Markus Iivonen

Skaalautuva verkkosovelluskehitys

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

26.11.2014

Tekijä(t) Otsikko Sivumäärä Aika	Markus Iivonen Skaalautuva verkkosovelluskehitys 59 sivua 26.11.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Vesa Ollikainen Professori Huaxia Rui

Tämän opinnäytetyön tavoitteina on TwitterSensor-palvelun kehittäminen vasteaikojen lyhentämiseksi ja suurten datamäärien hyödyntämiseksi. Lisäksi tavoitteena on teknologioiden arviointi ja mittaus suorituskyvyn parantamisen näkökulmasta. Työssä tutustuttiin asiakasohjelman ja palvelinpuolen teknologioihin, jotka mahdollistavat skaalautuvien sovellusten kehityksen. Työssä haluttiin tutkia mahdollisuutta vähentää palvelinpuolelle kohdistuvan kuormaa siirtämällä osa sovelluslogiikasta asiakasohjelman suoritettavaksi.

Työssä tutustuttiin vaihtoehtoisiin tietokantaratkaisuihin, jotka mahdollistavat erilaisen tietokantarakenteen käytön mahdollistaen suuremman ja monimuotoisemman datan käytön. Lisäksi työssä tutustuttiin Hadoop-sovelluskehykseen, joka on tarkoitettu Big Data -ratkaisujen monipuoliseen toteutukseen.

Perinteisissä verkkosovelluksissa asiakasohjelman käyttökokemusta saattaa häiritä latausajat jotka johtuvat palvelinpuolen suuresta kuormituksesta. Lisäksi perinteisessä verkkosovelluksessa jokainen käyttäjän aiheuttama toiminto aiheuttaa sivun uudelleen latauksen. Kaksi edellä mainittua asiaa yhdessä aiheuttavat viivettä asiakasohjelman toiminnassa ja asiakasohjelman käyttökokemus kärsii. Single-Page-Application (SPA) -sovelluskehitysmalli pyrkii vähentämään palvelimelle aiheutuvien kutsujen määrää ja turhia sivulatauksia.

Tutkittujen ja testattujen teknologioiden pohjalta lähdettiin toteuttamaan TwitterSensor-palvelusta uutta versiota, jossa hyödynnettiin toimivaksi todettuja teknologioita. TwitterSensor toteutettiin Single-Page-Applicationina, jossa käytettiin niin sanottua MEAN-pinoa jolloin sovellus toteutettiin JavaScript-ohjelmointikielellä. Asiakasohjelma toteutettiin AngularJS-sovelluskehyksellä ja palvelinpuolen toteutukseen käytettiin Node.js-sovelluskehystä. Tietokantatoteutukseen käytettiin MySQL-relaatiotietokantaa sekä MongoDB-NoSQL-tietokantaa.

Avainsanat	Verkkosovellus, Big Data, Hadoop, Single-Page-App, AngularJS, Node.js, MySQL, MongoDB, JavaScript, TwitterSensor

Author(s) Title Number of Pages Date	Markus Iivonen Scalable web-application development 59 26 November 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Vesa Ollikainen, Lecturer Huaxia Rui Professor

The goals of this Bachelor's Thesis were to improve a TwitterSensor -application by decreasing the response time and enabling larger usage of data. This Bachelor's Thesis also focuses on technologies where the goal is to improve application performance. One way to achieve better performance is to reduce the load on server and move part of the logic to the client side.

One area of this Bachelor's Thesis deals with different database technologies, which allow more flexible database structure. Using different database solutions allows usage of larger data volumes and more complex queries to the database. One other area to cover is the Hadoop -framework, which is a solution for dealing with Big Data.

User experience of a basic web-application may suffer because of several different reasons. A basic web application creates a request to the server each time the user does something on the site, which causes the site to reload. Usually, web applications fetch data from a database and in some cases heavy database queries cause delays or stop the web application completely while the server is handling the request. SinglePageApplication (SPA) tries to deal with this kind of a problem and it reduces requests to the server as much as possible.

After researching and testing new technologies to build web applications, a new version of the TwitterSensor -platform was started to be built. TwitterSensor was implemented as a SinglePageApplication by using MEAN -stack. Thus all application layers are written in JavaScript. The client side application was built by using AngularJS-framework and the server side used Node.js framework. The database layer was implemented by using MySQL relation database and MongoDB-NoSQL -database.

