi johtua esimerkiksi siitä, että kyselyssä ei ole hyödynnetty *findOne()*-funktiota, joka suoriutuisi toimintansa vuoksi kyselystä mahdollisesti nopeammin. Kummatkin ajat ovat kuitenkin ihan kohtuullisia, etenkin kun ne suoritetaan useimmiten sisäänkirjautumisen (mahdollisesti myös uuden käyttäjän luomisen) yhteydessä, mikä yleensä kestääkin pidempään ja siihen on useimmin käyttäjien keskuudessa totuttu.

Taulukko 5. Esimerkkikoodien 4 ja 8 suoritusaikamittausten tulokset.

`INSERT INTO kayttaja (id, tunnus, salasana) VALUES (0, :tunnus, :pwd);`			`db.kayttaja.insert({tunnus: "<parametri>", pwd:"<parametri>"})`		
MySQL			**MongoDB**		
Keskiarvo	Nopein	Hitain	Keskiarvo	Nopein	Hitain
128,15 ms	78 ms	203 ms	0 ms	0 ms	0 ms

Koodien 4 ja 8 tulokset on esitetty taulukossa 5. Näillä kyselyillä lisättiin uusi käyttäjä kayttaja-tauluun/kokoelmaan. Välittömästi huomataan, että MongoDB:ssä kyselyn suorittaminen on kestänyt 0 ms, eli se on tapahtunut välittömästi. MySQL:ssä kysely on taas kestänyt selvästi pidempään kuin aiemmin suoritetut. Tämä johtuu siitä, että MySQL-taulussa on olemassa skeema, johon uutta tietuetta verrataan ennen kuin se voidaan lisätä. MongoDB:ssä taas dokumentti lisätään suoraan kokoelmaan ilman minkäänlaista validointia (kts. aliluku 6.4 liittyen MongoDB:n skeemavalidointiin).

Kuten taulukoista nähdään, MongoDB on lähes koko ajan reippaasti nopeampi kyselyiden toteuttamisessa. Kaikki kyselyt on suoritettu käyttäen samaa laitteistoa, samaan aikaan päivästä ja samojen ohjelmien pyöriessä taustalla. On siis hyvin epätodennäköistä, että erot suorituskykymittauksissa johtuisivat mistään tietokantojen ulkopuolisesta tekijästä. MongoDB vaikuttaisi siis vain olevan enimmäkseen MySQL-tietokantaa nopeampi. Yllä olevat kyselyt ovat toki hyvin yksinkertaisia, mutta ero aikojen välillä on silti huomattava.

Näissä kyselyissä ei vielä käytetty indeksejä, mikä näkyi ainakin MySQL-kyselyissä selkeästi. Seuraavassa aliluvussa käydään läpi sitä, miten indeksit vaikuttavat asioihin, ja katsotaan vielä yksi suorituskykyvertailu.

6.4 Indeksointi

Hyvin yleisesti, etenkin isommissa sovelluksissa, on syytä käyttää indeksointia, koska se nopeuttaa kyselyiden suorittamista. Indeksointi on kohtuullisen monimutkainen konsepti, joten sitä ei tässä työssä käydä kovin tarkasti läpi. Noin pääpiirteittään indeksointi toimii niin, että taululle/kokoelmalle, ja niissä halutuille kentille, määritellään yksi indeksi (se voi olla vaikka id-parametri), jota käytetään. Kanta luo näistä indekseistä oman rekisterin, johon tallennetaan indeksin mukana myös osoitin varsinaiseen tietueeseen taulussa (DBMS - Indexing 2018). Kun indeksiä käytetään, kysely hakee indeksirekisteristä haluamansa tiedon ja sen avulla noutaa taulusta suoraan tiedot. Näin kyselyn ei tarvitse käydä koko taulua läpi, mikä nopeuttaa kyselyiden aikoja.

Suoritettavien kyselyiden tehokkuus riippuu siitä, kuinka monta tietuetta pitää käydä läpi, että löydetään haluttu tulos. Tämä ei ole ongelma pienissä kannoissa, mutta kun kannan koko nousee, nousevat myös suoritusajat. Ideaalitilanteessa suhde on 1:1, eli kysely käy läpi ainoastaan ne tietueet, jotka halutaan saada. Käytännössä tämä onnistuu hyvin harvoin, mutta aina on hyvä yrittää. Ideaali vaatii juurikin indeksoinnin käytön ja yleensä myös hyvin optimoidut kyselyt, jotka eivät yritä hakea ylimääräistä dataa tai hae sitä liian monimutkaisesti.

