TAMPEREEN
AMMATTIKORKEAKOULU

TIETOKANNAT JA SP-APPLIKAATIOT

Arttu Kaarre

Opinnäytetyö
Tammikuu 2017
Tietotekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

ARTTU KAARRE:
Tietokannat ja SP-sovellukset

Opinnäytetyö 29 sivua, joista liitteitä 0 sivua
Maaliskuu 2017

Opinnäytetyön tarkoituksena oli tutkia relaatiokantojen ja dokumenttikantojen toiminnallisia eroja ja selvittää, voiko relaatiokantoja käyttää reaaliajassa päivittyvien verkkoapplikaatioiden (toisin sanoen single-page -applikaatioiden) päätietokantoina.

Työssä vertaillaan relaatiokantoja ja dokumenttikantoja sekä esitellään kolme toisistaan eroavaa ajattelutapaa, joilla tyypillisesti hitaan relaatiokannankin kanssa voidaan kehittää reaaliajassa päivittyviä verkkoapplikaatioita.

Asiasanat: tietokannat, reaaliaikaiset verkkoapplikaatiot

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
ICT Engineering
Software Engineering

ARTTU KAARRE:
Databases and SP applications

Bachelor's thesis 29 pages, appendices 0 pages
March 2017

The purpose of this thesis was to examine relational databases and document-oriented databases and their functional differences in addition to finding out if relational databases could be used as the main database for real-time web applications (i.e. single-page applications).

The thesis compares relational databases to document-oriented databases and introduces three distinct ways of using a typically slow relational database with a real-time application.

Key words: databases, real-time web applications

SISÄLLYS

LYHENTEET JA TERMIT

SP(A)	Single-page application, verkkoapplikaatiotyyppi
CRUD	Create, read, update, delete, tyypilliset tietokantaoperaatiot
SQL	Structured Query Language, relaatiokantakyselykieli
MySQL	Relaatiokantaohjelmisto
CouchDB	Dokumenttikantaohjelmisto
ACID	Atomicity, consistency, isolation, durability, tietokannan datan eheyden turvaamisen periaate
NoSQL	Tietokantaohjelmistot, jotka eivät käytä SQL-tyyppistä kyselykieltä operaatioihin
Spring Framework	Pivotal Softwaren kehittämä ohjelmistokehys javalle
Springboot	Spring Frameworkista virtaviivaistettu ohjelmistokehys verkkoapplikaatioilla
Java	Sun Microsystemsin kehittämä ohjelmointikieli
HTTP	Hypertext Transfer Protocol, tiedonsiirtoon käytetty protokolla
TCP	Transmission Control Protocol, tietoliikenneprotokolla
WebSocket	Kaksisuuntainen, TCP-protokollan yli toimiva kommunikointiprotokolla
STOMP	WebSocketin avulla toimiva viestintäprotokolla
W3C	World Wide Web Consortium, verkkostandardeja ylläpitävä ja kehittävä organisaatio
Java bean	Uudelleenkäytettävä java-komponentti

1 JOHDANTO

Verkkosivujen kehittyessä hiljalleen enemmän ja enemmän verkkoapplikaatioiksi, selaimen kautta käytettäviksi ohjelmistoiksi, sivujen sujuvan käyttökokemuksen takaamiseksi taustalla pyörivä teknologiakin kehittyy jatkuvasti. Erityisesti single-page -tyyppiset applikaatiot ovat keränneet viime vuosina suosiota. Single-page -applikaatio tarkoittaa käytännössä sitä, ettei sivua päivitetä enää sivulle saapumisen jälkeen, vaan sisältö piirretään selaimeen dynaamisesti.

Single-page -applikaatioiden toiminnallisen luonteen vuoksi sivulla näkyvän datan olisi hyvä käyttäjäpäässä päivittyä reaaliajassa sitä mukaa, kun datalle tapahtuu muutoksia palvelimen tietokannassakin. Tälläistä toiminnallisuutta erityisesti tukemaan on kehitetty muun muassa dokumenttikantoja, joilla päivittyneestä datasta on helppo ilmoittaa käyttäjille, sillä perinteiset relaatiokannat eivät tälläiseen toiminnallisuuteen normaalisti jousta.

Työn tarkoituksena oli tutustua tarkemmin relaatiokantojen ja dokumenttikantojen toimintaan MySQL-relaatiokannan ja CouchDB-dokumenttikannan kautta, ja selvittää, onko relaatiokantojen käyttäminen reaaliaikaisten single-page -applikaatioiden kanssa mahdollista. Työssä esitellään muutamia tapoja joilla relaatiokantaa voitaisiin käyttää reaaliaikaisen applikaation päätietokantana ja analysoidaan tapojen vahvuuksia ja heikkouksia.

2 TIETOKANNAT

Tietokannoilla tarkoitetaan tietokoneen muistissa pysyvää kokoelmaa dataa, jota voidaan helposti hakea ja käsitellä käyttäjän määrittelemien ehtojen mukaan. Tietokantatyypistä ja kannan mahdollisista määritellyistä rajoitteista riippuen tallennettu data voi olla formaatiltaan mitä tahansa pelkästä tekstistä ja kokonaisluvuista kuviin, videoihin sekä ääneen. Perinteisesti tietokantoja käytetään esimerkiksi yrityksen asiakkaiden tietojen tallentamiseen, mutta käyttötarkoituksia on lukemattomia, joten tietokantatyyppejäkin löytyy useampia eri tarkoituksiin.

Tässä luvussa esitellään useimmille tutummat relaatiokannat sekä viime vuosina kasvavissa määrin suosiota keränneet dokumentti-pohjaiset kannat.

2.1 Relaatiotietokannat

Tyypillisesti tietokannoista puhuttaessa tarkoitetaan relaatiokantaa. 1970-luvulla kehitetty relaatioalgebraan perustuva tietokantamalli on ollut ensimmäisestä käyttöönotostaan lähtien vielä tähän päivään asti suosituin ja käytetyin tietokantatyyppi. (IBM100) Yksi suosituimmista relaatiokantaohjelmistoista on MySQL.

Relaatiokannat koostuvat kaksiulotteisista tauluista, joissa sarakkeet esittävät kenttiä ja rivit sisältävät itse tallennettua dataa. Yksinkertainen esimerkki yhdestä relaatiokantataulusta voisi olla esimerkiksi henkilöstötaulu (kuva 1).

