WEB-PALVELUIDEN SKAALAAMINEN

Jyri Mikkola

Opinnäytetyö
Marraskuu 2017
Tietojenkäsittely
Web-palvelut

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Web-palvelut

Jyri Mikkola:
Web-palveluiden skaalaaminen

Opinnäytetyö 65 sivua, joista liitteitä 0 sivua
Marraskuu 2017

Työn tavoitteena oli tarkastella mahdollisimman laajasti erilaisia ohjelmiston skaalaamiseen käytettäviä metodeja ja tutkia niiden soveltuvuutta pienten ja keskisuurten web-palveluiden toteuttamiseen. Varsinkin monimutkaisten ratkaisuiden kohdalla täytyy verrata mahdollisia hyötyjä ja haittoja, sekä arvioida, milloin kyseiset hyödyt ovat ohjelmiston kannalta haittoja tärkeämpiä. Työn tarkoituksena oli soveltaa useita erilaisia skaalautuvuutta parantavia ratkaisuja Coon-Tech Oy:n uusien markkinointijärjestelmien kehittämisessä. Web-ympäristö ja miljoonien asiakastietojen käsitteleminen muodostavat ohjelmistokehitykseen esteitä, joiden ratkaisemiseen käytettyjen metodien tarkastelu antaa mahdollisuuden tutkia aihealueen teoriaa käytännönläheisestä perspektiivistä.

Tämä opinnäytetyö on syntynyt käytännön tarpeen seurauksena, sillä kehitettävät järjestelmät eivät yksinkertaisesti toimineet ilman skaalautuvuutta varten suunniteltua ohjelmistoa web-ympäristölle ominaisten haasteiden takia. Työssä keskityttiin esittelemään kehitysympäristössä sovellettavien teknologioiden teoriaa, mahdollisuuksia ja toteutukseen liittyviä haasteita. Varsinkin teknologioita tarkkailtaessa pyritään erottamaan varsinainen toteutus teoriasta, jolloin esimerkiksi koodin rakenteessa kuvatut ratkaisut voidaan tulkita käytetystä ohjelmointikielestä riippumattomasti.

Uudet verkkoselainten tukemat ominaisuudet ovat mahdollistaneet entistä monipuolisempien verkossa toimivien sovellusten kehittämiseen. Osa perinteisesti käyttäjän tietokoneella toimivasta ohjelmistosta on siirtymässä Internetiin SaaS-palveluina, joten web-ympäristön kehittämishaasteiden ratkaiseminen tulee jatkuvasti ajankohtaisemmaksi. Opinnäytetyöprojektin yhteydessä toteutettiin skaalautuva markkinointijärjestelmä ja kartoitettiin selkeämpi kuva erilaisten skaalaamiseen käytettävien teknologioiden soveltuvuudesta yrityksen tulevaisuuden tarpeille. Työssä käsitelty skaalaamiseen liittyvä teoria on suureksi osaksi hyödynnettävissä myös web-palveluiden ulkopuolella, esimerkiksi tietokantajärjestelmiä tai yrityksen sisäisiä palvelinratkaisuja kehittäessä.

Asiasanat: web-palvelut, skaalautuvuus, jonot, tietokannat

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Web Services

MIKKOLA JYRI:
Scaling of Web Services

Bachelor's thesis 65 pages, appendices 0 pages
November 2017

The objective of this thesis was to explore methods used for scaling of software and the viability of these methods for implementation of small and middle-sized web services. When it comes to implementing advanced solutions, it must be carefully evaluated whether the pros outweigh the cons involved for the software in question. The purpose of this thesis was to implement several different solutions to improve the scalability of the new marketing systems developed for Coon-Tech Ltd. The web environment and handling of millions of client entries form certain obstacles for software development. The solving of these challenges allows one to observe the field and theory of scalability from a very practical viewpoint.

This thesis was written as the direct result of a practical need. Because of the unique challenges encountered in the web environment, the software systems being developed would simply not work without architecture planned for scalability. This thesis focuses on presenting the theory, possibilities and challenges involved in developing software in this chosen environment. When it comes to the technologies involved, this thesis strives to use abstractions to present the theory in a way that is not locked to a single strategy used for implementation. For example, the pictures representing the structure of the implementation of code can be interpreted as a sort of pseudo code that is usable regardless of the actual programming language involved.

Key words: web-services, scalability, queues, databases

SISÄLLYS

1 JOHDANTO ... 7
2 SKAALAAMINEN ... 9
3 WEB-YMPÄRISTÖN HAASTEET ... 11
3.1 HTTP ja REST ... 11
3.2 Ohjelmiston asetukset ... 13
3.3 Ajan ja tilan vaihtosuhdanne ... 14
4 TILAN TALLENTAMINEN ... 16
4.1 Asiakaspuolelle ... 17
4.1.1 Selaimen avulla ... 17
4.1.2 Suoralla kommunikaatiolla ... 18
4.2 Palvelinpuolelle tietokantojen avulla ... 18
4.2.1 Relaatiotietokannat ... 19
4.2.2 Oliotietokannat ... 21
4.3 Muut menetelmät ... 24
5 TIETOKANNAN TOIMINNAN VERTIKAALINEN SKAALAAMINEN ... 25
5.1 Object Relational Mapping ... 25
5.2 Rivien tallentaminen ... 26
5.3 Heuristiset menetelmät ... 29
5.3.1 Duplikaattien käsittely ... 30
5.3.2 Tiedoston sisällön vertaaminen tietokantaan ... 31
6 VERTIKAALINEN SKAALAAMINEN JONOILLA ... 34
6.1 Toimintaperiaate ... 34
6.2 Hyödyt ... 35
6.3 Haasteet ... 36
7 PALVELINTEN HORISONTAALINEN SKAALAAMINEN ... 39
7.1 Palvelinten välinen kommunikaatio ... 40
7.2 Kommunikaatiometodit ... 40
8 HAJAUTETUT VERKKOARKKITEHTUURIT ... 42
8.1 Master / slave ... 42
8.2 Vertaisverkot ja hybridit ... 43
8.3 Palvelukeskeinen arkkitehtuuri ... 44
8.4 Micropalvelut ... 45
9 TODENNUS JA TIETOTURVA LYHYESTI ... 46
9.1 Todennuspalvelinten käyttö ... 46
9.2 Keskitettyjen todennuspalvelimien riskit ... 47
9.3 Salasanojen turvallisuus verkkopalveluissa ... 47

10 TAAKAN JAKAMINEN HAJAUTETUISSA VERKOISSA 49
10.1 Prosessien delegoiminen 49
10.2 Delegoiminen jonoilla 49
10.3 Tietokantojen klusterointi 51
10.4 Tietokantojen sharding 51
11 HAJAUTETUN VERKON YLLÄPITO 54
11.1 Automaation tarve 54
11.2 Ohjelmiston asentaminen ja päivittäminen 55
11.2.1 Staging palvelimet 55
11.2.2 Verkon toiminnan varmistaminen päivityksen aikana 56
11.3 Hallintapalvelinten tehtävät 56
11.4 Palvelinten peilaaminen 57
11.5 Verkon toiminta pitkällä aikavälillä 57
12 JOHTOPÄÄTÖKSET JA POHDINTA 59
LÄHTEET 62

LYHENTEET JA TERMIT

Arkkitehtuuri	Ohjelmiston toteuttamiseen käytetty rakenne.
API	Application programming interface eli ohjelmointirajapinta. Ohjelman tarjoama määritelmä tiedon vaihtamiseen ihmisten tai toisten ohjelmien kanssa.
REST	(Representational State Transfer) Arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen. REST pyrkii hyödyntämään tilattomuutta muun muassa suorituskyvyn, skaalautuvuuden ja yksinkertaisuuden saavuttamiseksi.
SaaS	(Software as a service) Palvelumalli, jossa käytettävä ohjelmisto vuokrataan käytettäväksi tavanomaisen, lisenssipohjaisen paikallisen asennuksen sijaan.
Serialisaatio	Tiedon muuntaminen säilytettävään muotoon
Skaalautuvuus	Ominaisuus, joka mahdollistaa järjestelmän ympäristön laajentamisen aiheuttamatta häiriötä toimintaan. Syynä laajenemisen tarpeelle voi olla esimerkiksi järjestelmän käyttäjäkunnan kasvu tai tarve uusille toiminnoille.
SOA	(Service oriented architechture) Palvelukeskeinen arkkitehtuuri on tietojärjestelmien arkkitehtuurimalli, jossa eri toiminnot on suunniteltu itsenäisiksi palveluiksi.
Suunnittelumalli	Tietyn ongelman ratkaisemiseen suunniteltu, standardoitu koodirakenne.
Tilattomuus	Kommunikaation muoto, jossa kumpikaan osapuoli ei säilytä aktiiviseen kommunikaatioyhteyteen liittyvää tietoa.

1 JOHDANTO

Opinnäytetyö sai alkunsa Coon-Tech Oy:n uusien markkinointi- ja tietokantajärjestelmien kehityksestä. Monien nykyisten järjestelmien toimivuus rajoittuu suhteessa pieneen määrään tietoa. Esimerkiksi jotkin SaaS-palvelut rajoittavat yksittäisten asiakaslistojen koon 20 megatavuun, kun suurimmat yrityksen käsittelemät tiedostot voivat olla yli 100:n megatavun kokoisia. Jotta tiedostot saataisiin yhteensopivaksi näiden järjestelmien kanssa, jouduttaisiin tiedostot pilkkomaan pienempiin osiin, joka osoittautui tiedostojen määrän takia työlääksi. Edes omille palvelimille lisensoitavat markkinointijärjestelmät eivät ratkaisseet kaikkia ongelmia, sillä vaikka joihinkin järjestelmiin oli mahdollista tallettaa tiedostoja ilman tiedostokoon ylärajaa, tämä prosessi saattoi kestää jopa kymmeniä tunteja. Lisäksi järjestelmien tietokantarakenteet olivat käyttötarkoitukseen nähden huonosti optimisoituja, mikä johti alkuperäisten tiedostojen kokoon verrattuna moninkertaiseen levytilan käyttöön. Ensimmäinen yritykselle kehitetty tietokantajärjestelmän prototyyppi paljasti, että levytilaa voitaisiin säästää lähes 3 – 8 kertaisesti lisensoituun ratkaisuun verrattuna.

Lisensoitujen palveluiden lähdekoodin avoimuus mahdollisti ohjelmiston toiminnan tarkkailun, jonka perusteella pystyttiin selvittämää syitä järjestelmien hitauteen. Suurimmat toimintaan liittyvät ongelmat nousivat tietokantojen rivikohtaisista operaatioista, joita käsitellään tarkemmin luvussa 5. Rivikohtainen tiedonkäsittely johti niin suureen hidastukseen, että useita olemassa olevia ominaisuuksia ei voitu enää hyödyntää järjestelmien sisältämän tietomäärän kasvaessa. Nämä käytännön tarpeet synnyttivät idean yrityksen oman tietokantaratkaisun kehittämisestä.

Järjestelmän kehityksessä otettiin oppia prototyypin testaamisen yhteydessä kohdatuista ongelmista ja haasteista. Tämän tiedon avulla kehitettiin uusi järjestelmä, joka on alusta asti suunniteltu ratkaisemaan varsinkin tiedostokokojen skaalautumiseen liittyvät ongelmat. Tässä opinnäytetyössä esitellään käytännön ratkaisuiden pohjalta tehtyjä johtopäätöksiä, sekä yleistä skaalaamiseen liittyvää teoriaa. Työssä pyritään antamaan lukijalle laaja kuva erilaisista skaalautuvuuden toteuttamiseen liittyvistä mahdollisuuksista.

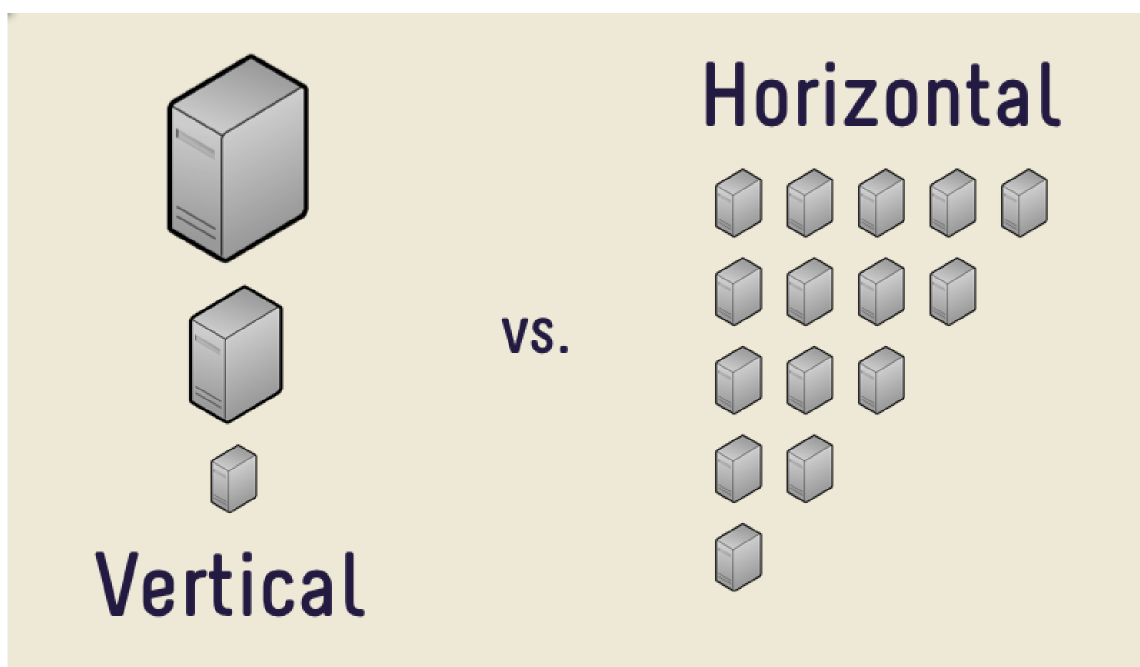
Kaikkia skaalaamiseen tarkoitettuja menetelmiä ei ollut kannattava soveltaa tietokantajärjestelmän kehittämiseen. Yksi skaalaamisen haasteista onkin selvittää, milloin tietyt

ratkaisut ovat toisia kannattavampia. Tästä syystä työssä pyritään selventämään tasapuolisesti esitettyihin menetelmiin liittyviä mahdollisuuksia ja haasteita, jotta lukija voi tarvittaessa hyödyntää käsiteltyä teoriaa itselleen oikean ratkaisun löytämiseen.

Skaalautuvien web-palveluiden tarkastelu ja toteuttaminen on nykyään entistä ajankohtaisempaa ainakin kahdesta syystä. Ensinnäkin: aikaisempi client–server -mallin arkkitehtuuri, jossa käyttäjän laitteelle asennettu ohjelma kommunikoi palvelun tarjoajan palvelinlaitteen kanssa Internetin välityksellä on väistymässä trendinä vuokrattavien, SaaS-palveluiden edeltä. Toisekseen: teknologioiden kehitys on aiheuttanut myös kerätyn ja saatavilla olevan tiedon määrän kasvuun. Tämä kasvu on puolestaan johtanut isoa tietomäärää kuvaavan big data -termin käytön yleistymiseen. Käytännössä tiedon määrän lisääntyminen tarkoittaa myös entistä suurempaa tarvetta ohjelmistolle, joka pystyy käsittelemään yli miljoonia tietoartikkeleita nopeasti.

2 SKAALAAMINEN

Ohjelmiston skaalaamisella tarkoitetaan ohjelmiston kehitystä suuntaan, joka mahdollistaa toimintojen ja/tai käyttäjäkunnan kasvun aiheuttamatta häiriötä kehitettävän sovelluksen toimintaan (Liu 2009: 1-2). Kuten useat muutkin tässä työssä esiteltävät tärkeät konseptit, skaalaaminen jakautuu kahteen yläkategoriaan. Skaalaaminen voidaan suorittaa pääasiallisesti kahdella eri tavalla: vertikaalisesti tai horisontaalisesti (kuva 1). Vertikaalisella skaalaamisella tarkoitetaan kaikkia parannuksia, joita voidaan tehdä yhden fyysisen sovellusympäristön sisällä. Vertikaalinen skaalaaminen pitää sisällään fyysisen ympäristön tehokkuuden parantelemisen asentamalla uusia ja/tai tehokkaampia osia käytettyyn laitteeseen. (Beaumont 2014)



KUVA 1. Palvelinten vertikaalinen ja horisontaalinen skaalaaminen (pc-freak.net 2014)

Ohjelmointikontekstissa lähes kaikki koodia optimoivia toimenpiteitä voidaan pitää vertikaalisen skaalauksen muotoina. Skaalauksen kannalta on olennaista, että olemassa olevat resurssit kyetään hyödyntämään tehokkaasti. Pelkällä resurssien lisäämisellä ei päästä pitkälle, mikäli nykyisistä resursseista ei voida hyödyntää tehokkaasti kuin murto-osa. Tämän työn ensimmäisellä puoliskolla keskitytään esittelemään vertikaalista skaalaamista.

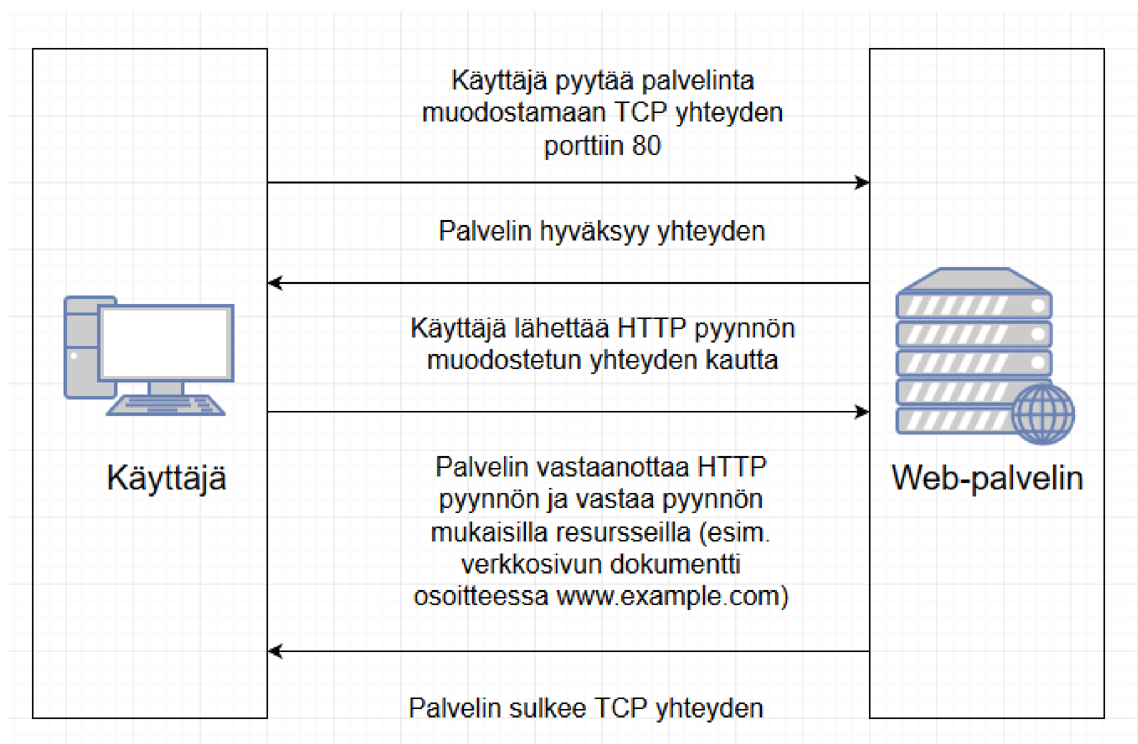
Horisontaalisella skaalaamisella tarkoitetaan prosesseja suoritettavien fyysisten laitteiden lisäämistä arkkitehtuuriin (Beaumont 2014). Horisontaalinen skaalaaminen mahdollistaa

teoriassa lähes loputtoman skaalautuvuuden sovellukselle, sillä vaikka yksittäisen laitteen tehot ovat rajoitetut teknologian puitteissa, varsinaisten laitteiden lukumäärällä ei ole muita rajoituksia kuin toteuttavan organisaation resurssit. Useiden tehottomampien laitteiden yhdistäminen prosessointitehojen kasvattamiseksi voi olla jopa halvempaa, kuin yhden tehokkaamman laitteen hyödyntäminen. Tällainen skaalaaminen ei ole kuitenkaan täysin ongelmatonta, sillä laitteita ei voi yhdistää saumattomasti yhdeksi kokonaisuudeksi ilman resurssihukkaa. Horisontaalisen arkkitehtuurin hyödyntäminen vaatii vertikaalista toteutusta monimutkaisempia ratkaisuja, jotta useat laitteet voivat keskustella ja täten toimia tehokkaasti yhdessä. Laitteiden välisen kommunikaatio lisäksi ohjelman suorittaminen pitää kyetä jakamaan usealle laitteelle. Tehtävästä riippuen tällaisen jaon tekeminen ei ole välttämättä mahdollista ja vaatii usein tavanomaisesta ohjelmointitavasta poikkeavia ratkaisuja. Työn loppupuolella keskitytään horisontaalisen skaalaamisen ongelmien havainnollistamiseen ja ratkaisuun.