Nämä asiat huomioon ottaen oli syytä testata, onko indeksoinnista tässä tapauksessa hyötyä. Aikarajoitusten vuoksi sitä testattiin ainoastaan yhdellä taululla, joka oli kayttaja-taulu. Koodeiksi valittiin koodit 3 ja 7, koska niillä haettiin ainoastaan yhtä arvoa yhdellä parametrillä, joka on suunnilleen unelmatilanne indeksin käyttöön. Taulusta (ja MongoDB:ssä kokoelmasta) otettiin identtinen kopio nimeltään k_idx, joka sisälsi kaiken testidatan ja muut parametrit. Kummassakin oli jo valmiiksi ohjelmistojen

luomat indeksit id-kentille, mutta niiden käyttö ei sopinut käytettyihin koodeihin. Tämän vuoksi päätettiin indeksoid kenttää *tunnus*. MySQL:ssä käytettiin *CREATE INDEX*-lausetta ja MongoDB:ssä *createIndex()*-metodia. Taulukossa 6 näkyy suorituskykymittausten tulokset indeksiä käyttäen.

Taulukko 6. Kyselyiden ajat käyttäen indeksejä.

`SELECT * FROM k_idx WHERE tunnus= :tunnus;`			`db.k_idx.find({tunnus: "<parametri>"})`		
MySQL			**MongoDB**		
Keskiarvo	Nopein	Hitain	Keskiarvo	Nopein	Hitain
0,46 ms	0,37 ms	0,67 ms	0 ms	0 ms	0 ms

Taulukon 6 arvot kertovat erittäin hyvin indeksoinnin toiminnasta. MongoDB:n ajat ovat tasaisesti 0 ms siinä, missä ne ilman indeksiä olivat suunnilleen 10 ms. SQL-kyselyidenkin ajat ovat laskeneet huomattavasti: ilman indeksiä kyselyiden keskiarvo oli suunnilleen 9 ms, nyt se on alle 0,5 ms. Ajat kertovat siitä, että indeksoinnista oli tosiaan tässä tapauksessa huomattavaa hyötyä, ja hyöty vain kasvaa tietomäärien kasvaessa.

Indeksoinnissa on tietenkin omat ongelmansakin. Se vie enemmän levytilaa, koska sitä varten täytyy luoda erillinen rekisteri. Se voi myös turhaan vaikeuttaa kyselyiden muodostamista ja tekemistä, etenkin jos sen hyöty ei ole niin suuri. Indeksointia kannattaa testata perusteellisesti ennen kuin sen ottaa käyttöön kannassa, mutta oikein tehtynä hyödyt ovat haittoja huomattavammat.

6.5 Tietoturva

Tietoturva on iso osa tietokantoja ja sovelluskehitystä. NoSQL-kantojen tietoturva ei ole ollut perinteisesti ihan niin hyvällä tasolla kuin relaatiokantojen, mutta se on parantumaan päin. MongoDB on suosittuna NoSQL-tietokantatuotteena erittäin hyvä kohde tietoturvarikkomuksille. On siis syytä tarkastella läheisemmin, kuinka kannasta

voidaan tehdä mahdollisimman turvallinen. Tässä tarkastellaan lähinnä itse kannan ja JavaScriptin yhteyttä sovellukseen, ja sen tuomia mahdollisia ongelmia ja niiden ehkäisyä. Erillisenä, tähän työhön kuulumattomana asiana on kantaan pääsy sovelluksen ulkopuolelta: kenellä on oikeuksia, mitä oikeuksia on ja miten niitä valvotaan. On myös suositeltavaa, että että tietokantapalvelin ja sovelluspalvelin pidetään selkeästi erillisinä ongelmien välttämiseksi.