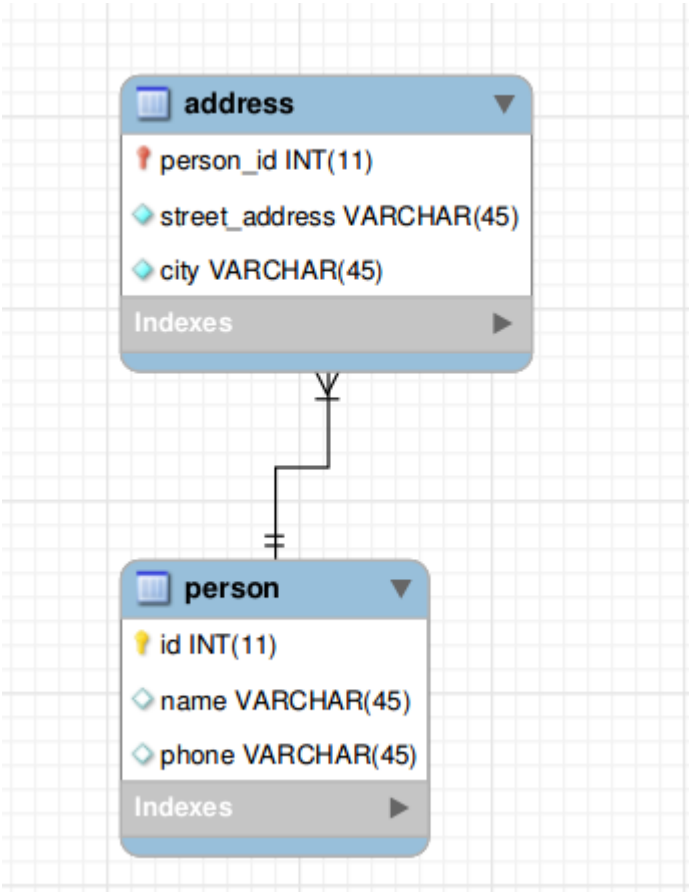
#	id	name	phone	address
1	1	Arttu	0400000000	Teiskontie, TAMPERE
2	2	Person2		Address, CITY
3	3	Person3	555-545454	Street 210, CITY 2

KUVA 1. Kuvakaappaus yksinkertaisesta henkilöstötaulusta MySQL Workbench -ohjelmasta

Sarakkeille voidaan määritellä tyypin lisäksi erilaisia rajoitteita (constraint), joilla kantaan tallennettavan datan muotoa voidaan ohjata haluttuihin suuntiin. Kuvan 1

esimerkissä sarakkeella id on rajoitteet NOT NULL, PRIMARY KEY ja AUTO_INCREMENT. Näiden mukaan sarakkeella id täytyy aina olla jokin arvo, sarakkeen arvo on yhden tietueen uniikki tunniste joten taulussa ei voi olla kahta riviä samalla id-arvolla, ja id:n arvo kasvaa itsestään tauluun lisättäessä dataa. Name-, phone- ja address -sarakkeet ovat tyypiltään tekstimuotoisia, name- ja address -sarakkeille on määritelty maksimipituudeksi 100 merkkiä ja phone-sarakkeelle on määritelty pituudeksi täsmälleen kymmenen merkkiä. Muutoin sarakkeiden arvo voi olla mitä tahansa, vaikka tyhjä. (MySQL Reference)

Relaatiokannoista erityisen tekee foreign key -rajoite. Nimessä oleva sana relaatio tulee todellisuudessa matematiikan puolelta relaatioalgebrasta, mutta sitä voidaan miettiä myös taulujen välisinä suhteina, jotka foreign key mahdollistaa. (Redmond, E., Wilson, J. R. 2012.) Käytännössä foreign key -määritelty sarake taulussa tarkoittaa sitä, että kyseinen arvo viittaa jonkin toisen taulun määriteltyyn uniikkiin arvoon, ja näin sitoo kahdesta erillisestä taulusta tietyt tietueet yhteen. Näillä taulujen välisillä suhteilla voidaan pitää tietyt datakokonaisuudet omissa rajoitteissaan ja mahdollistaa esimerkiksi 1-n -suhteet. Yksinkertainen esimerkki tälläisestä suhteesta voisi olla henkilötietojen erottaminen osoitetiedoista, eli kuvan 1 esimerkin address-sarake poistettaisiin kokonaan person-taulusta ja sille luotaisiin oma address-taulu sen tilalle, josta löytyisi osoitetiedot sekä person_id-niminen kenttä. Person_id on itse foreign key -määritelty avain, joka sitoo taulut person ja address yhteen. Siinä missä person-tauluun lisättäessä uudet datarivit saavat automaattisesti generoidun arvon id-sarakkeeseen, address-tauluun dataa lisättäessä person_id -sarakkeen arvoksi asetetaan osoitetietoja vastaavan henkilön rivin id person-taulussa. Esimerkiksi MySQL Workbenchillä voidaan generoida visuaalinen esitys taulujen välisistä suhteista tietokannassa (kuva 2). (MySQL Reference)



KUVA 2. Kuvakaappaus MySQL Workbenchin generoimasta suhdediagrammista

Diagrammista selviää myös taulujen 1-n -suhde. Henkilöllä voi olla useampi osoite tässä datakoosteessa, mutta osoitteilla voi olla vain yksi osoitteen ”omistava” henkilö. Muita mahdollisia taulujen välisiä suhteita ovat 1-1 sekä n-m, eli yhtä datariviä vastaa toisessa taulussa myös vain yksi datarivi, tai useampia datarivejä voi vastata toisessa taulussa kaksi tai useampia datarivejä. (MySQL Reference)

Tätä dataa haetaan, muokataan ja muutoin käsitellään SQL-lauseilla. SQL on lauseisiin ja ehtoihin perustuva tehokas kyselykieli (query language). Useimmille käyttäjille tutuin SQL-lause on SELECT, eli datan noutaminen tiettyjen ehtojen mukaisesti. Yksinkertaisimmillaan SELECT on kaiken datan hakeminen tietystä taulusta. Esimerkiksi henkilötaulun kaikki rivit (kuva 1) saa ajamalla komennon SELECT * FROM db.person;.

Tehokkaan kyselykielen SQL:stä tekee sen perustuminen matemaattisiin kaavoihin ja nimenomaan relaatioalgebraan, joka parhaiten käyttäjälle näkyy JOIN-lausekkeen kautta. (Redmond, E., Wilson, J. R. 2012.) JOIN-lausekkeella käyttäjä pystyy hakemaan tai manipuloimaan useampien taulujen dataa yhdellä ajettavalla komennolla. Person- ja

address -taulujen data voidaan esittää samassa taulussa esimerkiksi ajamalla komento SELECT * FROM db.person JOIN db.address WHERE person.id = address.person_id;

#	id	name	phone	person_id	street_address	city
1	4	Person 1	phone 1	4	Address 1	City 1

KUVA 3. Kuvakaappaus Workbenchillä haetusta datasta JOIN-lauseketta käyttäen

Tietokannassa ajettavat SQL-kyselyt voivat olla nk. ad hoc -kyselyitä, eli useimmiten koodissa määriteltyjä tilannekohtaisia kyselyitä, tai muistiin tallennettuja toimintoja (stored procedure), jotka ovat jossain mielessä tehokkaampia, mutta muuttumattomia kyselyitä. (MySQL Reference)

Relaatiokannat ovat ”design-first” -tyyppisiä tietokantoja. Datan muoto ja esitystapa, eli käytännössä tietokannan skeema, täytyy määritellä jo tietokannan luontivaiheessa. Kantaan syötettävän datan täytyy sopia tähän malliin, ja vaikka jälkikäteen skeeman muuttaminen onkin mahdollista on se huomattavasti vaivalloisempaa, erityisesti jos kannasta löytyy jo dataa. (MySQL Reference)

Operaatiot relaatiokannoissa tapahtuvat transaktioissa, jotka mahdollistavat lisäkerroksen datan eheänä ja korruptoitumattomana pitämisen varmistamiseksi. Käytännössä transaktionaalisuus tarkoittaa sitä, että jos kesken suoritettavan operaation tapahtuu jotain odottamatonta, mahdolliset jo tapahtuneet muutokset peruuntuvat ja tietokanta palautuu siihen tilaan, missä se oli ennen operaation alkamista. Näin vältytään esimerkiksi mahdollisilta sähkökatkosten aiheuttamilta datakorruptioilta. Transaktionaalisuuden periaattelle on määritelty muun muassa ACID-malli (atomicity, consistency, isolation, durability) jolla korostetaan tietokannan luotettavuutta. (MySQL Reference)