3 WEB-YMPÄRISTÖN HAASTEET

3.1 HTTP ja REST

Internetissä kommunikaatio web-palvelinten kanssa tapahtuu usein HTTP eli hypertext transfer -protokollan välityksellä. HTTP toimii TCP:n (usein Internetin asiayhteydessä TCP/IP) eli transmission control protocol käyttämänä kommunikaatiokielenä (Goralski 2009: 570). Varsinaista HTTP-kommunikaatioyhteyttä varten muodostetaan ensin TCP protokollan yhteys, jonka jälkeen varsinainen HTTP-kommunikaatio tapahtuu TCP yhteyden sisällä (kuvio 1).



KUVIO 1. HTTP kommunikaatioyhteyden muodostaminen

REST eli representational state transfer on Roy Fieldingin keksimä nimitys aikaisemmin HTTP-oliomalliksi (HTTP object model) kutsutulle konseptille (Fielding 2000). Alkuperäinen oliomalli kuvaa erilaisia web-palvelimien hallinnoimiin resursseihin kohdistuvia operaatioita Internetissä. Oliomalli sisältää itsessään palvelimelle annetun URI:n eli unifrom resource identifier:in ja operaatiota kuvaavan verbin, kuten GET (hae), DELETE (poista) ja niin edelleen. Näitä tietoja käyttämällä palvelin pystyy päättelemään, minkä toiminnon käyttäjä haluaa suorittaa. REST erottuu varsinaisesta oliomallista kuvaamalla

HTTP-oliomallin käyttämää arkkitehtuuria itse HTTP-protokollasta riippumattomasti. Web-ohjelmiston kehittämisen kannalta RESTin kiinnostavin ominaisuus on niin sanottu tilattomuus. Tilattomuudella tarkoitetaan, ettei palvelin säilytä mitään aikaisempia istuntoja tai niihin liittyviä tietoja (Goralski 2009: 570). HTTP-kommunikaation rakenteesta on merkittävä havaita, että sen pohjalla toimiva TCP ei ole tilaton protokolla, toisin kuin HTTP. Tämä mahdollistaa muun muassa HTTP 1.1 -versiossa lisätyn ominaisuuden hyödyntämisen useiden HTTP pyyntöjen lähettämiseen yhden TCP istunnon aikana (Goralski 2009: 571).

Tilattomuutta ja täten HTTP:tä voidaan kuvailla mallina, jossa A lähettää palvelupyynnön B:lle, johon B reagoi lähettämällä vastauksen takaisin A:lle, jonka jälkeen kommunikaatio on ohi. Tilaton kommunikaatio ei siis ota vastuuta kommunikaation aikaisen tiedon säilyttämisestä, vaikka tavanomaisen tietokoneella toimivan applikaation täytyy lähes aina säilyttää useita tilatietoja prosessin välimuistissa. Havainnollistavana esimerkkinä yksinkertainenkin ristinolla peli sisältää tilatiedot varatuista ruuduista, niiden merkeistä ja aktiivisen pelaajan vuorosta (kuvio 2).

<table>
<tr><td>X</td><td></td><td></td></tr>
<tr><td></td><td>X</td><td>O</td></tr>
<tr><td></td><td></td><td>O</td></tr>
</table>

KUVIO 2. Ristinolla peli

Niin useat ohjelmat tarvitsevat tilatietoja toimiakseen, että tilattomuuden asettamat rajoitteet poissulkevat usean ohjelmiston toiminnan kannalta pakollisia ominaisuuksia. Minkä takia maailman suurimman tietoverkon arkkitehtuuri pohjautuisi malliin, joka rajoittaisi web-ohjelmiston toimintaa huomattavasti? Käytännössä tilattomuus mahdollistaa web-palvelimille yksinkertaisemman virhekäsittelyn ja paremman suorituskyvyn, sillä palvelimen ei tarvitse ylläpitää aktiivisia yhteyksiä tai huolehtia yhteyden katkeamisen aiheuttamasta virhetilasta (Fielding 2000). Varsinkin Internetin alkuaikoina, nämä ominaisuudet olivat elintärkeitä kompensoimaan palvelinten rajatumpaa muistia ja suorituskykyä. Vaikka nykyiset laitteet kykenevät käsittelemään yhtäaikaisesti useampia yhteyksiä ja palvelupyyntöjä, REST ei ole silti menettänyt merkitystään makroskaalan suorituskyvyssä ja luotettavuudessa.

Suurimman rasituksen alaiset sivustot voivat joutua käsittelemään ruuhka-aikana useita miljoonia palvelupyyntöjä minuutissa, jolloin kaksisuuntaisten, aktiivisten yhteyksien ylläpitäminen kaikkien käyttäjien kanssa voi muodostua lähes mahdottomaksi. Ajankohdasta riippuen Google voi käsitellä 60 000 - 65 000 hakua sekunnissa (internetlivestats.com 2017 A). Minuutissa vastaava luku vastaa keskiarvoltaan 3,75 miljoonaa hakua. Tietysti pelkkiä hakuja laskiessaan tilasto ei ota huomioon, että tehdäkseen haun, tavallisen käyttäjän tulee ensin avata Googlen etusivu. Tällöin jokaisen haun tekeminen vaatii vähimmillään kaksi erillistä HTTP-palvelupyyntöä. Vaikka uuden teknologian myötä kehittyneet palvelimet tarvitsevat yhä vähemmän RESTin tarjoamia suorituskyvyllisiä hyötyjä, yksinkertaisuus ja luotettavuus tulevat olemaan aina ohjelmistosuunnittelussa suosittuja ominaisuuksia.

3.2 Ohjelmiston asetukset

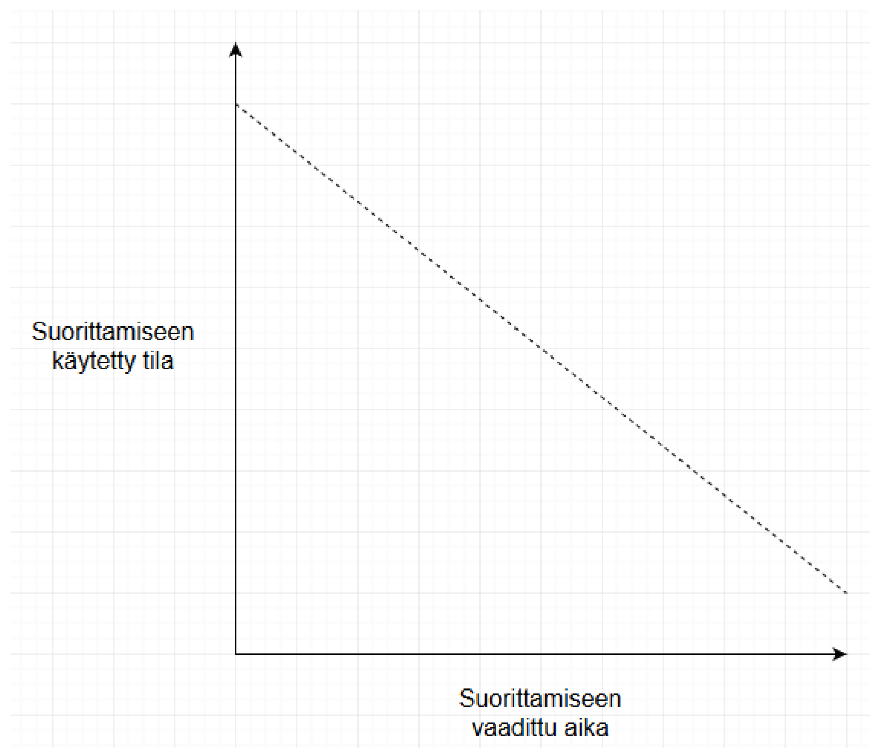
Web-palvelinten ohjelmisto asettaa rajoitteita HTTP-pyyntöjen käsittelyyn tiettyjen asetusten muodossa. Näiden asetusten tavoitteena on muun muassa estää vikatilanteisiin, tietoturvauhkiin tai liialliseen resurssienkulutukseen johtavien pyyntöjen käsittelyä (php.net 2017). Esimerkiksi tietoturvan näkökulmasta yleisimmät rajoitteet koskevat tiedostojen ominaisuuksia ja käsittelyä, kuten kokoa, tyyppiä, latausta tai lähettämistä. Web-palveluiden skaalaamisen kannalta olennaisia asetuksia ovat tiedostokokojen lisäksi pyyntöjen käsittelyn aikakatkaisun aika ja prosessille varattavan muistin yläraja. Asetuksia muuttaessa tulee aina miettiä tarkkaan, onko muutos ehdottomasti tarpeellinen sekä ymmärtää ja tasapainottaa siitä mahdollisesti koituvat haitat. Useat rajoitteet ja asetukset on suunniteltu suojaamaan kehittäjää, jonka takia on erityisen tärkeää välttää kiusausta saada ohjelmisto toimimaan ymmärtämättä muutoksista koituvia seurauksia. Ensisijaisesti tulee aina etsiä ratkaisuja, jotka eivät vaadi käyttörajojen kasvattamista, sillä muistin tai ajan loppuessa tavallisessa käytössä syynä on lähes poikkeuksetta virheellinen tai tehoton koodirakenne.

Tilanteen ja ympäristön mukaan muistin käyttörajojen kasvattaminen voi olla tarpeellista, mutta aikakatkaisurajan kasvattamista pidetään lähes poikkeuksetta huonona käytäntönä. Hidas aikakatkaisu aiheuttaa usein ylimääräistä resurssien kulutusta palvelimella, kun odottamaton virhetilanne muodostaa tukoksen, kuten muistia varaavan ikuisen kierteen.

Kehittäjän on myös turha olettaa, että käyttäjä odottaisi vastausta palvelupyyntöönsä useiden minuuttien ajan. Kun käyttäjän katkaisee yhteyden palvelimelle raskaan käsittelyn aikana, jäljelle jää usein monimutkainen tehtävä tilatietojen siivoamisesta, joka on usein ohjelmiston kehittäjän vastuulla. Vaikka osa palvelimien käyttämien ohjelmistojen kielistä mahdollistaakin komentosarjan ajamisen loppuun asti välittämättä käyttäjän yhteyden katkeamisesta, voivat tulokset olla odottamattomia ja haitallisia käyttökokemukselle. Lisäksi on huomioitava asiakaspuolen aikakatkaisu: vaikka palvelin olisi valmis käsittelemään pyyntöä useita tunteja, asiakkaan odotusaikaan vaikuttavat myös selaimen oletusarvot tai käyttäjän määrittämät asetukset.

3.3 Ajan ja tilan vaihtosuhdanne

Ajan ja tilan välinen vaihtosuhdanne (space-time tradeoff) kuvastaa ohjelmistokehityksen kahta resurssia. Minkä tahansa sovelluksen tehokas skaalaaminen vaatii kaikkien saatavilla olevien resursseiden tehokasta hyödyntämistä.



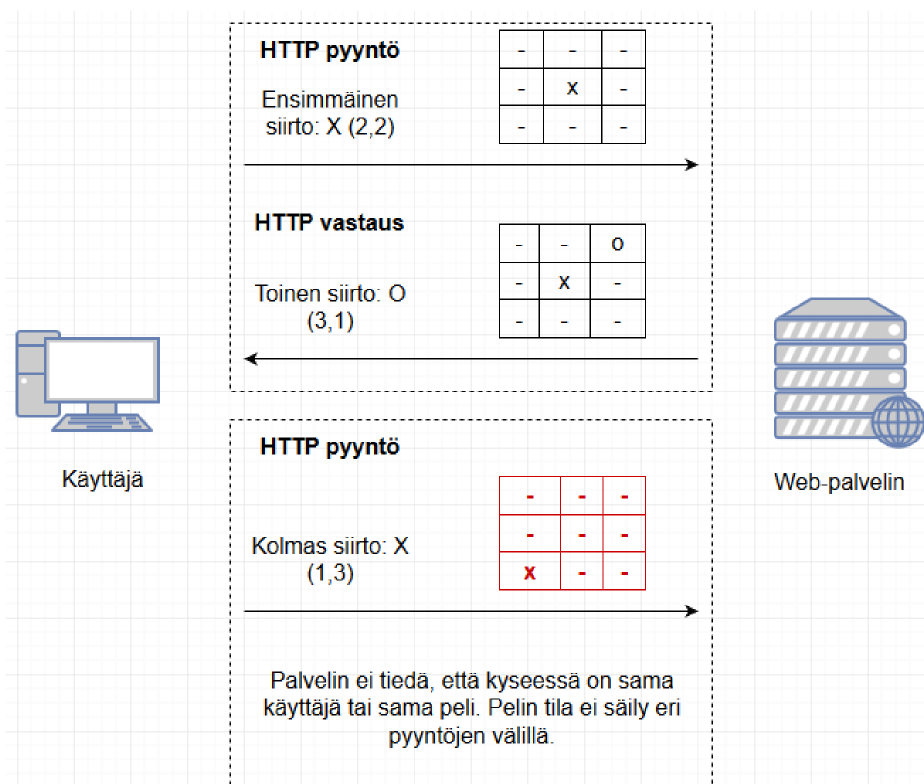
KUVIO 3. Ajan ja tilan välinen vaihtosuhdanne

Kuten kuvio 3:sta nähdään, tehtävissä, joissa vaihtosuhdanne pätee, voidaan vähentää tehtävän suorittamiseen kuluvaa aikaa lisäämällä tilan käyttöä. Vastaavasti prosessi voi myös säästää tilaa maksaessaan prosessointiajalla. (Savage 1988: 461)

Tämä malli voidaan havaita useissa tietotekniikan ohjelmistoissa ja ratkaisuissa. Erityisen yksinkertaisen ja käytännöllisen esimerkin aiheesta antaa aikaisemmin käsitelty REST: kaikkien tehtävien reaaliaikainen suorittaminen koostuu ajasta ja tilan tallentaminen esimerkiksi suoraan kovalevylle tai tietokantaan vastaa tilaa. Aika voidaankin rinnastaa saatavilla olevaan keskusmuistiin ja tila käytettävissä olevaan levytilaan. Tämä vaihtosuhdanne on helppo havainnoida myös tietokantojen toiminnassa: tieto voidaan tallentaa tavalla, joka vie vähemmän levytilaa mutta on vaivalloisempia hakea usean relaation takaa, tai vastaavasti tieto voidaan tehdä mahdollisimman helpoksi ja nopeaksi hakea tiedon toistamisen ja täten levytilan kustannuksella. Myös tiedon indeksointi tietokannoissa noudattaa tätä mallia, sillä indeksointi vaati ylimääräistä levytilaa indeksien tallennusta varten, mutta nopeuttaa indeksoitujen sarakkeiden hakuja.

4 TILAN TALLENTAMINEN

Mikäli ohjelman suorituksen aikaista tietoa halutaan käyttää myöhemmin, joudutaan se tallentamaan itse prosessista erilliseen varastoon. Jotkin yksinkertaiset sovellukset pystyvät säilyttämään kaiken tarvitsemansa tiedon ohjelman välimuistissa. Useat ohjelmat kuitenkin tallentavat käsittelemäänsä tietoa prosessin ulkoisiin pitkäkestoisiin varastoihin. Lähes kaikki asetuksia käyttävät sovellukset tallentavat käyttäjän asetusprofiilin ohjelman käyttökokemuksen parantamiseksi. Jos tieto säilytettäisiin vain sovelluksen muistissa, se katoaisi aina sovelluksen uudelleenkäynnistyksen yhteydessä. Kuten RESTiä käsittelevässä luvussa esitetty ristinollaesimerkki antaa ymmärtää, RESTin rakenne estää teoriassa kyseisen pelin toteuttamisen pelkän HTTP:n avulla.



KUVIO 4. HTTP-yhteydet palvelinta vastaan pelattavassa ristinolla -pelissä

Kuvio 4 osoittaa, että tunniste, merkityt ruudut ja aktiivisen pelaajan vuoro ovat kaikki varsinaisen kommunikaatioprotokollan ulkopuolisia tilatietoja, joita tarvitaan pelin ylläpitämiseen. Jokainen voi kuitenkin nopealla haulla todentaa, että Internetistä löytyy useita esimerkkejä toimivasta ristinollasovelluksesta. Tilan tallentaminen on usealle aloittelevalle web-kehittäjälle haaste, johon perinteinen ohjelmoija ei välttämättä törmää REST-arkkitehtuurin ulkopuolella. Käytännössä ongelma voidaan selvittää web-ympäristössä

käyttämällä yhtä tai useampaa ratkaisua kolmesta alakategoriasta: tallentamalla tila asiakaspuolelle, tallentamalla tila palvelimelle kommunikaatioprotokollan ulkopuolisella teknologialla tai hyödyntämällä toista kommunikaatioprotokollaa.

4.1 Asiakaspuolelle

4.1.1 Selaimen avulla

Asiakaspuolella (client-side) tarkoitetaan palvelua käyttävän henkilön laitteella suoritettavia prosesseja ja palvelinpuolella (server-side) palvelimella suoritettavia operaatioita (Liu 2009: 58-59). Tilaton RESTiä hyödyntävä HTTP ei kykene tavallisesti toimimaan itsenäisesti ilman käyttäjän syötettä. RESTin periaatteiden mukaisesti käyttäjä tai tavallisesti käyttäjän puolesta toimiva entiteetti (user agent), kuten selain, lähettää palvelimelle pyynnön tietyn sisällön lataamisesta. Sivun lataamiseen käytetyt palvelupyynnöt toimivat täten RESTin puitteissa täysin staattisesti: yhteys ei ole jatkuva, eikä samalla pyynnöllä ladata lisää sisältöä.

Jotta valmiiksi asiakaspuolelle ladattu sivu voisi muuttaa sisältöään dynaamisesti, tarvitaan JavaScriptiä tai muuta vastaavaa skriptikieltä. Skriptit eli komentosarjat toimivat ohjeina erilaisten toimintojen suorittamiseen suoraan asiakaspuolen ympäristössä, eli yleisesti käyttäjän selaimessa. Koska skriptikielet kykenevät muokkaamaan ladattuja HTML-dokumentteja eli sivustoja, ne voivat tallettaa ja hakea tietoa suoraan dokumentista.

Selaimet säilyttävät välimuistissaan palvelimen kanssa käytyyn keskusteluun liittyvää tietoa, kuten HTTP:n välityksellä ladattuja tiedostoja, sekä niin sanottuja evästeitä (cookies). Käytännössä evästeillä tarkoitetaan selaimen käyttölaitteelle tallentamia tietoja, joiden tallentamista palvelin on pyytänyt. Evästeiden sisältämiä tietoja käytetään kaikkein useimmin käyttäjien tunnistamiseen antamalla käyttäjälle uniikki tunniste. Täten evästeitä voidaan käyttää esimerkiksi käyttäjien tunnistamiseen, joka puolestaan mahdollistaa käyttäjätilien avulla sisäänkirjautumisen ympäristössä, jossa käyttäjistä ei voida tavallisesti tietää juuri mitään.

4.1.2 Suoralla kommunikaatiolla

HTTP:n ohella voidaan käyttää myös toista protokollaa tiedon välittämiseen asiakkaan ja palvelimen välillä. Verkkopistokkeella (network socket) voidaan muodostaa tälle välille tunnelimainen, kaksisuuntainen kommunikaatioyhteys. Vuonna 2011 standardisoidun WebSocket-protokollaa käyttävän yhteyden muodostaminen tapahtuu HTTP-pyynnöllä, jonka jälkeen TCP-yhteys jätetään auki jatkuvaa kommunikaatiota varten (Cadenhead 2015). Muodostettua yhteyttä käytetään sitten tiedottamaan käyttäjää palvelimella tapahtuneista muutoksista, joiden perusteella voidaan tehdä muutoksia käyttäjäpuolen ympäristössä.