Keywords	Web-application, Big Data, Hadoop, Single-Page-App, AngularJS, Node.js, MySQL, MongoDB, JavaScript, TwitterSensor

Sisällys

Lyhenteet

API	Application Programming Interface. Ohjelmointirajapinta jonka kautta ohjelmat pystyvät vaihtamaan tietoa keskenään.
AJAX	Asynchronous JavaScript And XML. Joukko verkkosovelluskehityksen tekniikoita joilla saadaan sivuista vuorovaikutteisempia.
CDN	Content Delivery Network. Kolmannen osapuolen tarjoama palvelu josta saadaan haettua nopeasti tietoa.
CSS	Cascading Style Sheets. Kuvauskieli jolla kerrotaan miten dokumentit voidaan esittää.
DOM	Document Object Model. Ohjelmointirajapinta joka mahdollistaa HTML-dokumenttien muokkauksen JavaScript-ohjelmointikielellä.
GFS	Google File System. Googlen kehittämä hajautettu tiedostojärjestelmä.
HDFS	Hadoop Distributed File System. Hadoop-sovelluskehyksen hajautettu tiedostojärjestelmä.
HTTP	Hypertext Transfer Protocol. Protokolla jota selaimet ja palvelimet käyttävät tiedonsiirtoon.
MEAN	Yhdistelmä MongoDB- Express-, AngularJS- ja Node.js-sovelluskehyksistä
REST	Representational State Transfer. HTTP-protokollaan perustuva arkkitehtuurimalli.
SPA	Single-Page-Application. Sovellus, jonka toiminta on toteutettu yhdellä sivulla.
URL	Uniform Resource Locator. HTTP-protokollan kanssa käytettynä merkeistä muodostuva osoite josta tietoa voidaan hakea.

1 Johdanto

Maailmassa syntyy koko ajan uutta tietoa. Tiedon määrän kasvuun on vaikuttanut teknologian kehitys ja lisääntynyt laitteiden kuten älypuhelinten ja tablettien määrä sekä lisääntynyt palveluiden määrä. Tietoa tuottavat esimerkiksi sosiaalisen median palvelut kuten Facebook ja Twitter, joissa jokaista käyttäjän julkaisemaa tilapäivitystä tai viestiä voidaan pitää osana suurempaa tietomassaa. Vastaavasti päivittäin käyttämämme palvelut, esimerkiksi kauppaketjut keräävät asiakkaistaan tietoa. [1.] Tiedon määrän kasvaessa siinä nähdään uusia yritysmahdollisuuksia sekä mahdollisuuksia parantaa jo olemassa olevia palveluita. Ei ole yllättävää, että suurin osa Big Dataan liittyvistä palveluista ja työkaluista tulevat yrityksiltä, kuten Amazonilta, Facebookilta, Googlelta ja Yahoolta, jotka ovat onnistuneesti pystyneet hyödyntämään ja kaupallistamaan heillä käytössä olevaa dataa. [2, s. 3.]

Oikein käytettynä suuret tietomäärät voivat tarjota yrityksille arvokasta tietoa asiakkaista tai mahdollistaa asiakkaille paremmin ja henkilökohtaisemmin suunnattujen palveluiden tuottamisen. Kohdennetusta mainonnasta esimerkkinä on Facebook, joka hyödyntää käyttäjistä kerättyä tietoa tarjoamalla kohdistettuja mainoksia asiakkaille. Facebook hyödyntää myös käyttäjistä kerättyä tietoa näyttäen uutisvirrassa sellaisten henkilöiden päivityksiä joiden kanssa kyseinen käyttäjä on useasti tekemisissä. Toisena voidaan mainita Twitter, joka suuntaa käyttäjille mainontaa perustuen heidän aikaisempiin viesteihin. Esimerkkinä voisi mainita tilanteen, jossa päivitetään Twitteriin viesti. Viestissä kehutaan suosikkiyhtyeen uutta levyä. Samalla he saattavat mainostaa, jos kyseinen yhtye on tulossa lähialueelle esiintymään. [3.]

Suuret tietomäärät aiheuttavat myös haasteita. Vaikka yrityksillä on pääsy suureen määrän erilaista tietoa, niitä ei osata hyödyntää oikein, tai tiedon hyödyntäminen kustannustehokkaasti on vaikeaa. IBM:n teettämän kyselyn mukaan puolet yritysten johtohenkilöistä ymmärtävät, etteivät he pysty hyödyntämään tietoa tavalla, joka parantaisi työn tulosta. [2, s. 3.]