2.2 Dokumenttikannat

Toinen tälle opinnäytetyölle relevantti tietokantamalli on dokumenttikanta (document-oriented NoSQL). Dokumenttikantoja tutkaillaan CouchDB:n kannalta, joka on Apachen Erlangilla kehittämä 2000-luvun puolivälissä julkaistu avoimen lähdekoodin tietokantaohjelmisto. (CouchDB Archive)

Kuten nimestä voi päätellä, NoSQL-kantojen suurimmat erot relaatiokantoihin ovat tiukasti määritellyn relaatio- sekä data-rakenteen ja SQL-tyyppisen kielen puuttuminen, ja dokumenttikannat ovat yksiä suosituimmista NoSQL-variaatioista. Siinä missä relaatiotietokantojen kanssa kehittäjä joutuu määrittelemään tiukat rajoitteet ja muotomäärittelyt kantaan syötettävälle datalle ennen kannan käyttöönottoa, dokumenttikannat eivät useimmiten ota käytännössä mitään kantaa datan rakenteeseen liittyen. (CouchDB Documentation)

Dataa talletetaan kantaan yksinkertaisina avain-dokumentti -pareina. Avain on luonnollisesti datalle uniikki, ja sen muoto vaihtelee käytetystä kannasta riippuen. Dokumentti, eli itse talletettu data, on useimmiten JSON- tai BSON (binary JSON) -muotoista tekstiä. Muita määritelmiä datan muodolle ei sinänsä ole; kunhan data on esimerkiksi validi JSON-objekti, voi itse objekti pitää sisällään mitä tahansa muutamasta lyhyestä kentästä massiivisiin hierarkisiin objektikokoelmiin. (CouchDB Documentation)

Useimpien dokumenttikantojen kanssa datan hakeminen ja käsittely on toteutettu mahdollisimman universaalilla ja liikuteltavalla tavalla; yleensä kaikki käsittelyt hoidetaan HTTP-kutsuja käyttäen. Esimerkiksi paikallisella koneella pyörivän CouchDB:n kanssa uuden tietokannan luominen onnistuu lähettämällä PUT-tyyppinen HTTP-kutsu osoitteeseen http://127.0.0.1:5984/nameOfDb. Tähän kantaan uuden datan lisääminen onnistuu yksinkertaisella POST-kutsulla samaan osoitteeseen, jossa HTTP-headereina kerrotaan lähetettävän datan muoto (JSON) ja bodyssä kuljetetaan itse talletettava data. (CouchDB Documentation)

Datan hakeminen tapahtuu näkymien (views) avulla. Näkymät ovat yksinkertaisia javascript-funktioita, jotka ehtolauseiden perusteella palauttavat haluttuja dokumentteja kannasta. Kaikissa CouchDB-kannoissa on valmiiksi by_id-niminen näkymä, joka ottaa parametrikseen id:n ja palauttaa kannasta dokumentin, jonka _id-kenttä täsmää annetun parametrin kanssa. Jos kannasta halutaan hakea esimerkiksi name-kentän perusteella dokumentteja, täytyy kehittäjän itse ohjelmoida by_name-näkymä, joka vertailee parametrina annettua tekstijonoa dokumenttien name-kenttien arvoihin. (CouchDB Documentation)

CouchDB, kuten muutkin suositut dokumenttikannat, ovat ”append-only” -tyyppisiä datan muokkaamisen suhteen. Vapaasti suomennettuna ”append-only” tarkoittaa ”lisäyksiin rajattua”, ja käytännössä se tarkoittaa sitä, että kaikki operaatiot – uuden datan lisääminen sekä vanhan muokkaaminen ja poistaminen – todellisuudessa lisäävät uusia dokumentteja tai dokumentti-revisioita tietokantaan. Esimerkiksi poistettaessa dokumentti, siitä todellisuudessa luodaan uusi revisio, joka on merkitty poistetuksi, esimerkiksi luomalla sille arvokenttä nimeltään _deleted ja asettamalla sen arvoksi true. Näin mahdollistetaan muun muassa helppo muutosten seuranta; tietokanta voidaan kuvitella pitkänä listana dokumenttimuutoksia, jonka loppuun lisätään aina rivi kun jokin, mikä tahansa, muutos tapahtuu. (CouchDB Documentation)

Monet dokumenttikannat, ja erityisesti CouchDB, ovat hajautetun luonteensa vuoksi hyvin skaalautuvia, ja CouchDB:n mainostetaankin pyörivän hyvin globaalin tason klustereissa, kevyillä älypuhelimilla ja kaikella niiden väliltä. (CouchDB Documentation)

2.3 Vertailua

Kuten selvistä toiminnallisista ja rakenteellisista eroista voikin päätellä, relaatio- ja dokumenttikannoilla on molemmilla laaja kirjo vahvuuksia ja heikkouksia toisiinsa verrattaessa. Molempien käsitteiden suhteellisen laajan kattavuuden vuoksi tarkemmat vertailut tehdään MySQL:n ja CouchDB:n välillä.

Tietokantaa valittaessa ensimmäisenä täytyy harkita applikaatiossa liikkuvan ja käsiteltävän datan rakennetta, tyyppiä ja mahdollista vapaamuotoisuutta. Relaatiokannat ovat useimmiten kehittäjien ensimmäinen valinta yksinkertaisesti jo niiden suuresta vuosikymmeniä jatkuneesta suosiosta johtuen, mutta relaatiokantoihin dataa tallennetaan tiukkojen, ennalta määritettyjen sääntöjen mukaan, joka lisää ennalta suunniteluun vaadittua aikaa sekä asettaa tiukkoja rajoja applikaation alalle (scope). Toisaalta tiukasti määritelty datan muoto mahdollistaa helpon integraation erityisesti oliokielten kanssa työskennellessä. Myös saapuvan datan muodosta ei kehittäjän tarvitse usein niinkään enää applikaation rajapintoja kehittäessään välittää, sillä tietokanta hylkää virheviesteineen automaattisesti dataa, joka ei muodoltaan sovi haluttuun tauluun.