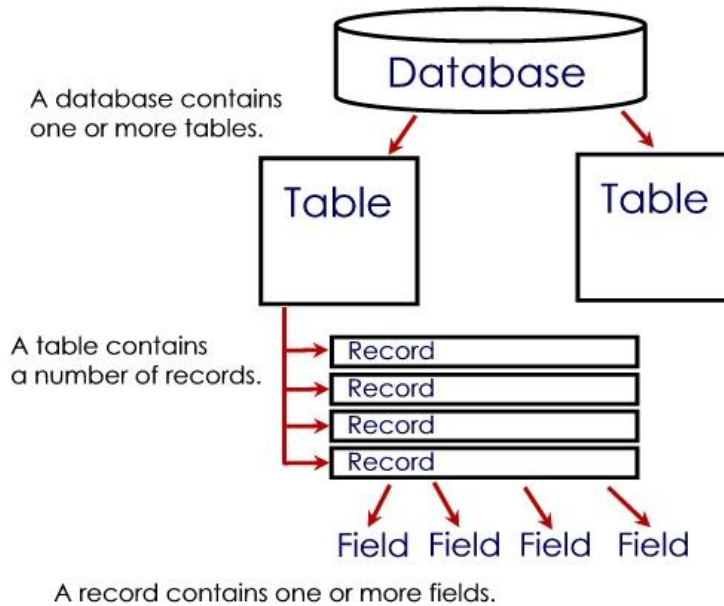
Selainympäristössä pistokkeiden hyödyntäminen on yleistymässä teknologioiden käytön helpottumisen ja selaintuen takia. Selainten uusien ominaisuuksien myötä pistokkeiden käyttöä varten on kehitetty myös kaupallisia palveluita. Pusher on esimerkki palvelusta, jossa käyttäjä voi vuokrata ulkoisen pistokeyhteyksiä hallinnoivan palvelun, jonka hintaluokka kasvaa muodostettujen yhteyksin määrän kanssa. Pistokkeisiin muodostettavia yhteyksiä varten on saatavilla myös avoimen lähdekoodin palvelinratkaisuja, kuten Socket.io 2.0 ja SocketCluster.

4.2 Palvelinpuolelle tietokantojen avulla

Ylivoimaisesti yleisin ratkaisu tilan tallentamiseen on jonkinlainen tietokanta. Tietokannat ovat tiedon säilyttämistä ja hallitsemista varten kehitetty hyvin pitkälle optimoitu teknologia. Ohjelmistopohjaiset tietokannat sisältävät menetelmät tiedon hakemiseen, tallentamiseen, poistamiseen sekä muokkaamiseen, joten niiden käyttö on usein suositeltavaa yksinkertaisten tiedostotallenteiden käytön sijaan. Mikäli ohjelma tallettaa tietoa esimerkiksi tekstitiedoston formaatissa, joutuu ohjelmoija kehittämään ohjelman sisäiset ratkaisut aikaisemmin manituille käsittelymenetelmille.

Tietokantojen tyypit voidaan jakaa karkeasti kahtia relaatio- ja oliotietokantoihin. Näistä kahdesta kategoriasta relaatiotietokannat ovat perinteisesti laajemmin käytettyjä (Elmasri & Navathe 2011: 47). Tietokannoissa tieto jäsennetään tietomalleihin, joita kutsutaan relaatiotietokannoissa tauluiksi ja oliotietokannoissa skeemoiksi. Tietomallit määrittävät

tiedon rakenteen ja niitä voidaan hyödyntää muun muassa tiedon tehokkaampaan käsittelyyn ja tiedon eheyden varmistamiseen. Yhtä tietomallin sisältämää tietojoukkoa voidaan kutsua tietokannasta riippuen muun muassa riviksi (yleisesti), tietueeksi (relaatiotietokannat), dokumentiksi (MongoDB), tai tyypiksi (Elasticsearch). Nämä rivit koostuvat yksittäisistä sarakkeista (columns) tai kentistä (fields).



KUVIO 5. Perinteisten relaatiotietokantojen hierarkia (Jollymore)

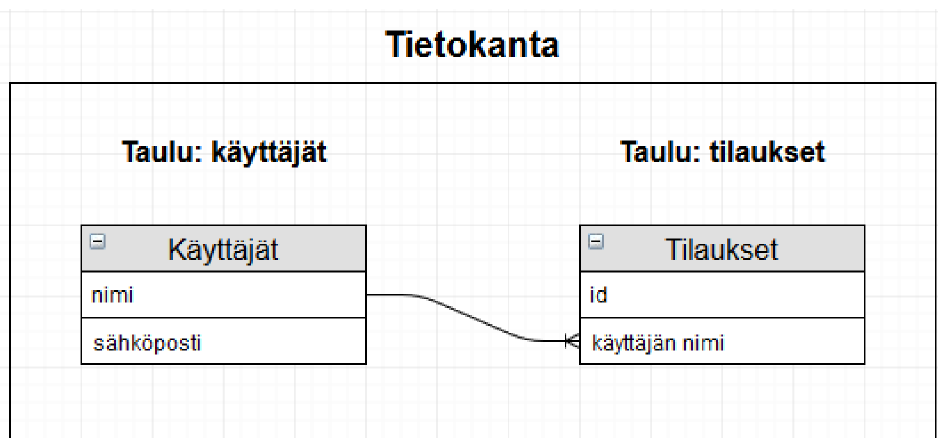
Kuvio 5 havainnollistaa näiden käsitteiden välisiä suhteita relaatiotietokannassa. Käyttäen relaatiotietokannoille olennaisia termejä esimerkkinä, käyttäjätiedot voidaan strukturoida käyttäjät -taulusta, jonka sisältämät rivit kuvaavat yksittäisiä käyttäjiä. Jokaiselle käyttäjälle ominainen tieto on tallennettu rivin sarakkeeseen. Käyttäjälle ominaisia tietoja voivat olla esimerkiksi id, sähköpostiosoite ja kryptattu salasana.

4.2.1 Relaatiotietokannat

Relaatiotietokannoista käytetään joskus epävirallisesti SQL-tietokanta nimitystä, sillä käytännössä kaikki relaatiotietokannat ymmärtävät SQL (structured query language) -kielen komentoja. Relaatiotietokannat soveltuvat hyvin yksinkertaisen tiedon, kuten finanssitietojen numeroiden käsittelyyn. Relaatiotietokannoissa tietokannan tauluille määritellään eri sarakkeiden käyttämät tietotyypit eheyden takaamiseksi (Elmasri & Navathe

2011: 63-64). Tietotyypit auttavat tietokantaa myös päättelemään, mitkä menetelmät soveltuvat parhaiten kyseisen tiedon käsittelyyn. Esimerkiksi henkilön omistamien asuntojen määrän tallettamista varten voidaan määritellä sarake, joka voi sisältää vain positiivisia kokonaislukuja, sillä henkilö ei voi omistaa negatiivista määrä tai puolikasta taloa.

Relaatiotietokantojen tarjoaman eheyden takia ne sopivat hyvin tavanomaisten sovellusten tarpeisiin, kuten käyttäjätietojen varmaan tallentamiseen. Relaatiotietokannat ovat levinneet niin laajasti, että käytännössä jokaisella vuokrattavalla web-palvelimella on valmiiksi asennettuna jokin relaatiotietokanta. Nimensä mukaisesti relaatiotietokannat perustuvat relaatiohin, eli tietueiden välisiin suhteisiin. Tietueiden välisiä suhteita kuvataan joko yhden suhteella yhteen, yhden suhteella moneen tai monen suhteella moneen. Relaatiotietokannan tauluille on olennaista määritellä sarake, jonka arvo kertoo, mille toisen taulun riville mikäkin taulun rivi kuuluu (kuvio 6).



KUVIO 6. Viittaukset relaatiotietokantojen tauluissa, yhden käyttäjän suhde moneen tilaukseen

Relaatiotietokantojen toiminnan kannalta olennainen konsepti on lyhenne ACID (Atomicity, Consistency, Isolation, Durability), eli suomeksi atomisuus, eheys, eristyneisyys, pysyvyys. ACID luettelee erinäiset vaatimukset ja rajoitteet joiden puitteissa relaatiotietokannat toimivat. Atomisuudella tarkoitetaan tietokannan toimintojen kaikki tai ei mitään mentaliteettia, joka takaa, ettei mitään tietokantaan kohdistuvaa toimintoa suoriteta osittain. Atomisuuden ja eheyden turvaamiseksi relaatiotietokannat tarjoavat usein mahdollisuuden niin sanottujen transaktioiden käyttöön. Transaktion aloittaminen tietokannassa mahdollistaa transaktion aikana annettujen komentojen suorittamisen niin, että koko prosessi voidaan keskeyttää yhden komennon suorittamisen epäonnistuessa, jonka jälkeen tietokanta voidaan palauttaa (rollback) transaktiota edeltäneeseen tilaan. Mikäli

transaktion aikaisten komentojen suorittaminen onnistuu normaalisti, voidaan muutokset hyväksyä päättämällä transaktio. (Elmasri & Navathe 2011: 732-733)

Eheydellä tarkoitetaan, että tietokannan tila siirtyy komentojen seurauksena aina toiminnallisesta tilasta uuteen toiminnalliseen tilaan. Eheyden saavuttamiseksi tietokanta valvoo, että sille määritetyt komennot eivät riko tietokannalle asetettuja rajoitteita tai asetuksia. Kyseisiin rikkeet voivat koostua väärän tietotyypin tallentamisesta, uniikiksi määritettyjen rivien kahdentamisesta, tietokannan asettamien relaatioiden rikkomisesta tai muusta vastaavasta syystä. (Elmasri & Navathe 2011: 732-733)

Eristyneisyydellä taataan, että tietokannan suorittamat komennot eivät vaikuta toisiinsa samanaikaisesti. Voidaan kuvitella tilanne, jossa käyttäjä A on tallentamassa tietokantaan tietoa samanaikaisesti, kun käyttäjä B pyytää tietokannalta tietoa talletetuista riveistä. Mikäli tietokanta ei eristäisi näiden kahden operaation toimintaa, voisi talletettujen rivien määrä vaihdella arvaamattomasti riippuen siitä, kuinka monta riviä samanaikaisesti toimiva talletusoperaatio on ehtinyt tallettaa. Eristyneisyyden myötä voidaan määrittää, palautetaanko rivien määrä ennen rivien lisäämistä, vai odotetaanko, että käynnissä oleva tallennusoperaatio on suoritettu loppuun asti. (Elmasri & Navathe 2011: 732-733)

Pysyvyys on puolestaan tiedon säilyvyyden takaava määrite. Pysyvyyden tarkoituksena on varmistaa, että virran katkeaminen tai järjestelmän kaatuminen ei johda datan menetykseen. Käytännössä tämä pyritään varmistamaan tallentamalla komentojen tulokset haihtuvasta (volatile) muistista pysyvään haihtumattomaan (non-volatile) muistiin, vaikka kyseinen komento johtaisi itse tietokannan ohjelmiston kaatumiseen. (Elmasri & Navathe 2011: 732-733)

4.2.2 Oliotietokannat

Olio- tai objektitietokannoiksi kutsutut tietokantaratkaisut on suunniteltu monimutkaisempien tietorakenteiden käyttöä varten. MongoDB:tä voidaan pitää vuonna 2017 eniten levinneenä oliotietokantana, jonka takia useat olio- ja relaatiotietokantoja vertailevat artikkelit tarkastelevat MongoDB:tä tärkeimpänä varteenotettavana vaihtoehtona perinteisille relaatiotietokannoille. Relaatiotietokannoista poiketen objektitietokannan rivien eli objektien välisiä yhteyksiä ei kuvata aina relaatioilla, vaikka tämä onkin mahdollista

MongoDB-tietokannassa (MongoDB 2017 A). On myös huomattava, että vaikka jotkin oliotietokannat tukevat relaatioita, niiden varmuus ei ole aina taattu ACID mallin mukaisesti. Oliotietokantaan voidaan tallettaa relaatioita rikkovaa tietoa, mikäli yhteyksien eheyttä ei tarkasteta ohjelmallisesti ennen tiedon tallettamista. Oliotietokannoille on tyypillisempää sisällyttää tietueeseen liittyvä tieto suoraan olion sisällä (nesting). Relaatiotietokannassa keskustelupalsta voi koostua viestiketju- ja viestitauluista, mutta oliotietokannassa viestiketjuolio voi sisällyttää itsessään kaikki kyseisen keskustelun viestit.

Tämä rakenne voi käyttötarkoituksesta riippuen helpottaa tai vaikeuttaa tiedon käsittelyä. Keskustelupalstaesimerkki havainnollistaa, kuinka oliotietokannan rakennetta voidaan hyödyntää sovelluskehityksessä. Käyttäjän avatessa keskustelun, ohjelman ei tarvitse hakea kuin viestiketjuolio, joka sisältää kaiken tarvittavan tiedon viestien näyttämiseen. Relaatiotietokantaa käyttäessä ohjelma joutuisi hakemaan viestiketjun lisäksi kaikki sille kuuluvat viestit. Olioiden sisältämien olioiden hakeminen on puolestaan usein vaivalloisempaa ja hitaampaa relaatioihin verrattuna. Mikäli keskustelupalstan tulisi esittää kaikki tiettynä päivänä kirjoitetut viestit, relaatiotietokanta voisi hakea kaikki tarvittavat rivit suoraan viestit taulusta päivämäärän perusteella. Vastaavasti oliotietokannan tapauksessa jouduttaisiin käsittelemään kaikki viestiketjuoliot, jotta päästäisiin käsiksi niiden sisältämiin viesteihin ja täten kirjoituspäivämääriin. Kuten relaatiotietokanta joutui aikaisemmassa esimerkissä toimimaan yhden ylimääräisen yhteyden kautta, nyt oliotietokanta vaatii vastaavan ylimääräisen askeleen.

Oliotietokantojen suurimmat edut relaatiotietokantoihin verrattuna ovat tiedon joustavuus, tiedon käsittelyn nopeus ja tietokanta-arkkitehtuurin skaalattavuus. Oliotietokannat on suunniteltu hyödynnettäväksi olio-ohjelmoinnin yhteydessä, joten applikaation ja tietokannan välinen jako ei ole yhtä selvä kuin relaatiotietokantoja käyttäessä. Tietokantaan tallennettavia olioita ei tarvitse muuntaa tallennusta varten, joten ohjelmoinnin ja tietokannan välinen yhtenevyys parantaa ohjelmoijan käyttökokemusta ja nopeuttaa ohjelman kehittämisprosessia (MongoDB 2017 B). Oliotietokantojen nopeudelliset edut muodostuvat tietokannat sisäisten osoittimien hyödyntämisestä sekä nestingin myötä vähennetyistä relaatioista tapauksissa, joissa tarvitaan paljon yhteen riviin liittyvää tietoa. Valitettavasti sanonnan mukaisesti ilmaista lounasta ei ole olemassa. Esimerkiksi MongoDB:n suorituskyvyn vertaaminen perinteiseen relaatiotietokantaan paljastaa käsittelynopeuden hinnan muodostuvan suuremmasta levytilan ja keskusmuistin hyödyntämisestä. MongoDB joutuu varaamaan käyttöönsä enemmän keskusmuistia suoritettavien

hakujen nopeuttamiseksi. (Farrugia 2012) Tietokannan käyttämän levytilan kasvu johtuu osakseen oliotietokantaan tallennettavien olioiden serialisaatiosta, joka vie itse tiedon lisäksi ylimääräistä tilaa jokaista avainta, pilkkua ja sulkua kohden (kuva 2).

```
{
    "id": 1,
    "first_name": "Matti",
    "last_name": "Meikalainen"
}
```

KUVA 2. JSON serialisaatio

MongoDB:n tapauksessa suurin rajoite nopeusetujen hyödyntämiselle vaikuttaisi olevan se, ettei MongoDB kykene nopeuttamaan tietokannan toimintaa, mikäli tietokannan nopeutta rajoittava tekijä on levyn I/O (input output eli kirjoitus- ja lukemisnopeus). Jotkut tietokannat pystyvät nopeuttamaan toimintaa käsittelemällä tietokantaan kohdistuvia operaatioita tietokannalle varatussa keskusmuistissa, mutta käsiteltävän tiedon määrän ylittäessä varatun keskusmuistin, tietokanta joutuu lataamaan uuden tiedon muistiin levyltä, jolloin levyn lukunopeus on rajoittava tekijä. Toisin sanoen, mitä enemmän tietoa yksittäiset operaatiot joutuvat käsittelemään, sitä vähemmän hyötyä tietokannalle varatusta keskusmuistista on. Tietokannan sisältämän tietomäärän kasvaessa myös operaatioiden koolla on tapana kasvaa, sillä esimerkiksi hakulauseet joutuvat käymään läpi suurempia määriä tietoa. (Farrugia 2012)

Lisäksi relaatiotietokannoilla on ajureita, jotka mahdollistavat tiedon kompressoinnin. Kompressointi voi joissain tapauksissa pienentää levytilan kulutuksen jopa puoleen kompressoitujen taulujen kohdalla (taulukko 1). Näitä ajureita tukevaa tietokantaa käyttämällä voidaan säilyttää mahdollisuus kompression hyödyntämiseen tulevaisuudessa. Vuokrattavia palvelimia käyttäessä levytila on usein jokseenkin rajoitettu resurssi, joten oliotietokannan tuomat edut eivät välttämättä oikeuta suurempaa levytilan käyttöä.

TAULUKKO 1. MongoDB- ja MySQL-tietokantoihin talletetun saman tiedon käyttämä levytila (Farrugia 2012)

MongoDB	MySQL InnoDB uncompressed	MySQL InnoDB 4KB block size compression
Data: 306MB	*Data:* 251MB	*Data:* 113MB
Index: 83MB	*Index:* 59MB	*Index:* 22MB
Total: 439MB	*Total:* 310MB	*Total:* 135MB
100%	71%	31%

Koska tätä työtä varten kehitettävä web-sovellus hyödyntää Laravel-ohjelmistokehystä, voidaan tietokantojen ajureja vaihtaa helposti hyödyntämällä Eloquent ORM abstraktiotasoa Eloquent mahdollistaa myös useiden erilaisten tietokantojen yhtäaikaisen käytön, jolloin tiedot voidaan jakaa erilaisten tietokantojen kesken tietotarpeiden mukaan.

4.3 Muut menetelmät

Relaatiotietokantoja ei ole kehitetty kaikenlaisen tiedon tallentamiseen. Esimerkiksi suurien binääri- ja tekstitiedostojen tallentaminen relaatiotietokantoihin voi tietokannasta riippuen hidastaa riviin kohdistuvia hakuja ja varata talletusta varten enemmän tilaa, kuin mitä varsinainen tieto vaatisi. Ohjelmat voivat käyttää hybridiratkaisua, jossa tietokantaan soveltuva tieto tallennetaan tietokantaa ja tietokantaan sopimaton tieto tallennetaan omaan tiedostoonsa, jonka sijaintiin voidaan viitata tietokannan sarakkeesta.

Hybridiratkaisun vaarana on tiedostoiden hallinnoinnin erillisyys tietokannan ACID mallista. Tietokanta ei voi itse vaikuttaa tietokannan ulkopuolisiin tiedostoihin, joten niiden poistaminen, päivittäminen ja luominen jää ohjelman vastuulle. Ohjelman toteuttamisessa on täten olennaista luoda ratkaisu, joka kykenee tarkkailemaan muutoksia tietokantaan ja toteuttamaan tarvittavat toimenpiteet tietokannasta erillisiin tiedostoihin. Tarkkailua varten voidaan hyödyntää esimerkiksi tarkkailijamallia (observer pattern), joka sidotaan ohjelmassa käytetyn tietokannan toimintaan.