Yksi isoimmista vaaranpaikoista on datan siirto kannan ja sovelluksen välillä. Tässä sovelluksessa suoritetaan ainoastaan yksi insert-komento, mutta myös haku-komennot tarvitsevat pääsyn kantaan, ja niiden avulla voi siis kantaan päästä myös ylimääräistä: tätä kutsutaan yleisesti tietokannoissa injektioksi. Mikäli sovellus olisi toteutettu käyttäen JavaScriptiä (tässä siis selaimen puolella oleva JS), kaikki siirrettävä tieto olisi nähtävissä (usein URL:ssa). Se voi olla hyväksyttävää jos puhutaan vaikka tiedon hakemisesta kirja-kokoelmasta, mutta muuttuu hyvin ongelmalliseksi kayttaja-kokoelman kanssa. Tunnukset itsessään eivät välttämättä ole mitenkään salaista, mutta salasanat ovat. Puhumattakaan siitä, että sovelluksella olisi tarkoitus jossain vaiheessa myös käsitellä sensitiivistä tietoa, eli esimerkiksi luottokorttietoa ostosten tekemistä varten.

MongoDB mahdollistaa uudemmissa versioissaan TLS/SSL-salauksen käytön. Kaikki tarpeellinen tulee valmiina Community Edition -versiossa, ja se on erittäin suositeltavaa ottaa käyttöön. Kryptaus mahdollistaa sen, että ainoastaan luotettu sovellus tai käyttäjä pääsee käsiksi kantaan. Enteprise Edition-versio tarjoaa myös mahdollisuuden kryptata kanta ja kaikki sen sisältämät tiedot silloinkin, kun kanta ei ole käytössä (Encryption at Rest 2018).

Äsken mainittua injektiota on myös syytä tarkastella enemmän. Se tarkoittaa siis haitallisen tiedon tahallista siirtämistä tietokantaan. NoSQL-kannoissa sen riski on perinteisesti ollut hyvin pieni, mutta se on silti olemassa. Relaatiokannoissa injektio antaa mahdollisuuden muokata dataa ja suorittaa SQL-komentoja. Mutta koska MongoDB mahdollistaa JS-injektiot, injektio voi ajaa myös kokonaisia koodinpätkiä. SQL-kannoissa injektioita on perinteisesti estetty esimerkiksi siivoamalla annettuja parametrejä sovelluksen puolella jo ennen kuin ne sijoitetaan kyselyihin (valmistellut

kyselyt). Esimerkiksi sovelluksessa käytetty PHP tarjoaa runsaasti eri käskyjä parametrien siivoukseen ja vertailuun.

MongoDB suosittelee myös auditoinnin käyttöä tietokannassaan. Auditointi pitää kirjaa käyttäjistä ja yhteyksistä, joita kantaan muodostetaan. Käytännössä sen avulla voi siis valvoa, mitä kannassa tapahtuu. Auditointi on kuitenkin mahdollista ainoastaan Enterprise Edition -versiossa.

Yksi suosittu vaihtoehto on myös se, että kokoelmista tehdään näkymät erillisellä välityspalvelimella ja kyselyitä tehdessä käytetäänkin näitä näkymiä. Se rajoittaisi kantaan pääsyä sovelluksen kautta, ja myös mahdollisesti nopeuttaisi kyselyiden suorittamista. Tärkeimpänä pointtina on kuitenkin juurikin se, että mahdollisuus päästä kantaan kiinni jatkuvan liikenteen vuoksi vähentyisi huomattavasti. Näin voitaisiin myös estää mahdolliset injektiot.

Myös monia pienempiä asioita, kuten skeemavalidointia ja eri kryptausmenetelmien käyttöä on syytä miettiä. Skeemavalidointi määrittelee perinteisen skeeman tapaan jo etukäteen, minkälaista tietoa kokoelmaan voi laittaa (Schema Validation 2018). Tällöin tietoa lisätessä ja muuttuessa kanta tarkistaa, että kaikki annetut arvot vastaavat validoinnissa määrättyjä parametrejä. Sitä voi myös käyttää kyselyitä tehdessä, esimerkiksi jos käytetään *$or*-operaattoria. Validointi saattaa hidastaa kannan toimintaa hieman, mutta kannassa, joka toimii niinkin nopeasti kuin MongoDB vaikuttaisi toimivan, lisäturvallisuus on sen arvoista.