Dokumenttikannat lähestyvät datan muotoa täysin toisesta suunnasta, unohtaen kaikki käyttäjälle tai kehittäjälle osoitetut rajoitteet. Ainoa kaikkia kantaan tallennettuja dokumentteja yhdistävä tekijä ovat CouchDB:n kohdalla dokumentiltä löytyvät _id ja _rev -kentät, dokumentin erottava uniikki tunniste sekä kyseisen dokumentin revisio. Muutoin kaikki dokumentit ovat mielivaltaisen kokoisia sekä muotoisia JSON-objekteja. Tämä mahdollistaa muun muassa sen, että kanta voidaan yksinkertaisesti vain ottaa käyttöön, ja applikaatiota päästään kehittämään nopeammin. Luonnollisesti tämä tarkoittaa kuitenkin myös sitä, että myöhemmin, jos datan täytyy olla muodoltaan tarkkaan määriteltyä – esimerkiksi tallennetuista asiakastiedoista täytyy aina löytyä tietyt kentät, aina tietyssä muodossa tai rajoitteiden sisässä – joutuvat kehittäjät käsin ohjelmoimaan validaattoreita. Myös datan eheyden tarkistaminen on huomattavasti vaivalloisempaa ilman relaatiokantojen tarjoamaa foreign key -tyyppistä ratkaisua jos kantaan talletetaan useita toisiin dokumentteihin viittaavia dokumentteja, sillä tietokanta itse ei ota dokumenttien sisältöön myöskään tältä osin mitään kantaa. (CouchDB Documentation)

Datan korruptoitumisen ja siltä suojautumisen suhteen molemmat kannat ovat varsin päteviä, joskin relaatiokannat vaativat enemmän työtä ja tietoa kehittäjiltään. Transaktiot ovat kriittinen tekijä mahdollisilta korruptoitumisilta suojautumiseen, eikä edes kaikki MySQL:n tukemat tietokantamoottorit tue transaktioita. Myös tietokantakäsittelyt ja rajapinnat palvelinkoodissa täytyy konfiguroida transaktionaalisiksi, joka vaatii kehittäjältä ymmärrystä niin transaktioiden toimintaperiaatteiden kuin käytössä olevan kielen ja ohjelmistokehyksenkin suhteen. Todellisuudessa nämä eivät kuitenkaan ole suuria esteitä, sillä MySQL käyttää oletusarvoisesti InnoDB:tä tietokantamoottorinaan josta löytyy tuki ACID-mallia noudattaville transaktioille, ja palvelinkoodin konfigurointi transaktionaaliseksi onnistuu esimerkiksi javaa käytettäessä helposti annotaatioilla. (MySQL Reference)

CouchDB:n ”append-only” -luonteen vuoksi korruptoitumisilta suojautumiseen ei kehittäjän tarvitse tehdä yhtään mitään – koskaa vanhaa dataa ei todellisuudessa koskaan muokata tai poisteta suoraan dokumenttikantaa käytettäessä, ei ole datan korruptoitumisen mahdollisuuttakaan. (CouchDB Documentation)

MySQL ja CouchDB eroavat myös varsin radikaalisti kyselyiden nopeuden sekä joustavuuden suhteen. Relaatiokantojen käyttämä SQL-kyselykieli on hyvin voimakas ja

mahdollistaa mitä monimutkaisimpien kyselyiden tekemisen, joten käyttäjälle palautuvan datan muoto on täysin kehittäjän määriteltävissä, ja kyselyistä voidaan tehdä hyvinkin yksityiskohtaisia tai tarkkoja – voidaan palauttaa niin yksittäisiä kenttiä yhdestä tietueesta kuin monimutkaisia, ehtolauseilla ja JOIN-lausekkeilla rakennettuja datakokoelmiakin. SQL-kyselyt käyvät kuitenkin nopeasti raskaiksi monimutkaisuuden lisääntyessä, joten jos kehittäjät päättävät käyttää kielen kaikkia työkaluja, on myös varauduttava todennäköiseen optimointityöhön myöhemmässä vaiheessa. (MySQL Reference)

CouchDB ei muodostaan johtuen käytä mitään määriteltyä kyselykieltä, vaan näkymiä. Näkymät ovat tehokas tapa hakea dataa kannasta, eikä kyselyitä sinänsä tarvitse koskaan optimoida, mutta tässäkin on kääntöpuolensa. Ensinnäkin, kaikille kyselyille täytyy määritellä erillinen näkymä – nimen, päivämäärän tai jonkun muun kentän perusteella hakemiseen kehittäjän täytyy itse kirjoittaa lyhyt javascript-funktio – näkymä – jossa annettua parametria verrataan haluttuun kenttään dokumenteissa. Vaikka näkymissä voi yrittää palautetun datan muotoa vähän rajoittaa, tyypillisesti CouchDB palauttaa myös kokonaisen dokumentin aina kyselyn vastaukseksi. Jos haluaa hakea tietyn blogipostauksen id:n perusteella, saattaa palautuva JSON-objekti olla huomattavasti odotettua suurempi johtuen siitä, että se sisältää esimerkiksi myös kaikki kommentit, jota kyseiselle postaukselle on tehty. (CouchDB Documentation)

3 RELAATIOKANNAT JA SINGLE-PAGE -APPLIKAATIOT

Relaatio- ja dokumenttikantojen eroavaisuuksista ja eri heikkouksista ja vahvuuksista voi jo päätellä vähintään sen, että niiden oikean maailman käyttökohteet ovat useimmissa tilanteissa täysin erilaiset. Jos kyseessä on kevyen web-applikaation tietokanta, jossa lähinnä ylläpidetään esimerkiksi käyttäjien lähettämiä viestejä tai blogisivujen sisältöä, miksi käyttää datan muodon suhteen niin kriittistä kantaa kuten MySQL? Ja vastaavasti jos kyseessä on iso määrä asiakasdataa jossa datan loogisuus ja täsmällinen muoto ovat ensisijaisia, miksi käyttää CouchDB:tä?

Nämä tilanteet eivät aina kuitenkaan ole niin yksioikoisia. Joskus yksi kanta tarjoaa erityisen haluttuja ominaisuuksia, mutta esimerkiksi datan muodosta johtuen on pakko tehdä kompromissi ja käyttää kuitenkin toista kantaa.

Tämän opinnäytetyön ajava kysymys on: voiko relaatiokantaa realistisesti käyttää päätietokantana käyttäjän päässä reaaliajassa päivittyvässä web-applikaatiossa, eli nk. single page -applikaatiossa? Tyypillisesti näiden applikaatioiden kanssa käytetään esimerkiksi dokumenttikantoja, joista löytyy useimmiten sisäänrakennetut notifikaatio-systeemit; jos käyttäjä tutkailee dataa y, ja data y muuttuu jonkin muun vaikuttajan toimesta, saa käyttäjä muutoksesta ilmoituksen joutumatta itse tekemään kyselyä datan senhetkisestä muodosta. Relaatiokannat eivät itsessään tälläistä käyttäytymistä tue, mutta joskus tulee vastaan tilanteita, joissa taustalla pyörivä tietokanta on pidettävä relaatiokantana esimerkiksi mahdollisista infrastruktuurisyistä, mutta käyttöliittymästä halutaan kuitenkin modernimpi, reaaliaikaisempi single page -applikaatio.

4 LÄHESTYMISTAPOJA

Tässä luvussa käydään läpi erilaisia lähestymistapoja reaaliaikaisten applikaatioiden ongelman ratkaisuun. Esimerkeissä käytetään tietokantoina MySQL-relaatiokantaa sekä CouchDB-dokumenttikantaa. Ohjelmointikielenä käytetään javaa, ja ohjelman konfiguroinnin yksinkertaistamiseksi käytössä on Spring frameworkin web-applikaatioiden helppoon kehittämiseen luotu Springboot, joka mahdollistaa myös yksinkertaisten REST-rajapintojen kehittämisen applikaatiolle. Frontend pidetään hyvin yksinkertaisena Reactilla toteutettuna henkilöstölistan esittävänä applikaationa.

Lähestymistapojen esimerkeissä ei tulla esittelemään yksityiskohtaisesti esimerkkiohjelman jokaikistä koodiriviä, vaan ennemminkin ideoita, esimerkille kriittisiä koodinpätkiä ja logiikkaa ohjelmointiratkaisujen takana.