5 TIETOKANNAN TOIMINNAN VERTIKAALINEN SKAALAAMINEN

5.1 Object Relational Mapping

Object relational mapping:in tai ORMin tarkoitus on luoda abstraktiorajapinta, jota käytetään tietokannan kanssa kommunikointiin applikaation omalla kielellä. Tätä rajapintaa voidaan käyttää tietokantoja hyödyntävien sovellusten ohjelmoimisen nopeuttamiseksi. Yleisesti käytettyjen hakujen yksinkertaistaminen omiksi luokikseen helpottaa koodin luettavuutta, yksinkertaistaa ylläpitoa ja auttaa ohjelmoijaa välttämään samojen hakulauseiden jatkuvaa toistamista. Hyvin toteutettua rajapintaa käyttäessä ohjelmoijan ei tarvitse edes tietää, millaista tietokantaa sovellus hyödyntää. Näistä syistä ORM on käytössä monissa suurten web-kielten (java, python, php) ohjelmistokehyksissä. ORMin abstraktion hyötyjä ja haittoja voidaan verrata helposti itse ohjelmointikielien kehittymiseen assembly-konekielistä korkeamman tason ratkaisuihin: alatason kielet mahdollistavat tarkemman koodin toiminnallisuuden ja muistin käytön hienosäädön, kun ylätason kielet puolestaan nopeuttavat itse koodausprosessia ratkaisemalla alatason ongelmat yleispätevillä ratkaisuilla (Kaushal 2015).

Ohjelmiston skaalaamisen kannalta abstraktioihin turvautuminen ei kuitenkaan anna ohjelmoijalle täyttä vastuunvapautta sivuttaa tietokannan käyttöön liittyvää tietämystä. Minkä tahansa ORMin tehokas ja skaalautuva käyttö vaatii ymmärrystä jokaisen hyödynnetyn tietokantaoperaation pohjalla toimivasta toteutuksesta. Hyvä ohjelmoija pystyy havaitsemaan tilanteet, joissa valmiita ratkaisuita tulee välttää niiden toteutukseen liittyvien ongelmien takia. Ohjelmoijan tulee osata laajentaa abstraktiomallia tai tarpeen mukaan ohittaa koko abstraktio kirjoittamalla komento suoraan tietokannan käyttämällä kielellä. Kokemattomuus itse abstraktiomallien käytössä voi nopeasti hidastaa tai pysäyttää sovelluksen toiminnan, jolloin järjestelmän kaatuminen voi toimia ohjelmoijalle hyvänä käytännönläheisenä oppimiskokemuksena, joka auttaa havaitsemaan nykyisen ratkaisun puutteet ja välttämään vastaavia ongelmia tulevaisuudessa.

Ohjelmistot joutuvat prosessin elinkaaren aikana lataamaan muistiinsa tietoja, kuten ohjelman ulkopuolisia asetustiedostoja ja koodissa määriteltyjä muuttujia. Eager loading:illa tarkoitetaan tiedon lataamista muistiin ennen tarvetta käyttää kyseistä tietoa. Eager loading lataa tiedon muistiin valmiiksi, jotta kyseinen tieto olisi helposti saatavilla

myöhemmin. Tietomäärien kasvaessa eager loadingin käyttäminen tulee vaarallisemmaksi.

```
$user→files→count();

$user→files()→count();
```

KUVA 3. Eloquent ORM

Kuvan 3 esimerkissä havainnollistetaan, kuinka ORMin toiminnan tuntemattomuus voi aiheuttaa muistinkäytölle asetettujen rajojen ylittymisen sovelluksessa. Ylempi lauseke lataa käyttäjälle kuuluvat tiedostot hyödyntäen olion määrittämää relaatiomallia ja laskee tiedosto-olioiden määrän. Alempi lauseke puolestaan palauttaa ensin käyttäjälle kuuluvat tiedostot valitsevan hakulauseen ja laskee sitten haun osumien määrän. Näiden kahden esimerkkilauseen väliset erot korostuvat tilanteessa, jossa halutaan listata käyttäjille kuuluvien tiedostojen määrä. Tällöin ensimmäinen esimerkki saattaa ladata jokaista käyttäjää kohden useiden megatavujen edestä tiedostoja, kun taas toinen esimerkki lataisi muutaman tavun kokoisen kokonaislukumuuttujan.

5.2 Rivien tallentaminen

Tietokantaan kohdistuvissa suurissa operaatioissa suoritusajan optimoimiseksi myös ohjelman käyttämä keskusmuisti tulee hyödyntää tietokantaoperaatioiden optimoimiseen ja tiedonkäsittelyyn. Havainnollistavana esimerkkinä voidaan kuvitella tilanne, jossa ohjelmoijan tehtävä on tallettaa tietokantaan suuri lista käyttäjätietoja.

Ensimmäinen esimerkkiratkaisu on suorittaa ohjelmassa silmukka, joka käy listan läpi ja tallentaa jokaisen käyttäjätietoja sisältävän rivin (kuva 4). Tämä ratkaisu minimoi käsittelevän ohjelman tarvitseman muistin, sillä käyttäjätiedot voidaan lukea ja tallettaa rivi kerrallaan säilyttämällä kerrallaan vain yhteen riviin liittyvä tieto ohjelman muistissa. Ratkaisun haittapuolet näkyvät nopeasti isoissa operaatioissa: jokainen yksittäisen rivin talletusoperaatio suorittaa erillisen yhteyden tietokantaan, hidastaen koko listan käsittelyä.

```
DB::transaction( function()
{
    for($n = 0; $n < 1000; $n++)
    {
        $user->id = $n;
        $user->save();
    }
});
```

KUVA 4. Rivien lisääminen tietokantaan yksi kerrallaan

Toinen vaihtoehto on käydä läpi käyttäjätiedot sisältävä lista silmukalla tallentaen jokainen rivi muuttujaan (muistiin) ja lopuksi tallettaa kaikki käyttäjätiedot kerrallaan bulk insert-operaatiolla (kuva 5). Tämä vaihtoehto on teoriassa noin talletettavien rivien määrän nopeampi, kun eniten aikaa kuluttava operaatio on itse tietokantayhteyden muodostaminen. Jos tehtävänä on tallettaa 1000 riviä käyttäjätietoja, bulk insert toimii joissain tapauksissa lähes tuhat kertaa nopeammin verrattuna rivien tallettamiseen yksi kerrallaan.

```
$users = [];
for($n = 0; $n < 1000; $n++)
{
    $user = ['id' => 1];
    $users[] = $user;
}
DB::table('users')->insert($users);
```

KUVA 5. Kaikkien rivien lisääminen tietokantaan yhdellä operaatiolla

Missä ensimmäinen esimerkkiratkaisu edustaa tilan minimoimaa vaihtoehtoa, toinen korostaa suoritukseen kuluvan ajan minimointia tilan kustannuksella. Täten myös toisen ratkaisun aiheuttamat ongelmat ovat suhteellisen helposti ennustettavissa: käyttäjätiedot sisältämän listan kasvaessa ohjelman käyttämä muisti kasvaa 1:1 suhteessa, jolloin suuren listan käyttäminen johtaa väistämättä muistin loppumiseen. Ohjelmapuolen muistin rajallisuus ei ole kuitenkaan ainoa ratkaisuun liittyvä ongelma. Käytännön testaaminen MySQL-tietokannalla osoittaa, että yksittäisen tietokannalle lähetettävän komennon kasvaessa myös komennon purkamiseen ja tulkitsemiseen kuluva aika kasvaa eksponentiaalisesti eikä lineaarisesti (taulukko 2). Huomioon tulee ottaa myös ohjelmiston, tietokannan ja itse verkon yksittäisten viestien koolle asettamat rajoitteet.

TAULUKKO 2. 10 000 rivin lisäämiseen kuluva aika link tracker -sovelluksessa

Yhden INSERT operaation koko	Operaatioiden määrä	Suoritusaika
10 000	1	35 s
1000	10	9.6 s
500	20	7.6 s
100	100	7.7 s
10	1000	12.1 s

Kaikkien resurssien tasapuoliseen hyödyntämiseen tulee käyttää kolmatta ratkaisua. Applikaation on hyödynnettävä esimerkkejä yksi ja kaksi tasoittaen tilan ja ajan käytön sellaisella tavalla, että ohjelma voidaan suorittaa mahdollisimman nopeasti, mutta myös käytettävän muistin rajoissa sekä luotettavalla virhemarginaalilla. Ohjelman muistinkäyttöä ei voi tietenkään laskea pelkän teoreettisen maksimin perusteella, vaan ratkaisun pitää olla tarpeeksi joustava toimimaan myös käyttäjälistan tietueiden tiedon koon muuttuessa. Yksinkertainen tapa täyttää nämä kriteerit tallettaa tieto erissä, joiden kokoa voidaan muuttaa ohjelman nopeimman toteutusmallin selvittämiseksi (kuva 6).

```
DB::transaction( function()
{
    $users = [];
    $maxInsertSize = 100;
    for($n = 0; $n < 1000; $n++)
    {
        $user = ['id' => 1];
        $users[] = $user;
        if($n != 0 && $n % $maxInsertSize == 0)
        {
            DB::table('users')->insert($users);
            $users = [];
        }
    }
});
```

KUVA 6. Rivien lisääminen tietokantaan 100:n erissä

Erien koon muuttaminen mahdollistaa suorittamiseen kuluvan ajan optimoimisen juuri haluttua käyttötarkoitusta varten. Tarkoituksena on siis suorittaa silmukkaa, joka tallentaa käyttäjätietoja, kunnes silmukan suorituskertojen määrä on esimerkiksi jaollinen talletuserän koolla. Kun silmukka täyttää tämän kriteerin, suoritetaan bulk insert-operaation välimuistiin talletetuilla käyttäjätiedoilla ja tyhjennetään välimuisti, joten ohjelman käyttämä muisti ei kasva yhden erän kokoa suuremmaksi.

5.3 Heuristiset menetelmät

Edellisessä luvussa esimerkkinä esitetty tiedoston sisällön käsittely on tehtävä, jolla voidaan havainnollistaa myös muita skaalaamiseen liittyviä haasteita. Muistin säästämiseksi on olennaista käyttää jäsentäjää (parser), joka kykenee käsittelemään tiedostojaa rivi kerrallaan säilyttämättä koko tiedoston sisältöä välimuistissa. Rivien erillisen käsittelyn seurauksena toistaiseksi käsittelemättömien rivien sisältöä ei voida hyödyntää. Mitä enemmän tietoa itse tiedostosta on saatavilla, sitä monipuolisempia ratkaisuja voidaan soveltaa käsittelyn nopeuttamiseen. Rivejä tallettaessa tällä ei tavallisesti ole juurikaan merkitystä, mikäli rivit eivät sisällä tarkkailtavia duplikaatteja. Myöskään rivien poistaminen ei ole erityisen kiinnostava prosessi, sillä poistettavien rivien määrän kasvaessa tietokannalle lähetettävän tiedon määrä ei usein kasva samalla tavoin, kuin rivejä lisätessä. Poikkeuksena tähän sääntöön on tilanne, jossa rivit koostuvan vain yhdestä avaimena käytetystä sarakkeesta ja useita rivejä täytyy poistaa avainkohtaisesti. Tällaisessa tilanteessa aikaisemmin esitetty ratkaisu eräkohtaisesta suorituksesta on jälleen toimiva.

Koko tiedoston sisällön säilyttäminen muistissa mahdollistaisi laajan tarkkailun toistuvien säännönmukaisuuksien havaitsemiseksi, mutta on usein mahdotonta muistin rajallisuuden ja tarkkailuun kuluvan ajan takia. Säännönmukaisuuksia etsiessä eräkohtaisella käsittelyllä tulee muistaa, ettei löydettyjä yhtenäisyyksiä voi aina soveltaa tehokkaasti koko tiedostoon tai muuhun vastaavaan tietosarjaan. Säännönmukaisuuksia voidaan hyödyntää heuristisilla menetelmillä, joilla tarkoitetaan ongelmanratkaisua etukäteen määritetyillä tavoilla, jotka pääsevät nopeasti lähelle parasta tulosta. Jos ongelmana on esimerkiksi löytää listalta kaikki a-kirjaimella alkavat nimet, joudutaan koko lista käymään läpi tarkastaen jokainen alkukirjain. Ongelmaan liittyvä tieto voi kuitenkin muuttua muuttamatta itse ongelman kuvausta. Aakkosellisesti jäsennetyn listan, jonka käsittely aloitetaan listan alusta, voidaan lopettaa heti, kun alkukirjain ei ole a. Tämä tieto voi sovellettuna säästää miljoonien rivien tarkastamiseen kuluvan ajan. Muita vastaavia säännönmukaisuuksia voidaan hyödyntää, jos tehtävänä olisi löytää s-kirjain. Nopeasti pääteltynä voidaan ajatella, että listan lukeminen on järkevintä aloittaa lopusta, sillä s on aakkosten loppupuolella. Tämä on kuitenkin itsessään merkityksetön tieto ilman lisätietoa eri alkukirjaimella alkavien nimien määristä.

5.3.1 Duplikaattien käsittely

Useissa tapauksissa, esimerkiksi henkilötietoja tallettaessa, tietokannassa halutaan säilyttää vain yksi kopio jokaisesta rivistä. Talletettava tieto voi kuitenkin sisältää kopioita olemassa olevista uniikeista riveistä, joko tietokannan tai itse tiedoston rivien sisällä. Kuten tietokannat osiosta selvisi, objektitietokantojen käyttäminen on tässä tilanteessa vaarallisempaa, sillä ilman erillistä ohjelmallista käsittelyä myös kopiot tallentuvat tietokantaan. ACID-mallia käyttävä MySQL-tietokanta puolestaan pysäyttää operaation käsittelyn ja tuloksena on `integrity constraint violation` eli eheyden rikkova virhe. Kummassakaan tapauksessa lopputulos ei vastaa haluttua.

```
INSERT INTO 'users' (id) VALUES (1) ON DUPLICATE KEY IGNORE;
```

KUVA 7. Toistuvien avaimien hylkääminen SQL tietokannoissa

Relaatiotietokantojen tapauksessa ongelma on yleisesti helppo korjata duplikaattiavaimet ohittavalla lauseella. Kuva 7 antaa esimerkin SQL-lausekkeesta, joka tallettaa uuden rivin users-tauluun id-sarakkeen arvolla 1, ohittaen esiintyvät duplikaattivirheet.

Relaatiotietokantojenkin kanssa kopioiden käsittelystä tulee huomattavasti haastavampi ongelma, jos ohitettujen rivien sisältämällä tiedolla on merkitystä, sillä tietokanta ei kirjaa ohitettujen rivien tietoja, vaan palauttaa talletettujen rivien määrän. Esimerkkinä tällaisesta tarpeesta voidaan kuvitella usealla paikkakunnalla järjestettävät arpajaiset. Jokaisella paikkakunnalla on oma tietokantansa osallistujista ja arpajaisten lopuksi arvotaan ensin voittajapaikkakunta ja sitten itse voittaja paikkakunnan osallistujista. Jälkikäteen arpajaisten järjestäjät haluavat estää huijarien osallistumisen seuraaviin arpajaisiin. Useilla paikkakunnilla osallistuneiden huijarien löytäminen osoittautuu kuitenkin vaikeaksi. Lisätessä kaikki tiedot yhteen tietokantaan tiedetään huijarien määrä, mutta ei itse henkilöiden tietoja. Rivit täytyy tallettaa yksi kerrallaan rivikohtaisten duplikaattien tarkistamiseksi, joka osoittautuu puolestaan tuskaisen hitaaksi suuren osallistujamäärän takia.

```
DB::transaction( function()
{
    $huijarit = [];
    foreach($osallistujat as $osallistuja)
    {
        if(DB::table('osallistujat')->find($osallistuja->nimi) != null)
        {
            $huijarit[] = $osallistuja->nimi;
        }
    }
});
```

KUVA 8. Rivikohtainen duplikaattien tarkastaminen

Kuva 8 paljastaa, kuinka jokaisen rivin tallettaminen vaatisi kaksi erillistä tietokantaoperaatiota: olemassa olevan tiedon tarkastamisen ja uuden tiedon tallentamisen Tässä tapauksessa suoritusnopeutta voidaan parantaa eräkohtaisella joukkatallennuksella, tarkkaillen samalla mahdollisia tietokannan duplikaattivirheitä. Tietokannan kohdatessa virheen talletuserä jaetaan kahtia ja yritetään tallentaa molemmat uudet erät. Tätä prosessia toistamalla saadaan lopulta selvitettyä virheen aiheuttanut rivi. Ratkaisua voidaan soveltaa tehokkaasti, kun toistuvien rivien määrä on suhteellisen pieni.

5.3.2 Tiedoston sisällön vertaaminen tietokantaan

Tiedoston sisällön vertaaminen tietokannan riveihin antaa toisen kiinnostavan mahdollisuuden prosessin tehokkuuden tarkkailuun. Rivikohtaiseen käsittelyyn verrattuna nopeampiin tuloksiin päästään jälleen heuristisia menetelmiä soveltamalla. Tilanteesta ja käyttötarkoituksesta riippuen lähes millä tahansa tavalla selkeästi järjestettyä tietoa voidaan käyttää erillisten tietokantayhteyksien määrän vähentämiseksi. Tässä tapauksessa tiedoston aakkosellista järjestystä voidaan hyödyntää merkkijonojen vertailuun. Yksittäisten rivien vertaamisen sijaan voidaan yrittää arvata mahdollisimman monen seuraavan verrattavan tietueen muoto ja ladata näiden tietueiden vertaamiseen vaadittava tieto yhdellä kerralla. Jos tarkoituksena on esimerkiksi selvittää, kuinka moni tiedoston tietue on jo valmiiksi talletettu tietokantaan, ja tiedetään tiedoston olevan aakkosjärjestyksessä, voidaan tietokannasta ladata valmiiksi kaikki käsiteltävän tiedon alkukirjaimella alkavat rivit. Tiedon käsitteleminen ohjelman omassa muistissa esimerkiksi taulukkona ei ole välttämättä tietokannan käsittelyä nopeampaa, mutta se vähentää tietokantayhteyksien mää-

rää, jolloin se voi olla silti nopeampi tapa käsitellä rivien vertaamista tiedostoon. Tietokannan osien tallentaminen ohjelman tietorakenteisiin muuttujina asettaa tietysti omat rajoitteensa: suuren tietokannan osat kasvavat nopeasti liian isoiksi, jolloin prosessi ylittää muistille asetetut rajat ja pysähtyy.

Muistin ylikuormittamisen estämiseksi tälle välimuistiin ladatulle tiedolle tulee asettaa jonkinlainen maksimikoko. Aakkosellisesti järjestetyn tiedon tapauksessa maksimikoon mukainen välimuisti voidaan ladata etsimällä tietokannasta tarkasteltavan merkkijonon alkukirjaimilla alkavat rivit. Vastaavien rivien määrä lasketaan tietokannasta aloittaen yhdestä alkukirjaimesta, jonka jälkeen hakulauseeseen lisätään yksi alkukirjain niin kauan, kunnes laskettujen rivien varaama muisti on välimuistille asetettua rajaa pienempi. Rivejä laskemalla vältetään itse tiedon lataaminen ohjelman muistiin ennen, kuin alkukirjaimia vastaava rivien määrä on riittävän pieni ohjelman käsiteltäväksi. Esimerkiksi taulukossa 3 testattava sana on 'testi' ja välimuistin maksimikoko 1000 riviä. T-kirjaimella alkavia sanoja on tietokannassa 2132, koska 2132 > 1000, lasketaan tietokannasta kahdella alkukirjaimella 'te' alkavat sanat (1789). Koska 1789 > 1000, prosessi toistetaan uudestaan laskemalla kolmella alkukirjaimella 'tes' alkavat sanat (847). Kolmella alkukirjaimella rivien määrä on pienempi kuin 1000, joten sama hakulause suoritetaan uudestaan ja tulos talletetaan välimuistiin.

TAULUKKO 3. Tiedon hakeminen sanan alkuosan perusteella.

Alkukirjaimia	Hakusana	Rivejä tietokannassa
1	t	2132
2	te	1789
3	tes	847

Välimuistin kokoa laskiessa on kuitenkin huomioitava itse hakujen suorittamiseen käytettävä aika. Suuria tietorakenteita käsitellessä tulee välttää raskaita hakulausekkeita, sillä muuten tietokantayhteyksien vähenemisestä saatavat nopeudelliset edut eivät tee välimuistin käytöstä kannattavaa. MySQL-tietokannan tapauksessa alkukirjainten perusteella hakeminen onnistuu suhteellisen tehokkaasti, sillä toisin kuin loppukirjaimia hakiessa, MySQL kykenee hyödyntämään indeksejä haun nopeuttamiseksi.

```
SELECT * FROM 'users' WHERE name LIKE 'tes%';
SELECT * FROM 'users' WHERE name LIKE '%sti';
```

KUVA 9. SQL haku merkkijonon alku- ja loppuosan perusteella.