Kaiken tämän lisäksi MongoDB suosittelee, että kehittäjä tietää kannan asetukset ja niiden merkityksen eikä tuudittaudu siihen, että oletusasetukset riittävät. NoSQL-tiedonhallintaohjelmistoilla on ehkä edelleen huono maine tietoturvan saralla, mutta se ei tarkoita, etteikö vaihtoehtoja löydy. Niitä täytyy vain olla valmis etsimään ja ennen kaikkea ymmärtämään.

7 Tulokset

7.1 Lopputulokset

Nyt kun kaikki mittaukset on tehty ja tiedot käyty läpi, on aika katsoa, mitä oikeastaan saatiin selville. Suorituskykymittausten perusteella voidaan päätellä, että NoSQL-tietokantatuote tulee todennäköisesti ainakin tämän sovelluksen kanssa toimimaan paremmin kuin aiempi kanta. MongoDB suoriutui kyselyistä tasaisesti MySQL:ää paremmin, mikä on hyvä merkki.

Tietoturvan osalta voi sanoa, että se tulee vaatimaan enemmän työtä kuin relaatiokannassa. Sovelluksessa pitäydytään edelleen PHP-skriptissä luvussa 6.4 mainitusta syystä, mutta tietoturvaa täytyy silti tarkastella enemmän.

Kaiken kaikkiaan NoSQL-tiedonhallintaohjelmisto on vaivan arvoinen. Se vaatii huomattavasti alkutyötä, mikäli aihe on vieras (kuten huomataan työn alun kappaleista), mutta tarjoaa tulosten mukaan ainakin rajoitetuissa olosuhteissa etuja suorituskyvyn suhteen.

7.2 Ongelmat

Työtä varten lukiessa ja sitä tehdessä mieleen tuli muutamia mahdollisia ongelmakohtia, jotka on myös syytä mainita. Ensimmäinen näistä on se, johtuuko SQL-kyselyiden kesto huonosta optimoinnista. Kyselyt ovat hyvin yksinkertaisia, joten hirveästi mitään optimoitavaa ei ole, mutta aina jotain voi tietenkin parantaa. Ehkä esimerkiksi indeksien käyttö alusta asti olisi nopeuttanut relaatiokannankin puolta.

Tietokantojen rakenne on myös hyvin yksinkertainen, ja tässä työssä käytettiin vain osia hyväksi. Kantojen kasvaessa ja muuttuessa monimutkaisemmiksi myös suoritusajat kasvavat. On myös syytä huomioida, että MongoDB:ssä on tässä tapauksessa toisteisuutta dokumenttien sisällä, mikä vaikuttaa mahdollisesti suorituskykyyn. Toisteisuuden voi ratkaista muuttamalla kannan rakenteita tai luomalla kokonaan uusia kokoelmia.

Kyselyitä suorittaessa täytyy myös miettiä, miten ne lukitsevat taulun/kokoelman suorituksen ajaksi. Se ei suoraan käynyt ilmi, etenkään MongoDB:n tapauksessa. Mutta vaikuttaisi siltä, että relaatiokanta lukitsee ainoastaan tietueen aina kun kanta tekee kyseiselle tietueelle jotain (voi myös joissain tapauksissa lukita koko taulun), mukaan lukien yksinkertaiset *SELECT*-lauseet. MongoDB taas vaikuttaisi lukitsevan kokoelman ainoastaan *insert*-operaation yhteydessä, mutta tästä ei tullut täyttä selvyyttä. Yksittäisten kyselyiden kanssa lukitus ei ole ongelma, mutta kun ruvetaan siirtymään moneen yhtäaikaiseen kyselyyn, mitä tapahtuu jatkuvasti verkkokaupasta puhuessa, lukoista voi tulla ongelma. Ongelman voi ainakin osittain ratkaista laitteiston muistia ja muita parametrejä kasvattamalla.

Tässä työssä on keskitytty hyvin vahvasti itse tietokantaan, kuten tarkoituskin oli, mutta siitä huolimatta muitakin asioita täytyy tarkastella. Yhtenä asiana on juurikin kannan (ja sovelluksen) taustalla oleva laitteisto. Nykyisellä laitteistolla ei ole mitään mahdollisuuksia ylläpitää verkkokauppaa, tosin pieni verkkokauppa voi hyvinkin toimia tavallisella pöytäkoneella. Parempi laitteisto on tietenkin maksullista, eikä tässä vaiheessa vielä mitenkään tarpeellista. Mutta jos tarkoituksena on saada aikaiseksi oikea, toimiva verkkokauppa, nykyiset puitteet eivät ole välttämättä riitä pidemmän päälle.