Esimerkkitestausta suoritetaan Lenovon ThinkPad X230 -mallisella kannettavalla tietokoneella, josta löytyy SSD-tyyppinen kovalevy.

4.1 Päivitysten jatkuva kyseleminen

Yksinkertaisin tapa tehdä ohjelmasta ainakin näennäisesti reaaliaikainen on konfiguroida frontendin koodi hakemaan näkyvälle datalle päivityksiä esimerkiksi ajax-kutsuilla jatkuvasti. Ajax-kutsuja voi lähettää automaattisesti tietyin väliajoin ja pyyntöjen lähettämistä voidaan rajoittaa esimerkiksi javascriptin windows.onblur- ja windows.onfocus -tapahtumilla niin, että pyyntöjä lähetetään ainoastaan, kun applikaatiosivu on aktiivisena.

Tälläisellä lähestymistavalla tietokantakyselyitä syntyy kuitenkin väistämättä satoja vain minuuteissa pienelläkin käyttäjämäärällä, ja on todennäköistä, että miltei kaikki näistä kyselyistä ovat vieläpä täysin turhia. Tyypillisesti nimenomaan paljon tauluja yhdistelevät relaatiokantakyselyt ovat yksiä suurimmista serverikoneiden toimintaa hidastavista tekijöistä, joten jos applikaation tietokantamallissa on vähänkin monimutkaisuutta ja foreign key -arvoilla toisiinsa sidottuja tauluja, ei tälläinen tapa toimi missään nimessä.

Jos käytössä on vain yksi suhteellisen yksinkertainen taulu josta haetaan ja johon tallennetaan dataa, voisi jatkuva pollaaminen olla teoriassa mahdollinen lähestymistapa. Nopeuden testaamiseksi luotiin yksinkertainen taulu jossa on numeromuotoinen id ja sen lisäksi 12 VARCHAR(45)-muotoista datakolumnia, jossa voi olla esimerkiksi nimiä, osoitteita, puhelinnumeroita tai vastaavia tietoja. Tauluun lisättiin 1000 riviä satunnaista dataa. Kuormitusta synnytettiin avaamalla selaimessa 10 eri välilehdelle node-applikaatio, joka hakee koko taulun sisällön ajax-kutsulla kerran sekunnissa, eli keskimäärin SELECT-kyselyitä suoritettiin 10 sekunnissa. Itse tietokannan nopeuden ja lähiverkon yli olevan todellisen nopeuden testaamiseksi tehtiin koko taulun valitsevia kyselyitä MySQL Workbenchillä sekä Postmanilla.

TAULUKKO 1. Kuorman alla olevan SQL-taulun vastausnopeuksia

	Hitain kysely (ms)	Nopein kysely (ms)	10 kyselyn ka. (ms)
MySQL WB	2,9	0,56	1,389
Postman	383	164	233,7

Tästä voitaneen päätellä että jos applikaatiossa käsitellään yksinkertaista dataa ja sillä on hyvin pieni määrä samanaikaisia käyttäjiä, voi applikaation näennäisen reaaliaikaisuuden rakentaa sinänsä jatkuvasti lähetettävien kyselyiden ympärille. Uuden datan näkymisen viive muilla käyttäjillä on kyselyn lähdöstä alle sekunnista muutamaan sekuntiin, jota voidaan useimmissa tapauksissa pitää hyväksyttävänä. Isoimpana vaikuttavana tekijänä tässä on tietysti se, kuinka usein kyselyitä lähetellään, ja applikaation tyypistä riippuen kerran muutamassa minuutissakin voi olla vielä toimiva tiheys.

Todellisuudessa tälläiselle lähestymistavalle ei kuitenkaan todennäköisesti koskaan ole mahdollista käyttökohdetta pelkästään jo sen takia, että relaatiokantojen sovittaminen reaaliaikaisiin applikaatioihin viestii siitä, että käytössä on jo dataa sisältävä relaatiokanta jota ei kuitenkaan voida siirtää helposti esimerkiksi uuteen dokumenttikantaan sen monimutkaisuudesta johtuen. Päivästä toiseen useita kertoja minuutissa tapahtuvat kyselyt synnyttävät myös suunnatonta kuormitusta serverikoneen kovalevylle, joka ei tyypistä riippuen todennäköisesti kauaa kestäisi tälläistä kuormaa.

4.2 Päivitysten tilaaminen WebSocket-yhteyttä hyödyntäen

Loogisempi tapa lähestyä ongelmaa on yrittää löytää tapa implementoida dokumenttikantojen tarjoama ilmoituspalvelu itse. Yksi mahdollinen tapa ratkaista tämä puhtaasti koodimuutoksissa pysyen on WebSocket-protokollan tarjoamat kanavatilaukset.

WebSocket on ensimmäisen kerran vuonna 2009 W3C-organisaation spesifikaatiossa esitelty uudentyyppinen TCP-protokollan päällä toimiva protokolla, joka mahdollistaa serverin ja asiakkaan välisen kaksisuuntaisen liikenteen. Käytännössä tämä tarkoittaa sitä, että asiakas avaa ensimmäisen serverin kanssa käytävän HTTP-kättelyn aikana myös WebSocket-yhteyden serveriin, jonka jälkeen datan molemminpuoliseen liikkumiseen ei tarvita enää muita HTTP-yhteydenottoja.

WebSocket-protokollaa tukeva serveri voi tarjota erinäisiä url-muodossa esitettyjä kanavia asiakkaiden tilattavaksi. Tyypillisesti kanavia käytetään esimerkiksi ”olet saanut uuden yksityisviestin” -tyyppisten ilmoitusten lähettämiseen käyttäjälle. Asiakkaan päässä tilaus tapahtuu javascriptissä SockJS- sekä STOMP -kirjastojen avulla helposti (kuva 4).

```
var socket = new sockjs('http://localhost:8080/stompEndPoint');
stompClient = Stomp.over(socket);
stompClient.connect({}, function (frame){
    // WebSocket-yhteys avattu
    stompClient.subscribe('/topic/personListener', function(event){
        // Callback
    });
})
```

KUVA 4. personListener-kanavan tilaaminen SockJS:ää ja STOMP:ia hyödyntäen

Tilaukseen käytetty subscribe-funktio kutsutaan kahdella parametrilla – ensimmäisenä tilattavan kanavan osoite, toisena callback-funktio, jota kutsutaan aina, kun serveri lähettää viestin kyseiselle kanavalle. Perusideana olisi siis se, että asiakkaan hakiessa normaalilla ajax-kutsulla jotain dataa serveriltä asiakas myös tilaa kyseiseen dataan sidotun kanavan. Kun tietokannassa olevassa datassa tapahtuu muutoksia, serveri lähettää kaikille kanavan tilanneille ilmoituksen muutoksesta, jotta asiakkaat osaavat hakea datan uudestaan (kuva 5).

```
getPersonnel = () => {
    // Haetaan näytettävä data
    jquery.ajax({
        url: 'http://localhost:8080/person',
        success: function(data){
            that.setState({
                displayedPersonnelData: data
            })

            // Tarkistetaan, onko kanava tilattu jo
            if (jquery.inArray("personListener", subs) == -1){
                // Tilataan kanava
                stompClient.subscribe('/topic/personListener', function(event){
                    // Callback; kutsutaan tätä samaa funktiota aina, kun serveri
                    // ilmoittaa henkilöstötiedoissa tapahtuneesta muutoksesta
                    that.getPersonnel();
                });
                // Lisätään kanava tilattujen listalle, ettei kanavaa tilata
                // uudestaan ja uudestaan turhaan
                subs.push("personListener");
            }
        },
        error: function(error){
            // ajax-kutsun virheenkäsittely
        }
    })
}
```

KUVA 5. Datan hakeva ja samalla kanavan tilaava funktio

Serveripäässä WebSocketin käyttöönotto, konfigurointi ja itse käyttäminen vaativat enemmän työtä frontendiin verrattuna, määrä riippuen käytetystä ohjelmakehikosta. Springbootin kanssa käyttöönotto onnistuu suhteellisen vaivattomasti – projektin riippuvuuksiin lisätään Springin oma WebSocket-projekti ja WebSocketille itselleen luodaan yksinkertainen konfiguraatio-luokka (kuva 6).