Kuvan 9 SQL-lausekkeista ensimmäinen hyödyntää indeksejä hakujen nopeuttamiseen, mutta loppuosan perusteella hakiessa joudutaan suorittamaan niin sanottu full table scan, jolloin haku hidastuu.

6 VERTIKAALINEN SKAALAAMINEN JONOILLA

Tavallisesti selaimen tai muun vastaavan asiakassovelluksen sulkeminen keskeyttää palvelimelta pyydetyn tehtävän suorittamisen, joka on ongelma varsinkin pidempiä prosesseja käsitellessä. Jos asiakkaan yhteyden katkaisu jätetään huomiotta esimerkiksi PHP:ssa `ignore_user_abort(true)` -komennolla, palvelin voi jäädä suorittamaan useita pitkäaikaisia prosesseja. Prosessien välisten vaikutusten ja käyttäjän toiminnan arvaamattomuuden takia onkin suositeltavaa suorittaa tietokantaan kohdistuvat operaatiot atomisesti transaktioilla. Jonoja hyödyntävän applikaation käyttäjän ei kuitenkaan tarvitse odottaa itse tehtävän suoritusta, vaan pelkkä tehtävän muodostaminen riittää. Kun tehtävä on tallennettu onnistuneesi jonoon, se voidaan suorittaa myöhemmällä ajankohdalla omana prosessinaan.

6.1 Toimintaperiaate

Jonon päätarkoitus on tiedon tallentaminen järjestettyyn tietorakenteeseen myöhempää käsittelyä varten (Knuth 1997: 238-243). Itsessään laaja määritelmä mahdollistaa jonojen toteuttamisen lukuisilla erilaisilla tilaa tallentavilla tekniikoilla. Jonon käyttämä tieto voidaan tallentaa esimerkiksi tiedostoon, tietokantaan, evästeeseen tai mihin tahansa muuhun saatavaan pysyvään varastoon. Jonon abstraktin konseptin varsinaista toteutusta edellä mainituilla tekniikoilla kutsutaan Laravel-ohjelmistokehyksessä ajuriksi (Laravel 2017). Koska jonoihin talletetut tehtävät on tarkoitettu suoritettavaksi myöhemmällä ajankohdalla, ne suoritetaan usein talletuksesta erillisellä prosessilla.

Käytännössä jono voidaan toteuttaa useilla eri tavoilla, mutta kaikkein yleisin ratkaisu on jonkinlainen tietokanta. Varsinaisesti jonojen käsittelyyn suunnitellut tietokannat kuten Redis kykenevät perinteisiä tietokantoja nopeampaan käsittelyyn. Sen sijaan että tietokanta ladattaisiin levyltä, muun muassa Redis ja SQLite pystyvät lyhentämään hakuaikoja säilyttämällä tiedon suoraan muistissa. Muistin hyödyntäminen nopeuttaa tehtävien tallennusta ja niiden hakemista jonosta. Muisti on kuitenkin levytilaan verrattuna liian kallis resurssi suurta tietomäärää tallentaessa (Elmasri & Navathe 2011: 568). Jos tehtävä käsittelee huomattavaa tietomäärä, tulee tieto hakea käsittelyn aikana perinteisestä tietokannasta muistin säästämiseksi. Tiedon tallentamisen menetelmät voidaan pohjimmiltaan jakaa kahteen pääkategoriaan: viestijonoon ja käsittelijäjonoon.

Viestijonon tarkoituksena on toimia väliaikaisena sijoituspaikkana applikaation erilaisille tapahtumille. Esimerkiksi käyttäjän rekisteröidyttyä jonoon talletetaan "käyttäjä rekisteröityi" -tapahtuma, sekä rekisteröityneen käyttäjän tiedot. Viestijonoja varten rekisteröidään erillisiä tapahtumia kuuntelevia prosesseja (listener). Viestin saadessaan kukin kuuntelija määrittää, miten se haluaa käsitellä saamansa viestin. Käyttäjän rekisteröitymisen tapauksessa kuuntelija voi esimerkiksi lähettää käyttäjälle tilin aktivoivan varmistusviestin. Viesti- tai tapahtumajonoille on ominaista muodostaa erilaisia sääntöjä, jotka märittelevät mille ryhmille tietyt viestit ovat saatavilla. Tapahtumat tai viestit voidaan jakaa myös yleislähetyksenä (broadcasting), jolloin ne ovat kaikkien halukkaiden kuuntelijoiden saatavilla. Viestijonot hyödyntävät tilallisia (stateful) menetelmiä, kuten pistokkeita tehtävien tiedottamiseen käsittelijöille käsittelijäjonoja useammin.

Käsittelijäjonossa jonoon tallennettava tieto muodostetaan niin, että jonoon talletettava tehtävä säilyttää itsessään sen suorittamiseen tarvittavat tiedot. Tämän takia käsittelijäjono tarvitsee tavallisesti toimintaansa viestijonoa enemmän tilaa. Koko tehtävän koodin tallettaminen olisi tilan käytön kannalta tehotonta, jonka takia jonoon talletetaan usein vain viittaus tehtävän käyttämään koodiin, sekä skriptin käyttämiseen tai luokan muodostamiseen tarvittavat muuttujat. Käsittelijä on ajoitettu, toistuva prosessi, joka tarkastaa jatkuvasti jonon sisältöä uusien tehtävien varalta. Käsittelijän havaitessa jonossa käsiteltävää sisältöä se muodostaa ensin suoritettavan tehtävän jonon sisällöstä ja suorittaa sitten kyseisen tehtävän. Suurin ero viesti- ja käsittelijäjonojen välillä on vastuu tehtävän suorittamisessa. Viestijonon tapauksessa kuuntelijaprosessi päättää jatkotoimenpiteistä, kun taas käsittelijäjono suorittaa tehtävän saamiensa ohjeiden mukaisesti.

6.2 Hyödyt

Koska tehtävät suoritetaan myöhemmällä ajankohdalla, itse pyyntöjen käsittely nopeutuu, sillä käyttäjän ei tarvitse odottaa varsinaisen tehtävän suoritusta. Ohjelman suorituksen jakaminen tehtäväluokkiin selkeyttää koodin rakennetta ja toimii yhden vastuun mallin (single responsibility principle) mukaisesti pakkaamalla yhden toiminnon yhteen tehtäväluokkaan. Talletetut tehtävät voidaan käsitellä web-ympäristön kannalta poikkeuksellisesti ilman käyttäjää, sillä alkuperäinen käyttäjäsyöte voidaan tallettaa muotoon, jota voidaan hyödyntää myöhemmin tehtävän suorittamiseen. Tämä puolestaan mahdollistaa useiden tehtävien yhtäaikaisen suorituksen monisäikeisyydellä.

Jonojen käyttö soveltuu erityisen hyvin sähköpostin lähettämisen kaltaisiin tehtäviin, sillä useat yhtäaikaisesti palvelimelta lähtevät sähköpostit aiheuttavat helposti pullonkaulan lähetyspalvelimella. Sen sijaan, että käyttäjä joutuisi viestin lähetystä odottaessa katsomaan käyttöliittymän latausikkunaa useita minuutteja, voidaan vastaus onnistuneesta rekisteröitymisestä palauttaa heti. Nopeampi käsittely parantaa käyttökokemusta. Jonoja voidaankin hyödyntää lähes koko applikaation toiminnan nopeuttamiseen, palautetta vaativia tehtäviä lukuun ottamatta. Viivästetty suoritus ei sovellu esimerkiksi laskinsovellukseen, koska käyttäjän tavoitteena on saada laskun tulos mahdollisimman nopeasti. Myöskään erittäin yksinkertaisten operaatioiden nopeutta ei voida välttämättä parantaa, sillä itse tehtävän tallettaminen jonoon voi olla raskaampi prosessi kuin pyynnön välitön käsittely.

Jonoja hyödyntävän arkkitehtuurin tehokas toteuttaminen vaatii suoritettavien tehtävien onnistunutta jakamista pienempiin paloihin. Jos tavoitteena olisi 100 000 rivin tallentaminen tietokantaa, yksittäinen tehtävä voisi olla tuhannen rivin tallentaminen. Pienempien tehtävien suorittamisella on joitakin etuja raskaisiin prosesseihin verrattuna: esimerkiksi muistin- tai ajankäytölle asetettuja rajoitteita ei tarvitse muuttaa tehtävien suorittamisen mahdollistamiseksi, joten näiden asetusten tuoma turvallisuus pystytään hyödyntämään applikaatiossa. Yksittäisten tehtävien muodostaminen ei ole kuitenkaan aina näin selvää, sillä joskus tehtävän muodostamiseen voidaan vaatia toisen tehtävän tulos. Kun suoritusjärjestyksellä on väliä, voidaan tehtävät muodostaan ketjuttamalla. Ketjuttaessa muodostetaan loogisesti järjestetty ketju suoritettavista tehtävistä, jonka suorittaminen aloitetaan ensimmäisestä tehtävästä. Käsittelyn jälkeen jokainen tehtävä lisää jonoon ketjun seuraavan tehtävän ja antaa uudelle tehtävälle oman tuloksensa. Esimerkiksi erityisen vaativan laskun ratkaisemiseksi voidaan muodostaa laskujärjestyksen mukainen ketju.

6.3 Haasteet

Jonojen hyödyntäminen sovelluskehityksessä mahdollistaa muuten liian raskaiden tehtävien suorittamisen web-ympäristössä. Tehtäväpohjainen toimintamalli aiheuttaa kuitenkin kehitykseen haasteita, joiden selvittäminen vaatii jono-arkkitehtuuriin ohjelmoijalta useita tunteja. Isojen prosessien pilkkominen itsenäisiksi tehtävikseen vaatii tarkkaa suunnittelua. Tehtävien irtonaisen luonteen ja web-ympäristön tilattomuuden takia tietokantaa tai muuta vastaavaa tilallista varastoa joudutaan hyödyntämään entistä enemmän

muun muassa tehtävien käyttämän tiedon säilyttämiseen. Lisäksi jonojen toiminnan tarkkailemiseksi joudutaan tallettamaan usein metatietoa, eli tietoa kuvaavaa tietoa. Metatietoa voidaan käyttää esimerkiksi paloitellun tehtävän suorituksen tarkkailussa: paloiteltuja alatehtäviä luodessa talletetaan myös kyseisten tehtävien määrä. Jokaisen osatehtävän valmistuessa voidaan päivittää myös suoritettujen osatehtävien määrä. Näillä metatiedoilla voidaan seurata prosessien etenemistä vertaamalla suoritettujen tehtävien määrää prosessin koko tehtävämäärään. Tämän tiedon säilyttäminen mahdollistaa myös prosessien etenemisen esittämisen käyttöliittymässä.

Tehokasta sovelluskehitystä varten on olennaista, että myös jonot ovat helposti testattavissa. Testaamisen takia jonojen tulisi käyttää rajapintoja ajurien määrittelemiseksi. Jonojen testaaminen onnistuu usein helpoiten reaaliaikaisella ajurilla, joka kykenee käsittelemään jonon sisältöä välittömästi testien aikana tavanomaisen erillisen prosessin sijaan. Metatietoja käsittelevät luokat, kuten pienempiin osiin jaetut tehtävät tulisi toteuttaa yläluokkien tai rajapintojen avulla. Esimerkiksi olio-ohjelmoinnissa tietokantaa päivittävät toiminnot voidaan tallettaa tehtävän yläluokkaan, vähentäen uuden tehtäväluokan toteutusvirheiden riskiä.

```php
abstract class ParentJob
{
    private $id;

    public function handle()
    {
        $this->execute();
        $this->updateProgress();
    }

    private function updateProgress()
    {
        DB::table('jobs')->find($this->id)->update(['status' => 'done']);
    }

    protected abstract function execute();
}

class Job extends ParentJob
{
    protected function execute()
    {
        // Functionality
    }
}
```

KUVA 10. Yläluokan hyödyntäminen jonojen toiminnan tarkkailemiseen

Kuva 10 havainnollistaa yläluokan hyödyntämistä tähän tarkoitukseen: ohjelmoijan ei tarvitse muistaa kutsua erikseen tietokantaa päivittävää `updateProgress` -metodia, sillä yläluokka kutsuu metodia automaattisesti suorituksen yhteydessä. Vastaavilla ratkaisuilla voidaan vähentää inhimillisistä virheistä aiheutuvien ongelmien määrää.

Virhetilanteiden hallinta on yksi jonojen käyttöön liittyvistä haasteista. Virhetilanteissa yhden tehtävän epäonnistuminen voi aiheuttaa odottamattomia seurauksia, joiden syitä on usein vaikea löytää. Ohjelmistosuunnittelijan vastuulla on miettiä, hylätäänkö koko prosessin tulos yksittäisen osatehtävän epäonnistuessa, vai ilmoitetaanko prosessin lopputulos käyttäjälle ilmenneistä virheistä huolimatta? Myös mahdolliset metatietoja käyttävät järjestelmät voivat sekoittua, mikäli ne eivät kykene huomioimaan virhetilanteita. Aina ei ole edes selvää, onko virhe peräisin tehtävän muodostamistilanteesta vai ilmeneekö se tehtävän suorittaman koodin seurauksena. Näiden ongelmien takia jonojen kanssa on suositeltavaa käyttää komentomallin (command pattern) mukaista koodirakennetta, joka mahdollistaa edellisen komennon kumoamisen, jolloin ohjelmisto voidaan saattaa takaisin tehtävän suorittamista edeltäneeseen tilaan.

Ketjutetut tehtävät ovat vielä paljon tavallisia tehtäviä haasteellisempia. Monisäikeisyyttä ei voida hyödyntää ketjutettujen tehtävien suorittamiseen, koska tehtävien suoritusjärjestyksellä on väliä ja jokainen tehtävä voi vaatia edellisen tehtävän tuloksen toimiakseen. Ketjuttaminen vaikeuttaa myös tehtävien suorituksen tarkkailua, sillä toteutuksesta riippuen jonossa voi näkyä vain yksi tehtävä, vaikka itse ketju saattaisi olla satojen tehtävien pituinen. Tämä johtuu tehtävien dynaamisesta muodostamisesta: jos ketjun tehtävien muodostamiseen tarvittavaa tietoa ei ole olemassa ennen aikaisemman tehtävän suoritusta, ei tehtäviä voida tallettaa jonoon ennen edellisen tehtävän valmistumista. Ketjutettujen tehtävien tarkkailua varten toteutettava ratkaisu voi olla tapauksesta riippuen hyvinkin monimutkainen, jotta jäljellä olevien tehtävien määrä voitaisiin laskea luotettavasti.

7 PALVELINTEN HORISONTAALINEN SKAALAAMINEN

Käyttäjämäärien kasvaessa yksittäinen palvelin ei ole enää tarpeeksi tehokas käsittelemään kaikkea applikaation toimintaa. Käyttötarkoituksesta riippuen yksittäinenkin käyttäjä voi vaatia sovellukselta enemmän laskentatehoa, kuin mitä olisi kustannustehokasta toteuttaa yhdellä tehokkaalla palvelimella. Yhä useimmilla sosiaalisen media ja verkkokauppojen aikakauden palveluilla on useita satojatuhansia käyttäjiä. Internetin käyttäjäkunnan kasvamisen ja jatkuvasti kunnianhimoisempien sovellusten takia tarve hajautetuille järjestelmille on kasvussa. Tällaisia järjestelmiä tarjotaan nykyään usein pilvipalveluiden muodossa. Vuonna 2017 lähes kaikki merkittävät teknologiayritykset, kuten Microsoft, Google, IBM ja Amazon tarjoavat jonkinlaisia palvelinratkaisuja pilvessä. Esimerkiksi Microsoft on keskittynyt strategisesti pilvipalveluiden kehittämiseen jo vuodesta 2013 asti (Microsoft 2013: part 1 – item 1A, 15). Tämä linja on jatkunut toistaiseksi ainakin vuoteen 2017, jolloin Microsoft on keskittämässä suuntautumistaan pilviteknologian lisäksi tekoälyyn aikaisemman mobiililaitteiden integraation sijaan (Microsoft 2017: part 1 item 1, 3). Näiden teknologiajättien käyttäytymisen perusteella on helppo havainnoida ohjelmiston ja teknologisen kehityksen trendejä, jotka viittaavat vahvasti hajautettujen ratkaisuiden laajempaan käyttöön.

Web-palvelinten hyödyntämien teknologioiden kehittymisen myötä myös perinteinen käyttäjä–palvelin (client–server) -arkkitehtuuri on syrjäytymässä pilvipalveluiden tarjoamien SaaS (software as a service) eli ohjelmisto palveluna -ratkaisuiden edeltä (Wainerwright 2014). Mitä paremmin käyttäjien laitteet ja selaimet tukevat uusia ominaisuuksia, sitä nopeammin perinteiset käyttöliittymät ovat muuttamassa verkkoon omiksi vuokrattaviksi palveluikseen. SaaS-ratkaisut tarjoavat käyttäjälle alustariippumattoman selainympäristön ja pienemmät asennuskulut. Web-sivustoiden toiminnallisuus ei ole riippuvainen käyttäjän käyttöjärjestelmästä, joten sovelluskehittäjien ei tarvitse kehittää useita eri työpöytäversioita, jolloin myös käyttäjä pystyy puolestaan käyttämään sovellusta kaikilla laitteillaan. Koska palvelu vuokrataan tai ostetaan käytettäväksi ulkoisilta palvelimilta, käyttävän tahon ei tarvitse asentaa sovellusta paikallisesti omalle laitteellensa. Varsinkin organisaatioiden kohdalla voidaan säästyä useille laitteille suoritettavilta asennuksilta ja asennuksen yhteydessä ilmeneviltä ongelmilta. Lisäksi sovellukset eivät vie asennukseen tarvittavaa levytilaa käyttäjän laitteelta.

Hajautetun arkkitehtuurin toteuttajalla on ratkaistavanaan kolme päähaastetta: palvelinten välinen kommunikaatio, ohjelmiston päivittäminen ja prosessien hajauttaminen. On tietysti huomioitava, että näiden ratkaisuiden integrointi mihin tahansa ohjelmistoon vaatii yksittäisellä laitteella toimivien järjestelmien kehittämiseen verrattuna paljon aikaa ja voi olla arkkitehtuurista riippuen hankala toteuttaa jälkikäteen. Myöhempää integraatiota voidaan helpottaa käyttämällä useita rajapintoja ohjelmiston rakenteessa.

7.1 Palvelinten välinen kommunikaatio

Hajautetun arkkitehtuurin toiminnan kannalta on olennaista, että arkkitehtuurin palvelimet kykenevät kommunikoimaan tehokkaasti keskenään. Tätä kommunikaatioita tarvitaan lähes kaikkien hajautettuun arkkitehtuurin ongelmien ratkaisussa, joten tehokkaiden kommunikaatiomenetelmien hyödyntäminen on olennaista esitellä ennen muita haasteita. Palvelinten tilan tarkkailu on erityisen tärkeää useita palvelimia sisältävässä arkkitehtuurissa, sillä ilman toimenpiteitä, yhden palvelimen virhetila voi aiheuttaa järjestelmän seisahtumisen. Ohjelman kriittisiä osioita ei tietenkään tule jättää yhden palvelimen varaan, mutta ilman liikenteen ja kuormituksen tarkkailua, palvelinten lähettämiä viestejä ei voida ohjata oikeille palvelimille. Kuten jonojen tapauksessa, myös hajautettujen prosessien seuraaminen käyttöliittymästä olisi mahdotonta ilman tarkkailua.