7.3 Esiin tulleet kysymykset

On hyvä käydä myös läpi muutamia kysymyksiä, joihin ei ehkä tullut tässä työssä selkeitä vastauksia, mutta jotka kuitenkin ovat jatkokehityksen kannalta mahdollisesti kiinnostavia.

Ensimmäisenä on tietoturva. Onko se kuitenkaan tarpeeksi hyvä MongoDB:ssä, etenkin kun kyseessä on Community Edition? Maksullinen versio tarjoaa mm. enemmän salaus- ja auditointimahdollisuuksia, joiden käyttäminen toisi turvaa jatkokehitystä ajatellen. MongoDB:n sivuilta ei suoraan käy ilmi maksullinen version hinnoittelua, ei edes pilviversiolle, joten sen käyttöönoton hinta jää avoimeksi.

Seuraavana on se, miten kanta pilvessä ja kanta omalla palvelimella eroaa toisistaan. Pilven käyttö tuo tietenkin omat haasteensa, mutta se voi silti olla parempi vaihtoehto, etenkin isompiin järjestelmiin, jos ei ole mahdollisuutta ylläpitää isoja palvelimella. Tässä yhteydessä ei kuitenkaan testattu pilven käyttöä ollenkaan, joten ei ole selvää, miten se vaikuttaisi esimerkiksi suorituskykyyn.

Suorituskykyyn liittyen täytyy myös pohtia, kuinka valmisteltujen kyselyiden käyttö vaikuttaa kyselyiden suoritusaikaan. Tässä työssä testit tehtiin niin, että parametreiksi annettiin suoraan halutut arvot, mutta näin ei voi tietoturvan vuoksi toimia oikeassa sovelluksessa.

Viimeisenä voisi miettiä sitä, ovatko mittausten antamat ajat normien puitteissa. Tähän ei ole löytynyt mitään selkeää vastausta, sillä ajat riippuvat monesta tekijästä (kts. aliluku 7.2 mahdollisista ongelmista). Ei siis ole mahdollista sanoa suoraan, onko kyselyt optimoituja ja kannat sopivan kokoisia. Tämä vaikuttaa olevan lähinnä sovelluskohtainen asia, eli yksittäistä, selvää vastausta normaaliaikoihin ei ole olemassa.

7.4 Jatkokehitys

Näillä eväillä ei ole suoraan mahdollista toteuttaa MongoDB.n käyttöönottoa, vaan ideana oli lähinnä tehdä alustavaa työtä siihen liittyen. Jatkokehityksen kantilta seuraava askel on suorittaa kuormituskykymittauksia, jotka antavat viitettä siitä, miten MongoDB-kanta oikeasti tulee suoriutumaan verkkokauppasovelluksen tietokantana. Tämän jälkeen täytyy myös miettiä samanaikaista käyttöä ja mahdollisesti monisäe-ajon tarvetta.

On myös tulosten valossa syytä pohtia, onko järkevää tehdä täydellistä siirtymää, eli siirtää kaikkia kantoja NoSQL-ympäristöön. Yhtenä mahdollisuutena on, että siirretään ainoastaan kirja- ja kirjailija-taulut ja jätetään kayttaja-taulu SQL-ympäristöön. Tämä helpottaisi esiintulleiden tietoturvakysymysten kanssa. Koska indeksikyselyiden perusteella relaatiokanta suoriutuu kyseisen taulun kanssa hyvin, ei ole mitään erityistä syytä tehdä asioita turhan monimutkaiseksi kantapuolella. Osittainen käyttöönotto on mahdollinen, joskin se vaatii hieman enemmän sovellukselta ja backend-koodilta. Yksi vaihtoehto on myös tehdä niin, että uudet käyttäjäprofiilit menevät ensin NoSQL-kantaan, jonka jälkeen ne voidaan käsitellä erillisellä välityspalvelimella ja siirtää relaatiokantaan.