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config){
        config.enableSimpleBroker( ...destinationPrefixes: "/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry){
        registry.addEndpoint( ...strings: "/stompEndPoint").setAllowedOrigins("*").withSockJS();
    }

    @Bean
    public StompConnectEvent stompConnectEvent() { return new StompConnectEvent(); }
}
```

KUVA 6. WebSocketin konfiguraatio-tiedosto

Itse viestin lähettäminen kanavan tilaajille hoidetaan Springin tarjoaman SimpMessagingTemplate-luokan instanssin tarjoamalla convertAndSend-metodilla (kuva 7).

```
// Vastaanottaa /person -osoitteeseen lähetetyn jsonin
@RequestMapping(method = RequestMethod.POST, value = "/person")
public void savePerson(@RequestBody PersonEntity personEntity){
    // Json mapataan henkilöentiteettiin ja entiteetti tallennetaan kantaan
    PersonEntity pe = personService.save(personEntity);

    // Jos tallennus onnistui, ilmoitetaan personListener-tilaajille muutoksesta
    if (pe != null) {
        template.convertAndSend( destination: "/topic/personListener",  payload: "UPDATE");
    }
}
```

KUVA 7. Uuden henkilön kantaan tallentava metodi

SavePerson-metodi on REST-rajapintaosoitteeseen ”/person” saapuvat POST-tyyppiset HTTP-kutsut käsittelevä metodi. Rajapintaan lähetetään JSON-muodossa kantaan tallennettavaksi halutun henkilön tiedot, jotka @RequestBody-annotaation kautta kartoitetaan automaattisesti henkilö-tyyppiseen entiteettiin helpon tallentamisen mahdollistamiseksi. SimpMessagingTemplaten instanssilla template lähetetään tieto personListener-kanavan tilaajille muutoksesta, jolloin kuuntelijat tietävät tehdä uuden ajax-kutsun päivittyneen henkilöstölistan hakemiseksi.

Päivityksestä kertovien viestien lähettäminen täytyy luonnollisesti tälläisellä lähestymistavalla implementoida manuaalisesti jokaiselle kyseistä dataa muokkaavalle metodille, ja dataa muokkaavia metodeja saattaa erityisen monimutkaisten entiteettien kanssa olla jopa kymmeniä. Myös viestillä itsellään ei kuvan 7 mukaisessa implementaatiossa ole muuta virkaa kuin ilmoittaa jotain tapahtuneen datan suhteen, joten tilaavat asiakkaat päätyvät kuitenkin jokainen tekemään koko taulun kattavan tietokantahaun viestin saadessaan.

STOMP-viestin mukana voidaan kuitenkin kuljettaa myös itse päivitettyä dataa, jolloin tietokantahakuja ei tarvitsisi välttämättä ensimmäistä lukuunottamatta enempää tehdä. Yksinkertainen lähestymistapa olisi kevyen abstraktion esittely koodiin STOMP-viestien lähettämiseen tarkoitetulla metodilla (kuva 8).

```
private void stompMessageSender(String destination,
                                MessageType type,
                                Long id, PersonEntity entity){
    ObjectMapper mapper = new ObjectMapper();
    ObjectNode node = mapper.createObjectNode();
    node.put("type", type.toString());
    node.put("id", id.toString());
    try {
        node.put("data", mapper.writeValueAsString(entity));
    } catch (Exception e){
        //noop
    }

    template.convertAndSend(destination, node);
}
```

KUVA 8. Tilaajille viestien lähettämiseen tarkoitettu metodi

StompMessageSender-metodia voidaan käyttää kaikissa datan päivittymiseen liittyvien viestien lähettämisessä. Metodi ottaa parametreinä kanavan url-osoitteen, datan käsittelyn tyypin (esimerkiksi DELETED, CREATED, UPDATED), käsitellyn datan tunnisteen sekä luonti- ja päivitystapahtumissa itse muuttuneen datan. Nämä arvot muunnetaan JSON-objektiksi, joka lähetetään STOMP-viestin arvona. Viesti on helppo parsia asiakaspäässä javascriptin omilla JSON-kirjastoilla, ja eri tyyppisten käsittelytapahtumien vaatimat toimenpiteet voidaan suorittaa esimerkiksi switch-case -ehdoilla (kuva 9).

```
stompClient.subscribe('/topic/personListener',
function(event){
  var json = JSON.parse(event.body);

  switch (json.type){
      case "CREATED":
          // Lisätään uutta dataa
          displayedData.push(JSON.parse(json.data));
          break;
      case "UPDATED":
          // Päivitetään olemassa olevaa dataa
          var index = displayedData.
                      findIndex(p => p.id === json.id);
          displayedData[index] = JSON.parse(json.data);
          break;
      case "DELETED":
          // Poistetaan olemassa olevaa dataa
          var index = displayedData.
                      findIndex(p => p.id === json.id);
          displayedData.splice(index, 1);
          break;
  }
});
```

KUVA 9. STOMP-viestin JSON-datan käsittely frontendissä

Näin vältetään ensimmäisen kerran pakollista tietokantakyselyä lukuunottamatta käytännössä kaikki loput tietokantakyselyt.

4.3 Dokumenttikannan käyttäminen relaatiokannan rinnalla

Vaikka verkkoapplikaatio päätoimisesti käyttäisikin relaatiokantaa tärkeimpien tai suurimpien datamäärien säilyttämiseen, ei se tarkoita sitä, että kaiken datan pitäisi aina löytyä kyseisestä relaatiokannasta. Erityisesti reaaliajassa käyttäjille päivittyväksi tarkoitettu data voi joko kokoaikaisesti tai useampien käyttäjien samanaikaisten sessioiden ajan olla löydettävissä erillisestä dokumenttikannasta. Yksi tähän soveltuva dokumenttikanta on Apachen CouchDB.

Springiltä ei löydy suoraa valmiiksi määriteltyä spring-data -tukea CouchDB:lle, mutta java-alustoille löytyy lukuisia erinäisiä kirjastoja tämän ongelman ratkaisemaan, joista yksi suosituimmista on Ektorp. Ektorpin käyttöönotto projektissa ei vaadi muuta kuin itse kirjaston lisäämisen projektin riippuvaisuuksiin ja kevyen konfiguraatio-luokan luomisen, jossa CouchDB-yhteys luodaan ja alustetaan javabeaniksi, jotta sitä voidaan käyttää muissakin applikaation moduuleissa (kuva 10).