7.2 Kommunikaatiometodit

Palvelinten väliseen kommunikaatioon voidaan käyttää käyttötarpeiden mukaan useita eri protokollia. Ehkä yksinkertaisin kommunikaatiometodi voidaan tarjota HTTP API:lla. Lyhenne API tulee englannin kielen sanoista application programming interface ja kääntyy suomeksi ohjelmointirajapinnaksi. Yksinkertaisesti selitettynä ohjelmointirajapinnalla tarkoitetaan ohjelman määrittelemää ratkaisua, jolla toiset ohjelmat pystyvät kommunikoimaan rajapinnan tarjoavan ohjelman kanssa. Ohjelmointirajapinta määrittää, mitä ohjelman ominaisuuksia muut tahot voivat hyödyntää rajapinnan kautta. Rajapintojen on syytä kommunikoida sen käyttäjälle olennainen rajapinnan käyttöön liittyvä tieto. (Mikkonen 2005: 58)

Tavallista web-sivustoa voidaan pitää palvelimen käyttäjälle tarjoamana RESTiä hyödyntävänä rajapintana, sillä palvelimen määrittämä rajapinta on sivuston osoite, jonka perusteella palvelin vastaa käyttäjän pyyntöön. Samaa menetelmää käyttäen verkkosivusto voi tarjota useita eri toimintoja HTTP-rajapinnalla, joten HTTP API on web-ohjelmoijalle helposti ymmärrettävä käsite. Varsinaisen ohjelmointirajapinnan tarjoaminen eroaa usein HTTP-rajapinnasta vain rajapinnan palauttaman vastauksen tyypillä, joka on harvoin suunniteltu suoraan ihmisten luettavaksi. HTTP-kommunikaatiota hyödyntämällä eri palvelimet voivat toimia myös toistensa asiakkaina lähettämällä toisilleen pyyntöjä ja vastaamalla näihin pyyntöihin. HTTP APIa voidaan käyttää myös eri palvelimien prosessien valmistumisen tarkkailuun. Pyynnön lähettävä palvelin voi tarjota viestissään verkkokoukun (webhook), jonka tarkoituksena on suorittaa jokin toiminto toiselle palvelimelle lähetetyn pyynnön käsittelyn valmistuessa. Kaikessa yksinkertaisuudessaan verkkokoukku koostuu siis verkko-osoitteesta, eli linkistä, johon suorittava palvelin lähettää HTTP-pyynnön.

Jatkuvaan tilan tarkkailuun voidaan käyttää HTTP APIen tai muiden REST-rajapintojen tapauksessa pollingia. Tässä tapauksessa polling tarkoittaa palvelimen suorittamaa jatkuvaa ajoitettua palvelupyyntöjen lähettämistä toiselle palvelimelle. Polling voi toimia kummasta suunnasta tahansa: joko tarkkailtava palvelin lähettää tarkkailijalle tiedotteen tai tarkkailija kysyy tarkkailtavan palvelimen tilaa (kuva 11).

```
GET http://dbclient.dev/opdemo

 -- response --
200 OK
```

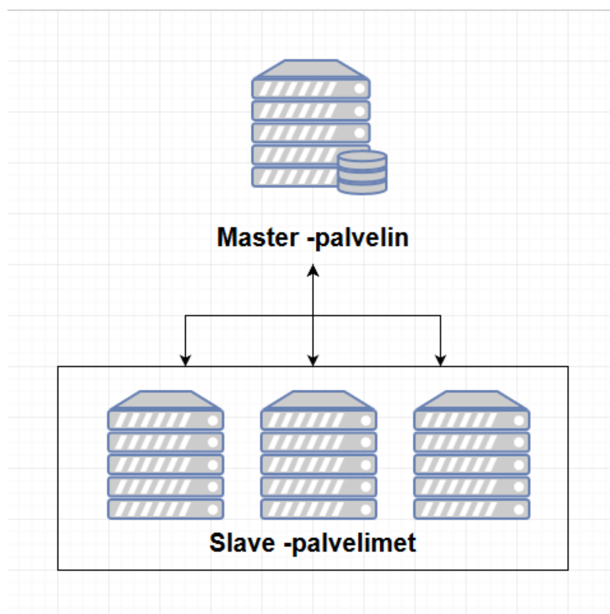
KUVA 11. HTTP-palvelupyyntö yksinkertaiseen APIin ja pyynnön vastaus

Tarkkailtavan palvelimen raportoidessa tilastaan kyseessä on sydänääni (heartbeat). Sykemittarin tavoin tarkkaileva palvelin huomaa, mikäli takkailtavan palvelimen pulssi eli jatkuvien viestien virta seisahtuu. Suurin ero näiden kahden menetelmän välillä on vastuu pyyntöjen lähettämisestä. Sydänääni voi olla hyödyllinen kriittisten prosessien tarkkailuun, mutta tarkkailun määrittäminen tarkkailevalta palvelimelta helpottaa prosessin kontrolloimista, sillä tarkkailijan ei tarvitse lähettää tai käsitellä pyyntöjä tarpeettomasta, vaan se voi kysyä toisten palvelimien tilaa tarpeidensa mukaisesti ja säästää näin ollen resursseja. Lisäksi vastuun jakaminen tarkkailijalle muodostaa yksittäisen paikan, josta tarkkailukriteereitä ja palvelimien tilaa on helpompi seurata tai muuttaa.

8 HAJAUTETUT VERKKOARKKITEHTUURIT

8.1 Master / slave

Yleisin ja ihmisten kannalta helpoin tapa palvelinten toiminnan tarkkailuun on kiistanalaisesti nimetty master/slave kommunikaatiomalli. Mallilla tarkoitetaan yhtä master eli pää- tai hallinnointipalvelinta, joka hallinnoi useita orjana (slave) toimivia alipalvelimia (kuvio 7).



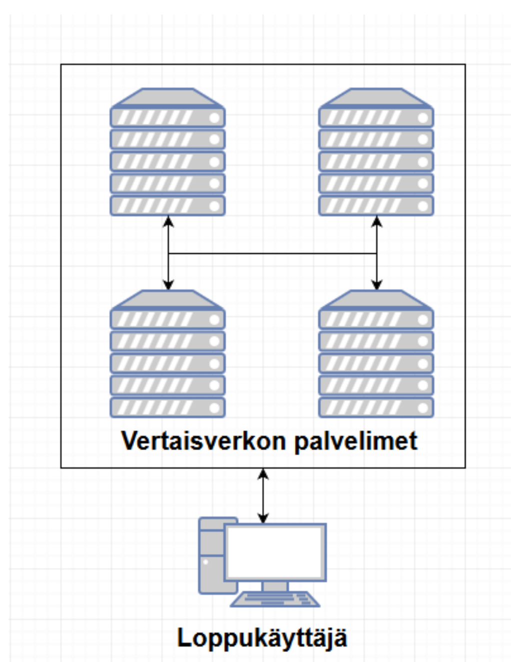
KUVIO 7. Master / slave -arkkitehtuurin topologia

Master/slave arkkitehtuuri tekee tarkkailusta ja hallinnoinnista helppoa, sillä verkkoa hallinnoiva käyttäjä pystyy hankkimaan kokonaisuuteen liittyvät tiedot yksittäiseltä pääpalvelimelta ilman tarvetta kerätä tietoja manuaalisesti useilta palvelimilta. Hallinnointipalvelimen tehtävänä on ohjeistaa orjapalvelimia toteuttamaan erinäisiä verkon sisäisiä tehtäviä. Vastaavasti alipalvelimet raportoivat kaiken tarvittavan tiedon pääpalvelimelle. Alipalvelimet voivat raportoida esimerkiksi tehtävien etenemisestä, aiheutuneista virheistä ja sen hetkisestä työtaakastaan. Näiden tietojen perusteella pääpalvelin kykenee jakamaan uusia tehtäviä tehokkaasti, tai jopa luomaan uusia virtuaalisia alipalvelimia dynaamisesti tarpeen mukaan.

8.2 Vertaisverkot ja hybridit

Vertaisverkko edustaa vaihtoehtoa perinteiselle asiakas–palvelin -arkkitehtuurille. Vertaisverkossa jokainen laite toimii sekä palvelimena että asiakkaana ja omaa teoriassa samat oikeudet kuin muutkin verkon laitteet. Jokainen laite voi pyytää ja kerätä tarvitsemansa tiedot muilta laitteilta, suorittaa verkon tehtäviä ja delegoida tehtäviä muille vertaisverkon laitteille. (Verma 2004: 7) Oikein toteutettuna vertaisverkon etuna on mahdollisten kriittisten virhepisteiden minimointi. Koska verkko ei ole varsinaisesti vastuussa ohjaavalle pääpalvelimelle, teoriassa jokainen verkon laite voi suorittaa perinteisesti pääpalvelimelle kuuluvia tehtäviä. Täten verkko ei ole riippuvainen minkään yksittäisen laitteen toiminasta. Vertaisverkossa laitteet voivat varmuuskopioida tarvittavat tiedot useille laitteille, jolloin tärkeä tieto on aina saatavilla yksittäisten laitteiden tilasta huolimatta.

Vertaisverkot eivät ole kuitenkaan usein käytännöllinen ratkaisu, sillä verkko vaatii osallistuvilta laitteilta käyttöoikeuksia, joka tekee ohjelmiston asentamisesta monimutkaista loppukäyttäjälle (Ycombinator 2014). Hybridiratkaisun tavoitteena on välttää verkon toimintaan liittyviä ongelmia sisällyttämällä verkkoon useita toisista laitteista eriarvoisia hallintalaitteita, jotka kykenevät hallintapalvelimien tavoin suorittamaan muille laitteille sopimattomia tehtäviä tai hallinnoimaan verkon toimintaan liittyvää, salaista tietoa. Vertaisverkon hybridi voi toimia myös palveluntarjoajan verkon sisällä, jolloin jokainen palvelinlaite on keskenään samanarvoinen, mutta ulkopuoliset laitteet eriarvoisia (kuvio 8).



KUVIO 8. Vertaisverkon hybridimalli palveluntarjoajan verkossa

Hybridiratkaisut hyötyvät vertaisverkon tarjoaman dynaamisemman laskentatehojen hajauttamisen lisäksi mahdollisuudesta kerätä ja suorittaa joitain raskaampia tai harvinaisempia verkon tehtäviä yksittäisillä tehtävää varten suunnitelluilla palvelimilla. Esimerkiksi verkon sisäistä tietoa keräävät palvelimet voivat näin ollen hyötyä master/slave -mallin tarjoamasta ihmisläheisemmästä hallinnasta ja samalla välttyä jakamasta kerättyä toisille laitteille tarpeetonta tietoa kaikkien verkon laitteiden kanssa. Tiedon keräämistä varten suunniteltujen palvelimien käyttäminen on kriittistä, sillä tällaiset palvelimet vaativat huomattavasti enemmän levytilaa kuin tavanomaiset verkon laitteet, eikä niiden sisältämän tiedon jakaminen useille palvelimille ole kustannustehokasta. Tiedon keräämiseen suunnitellut palvelimet ovat erityisen hyödyllisiä, kun tavoitteena on kerätä suuria määriä laitteiden tai verkon toimintaan liittyvää tietoa verkon laitteilta, joka ei ole välittömästi tarpeellista muille verkon laitteille.

8.3 Palvelukeskeinen arkkitehtuuri

SOA on lyhenne englannin kielen sanoista Service Oriented Architechture eli suomeksi palvelukeskeinen arkkitehtuuri. Palvelukeskeinen arkkitehtuuri koostuu nimensä mukaisesti useista erillisistä palveluista, joiden toimintoja yhdistelemällä voidaan muodostaa modulaarinen, useista osista koostuva kokonaisuus. Eri palveluita käytetään niiden tarjoamien ohjelmointirajapintoja avulla. Rajapintojen hyödyntäminen jokaisen palvelun kohdalla lisää arkkitehtuurin abstraktiota, jolloin applikaatio on vähemmän riippuvainen yksittäisistä järjestelmistä ja tekee vanhojen järjestelmien muuntelusta, uudelleen käytöstä ja korvaamisesta helpompaa (Bean 2010: 23). Esimerkiksi tietokantapalvelu voi määrittää rajapinnassa menetelmät tiedon hakuun, tallettamiseen ja poistamiseen. Rajapintaa käyttävän ohjelman ei tarvitse tietää, mitä tietokantaa palvelu käyttää, sillä palvelu toimii rajapinnan määritteiden mukaisesti tietokantatoteutuksesta riippumatta. Täten tietokannan toteutuksen muuttaminen ei vaadi palvelua ja rajapintaa hyödyntävien palveluiden muokkaamista.

Yksi palvelukeskeisen arkkitehtuurin hyödyistä on eri järjestelmien välinen toiminnallinen integrointi, jonka takia SOA on hyödyllinen malli useita laitteita sisältäviä verkkoja rakentaessa. SOA-verkkoon on helppo lisää myös ulkoisten tahojen palveluita. Arkkiteh-

tuurin painottama abstraktio lisää palvelukomponenttien uudelleenkäytettävyyttä ja nopeuttaa täten uusien järjestelmien kehittämistä. Lisäksi palveluihin jaettu järjestelmä helpottaa tehtävien jakamista ja valmiiden komponenttien yhdistämistä muuhun koodiin. Kokonaan modulaarisista komponenteista koostuvaa järjestelmää on helppo skaalata horisontaalisesti järjestelmän kasvaessa, koska eniten käytön aiheuttamaa rasitusta kokevia palveluita voidaan lisätä käyttötarpeiden mukaan.

8.4 Micropalvelut

Micropalvelut (microservices) ovat SOA:lle läheinen malli, jonka suurin ero SOA:an verrattuna on yksittäisten komponenttien koko. Micropalvelut-arkkitehtuurissa modulaarisuus on koko järjestelmän määrittävä tekijä: pienimmätkin ominaisuudet tulee jakaa omiksi komponenteikseen. Näiden palveluiden kehityksen alussa tarvittavien rajapintojen ohjelmoimiseen ja komponenttien rajaamiseen kuluva aika tekee Micropalveluista SOA-arkkitehtuuria radikaalimman ratkaisun, jossa yksittäisen komponentin on tarkoitus suorittaa vain yksittäinen toiminto, toisin kuin SOA-arkkitehtuurissa, jota voitiin kuvailla esimerkiksi tietokantapalveluna. Pienemmillä komponenteilla saavutetaan käytännössä samat edut kuin useita ominaisuuksia tarjoavilla komponenteilla, mutta niiden rakenteen hienojakoisuus tekee järjestelmästä vielä entistäkin joustavamman.

Useille monimutkaisille arkkitehtuureille tyypillisesti micropalveluidenkaan hyödyntäminen ei ole valitettavasti täysin ongelmatonta. Yksittäisen komponentin toimintaa on suhteellisen helppo testata, mutta useista osista koostuvan kokonaisuuden testaaminen on vaikeaa (Melo 2014). Lisäksi pienistä komponenteista koostuvan ohjelmiston yksittäisen osan voi olla vaikea päästä käsiksi tarvittaviin tietoihin, monimutkaistaen kehitystä entisestään. Ohjelmiston kasvun myötä omille palvelimilleen jaettujen komponenttien määrä kasvaa, jonka myötä ohjelmiston toiminta vaatii yhä enemmän komponenttien välistä kommunikaatiota (Fowler 2014). Järjestelmän palvelimien välisessä liikenteessä komponentit joutuvat muodostamaan ja käsittelemään useita palvelupyyntöjä, joka pakottaa järjestelmän käyttämään kevyitä viestiprotokollia.

9 TODENNUS JA TIETOTURVA LYHYESTI

9.1 Todennuspalvelinten käyttö

Useat SaaS-palvelut tarvitsevat käyttäjätilejä muun muassa laskuttamista ja sisäänkirjautumista varten. Useita palvelimia hyödyntävässä arkkitehtuurissa käyttäjien suorittamien maksujen ja käyttäjätilien valvominen voi muodostua vaivalloiseksi, mikäli käyttäjätiedot toistuvat tai hajautuvat useiden käytettävien palvelinten tietokantoihin. Maksu- ja käyttäjätietojen kahdentaminen on vaarallista mahdollisten synkronointiin liittyvien toteutusvirheiden takia, sillä tilanteen selvittämisestä tulee miltei mahdotonta, jos käyttäjän suorittamasta maksusta on ristiriitaista tietoa useilla palvelimilla. Lisäksi eri palvelinten ohjelmistojen versiot saattavat poiketa toisistaan kasvattaen virheiden mahdollisuutta.

Yksi mahdollinen ratkaisu näihin ongelmiin on käyttäjätilien hallintaan ja maksamiseen liittyvien ominaisuuksien sijoittaminen omalle, varsinaisesta SaaS-palvelimesta irtonaiselle palvelimelleen. Käyttäjätilien hallinnan vastuun siirtäminen omalle palvelimelleen vapauttaa levytilaa käyttöpalvelimilta ja helpottaa eri palvelinten ohjelmiston versiointia. Hinnoitteluun tehtävät muutokset on helpompi hallinnoida omalta palvelimeltaan kaikkien erillisten palvelimien päivittämiseen verrattuna. On myös mahdollista, että SaaS-palvelu tarjoaa käyttäjilleen yksityisiä palvelimia, jolloin verkkoon liittyvien tietojen sijoittaminen asiakkaan palvelimelle ei ole järkevää.

Käyttäjä voidaan aina velvoittaa rekisteröimään käyttäjätunnus jokaiseen palveluun, mutta useita erillisiä palveluita sisältävässä järjestelmässä yksittäisen organisaatiotunnuksen tai tilin käyttäminen on usein huomattavasti kätevämpää useiden erillisten käyttäjätilien luomiseen verrattuna. Esimerkiksi OAuth-protokolla on suunniteltu käyttöpalvelimesta erillisen todentamispalvelimen toteuttamiseen. OAuthia hyödyntävä käyttöpalvelin voi pyytää käyttäjää kirjautumaan sisään todennuspalvelun hallinnoimilla tunnuksilla tai tarpeen mukaan pyytää käyttäjän tilitietoja todennuspalvelimelta, jolloin todennuspalvelin voi jakaa käyttöpalvelimelle käyttäjän hyväksymiä tietoja paljastamatta käyttäjän salasanaa tai muita salaista tietoja käyttöpalvelimelle. (Bihis 2015)

9.2 Keskitettyjen todennuspalvelimien riskit

Tietoturvan kannalta keskitetyillä todennuspalvelimilla on sekä hyviä että huonoja puolia. Käyttäjätietojen tallettaminen yhteen palveluun pienentää mahdollisen tunkeilijan hyökkäyspinta-alaa, sillä käyttöpalvelin ei ikinä säilytä käyttäjien kriittisiä tilitietoja. Tunnistetietojen ja todennuksien hallinnointi erillisiltä palvelimilta helpottaa myös erilaisten varmenteiden toteuttamista. Esimerkiksi sähköposti-, laite- tai ip-varmenteita ei tarvitse ohjelmoida kuin todennuspalvelimille.

Valitettavasti yksi tietoturvan suurista haasteista on käyttökokemuksen ja turvallisuuden välinen vastakkainasettelu. Yhtä salasanaa kaikille palvelimelleen kirjautumiseen käyttävä asiakas riskeeraa kaikki palvelimensa salasanan kadotessa. Todennukseen käytettäviä palvelimia toteuttaessa on aina aiheellista ottaa huomioon verkon käyttäjien määrä ja sen aiheuttama rasitus. Kryptografiassa asymmetristen avaimien käyttäminen on avainten suuremman koon takia eksponentiaalisesti hitaampaa symmetrisiin verrattuna. RSA-avainpareja käyttävät palvelimet voivat pahimmassa tapauksessa muodostaa pullonkaulan. Asymmetrisia avainpareja käytetäänkin tavallisesti yhteyden muodostamisen alussa varsinaisesti yhteyden suojaamiseen käytettävän AES (tai muun symmetrisen algoritmin) -avaimen neuvotteluun, viestien eheyden säilyttämiseen ja lähettäjien varmentamiseen. Asymmetrisia avaimia voidaan käyttää myös palvelinten tunnistamiseen verkon sisällä, mutta salakuuntelun mahdollisuuden takia yhteyttä ei voida muodostaa turvallisesti ilman salattua HTTP eli HTTPS-yhteyttä.

9.3 Salasanojen turvallisuus verkkopalveluissa

Pitkien salasanojen käyttäminen vaikeuttaa salasanan murtamista, mutta niiden muistaminen on käyttäjällä vaikeaa. Tästä syystä jotkut tietoturvan asiantuntijat ovat suositelleet salasanojen korvaamista salalauseilla (passphrase) (welivesecurity 2016). Tarkoituksena on muodostaa pitkä, mutta samanaikaisesti helposti muistettava lause.