Tässä vaiheessa kyselyt ovat vielä yksinkertaisia, mutta sovelluksen kehittyessä ja kasvaessa täytyy myös miettiä kyselyiden optimointia. Ei ole järkeä ottaa käyttöön uutta tietokantaa, mikä on suuri työ backend-koodin ollessa täysin valmis, jos siitä ei saada mitään selkeitä hyötyjä. Jos kyselyt ovat huonot, ne vievät kantatyypistä huolimatta resursseja ja aikaa.

On myös syytä miettiä sitä, että kanta olisikin pilvipalvelussa. Kuten luvussa 7.3 mainitaan, kummatkin kannat ovat tällä hetkellä omalla palvelimella, jonka tehot eivät riitä tuotantokäytössä olevan verkkokaupan pyörittämiseen. Yhtenä vaihtoehtona siis on kannan (ja mahdollisesti myös koko sovelluksen) siirtäminen pilvipalveluun.

8 Yhteenveto

Työssä on pyritty käsittelemään laajasti tietokantojen eri piirteitä keskittyen erityisesti SQL- ja NoSQL-tietokantojen eroihin. Tarkoituksena oli selvittää, onko mahdollista ja järkevää ottaa käyttöön NoSQL-tietokanta tilanteessa, jossa sovelluspuoli on jo valmis.

Aluksi oli tärkeää selvittää yleiset erot eri kantatyyppien välillä, ja sen jälkeen keskittyä NoSQL-tietokantojen erityispiirteisiin. Kun oli löydetty sopiva NoSQL-tietokantatuote, piti käyttöönottoa ruveta miettimään tarkemmin. Muodostettiin neljä kyselyä, jotka vastasivat sovelluksesta poimittuja SQL-kyselyitä. Tämän jälkeen tehtiin suorituskykymittauksia kummankin kannan lauseilla, jotta saatiin parempaa vertailupintaa kannoille. Mittausten perusteella todettiin, että NoSQL-kanta todellakin voi ainakin joissain olosuhteissa olla tehokkaampi kuin relaatiokanta.

Suorituskyvyn jälkeen tärkein aihe oli tietenkin tietoturva, jota käytiin kohtalaisen yleisellä tasolla läpi. Katsottiin kuitenkin muutamia mahdollisia parannuksia ja ehdotuksia, joilla NoSQL-kannasta saa turvallisemman. Lopuksi on huomautettava, että koska taulujen rakennetta muutettiin siirtyessä kokoelmiin ja kyseessä on hyvin yksinkertainen tietokanta, kyseessä ei ole migraatio. Mutta koska siis tässä tapauksessa yksi ihminen hoitaa koko verkkokaupan (sovellus, kannat, käyttöliittymä) eikä se ole tuotantokäytössä, on täysin mahdollista tehdä muutoksia kantaan.

Kokonaisuudessaan työ auttoi ensinnäkin selvittämään NoSQL-kantojen toimintaa hyvin laajasti, ja toiseksi se antoi hyvin osviittaa siitä, mitä on odotettavissa, jos halutaan todellakin ottaa valmiiseen sovellukseen NoSQL-tietokanta käyttöön.

Lähteet

A Timeline of Database History. 2017. Verkkoaineisto.
<https://www.quickbase.com/articles/timeline-of-database-history>. Luettu 17.6.2018.

An introduction to Redis data types and Abstractions. 2018. Verkkoaineisto.
<https://redis.io/topics/data-types-intro>. Luettu. 2.7.2018.

Codd's 12 Rules. 2018. Verkkoaineisto.
<https://www.tutorialspoint.com/dbms/dbms_codds_rules.htm.>. Luettu 2.9.2018.

DBMS - Database Schemas. 2018. Verkkoaineisto.
<https://www.tutorialspoint.com/dbms/dbms_data_schemas.htm>. Luettu 15.6.2018.

DBMS - Indexing. 2018. Verkkoaineisto.
<https://www.tutorialspoint.com/dbms/dbms_indexing.htm>. Luettu 2.8.2018.

Document Database. 2017. Verkkoaineisto. <https://www.mongodb.com/document-databases>. Luettu 12.6.2018.

Document Databases Explained. 2018. Verkkoaineisto.
<http://basho.com/resources/document-databases/>. Luettu 12.6.2018.

Encryption at Rest. 2018. Verkkoaineisto.
<https://docs.mongodb.com/manual/core/security-encryption-at-rest/>. Luettu 1.9.2018.