```
@Bean
public StdCouchDbConnector connector() throws Exception {
    String url = "http://localhost:8082/";
    String databaseName = "personnel";

    Properties properties = new Properties();
    properties.setProperty("autoUpdateViewOnChange", "true");

    HttpClientFactoryBean factory = new HttpClientFactoryBean();
    factory.setUrl(url);
    factory.setProperties(properties);
    factory.afterPropertiesSet();
    HttpClient client = factory.getObject();

    CouchDbInstance dbInstance = new StdCouchDbInstance(client);
    return new StdCouchDbConnector(databaseName, dbInstance);
}
```

KUVA 10. CouchDB-yhteyden konfigurointi

Kaikki CouchDB-operaatiot tapahtuvat näin connector-nimisen java beanin kautta. Tämän beanin kautta päästään käsiksi tyypillisten CRUD-operaatioiden lisäksi myös kaikkiin dokumenttikannoille erityisiin toiminnallisuuksiin, joista tämän applikaation käyttötarkoituksiin tärkeimpänä on ChangesFeed.

ChangesFeed-olion avulla voidaan ohjelman sisällä seurata yhdistetyn CouchDB-tietokannan _changes-syötettä, jonka loppuun kanta automaattisesti lisää aina tietoja kannan datassa tapahtuneista muutoksista. Muutoksien ilmoittamiseen voidaan luoda esimerkiksi oma asynkroninen ja omassa säikeessään pyörivä metodi, joka aiemman lailla lähettää STOMP-viesteinä datan päivityksille tarkoitetun kanavan kuuntelijoille uutta dataa JSON-muodossa (kuva 11).

```
@Async
private void continuousChangesChecker(ChangesFeed feed){
    while (feed.isAlive()){
        try {
            DocumentChange change = feed.next();
            PersonCouchEntity person = personCouchRepo.findOne(change.getId());