Salasanoja yritetään murtaa usein yleisimpien salasanojen listoja hyödyntäen muuntamalla joitain merkkejä erikoismerkkikriteereiden täyttämiseksi. Tämän päätelmän perusteella käyttäjää ei tulisi pakottaa muistamaan palvelun vaatimia ihmeellisiä merkkivaatimuksia, vaan salasanalistoja voidaan hyödyntää myös turvallisten salasanojen luomiseen.

Esimerkiksi rekisteröitymisen yhteydessä käyttäjän salasanaa voidaan verrata Levenšteinin etäisyydellä tai muulla vastaavalla merkkijonojen samankaltaisuutta analysoivalla algoritmillä yleisimmistä salasanoista koostuvaan listaan. Mikäli samankaltaisuus ylittää tietyn prosentin, voidaan käyttäjää vaatia laatimaan turvallisempi salasana. Tämän ratkaisun haittana on salasanoja murtavan tahon salasanalistan ohittamisella säästämä aika, mutta listan sisältämien salasanojen määrä on kuitenkin huomattavasti pienempi kuin muista mahdollisista salasanoista muodostuva sana-avaruus.

10 TAAKAN JAKAMINEN HAJAUTETUISSA VERKOISSA

10.1 Prosessien delegoiminen

Web-ohjelmiston käyttäjämäärän kasvaessa tarvittavat resurssit vaihtelevat sovelluksen käyttötarkoituksen mukaan, mutta järjestelmän hidastuminen on useissa tapauksissa ensimmäinen havaittava rajoite. Kasvaneen kuormituksen aiheuttama hidastus vaikuttaa negatiivisesti käyttökokemukseen ja saa täten hyvinkin suunnitellun ohjelmiston näyttämään huonolta käyttäjän silmissä. Tällaisissa tapauksissa hajautetun arkkitehtuurin hyödyt muodostuvat nimenomaan erillisten pyyntöjen käsittelyyn saatavilla olevien resurssien kasvamisesta uusien palvelinten määrän myötä.

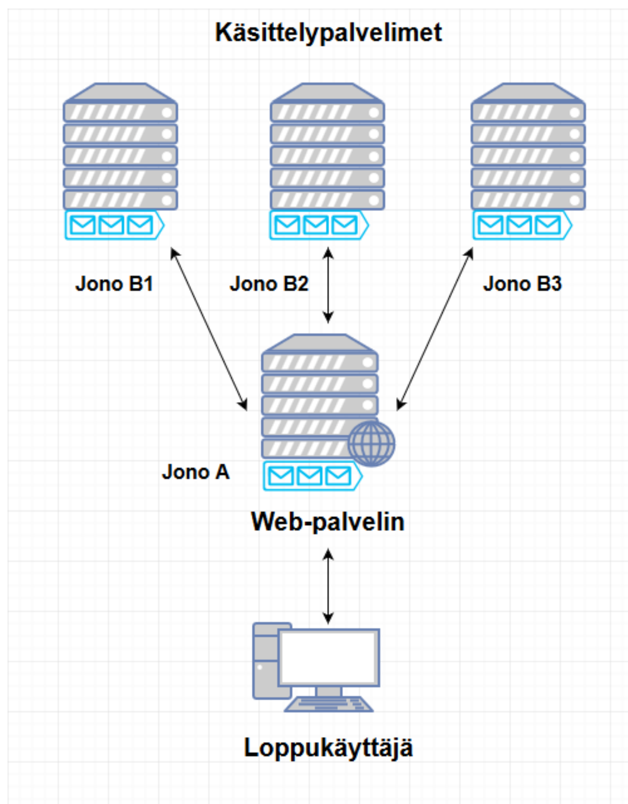
Verkkosivuja tarjoavien palvelimien kuormituksen jakaminen sivun lataajien sijainnin perusteella on yksi yksinkertaisimmista esimerkeistä taakanjaon hyödyntämisessä. Asiakkaan sijainnin avulla yksittäisen pyynnön käsittelyä voidaan nopeuttaa kahdella tavalla: lataamalla sivusto palvelimelta, joka sijaitsee fyysisesti lähempänä asiakasta (Amazon Web Services 2016), jolloin yksittäisen palvelimen ei myöskään tarvitse palvella kuin sille määritetyn alueen asiakkaita, jolloin yksittäiseen palvelimen kokema rasitus vähenee.

Verkkoarkkitehtuurin palvelimiin kohdistuvan rasituksen ja ohjelmiston suorittamien tehtävien jakamiseen on useita erilaisia menetelmiä. Pohjimmiltaan tehtävien delegoimisen haaste perustuu kuitenkin loogisten suoritettavien kokonaisuuksien määrittelyyn ja näiden kokonaisuuksien jakamiseen verkon palvelimille, jotka kykenevät suorittamaan kyseiset tehtävät.

10.2 Delegoiminen jonoilla

Jonoja koskevassa luvussa esiteltiin menetelmiä yksittäisten raskaiden operaatioiden pienempiin osiin jakamista varten. Erityisen suurien prosessien tapauksessa näitä menetelmiä voidaan käyttää myös horisontaalisen skaalaamisen puitteissa jakamalla luodut pienemmät prosessit useamman palvelimen käsiteltäväksi. Toisin sanoen tehtäviä muodos-

taessa ne jaetaan eri palvelimien käsittelijäjonoihin. Tarvittaessa myös tehtävänannot voidaan tallettaa jonoon, mikäli muodostettavien tehtävien määrä on niin suuri, ettei yksittäisten tehtävien muodostaminen yhdellä palvelimella kannata. Ohjelmoijan tulee kuitenkin huomioida, että kommunikaatioon ja tiedon lähettämiseen kuluvan ajan takia useiden palvelimien käyttäminen tehtäväjonojen prosessoimiseen ei välttämättä nopeuta kevyiden tehtävien prosessointia.



KUVIO 9. Delegoiminen jonoilla

Käytännössä tehtävänannon käyttäminen tehtävänä tarkoittaa prosessin entistä hienojakoisempaa käsittelyä jättämällä suuri osa käsittelystä muille verkon palvelimille. Kuvio 9 kuvastaa tätä prosessia. Esimerkiksi suurta tiedostoa käsitellessä tehtävänanto voidaan muodostaa laskemalla muille palvelimelle käsiteltäväksi lähetettävien palojen koko, tallentamalla palojen lähetys alkuperäisen palvelimen jonoon ja käyttämällä ulkoisten palvelinten rajapintoja palojen käsittelymenetelmien määrittelyyn. Käsittelyn lopuksi ulkoiset palvelimet voivat lähettää lopputuloksen takaisin alkuperäisellä palvelimelle verkkokoukkua tai muuta vastaavaa kommunikaatiomallia hyödyntäen.

10.3 Tietokantojen klusterointi

Klusteroiminen (clustering) on jokseenkin hämmentävä termi, sillä eri tietokannat käyttävät sitä eri rakenteiden kuvaamiseen. Tietokantaratkaisujen kanssa on oleellista testata klusteroitujen tietokantojen toimintaa käyttötarkoituksen mukaisessa tilanteessa ja verrata tuloksia klusteroimattomaan ratkaisuun todenmukaisten tulosten ja ratkaisun kannattavuuden mittaamiseksi.

Oraclen tietokannassa klusteroidessa relaatiolla yhteen kuuluvat tietueet yhdistetään yhdeksi klusteriksi relaatiossa käytetyn sarakkeen perusteella. Tämä klusterointi muistuttaa tavallaan oliotietokantojen nested tietorakenteita, sillä klusteri yhdistää toisiinsa kuuluvat tiedot helposti kerralla haettavaksi kokonaisuudeksi. Yksittäinen klusteri toimii teoriassa omana kokonaisuutenaan, joten yksittäisen osan virheet eivät vaikuta kuin yhteen tietokannan osioon. Klusterin sisäisten taulujen tietojen perusteella tehtävät haut nopeutuvat, sillä tietokannan ei tarvitse suorittaa lukuoperaatioita relaatioavaimien perusteella. (relationaldbdesign.com) Tämän ratkaisua toteuttaessa tulee löytää kriittinen piste, joka kertoo, kuinka paljon tietoa tulee säilöä tietokantaan ennen kuin klustereiden muodostaminen on kannattavaa.

Muissa toteutuksissa, kuten SQL server -tietokantapalvelimessa, klustereilla tarkoitetaan usein joko paikallisesti monistettua tietokantaa tai ratkaisua, jossa yksi tietokanta peilataan toisille tietokantapalvelimille. Näiden klustereiden tavoitteena on tarjota tietokannalle parempi virheensietokyky. Paikallisesti toteutetut klusterit joutuvat jakamaan keskenään yhden laitteen levytilan ja muistin. Näitä klustereita käyttäessä tietokantaa ohjaava pääprosessi tarkastaa, mistä klusterista haettava tieto löytyy. Paikallisissa klustereissa etsiminen aiheuttaa tämän haun takia ylimääräisen viiveen. (Bates 2009) Virhetilanteessa paikallisesti monistettu tietokanta voi siirtää toiminnan toisella prosessille ja peilattujen palvelimien tapauksessa klusteri siirtää vastuun tietokantapalveluista muille klusterin tietokantapalvelimille. Peilaamista käsitellään tarkemmin luvussa 11.4.

10.4 Tietokantojen sharding

Ohjelmiston keräämien tietomäärien kasvaessa yksitäisen palvelimen levytila ei enää riitä kaiken tiedon tallettamiseen ja/tai suuri tietomäärä alkaa hidastaa tiedon käsittelyä

hauissa ja muissa tietokantaan kohdistuvissa operaatioissa. Levytilan puute voidaan ratkaista skaalaamalla tietokantapalveluita horisontaalisesti useille eri tietokantapalvelimille. Horisontaalisen skaalaamisen mallista käytetään myös nimitystä sharding, joka viittaa applikaation tai verkon käyttämän tiedon jakamiseen usean tietokantapalvelimen kesken. Sharding hidastaa hakuja palvelinten välisen kommunikaatioviiveen takia, joten sen hyödyt ovat rajalliset, kun tietokannan sisältämän tiedon määrä on suhteellisen pieni (tietokannan rakenteesta ja käyttötapauksesta riippuen esimerkiksi 100 gigatavua).

Jotta erillisistä tietokannoista olisi sovelluksen nopeuden kannalta hyötyä, tulee ne jakaa tietokantaan suoritettavia hakuja tukevalla tavalla. Numeroita sisältävät rajalliset tietueet voidaan jakaa järjestyksellisesti arvoalueen perusteella: ensimmäinen tietokanta voi sisältää esimerkiksi kaikki rivit 1 - 100 000 000 väliltä, seuraava 100 000 001 - 200 000 000 ja niin edelleen. Merkkijonojen tapauksessa tieto voidaan jakaa esimerkiksi alkukirjaimien perusteella. Jos tieto jaettaisiin englannin kielen aakkosjärjestelmän perusteella neljään ryhmään, saataisiin merkeillä (A - F), (G - L), (M - R) ja (S - Z) alkavat ryhmät. Sharding:in hyöty perustuu aina ohjelmiston käyttötarkoitukseen. Jos tallennettavat merkkijonot olisivat englannin kielen sanoja, t-kirjaimella alkavia sanoja on arviolta 15.978% kaikista sanoista, kun z-kirjaimelle sama suhdanne on 0.045% (Norvig 2012). Täten kirjainten ryhmät olisi loogista muodostaa niin, että jokaisen tietokannan kirjainten yhteenlaskettu esiintyvyys vastaa muita ryhmiä. Oikeanlainen sharding auttaa tietokantoja säilyttämään keskenään samankaltaisen koon ja toiminnallisuuden ilman rasituksen aiheuttamia piikkejä.

Taulukko 4 paljastaa ennen pitkää suoraan aakkosellisten ryhmien perusteella muodostettujen tietokantojen välille syntyvän eron. Tilasto osoittaa, että jos tietokantojen sharding toteutettaisiin suoraan englanninkielisiä sanoja sisältävän avaimen perusteella, toinen tietotokanta sisältäisi ennen pitkää vain noin puolet ensimmäisen tai viimeisen tietokannan tietomäärästä. Jos sharding suoritettaisiin yhtä paljon levytilaa hyödyntäville palvelimille, ensimmäinen ja viimeinen tietokanta täyttyisivät noin kuusi prosenttiyksikköä nopeammin verrattuna tilanteeseen, jossa eri palvelimien tietokantojen sisältämä tieto jakautuisi optimaalisesti 25 -prosenttisesti kaikkien neljän klustereiden kesken.

TAULUKKO 4. Englanninkielisten sanojen jakautuminen alkukirjainryhmien perusteella jaettuihin tietokantoihin (Taulukko muodostettu Norvig 2012 tilastoiden perusteella)

Tietokanta	Yleisyys englanninkielisen sanan ensimmäisenä kirjaimena (% kaikista)
A - F	~31,354%
G - L	~16,518%
M - R	~21,107%
S - Z	~31,021%

Hyvän tosielämän esimerkin tiedon jakamisesta antaa mikroblogipalvelu Twitter. Twitterissä luotujen viestien määrä on arviolta noin 6000 sekunnissa (internetlivestats.com 2017 B). Tällä tahdilla palvelu joutuu tallentamaan vuodessa jopa 2 miljardia twiittiä. Talletettavien viestien määrä on niin massiivinen, että palvelu joutuu käyttämään useita muunneltuja palvelimia ja tietokantaratkaisuja (Hashemi 2017). Twitterin kaltaisessa palvelussa viestit voitaisiin jakaa horisontaalisesti useille palvelimille esimerkiksi viestien kirjoitusajankohdan perusteella. Täten palvelu voisi käyttää tehokkaimpia palvelimiaan uusien viestien tallentamista ja lukemista varten, sillä vanhoja viestejä luetaan huomattavasti uusia viestejä vähemmän, koska suurin osa ihmisistä näkee viestin sisällön vuorokauden kuluessa sen kirjoittamisesta. Jos tiedon säilyttäminen on tarpeellista, mutta sen varsinaisen käytön elinkaari rajoitettu, aikajanamainen sharding on tehokas ratkaisu. Massiivisessa skaalassa vanhentuvat tiedot voidaan tallettaa levytilaa ajatellen rakennetuille palvelimille, jotka eivät hakujen rajatun määrän vuoksi vaadi juurikaan keskusmuistia tiedon käsittelemiseen.

11 HAJAUTETUN VERKON YLLÄPITO

Pieniä verkkoja rakentaessa palvelinympäristöjä voidaan luoda ja testata vielä suhteellisen helposti manuaalisesti. Mikäli verkossa on paljon samaan tarkoitukseen suunnattuja, kehitettävää ohjelmistoa tarvitsevia rinnakkaisia palvelimia, voidaan tällaisten palvelinten ohjelmistoa päivittää verkon elinkaaren alussa useilla SSH eli secure shell -yhteyksillä. Multiplexing mahdollistaa saman komennon yhtäaikaisen suorittamisen useilla palvelimilla synkronoimalla eri istunnoille lähetettävät komennot. Mitä monimutkaisemmiksi verkot kasvavat, sitä hankalammaksi niiden ylläpito muodostuu. Vaikka manuaalinen toiminta on vielä aloitusvaiheessa mahdollista, pienempääkin ohjelmistokokonaisuutta testatessa automaation käyttäminen maksaa hyvinkin nopeasti takaisin sen kehittämiseen kuluvan ajan.

Tämän työn puitteissa rakennettu arkkitehtuuri sisältää toistaiseksi kolmeen eri tarkoitukseen rakennettuja palvelinohjelmistoja. Ensimmäinen näistä palvelintyypeistä tarjoaa asiakkaalle käyttöliittymä tiedonkäsittelyä varten. Toinen ohjelmisto keskittyy markkinointiviestien lähetystoiminnan tarkkailuun ja lähetysasetusten automaattisen päivittämiseen. Kolmas palvelintyyppi hoitaa käyttäjätietojen ja maksujen hallintaa aikaisemmin esitellyn OAuth-protokollan uudemman 2.0 -version mukaisesti. Jo näiden kolmen palvelutyypin jatkuva manuaalinen integroiminen palvelinympäristöön aiheutti kehittämisen viivästyksiä, joilta olisi voitu välttyä ottamalla prosessin automatisoiva järjestelmä käyttöön kehityksen alkuvaiheissa.

11.1 Automaation tarve

Manuaalisen päivityksen hankaluus riippuu pitkälti verkon palvelinten välisen yhteistyön määrästä. Varsinkin dynaamisesti skaalautuvassa verkossa eniten rasitusta kokevia osa-alueita hoitavia palvelimia lisätään tai vähennetään käyttötarpeen mukaisesti, joten verkon toiminnan automatisoiminen on käytännössä välttämättömyys. Jos yksittäisen palvelimen toiminnallisuus on tiukasti kytköksissä muiden palvelimien toimintaan, joudutaan uusia ominaisuuksia kehittäessä päivittämään jatkuvasti useiden eri palvelinten ohjelmistokoodia. Tämä on ongelma varsinkin SOA- tai micropalvelu -järjestelmien käyttäjille, sillä komponenttien määritysten eli ohjelmointirajapintojen muuttaminen vaatimusten mukaisiksi vaikuttaa myös useisiin palvelua käyttäviin komponentteihin.

Ketteriä kehitysmalleja noudattavissa projekteissa ei voida tietää etukäteen kaikkia ohjelmistolle kehitettäviä ominaisuuksia ja vaatimuksia, joten ohjelmoijat eivät voi keskittyä yksittäisen palvelimen toiminnan viimeistelyyn ennen kokonaiskuvan hahmottumista. Ennemmin tai myöhemmin ohjelmiston kehityksessä kohdataan uusia tarpeita, joten muutoksiin kannattaa varautua jo projektin alkuvaiheesta lähtien. Verkkoarkkitehtuuria voi monimutkaistaa myös tarve palvelinten virtualisointiin tai muiden palvelinympäristöä muuttavien teknologioiden hyödyntämiseen.

11.2 Ohjelmiston asentaminen ja päivittäminen

Ohjelmiston käyttöönotto on usein pelkkää versionhallinnasta kopioimista monimutkaisempaa. Palvelimille voidaan joutua asentamaan ohjelmiston lisäksi muita sen toiminnan kannalta pakollisia komponentteja. Web-ympäristöissä toimiva ohjelmisto käyttää lähes aina tietokantapalveluita, jotka tarvitsevat sekä omat tietokantansa että käyttäjätiedot näille tietokannoille. Tietokantojen käyttämisen automatisoimiseksi on välttämätöntä käyttää ohjelmallisesti tietokantarakenteet luovia skriptejä. Lisäksi ohjelmat voivat vaatia ympäristöön liittyviä lisätietoja toimiakseen oikein ja tehokkaasti. Esimerkiksi tarkasti optimoidulle ohjelmistolle voi olla tärkeää tietää, kuinka paljon muistia ja säikeitä sen on mahdollista hyödyntää. PHP ja muut ohjelmistot voivat vaatia oletusasetusten muokkaamista laitteen resurssien ja erilaisten ominaisuuksien, kuten tiedostojen lataamisen käyttämiseksi.

11.2.1 Staging palvelimet

Linux-palvelimilla ohjelmiston kansiot ja tiedostot vaativat oikeat käyttöoikeudet ja Apache2-palvelimelle saatetaan kansiorakenteesta riippuen joutua asettamaan htaccess direktiivejä. Näiden tekijöiden ja muiden todenmukaiseen asennukseen liittyvien prosessien testaamiseksi voidaan käyttää niin sanottuja staging-palvelimia. Staging-palvelimen tarkoituksena on asentaa ohjelmiston uusi versio käyttöympäristöä vastaavalle palvelimelle ohjelmiston toiminnan testaamiseksi. Ohjelmistoa kannattaa tietysti testata staging-palvelinten lisäksi myös oikeassa käyttöympäristössä mahdollisten palvelimien välisistä eroista muodostuvien ongelmien havaitsemiseksi. Mikäli ominaisuustestit (feature tests) aiheuttavat verkon toimintaan vaikuttavaa rasitusta, jonka voidaan odottaa häiritsevän

käytössä olevan ohjelmiston toimintaa, voidaan ohjelmistolle luoda erilaisia ympäristöstä riippuvia testaamislaajuuksia täydellisen testaamisvarmuuden kustannuksella.