Getting Started With Graph Databases. 2018. Verkkoaineisto.
<https://academy.datastax.com/resources/getting-started-graph-databases>. Luettu 1.7.2018.

Guide to Wide-Column Store. 2017. Verkkoaineisto. <http://nosql-guide.com/nosql-databases-explained-wide-column-stores/>. Luettu 15.6.2018.

Hakkarainen, Anssi. 2011. Oracle-tietokannan tehokas hallinta. Helsinki: readme.fi.

High Availability (HA). 2016. Verkkoaineisto. <https://www.techopedia.com/definition/1021/high-availability-ha>. Luettu. 15.7.2018.

JavaScript Object Notation (JSON). 2016. Verkkoaineisto. <https://www.techopedia.com/definition/3930/javascript-object-notation-json>. Luettu 5.6.2018.

Key-value Databases Explained. 2017. Verkkoaineisto. <http://basho.com/resources/key-value-databases/>. Luettu 13.6.2018.

NoSQL Databases Explained. 2018. Verkkoaineisto. <https://www.mongodb.com/nosql-explained>. Luettu 7.10.2018.

RavenDB. 2018. Verkkoaineisto. <https://ravendb.net/features>. Luettu 1.7.2018.

RavenDB Product. 2018. Verkkoaineisto. <https://ravendb.net/buy>. Luettu 1.7.2018.

Redis FAQ. 2018. Verkkoaineisto. <https://redis.io/topics/faq>. Luettu 2.7.2018.

Schema Validation. 2018. Verkkoaineisto. <https://docs.mongodb.com/manual/core/schema-validation/>. Luettu 1.9.2018.

Structured Query Language (SQL). 2016. Verkkoaineisto. <https://www.techopedia.com/definition/1245/structured-query-language-sql>. Luettu 5.6.2018.

What is MongoDB?. 2018. Verkkoaineisto. <https://www.mongodb.com/what-is-mongodb>. Luettu 20.6.2018.

What is NoSQL?. 2016. Verkkoaineisto. <https://www.mongodb.com/nosql-explained>. Luettu 1.7.2018.

Why NoSQL Database?. 2018. Verkkoaineisto. <https://www.couchbase.com/resources/why-nosql>. Luettu 3.10.2018.

SQL-vedokset

Tässä liitteessä on SQL-tietokannassa olevien taulujen it, eli rakenne. it eivät sisällä mitään tietueita, koska ne luotiin uudestaan aiempien taulujen tilalle tätä työtä varten. Niissä on määritelty kaikki parametrit ja avaimet, jotka tauluissa oli suorituskykymittausten aikana.

```
CREATE TABLE `kayttaja` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `tunnus` varchar(255) NOT NULL,
  `salasana` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=24 DEFAULT CHARSET=utf8;
```

Esimerkkikoodi 1. Taulun kayttaja-luontiskripti.

Esimerkkikoodilla 1 luotiin SQL-tietokanta, koska se piti tätä työtä varten siirtää muualta uuteen MySQL-ympäristöön.

```
CREATE TABLE `kirja` (
  `kirja_id` int(11) NOT NULL AUTO_INCREMENT,
  `kirja_nimi` varchar(255) NOT NULL,
  `ISBN` bigint(255) NOT NULL,
  `hinta` decimal(10,2) NOT NULL,
  `julkaisuvuosi` int(11) NOT NULL,
  `lkm` int(11) NOT NULL DEFAULT '100',
  `kirjailija_id` int(11) NOT NULL,
  PRIMARY KEY (`kirja_id`),
  KEY `FOREIGN` (`kirjailija_id`)
) ENGINE=InnoDB AUTO_INCREMENT=27 DEFAULT CHARSET=utf8;
```

Esimerkkikoodi 2. Taulun kirja-luontiskripti.

Yllä oleva skripti (koodi 2) on kirja-taulun luonnissa käytetty skripti.

```
CREATE TABLE `kirjailija` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `etunimi` varchar(255) DEFAULT NULL,
  `sukunimi` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=16 DEFAULT CHARSET=utf8;
```

Esimerkkikoodi 3. Taulun kirjailija-luontiskripti.