            template.convertAndSend( destination: "/topic/personListener", person);
        } catch (InterruptedException ex){
            // Yhteys katkesi, käsittele virhe
        }
    }
}
```

KUVA 11. Dokumenttikannan datan muutoksista ilmoittava metodi

WebSocket-rajapintojen kautta aktiivisia käyttäjäsessioita on helppo seurata WebSocket-protokollan jatkuvan luonteen vuoksi, joten tyypillisen muutaman käyttäjän samanaikaisen session kulku voisi olla esimerkiksi seuraavanlainen:

1. Ensimmäinen käyttäjä ottaa applikaatioon yhteyden. Reaaliajassa päivitettävä data tai mahdollisesti jopa kaikki tarpeellinen data kopioidaan sellaisenaan pysyvästä relaatiokannasta yhteisen session ajaksi käytettävään väliaikaiseen dokumenttikantaan.
2. Uusi käyttäjä ottaa applikaatioon yhteyden. Serveri huomaa, että käytössä on väliaikainen dokumenttikanta, joten applikaatio palauttaa uudelle käyttäjälle kaiken datan väliaikaiskannasta relaatiokannan sijaan.
3. Käyttäjät tekevät dataan muutoksia, lisäyksiä tai poistoja, jotka talletetaan väliaikaiskantaan pysyvän relaatiokannan sijasta. Muut käyttäjät saavat päivitykset automaattisesti STOMP-viesteinä väliaikaiskannan muutossyötteen kautta.
4. Kun viimeinen käyttäjä sulkee applikaation tai tietty määritelty aikaraja tulee täyteen, väliaikaiskannan data muunnetaan kokonaisuudessaan relaatiokannalle sopivaksi ja uudet sekä muuttuneet tietueet päivitetään pysyvään relaatiokantaan. Väliaikaiskanta pyyhitään tyhjäksi.

Tälläisessä skenaariossa relaatiokantakyselyt pysyvät äärimmäisen vähäisinä, ideaalitilanteessa niitä olisikin vain kaksi – ensimmäinen kaiken datan hakeva kysely ja toinen kaiken muuttuneen datan päivittävä kysely.

5 POHDINTA

Opinnäytetyön tavoitteena oli ottaa selvää onnistuuko tyypillisesti hitaiden relaatiokantojen käyttö modernien single page -applikaatioiden kanssa, joilta useasti vaaditaan datan muutosten reaaliaikaista raportointia käyttäjille, ja tulokset vaikuttavat varsin positiivisilta. Ongelman ratkaisuun tai ainakin tutkimiseen on varmasti lukemattomia eri tapoja, joista tässä työssä esiteltiin kolme – datan jatkuva uudelleenhakeminen, datamuutosten lähettäminen WebsSocket-protokollan yli sekä dokumenttikannan käyttö relaatiokannan rinnalla.

Ensimmäinen lähestymistapa, datan uudelleenhakeminen, on tavoista selvästi yksinkertaisin ja helpoin toteuttaa, mutta myös hyvin vaikea suositella käytettäväksi oikeastaan missään tilanteessa. Jatkuva kysely tuottaa turhaa kuormaa niin frontendille kuin palvelinkoneellekin, erityinen paino ollessa jälkimmäisellä – siinä missä tyypillinen web-applikaation käyttäjäsessio saattaisi normaalisti synnyttää joitain kymmeniä tietokantakyselyitä, jatkuva päivitysten kysely nostaa kyselytiheydestä riippuen luvun helposti jopa tuhansiin, joka kuormittaa palvelinkoneita todella suuressa mittakaavassa.

Mahdollista variaatiota voitaisiin esitellä kyselylähestymistapaan esimerkiksi muuttamalla kyseleminen nk. long pollaamiseksi. Long polling tarkoittaa käytännössä sitä, että palvelin jättää tyystin vastaamatta käyttäjien lähettämiin kyselyihin aina siihen asti, että uutta dataa todellisuudessa ilmestyy kantaan. Tietokantakyselyjen määrä putoaa murto-osaan aiemmasta, mutta skaalautuvuus kärsii niin suunnattomissa määrin, ettei long polling todellisuudessa ole realistinen vaihtoehto tälläisessä tilanteessa. Palvelimen pitäisi seurata tietokannan tilaa ja sitä, onko siellä oikeasti tapahtunut muutoksia sekä pahimmillaan jokaisen yksittäisen käyttäjän senhetkistä tilannetta – mitä dataa on jo lähetetty, mitä ei kuulu lähettää ja niin edelleen. Palvelinkoodia täytyy joka tapauksessa muokata ja palvelimen toimintaa konfiguroida tukemaan long polling -tapaa, joten ehkä järkevämpää olisi samantien siirtyä käyttämään uudempia tekniikoita, erityisesti WebSocketia.

WebSocket-protokolla on uudenaikaisempi tapa ratkoa datan lähettämiseen liittyviä ongelmatilanteita, ja se on kriittinen tekijä kahdessa jälkimmäisessä lähestymistavassa,

WebSocketien kautta päivitysten tilaamisessa sekä relaatiokannan rinnalla toimivan dokumenttikannan käyttämisesä.

Nämä kaksi lähestymistapaa ovat frontend-koodin suhteen käytännössä tismalleen samanlaisia, vaatien selainkoodilta lähinnä tuen kanavien tilaamiselle ja STOMP-viestien käsittelylle. Palvelimen osalta molemmat lähestymistavat ovat kuitenkin varsin erilaiset, ja molemmilla on selvät heikkoutensa ja vahvuutensa.

Luvussa 4.2 esitelty yksinkertainen päivityskanavien tilaaminen on WebSocket-tyyleistä selvästi helpompi toteuttaa ainakin pienen mittakaavan applikaatioissa. Kunhan tietokannan taulut ja datan java-koodissa käsittelyyn tarkoitetut entiteettiluokat ovat tarkasti määriteltyjä, voidaan myös yksittäisten tietueiden eheydestä ja oikeellisuudesta olla varmoja, sillä data talletetaan kantaan ennen sen lähettämistä dataan liitettyä kanavaa kuunteleville käyttäjille. Relaatiokannan kyselyt pysyvät käytännössä tismalleen samoissa lukemissa kuin normaalissa relaatiokannan ympärille rakennetussa applikaatiossakin – palvelin vain suorittaa yhden ylimääräisen kantoihin liittymättömän toimenpiteen, eli STOMP-viestin rakentamisen ja lähettämisen, dataa muutettaessa.

Tästä syystä skaalautuvuus on kuitenkin myös suurin huolenaihe tälläisen lähestymistavan kanssa. Koska kyseessä on täysin applikaatiolle, käyttötarkoituksille ja käyttötilanteilla mukautettu tapa ratkaista ongelma, täytyy applikaation kehittäjän tuottaa monia erilaisia implementaatioita STOMP-viestien lähettämisestä eri CRUD-operaatioille ja eri applikaation käyttämille datakokonaisuuksille. Tästä syntyy paljon manuaalista työtä ja kuten aina mukautetun koodin kanssa, myös suurempi tarve applikaation toiminnan tarkalle dokumentaatiolle, erityisesti mahdollisia myöhemmin projektiin tulevia kehittäjiä silmälläpitäen.

Kokonaisuutena datan eheydestä tai tarkasta järjestyksestä ei voida myöskään olla täysin varmoja tilauslähestymistavan kanssa. Jos applikaation dataa muutellaan hyvin tiheään tahtiin, saattavat eri käyttäjät saada yksittäisiä muutospäivityksiä eri järjestyksessä, joten jos datan tietty esitysjärjestys applikaatiossa on elintärkeää, täytyy datassa itsessään olla tallessa tieto esimerkiksi listasijainnista.

Viimeisen lähestymistavan kanssa ei tälläistä ongelmaa synny. Kun dataa lähetetään jatkuvasti suoraan relaatiokannan rinnalla toimivan dokumenttikannan muutossyötteestä,

voidaan frontend-päässä olla täysin varmoja, että kaikki käyttäjät näkevät datan tismalleen samassa muodossa ja järjestyksessä.

Myös skaalautuvuus, niin koodin kuin palvelinten suorituskyvynkin suhteen, on huomattavasti pienempi ongelma käytettäessä dokumenttikantaa sessioiden aikaisena tilapäiskantana. Koska muutossyöte, eli muutoksista ilmoitusten työntäminen löytyy sisäänrakennettuna taulukohtaisesti CouchDB-dokumenttikannasta, palvelimen koodin suhteen ei kehittäjän tarvitse nähdä paljoa vaivaa – riittää, että kantaan liitetään kuuntelija, joka ilmoittaa eteenpäin aina kun kannan muutossyötteeseen lisätään uusi rivi. Dokumenttikantojen ”append only” -luonteesta johtuen myös poistot ja vanhojen tietuiden muokkaukset nähdään lisäyksinä muutossyötteessä. Kannan ja kuuntelijan implementoinnin jälkeen kanta osaa automaattisesti ilmoittaa kaikista tiettyyn tauluun tapahtuneista muutoksista itsekseen eteenpäin.

Dokumenttikannan käyttö väliaikaiskantana tuo mukanaan myös selviä lisäongelmia ratkaistavaksi. Ensinnäkin toisen kannan käyttöönotto vaatii työtä – luvun 4.2 lähestymistavan WebSocket-konfiguroinnin lisäksi kehittäjän täytyy valita käytettävä kanta joko ominaisuuksien tai käytetyn alustan kanssa yhteensopivuuden mukaan, tutustua kannan käyttöön jos kehittäjä on lähinnä relaatiokantojen kanssa toiminut menneisyydessä ja konfiguroida kanta oikeasti myös tekemään jotain. Riippuen siitä, kuinka tarkasti määriteltyä datan muoto on, joutuu kehittäjä implementoimaan manuaalisia validointimetodeja kaikelle talletettavalle datalle dokumenttikantojen muotokaavojen puutteen vuoksi. Joissain tilanteissa pysyväiskäytössä olevan relaatiokannan entiteettiluokat ja dokumenttikannan entiteettiluokat saattavat myös lievästi erota toisistaan, jolloin näiden luokkien välille täytyy kirjoittaa mappaus-metodeja. Siinä vaiheessa kun suuria osia relaatiokannan datasta aletaan siirtämään väliaikaiseen dokumenttikantaan herää myös kysymys, että voisiko sitä siirtyä pysyvästi dokumenttikannan käyttöön kyseisen applikaation kanssa.

Vaikka jatkuva kyseleminen onkin ehdottomasti suosituslistan alapäässä, lukujen 4.2 ja 4.3 lähestymistavoille löytyisi varmasti omat paikkansa monien web-applikaatioiden toimintalogiikasta. Toiminnallisesti ja tämän opinnäytetyön suhteen luvun 4.2 lähestymistapa, kanavien tilaus, putoaa selvästi lähemmäksi tavoitteita ja päämäärää vaikka se tuokin mukanaan jonkin verran lisää manuaalista työtä ja dokumentaation tarvetta. Kanta ei joudu kestämään yhtään normaalia, ei-reaaliaikaista applikaatiota

enempää rasitusta, ja tekstimäärällisesti kattavienkin JSON-objektien todellinen bittikoko on useimmiten niin häviävän pieni, että datan jatkuva lähettely STOMP-viesteillä on käytännössä ilmaista. Mikään ei sinänsä estä kehittäjiä liittämästä STOMP-viesteillä datan kuljettamista käytännössä kaikkeen applikaation dataan, mikä voidaan JSON-muotoiseksi serialisoida.

Luvun 4.3 dokumenttikantalähestyminen sopii paremmin tukemaan jo olemassa olevaa täysin normaalisti relaatiokannan päällä toimivaa applikaatiota. Siinä missä relaatiokannassa säilytetään kaikki rakenteellisesti tarkoin määritelty data, voidaan dokumenttikantaan tallentaa kaikki, mitä halutaan reaaliajassa käyttäjille lähettää ja minkä muodolla joko ei niinkään ole väliä, tai mille on voitu kirjoittaa tarkat validointimetodit. Tälläistä dataa on esimerkiksi yksityis- ja keskusteluhuoneviestit.

LÄHTEET

MySQL Reference. Luettu 30.1. https://dev.mysql.com/doc/refman/5.7/en/

CouchDB Documentation. Luettu 31.1. http://docs.couchdb.org/en/2.0.0/index.html

CouchDB Archive. https://archive.apache.org/dist/couchdb/

Redmond, E., Wilson, J. R. 2012. Seven Databases in Seven Weeks. Yhdysvallat: Pragmatic Programmers, LLC

IBM100 Icons of Progress, Relational Database. Luettu 29.1. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/reldb/