11.2.2 Verkon toiminnan varmistaminen päivityksen aikana

Uutta ohjelmistoa asentaessa palvelulle aiheuta häiriö voidaan välttää minimoimalla varsinaiseen käyttöönottoon kuluva aika asentamalla vanha ja uusi ohjelmisto rinnakkain palvelimille. Uutta ohjelmistoa asentaessa vanhan ohjelmiston käyttöä jatketaan niin pitkään, kunnes uuden ohjelmiston asennus on valmis (zero downtime deployment). Tämän jälkeen palvelin asetetaan ohjaamaan kaikki palvelupyynnöt uuden ohjelmiston käsiteltäväksi ja vanhan ohjelmiston kansiot poistetaan palvelimelta. Versioita päivittäessä on tyypillistä merkitä ohjelmiston eri versioiden sisältämät kansiot uniikeilla tunnisteilla. Tunnisteena voidaan käyttää esimerkiksi versionhallinnan käyttämiä tunnisteita, kuten Git:in hashia.

11.3 Hallintapalvelinten tehtävät

Yksittäinen hallintapalvelin soveltuu hyvin automatisoituun testaamisen ja verkon palvelinten muokkaukseen. Hallintapalvelimen tarkoituksena on ylläpitää listaa verkkoon kuuluvista palvelimista ja antaa tarpeen mukaan mahdollisuus uusien palvelinten käyttöönottoon tai vanhojen poistamiseen. Palvelinympäristön muodostamisen yhteydessä uusille palvelimille voidaan tallettaa hallintapalvelimen sisältämät tarpeelliset tiedot, jotka koskevat uutta palvelinta.

Asennuksen yhteydessä suoritettavat toimenpiteet on helppo määritellä hallintapalvelimen skriptissä, joka voidaan suorittaa verkon palvelimella esimerkiksi SSH-etäyhteyden avulla. Skriptin suorittamisen eri vaiheiden välille voidaan määritellä verkkokoukkuja tai muita kommunikaatiomenetelmiä, joiden avulla voidaan tarkkailla asennuksen aikaisia virheitä. Useat webhotellit vuokraavat asiakkailleen jaettuja palvelimia eli samalla fyysisellä palvelinlaitteella sijaitsevia virtuaalisia palvelimia. Palveluntarjoajien virtualisaatioratkaisujen kehittyminen on johtanut entistä helpompaan uusien palvelimien käyttöönottoon, jota voidaan käyttää näiden palveluntarjoajien ohjelmointirajapintojen kautta oman hallintapalvelimen kehittämiseen.

11.4 Palvelinten peilaaminen

Verkon kriittisten osioiden jatkuva toiminta virhetilanteissa voidaan varmistaa käyttämälle palvelinten peilaamista (mirroring). Peilaamisen tarkoituksena on synkronoida kaksi tai useampi palvelin niin, että jokaisen palvelimen tiedot vastaavat toisiaan. Tällaiset palvelimet nimetään usein käyttötarkoituksen ja käyttöprioriteetin mukaan esimerkiksi autenktikaatio1 ja autentikaatio2 -palvelimiksi. Autentikaatio1-palvelin on tavallisesti vastuussa verkon käyttäjien tunnistamisesta ja sisäänkirjautumisista, mutta palvelimen kaatuessa verkko siirtyy ohjaamaan liikenteen seuraavalle vastuulliselle palvelimelle. Koko verkon käyttämien palvelinten peilaaminen on kriittistä verkon toiminnallisuuden varmistamiseksi ja toimii samanaikaisesti myös palvelimen tiedon varmuuskopiona.

Peilaamiseen käytettyjen ratkaisujen haittapuolet koostuvat tavallisesti harvoin hyödynnettävien palvelinten ylläpitämisestä ja synkronoinnin aiheuttamasta kuormituksesta. Toisin kuin rinnakkaisesti samaan tarkoitukseen käytettyjä palvelimia, tiedon eheyden takaamiseksi varalla pidettäviä peilattuja palvelimia ei hyödynnetä jatkuvassa käytössä. Synkronoinnin tulee olla mahdollisimman reaaliaikaista, jotta tärkeimpien tietojen säilymisestä voidaan olla varmoja. Tästä syystä synkronointi ei sovellu erityisen suurille tietokannoille, vaan toimii paremmin pienen tietomäärän varmentamisessa. Vähemmän kriittisten tai erityisten suurien tietomäärien varmentamiseen voidaan käyttää ajoitettuja tehtäviä, jotka siirtävät tarvittavat tiedot toiselle palvelimelle, josta ne voidaan ladata käyttöpalvelimen tilan palauttamiseksi virhetilanteen jälkeen. Suuria tietomääriä käsitellessä varmuuskopiointi ajoitetaan usein käyttäjien aikavyöhykkeen yölle tai muulle vastaavalle ajalle, jolloin sovelluksen kokema rasitus aiheuttaa käyttäjille mahdollisimman vähän haittaa.

11.5 Verkon toiminta pitkällä aikavälillä

Verkon toiminnan hidastuminen on todennäköistä pitkällä aikavälillä. Hidastumiseen varaudutaan suunnittelemalla valmiiksi odotettuja ongelmakohtia tarkkailevat järjestelmät. Toiminnan kannalta merkittävät tilastot ovat aina ohjelmistokohtaisia ja verkon toimintatarkoituksesta riippuvaisia. Yleistä tarkkailua voidaan kuitenkin suorittaa esimerkiksi

mittaamalla palvelinten välisten palvelupyyntöjen vastausaikoja. Resurssien rajallisuuden takia tarkkailu tulee priorisoida ensimmäisenä toiminnan kannalta kriittisiin pisteisiin, kuten todennuspalvelimiin. Todennuspalvelimelle lähetettyjen sisäänkirjautumispyyntöjen käsittelyviiveen keskiarvoa ja pisintä käsittelyaikaa voidaan käyttää muodostamaan yleisluontoinen kuva järjestelmän toiminnasta ja asiakkaan käyttökokemuksesta. Näiden arvojen kasvaessa ohjelmiston toiminnan tutkiminen tai skaalaaminen on aiheellista.

Useiden monimutkaisten verkkoarkkitehtuurien tarvitsee tallettaa käytön aikaista tietoa. Varsinkin kehityksen alkuvaiheessa kannattaa varautua palvelimille muodostuviin irtotiedostoihin, sillä koodivirhe voi keskeyttää ohjelman toiminnan ennen väliaikaisten tiedostojen poistamista. Ohjelmiston käyttämien tiedostojen määrän tai niiden käyttämän levytilan laskeminen auttaa kehittäjää varautumaan ja havaitsemaan mahdollisten irtotiedostojen muodostumisen, sekä tarkkailemaan levytilan täyttymistä käytössä. Käytön myötä paljon logi- tai väliaikaistiedostoja kerääville ohjelmistoille kannattaa suunnitella tietyllä aikaviiveellä vanhoja tiedostoja poistava tehtävä.

12 JOHTOPÄÄTÖKSET JA POHDINTA

Useat skaalaamiseen käytettävät tekniikat havaittiin ylimitoitetuiksi Coon-Tech Oy:lle tuotettuja palveluita varten. Varsinkin monet horisontaaliset ratkaisut olisivat hidastaneet ohjelmiston kehittämistä ja joissain tapauksissa jopa toimintaa palvelinten välisten viestien muodostamiseen, lähettämiseen ja purkamiseen kuluvat ajan takia. Hajautetut palvelinratkaisut tekisivät järjestelmästä skaalautuvamman ja varmemman, mutta markkinointijärjestelmien tapauksessa järjestelmän jatkuva toimivuus ei ole samalla tavoin kriittistä, kuin esimerkiksi pankki- tai maksujärjestelmissä. Skaalautuvuuden parantamiseksi ei ole vielä toistaiseksi kannattavaa käyttää lisää resursseja ennen todellista tarvetta.

Asiakastietojen turvaamiseksi järjestelmän tietoturvaa parannettiin erilaisilla varmenteilla. Asiakastietojen säilyttämien ei vaatinut myöskään peilaamista palvelimien välillä, sillä järjestelmän käyttönopeus oli täysin ajantasaisia varmuuskopioita tärkeämpää. Tästä syystä ajoitetut varmuuskopiot ulkoiselle palvelimelle olivat riittävä ratkaisu asiakastietojen ja järjestelmän asetusten turvaamiseksi. Asiakaslistoista säilytetään myös alkuperäiset tiedostot, joten pahimmassa tapauksessa vain muutaman tiedoston sisältö jouduttaisiin lataamaan uudestaan järjestelmään.

Tilan ja ajan välinen vaihtosuhdanne osoittautui puolestaan erittäin käytännönläheiseksi teoriaksi, varsinkin tietokantaan kohdistuvia operaatioita ohjelmoidessa. MySQL-tietokannassa testattujen operaatioiden nopeuden vertailu paljasti, että tietokannan suorittamien komentojen koolla ja määrällä on väliä suoritusnopeuden optimoinnissa. Yksittäisten operaatioiden koon kasvaessa myös niiden suorittamiseen kuluva aika kasvaa eksponentiaalisesti. Suoritusnopeuksia tarkkaillessa tulee ottaa kuitenkin huomioon erilaisten ohjelmistojen väliset erot, sillä parhaiten toimivat operaatiokoot ovat riippuvaisia useista tekijöistä. Ohjelmiston suorituskykyä parannellessa on kannattavaa luoda erilaisia toimintoja ja ominaisuuksia varten kehitettyjä testejä, joiden perusteella voidaan päätellä, minkä kokoisten tietokantaoperaatioiden suorittaminen johtaa parhaaseen käsittelynopeuteen. Tilan ja ajan välisen vaihtosuhdanteen teoriaa on mahdollista soveltaa myös tietorakenteiden perusteella käytettävien nopeampien tiedonkäsittelymenetelmien etsimiseen.

Oliotietokantojen mainostettu nopeus oli pettymys. Jotkin oliotietokannat, kuten MongoDB, vaativat käyttöönsä paljon enemmän keskusmuistia. Keskusmuistia varaamalla

tietokanta pystyy teoriassa toimimaan nopeammin, mutta tietomäärän kasvaessa keskusmuistissa säilytettävä tietomäärä kutistuu nopeasti erittäin pieneksi suhteessa käsiteltävään tietomäärään, jolloin MongoDB joutuu jatkuvasti lataamaan uutta tietoa keskusmuistiin levyltä. Ohjelma pystyy lataamaan tiedon nopeasti oliotietokannan varaamasta keskusmuistista, mutta itse keskusmuistiin ladattava tieto rajoittuu silti varsinaisen levyn lukunopeuteen. Mitä enemmän tietoa joudutaan lataamaan varattuun keskusmuistiin suoraan levyltä, sitä vähemmän hyötyä varatusta keskusmuistista on käsittelynopeudelle. Loppujen lopuksi oliotietokannan tuomat edut katsottiin skaalaamisen kannalta jokseenkin pieniksi mahdollisiin haittapuoliin, kuten ACID-mallin puutteeseen ja moninkertaiseen levytilan kulutukseen verrattuna. Kuten oliotietokantoja koskevassa luvussa pääteltiin, varsinaiset nopeudelliset edut saattaisivat jäädä hyvinkin pieniksi ilman erityisesti oliotietokannan nopeutta varten suunniteltuja tietorakenteita.

Käytännön tulosten perusteella voidaan päätellä, että jonot ovat yksi yleisesti hyödynnettävimmistä teknologioista. Jonoja voidaan käyttää niin käyttöliittymien nopeuttamiseen kuin monimutkaisten tehtävien suorittamiseen ilman tarvetta yksittäisille, pitkäaikaisille prosesseille. Jonoja käyttämällä on siis mahdollista nopeuttaa pienten applikaatioiden toimintaa, mahdollistaa suurempien kokonaisuuksien prosessoimisen keskikokoisissa ohjelmistoissa sekä delegoida tehtäviä suuressa, SOA-arkkitehtuuria hyödyntävässä ratkaisussa. Lisäksi uudet ohjelmistokehykset ovat tehneet jonoja hyödyntävän arkkitehtuurin kehittämisestä entistä helpompaa. Tarpeen mukaan jonot voidaan toteuttaa monilla erilaisilla teknologioilla, kuten keskusmuistia hyödyntävillä tietokannoilla nopeuden parantamiseksi tai tavanomaisemmilla relaatiotietokannoilla, kun yksittäisen tehtävän täytyy sisältää paljon tietoa. Nämä ominaisuudet tekevät jonoista yhden joustavimmista ja samalla tärkeimmistä skaalaamiseen käytettävistä menetelmistä web-ympäristössä.

Ohjelmiston kehityksen aikana jonoja jouduttiin käyttämään ohjelmiston asetuksiin liittyvien rajoitteiden takia, mutta saavutetut hyödyt tekivät teknologiasta heti toteuttamiseen liittyvien haasteiden arvoisen ratkaisun. Suurten tiedostojen käsittely ei ollut alun perin mahdollista, sillä käsittelemiseen kuluva aika olisi huomattavasti isompi, kuin mikä olisi järkevä määrittää suoritukseen käytettävän ajan ylärajaksi. Lisäksi joillain käsittelyyn käytetyillä ohjelmistokomponenteilla oli tapana ladata koko tiedoston sisältö kerralla käsittelyä varten, jolloin myös sallitun muistinkäytön raja rikottiin. Nämä ongelmat ratkaistiin jakamalla suuri tiedosto pienempiin osiin latauksen yhteydessä ja luomalla rajapinta, jonka avulla useat tiedoston palaset voidaan käsitellä yksittäisenä tiedostona ohjelmiston

koodissa. Tiedostoa käsitellessä voitiin täten luoda yksittäinen tehtävä jokaiselle tiedoston palalla, jolloin yksittäisen tehtävän koko pysyi helposti asetusten määrittämien rajojen sisällä. Lisäksi pienet, toisistaan riippumattomat tehtävät mahdollistavat rinnakkaisen prosessoinnin monisäikeisyydellä.

Kaiken kaikkiaan ohjelmiston kehittäjälle skaalaamisen suurimmaksi haasteeksi muodostuu aina kysymys: milloin skaalaaminen on tarpeellista? Tässä työssä esiteltyä markkinointijärjestelmää kehittäessä yksi suurimmista eduista oli se, että yrityksen tarpeet ja järjestelmän toteuttamiseen liittyvät haasteet olivat jo tiedossa aikaisemman prototyypin testaamisen takia. Skaalaamiseen ja optimointiin tottumattoman ohjelmoijan kannalta näitä haasteita on vaikea havaita pelkän teorian perusteella, joten jotkin skaalaamiseen liittyvät haasteet tulee valitettavasti oppia kokemuksen kautta. Kehittäjän tulee varautua yhtäaikaisesti skaalaamiseen liittyviin haasteisiin, mutta välttää myös ylimääräisen työn tekemistä ja ylikehittämistä (over engineering). Ylikehittämisen välttämisen kannalta skaalaaminen on haaste, jonka ratkaiseminen on helpompaa ketteriä kehitysmalleja käyttäessä. Laaja katsaus erilaisiin skaalaamismenetelmiin on antanut Coon-Tech Oy:lle mahdollisuuden varautua tulevaisuuden haasteisiin ja harkita uusien ratkaisujen käyttöönottoa, kun siitä tulee ajankohtaista.

LÄHTEET

Bates, G. 2009. Advantages And Disadvantages of Clustering SQL Server. Julkaistu 16.1.2009. Luettu 19.11.2017. http://www.sql-server-performance.com/2009/advantages-and-disadvantages-of-clustering-sql-server/

Bean, J. 2010. SOA and Web Services Interface Design. Morgan Kaufmann Publishers. ISBN: 978-0-12-374891-1

Beaumont, D. 2014. How to explain vertical and horizontal scaling in the cloud. Luettu 19.11.2017. https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/

Bihis, C. 2015. Mastering OAuth 2.0. Packt Publishing. ISBN: 978-1-78439-230-7

Cadenhead, T. 2015. Socket.IO Cookbook. Packt Publishing. ISBN: 9781785880865

Elmasri R. & Navathe S. 2011. Database Systems. Pearson. ISBN: 978-0-13-214498-8

Farrugia, C. 2012. MySQL vs. MongoDB Disk Space Usage. Luettu 19.7.2017. https://www.revulytics.com/blog/mysql-vs-mongodb-disk-space-usage

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.

Fowler, M. 2014. Microservices. Julkaistu 25.3.2017. Luettu 19.11.2017. https://martinfowler.com/articles/microservices.html

Goralski, W. 2009. The Illustrated Network : How TCP/IP Works in a Modern Network. Morgan Kaufmann. Amsterdam. ISBN: 9780080923222

Hashemi, M. The Infrastructure Behind Twitter: Scale. Julkaistu 19.1.2017. Luettu 19.11.2017. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

internetlivestats.com 2017 A. Google Searches In 1 Second. Luettu 19.7.2017. http://www.internetlivestats.com/one-second/#google-band

internetlivestats.com 2017 B. Twitter Usage Statistics. Luettu 19.7.2017. http://www.internetlivestats.com/twitter-statistics/#trend

Jollymore, W. Using MySQL Databases. Luettu 21.11.2017. http://www-acad.sheridanc.on.ca/~jollymor/syst28043/mysql1.html

Kaushal, N. 2015. High level languages vs Low level languages (Infographics). Luettu 19.11.2017. https://www.educba.com/high-level-languages-vs-low-level-languages/

Knuth, D. 1997. The Art of Computer Programming, Volume 1: Fundamental Algorithms, Third Edition. Addison-Wesley. ISBN: 0-201-89683-4

Laravel 2017. Documentation (5.5): Queues. Luettu 19.11.2017. https://laravel.com/docs/5.5/queues

Liu, H. 2009. Software Performance and Scalability: A Quantitative Approach. John Wiley & Sons, Inc. New Jersey. ISBN: 978-0-470-46253-9

Melo, F. 2014. Developing Microservices for PaaS with Spring and Cloud Foundry. Julkaistu 16.10.2014. Luettu 19.11.2017. https://www.infoq.com/presentations/microservices-pass-spring-cloud-foundry

Microsoft 2013. Annual Report Pursuant To Section 13 Or 15(D) Of The Securities Exchange Act Of 1934, For the Fiscal Year Ended June 30, 2013. Luettu 20.11.2017. https://www.sec.gov/Archives/egar/data/789019/000119312513310206/d527745d10k.htm

Microsoft 2017. Annual Report Pursuant To Section 13 Or 15(D) Of The Securities Exchange Act Of 1934, For the Fiscal Year Ended June 30, 2017. Luettu 20.11.2017. https://www.sec.gov/Archives/egar/data/789019/000119312513310206/d527745d10k.htm

Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Talentum. Jyväskylä. ISBN: 952-14-0862-6

MongoDB 2017 A. Documentation 3.4. Luettu 19.7.2017. https://docs.mongodb.com/manual/core/data-model-design/#data-modeling-referencing

MongoDB 2017 B. Advantages Of NoSQL. Luettu 19.7.2017. https://www.mongodb.com/scale/advantages-of-nosql

Norvig, P. 2012. English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU. Luettu 19.11.2017. http://norvig.com/mayzner.html

pc-freak.net 2014. What is Vertical scaling and Horizontal scaling – Vertical and Horizontal hardware / services scaling. Luettu 19.11.2017. http://www.pc-freak.net/blog/vertical-horizontal-server-services-scaling-vertical-horizontal-hardware-scaling/

php.net manual 2017. Resource Limits. Luettu 19.11.2017 http://php.net/manual/en/ini.core.php

Relationaldbdesign.com. Advantages of Clustered Tables in Oracle. Luettu 19.11.2017. https://www.relationaldbdesign.com/extended-database-features/module3/clustered-tables-advantages.php

Savage, J. Models Of Computation Exploring the Power of Computing. Addison-Wesley. ISBN: 978-0201895391

Verma, D. 2004. Legitimate Applications of Peer-to-Peer Networks. John Wiley & Sons, Inc. ISBN: 9780471653790

Wainerwright, P. From client-server to cloud: SaaS evolution. Luettu 19.11.2017. https://diginomica.com/2014/01/10/client-server-cloud-saas-evolution/

Welivesecurity 2016. Forget about passwords: You need a passphrase! Julkaistu 5.5.2016. Luettu 19.11.2017. https://www.welivesecurity.com/2016/05/05/forget-about-passwords-you-need-a-passphrase/

Ycombinator 2014. Why don't more apps use peer to peer networking? Julkaistu 14.8.2014. Luettu 19.11.2017. https://news.ycombinator.com/item?id=8175453