Koodilla 3 on luotu kirjailija-taulu. Tässä liitteessä olevista luontiskripteistä näkee siis kuinka käytössä olevat SQL-kannan taulut on luotu.

Suorituskykymittausten tulokset

Tässä liitteessä on taulukoitu suorituskykymittausten tarkat ajat, joista on otettu alilukuun 6.3 keskiarvot sekä alimmat ja korkeimmat ajat.

Taulukko 1. Koodien 1 ja 5 suoritusajat.

Kerrat	**Suoritusaika (ms)**	
	MySQL	**MongoDB**
1	66,81012	25
2	56,35917	18
3	54,47416	18
4	51,80572	18
5	64,583	18
6	62,6	18
7	49,86341	18
8	65,97452	18
9	78,13173	18
10	55,6	17
11	11,65217	21
12	13,733133	21
13	14,35182	21
14	17,09852	21
15	10,61259	21
16	55,5	18
17	50	18
18	54,7	18
19	65,7	18
20	58,8	18

Taulukossa 1 on tarkat suoritusajat koodien 1 ja 5 ajoista. Ne on merkitty millisekunteina (ms). Niiden perusteella on muodostettu taulukko 2 aliluvussa 6.3.

Taulukko 2. Koodien 2 ja 6 suoritusajat.

Kerrat	Suoritusaika (ms)	
	MySQL	MongoDB
1	70,9633	18
2	68,07377	19
3	66,00317	24
4	63,79273	17
5	70,72853	17
6	65,23428	17
7	71,18567	17
8	68,43512	17
9	84,84259	18
10	75,80326	18
11	67,5991	17
12	66,3038	18
13	67,98268	18
14	64,67066	17
15	64,67665	17
16	71,43113	17
17	63,57421	18
18	73,71213	17
19	68,03058	17
20	69,20358	17

Taulukossa 2 on on tarkat suoritusajat koodien 2 ja 6 ajoista. Ne on merkitty millisekunteina (ms). Niiden perusteella on muodostettu taulukko 3 aliluvussa 6.3.

Taulukko 3. Koodien 3 ja 7 suoritusajat.

Kerrat	Suoritusaika (ms)	
	MySQL	MongoDB
1	7,36215	10
2	7,56869	10
3	7,43741	10

4	8,2435	10
5	8,74255	10
6	11,29766	10
7	9,95703	10
8	8,71732	10
9	7,55031	10
10	10,33805	10
11	9,17617	10
12	8,94054	10
13	9,8467	10
14	8,4	10
15	11,51362	10
16	7,66	10
17	7,63	10
18	7,54	10
19	7,9	11
20	9,4	10

Taulukossa 3 on on tarkat suoritusajat koodien 3 ja 7 ajoista. Ne on merkitty millisekunteina (ms). Niiden perusteella on muodostettu taulukko 4 aliluvussa 6.3.

Taulukko 4. Koodien 4 ja 8 suoritusajat.

Kerrat	**Suoritusaika (ms)**	
	MySQL	**MongoDB**
1	156	0
2	141	0
3	125	0
4	125	0
5	203	0
6	110	0
7	110	0
8	141	0
9	140	0
10	109	0

11	156	0
12	110	0
13	110	0
14	156	0
15	125	0
16	125	0
17	109	0
18	125	0
19	78	0
20	109	0

Taulukossa 4 on on tarkat suoritusajat koodien 4 ja 8 ajoista. Ne on merkitty millisekunteina (ms). Niiden perusteella on muodostettu taulukko 5 aliluvussa 6.3.

Taulukko 5. Indeksikyselyiden suoritusajat.

Kerrat	**Suoritusaika (ms)**	
	MySQL	**MongoDB**
1	0,54	0
2	0,5	0
3	0,416	0
4	0,37	0
5	0,417	0
6	0,554	0
7	0,566	0
8	0,39	0
9	0,4	0
10	0,399	0
11	0,669	0
12	0,48	0
13	0,412	0
14	0,49	0
15	0,427	0
16	0,44	0
17	0,38	0

18	0,48	0
19	0,41	0
20	0,46	0

Taulukossa 5 on on tarkat suoritusajat indeksikyselyiden ajoista. Ne on merkitty millisekunteina (ms). Niiden perusteella on muodostettu taulukko 6 aliluvussa 6.3.1.