Kasvavat tietomäärät eivät aiheuta ongelmia pelkästään yrityksen taustajärjestelmien tasolla. Palveluita toteuttaessa on huomioitava, miten asiakasohjelma kykenee käsittelemään palvelimelta saatua tietoa ja tarjoamaan sen käyttäjälle verrattain nopeassa ajassa ilman käyttökokemuksen kärsimistä. Sivustojen vasteajalla eli sillä, miten nopeasti käyttäjän toimenpiteisiin pystytään vastaamaan, on todettu olevan yritysten kannal-

ta taloudellisia vaikutuksia. Googlen ja Microsoftin yhteistyössä tekemän tutkimuksen mukaan 2,0 sekunnin viive Bing-hakukoneessa vähentää tuloja 4,3 % asiakasta kohden. Samassa tutkimuksella Google havaitsi 4,0 sekunnin viiveen vähentävän yhden käyttäjän suorittamien hakujen määrää 0,59 % päivässä. [4.]

Työn tavoitteina on TwitterSensor-palvelun kehittäminen vasteaikojen lyhentämiseksi ja suurten datamäärien hyödyntämiseksi. Lisäksi tavoitteena on teknologioiden arviointi ja mittaus suorituskyvyn parantamisen näkökulmasta.

TwitterSensor on tutkimusalusta jonka perustaja on professori Huaxia Rui (Simon Business School at University of Rochester) [5.]. Tutkimusalusta on tarkoitettu helpottamaan Big Dataan liittyvää tutkimustyötä, joka perustuu mikroblogipalvelu Twitteristä kerättyyn dataan.

Tutkimusalustan kehittäminen kattaa kaksi erillistä osa-aluetta, joista ensimmäinen on kehittää palvelun verkkosovellusta, mikä tarkoittaa lyhyempää vasteaikaa asiakasohjelman ja palvelimen välisessä kommunikoinnissa. Toinen osa-alue on kehittää palvelun taustajärjestelmää, mikä mahdollistaa laajemman tietomäärän käyttöönottoa, tiedon reaaliaikaista tallentamista Twitter-palvelusta, reaaliaikaisia tietokantakyselyitä sekä syvempää tiedon analysointia siihen soveltuvilla työmenetelmillä.

Työssä tutkitaan käytettävissä olevia teknologioita, jotka mahdollistavat ylempänä mainittujen pidemmän aikavälin tavoitteiden saavuttamisen. Työssä tutkitaan ja kokeillaan millaisia parannuksia asiakasohjelmaan voidaan saada uudemmilla verkkosovellusteknologioilla, tavoitteena vähentää käyttäjän kokemaa viivettä. Lisäksi tutkitaan ja testataan millaisia vaihtoehtoisia palvelinsovellusteknologioita on saatavilla ja millaisia hyötyjä palvelintoteutusta vaihtamalla voidaan saavuttaa. Työssä myös tutkitaan ja testataan NoSQL-tietokantaratkaisua verrattuna perinteiseen MySQL-tietokantaan. Palvelinpuolen ja tietokantojen ratkaisuissa halutaan kiinnittää huomiota toteutusten skaalautumiseen ja miten hyvin kyseiset teknologiat voisivat sopia Big Data ratkaisuihin. Työssä tutkitaan Hadoop-sovellusarkkitehtuurin rakennetta ja sen ympärille rakennettuja liitännäisiä, halutaan selvittää miten kyseistä arkkitehtuuria voitaisiin hyödyntää tulevaisuudessa. Hadoop-arkkitehtuurin testaus ei kuulu tämän työn sisältöön vaan sen käyttöön on tarkoitus siirtyä pidemmällä aikavälillä.

Lopputyössä TwitterSensor-palvelu toteutettiin yhden sivun sovelluksena ”Single-Page App, SPA”. Toteutukseen käytettiin niin sanottua MEAN-pinototeutusta, joka tarkoittaa

MongoDB-tietokannan, Express-, AngularJS- ja Node.js-sovelluskehyksen käyttöä. MEAN muodostaa ohjelmistopinon, jossa on käytetty toteutukseen pelkästään JavaScript-ohjelmointikieltä. Työssä toteutettiin asiakasohjelman käyttöliittymä ja sovelluslogiikka AngularJS-sovelluskehyksellä, jonka lisäksi asiakasohjelman kuvaajat toteutettiin D3.js- ja Google Charts JavaScript -kirjastoilla. Palvelinohjelmisto toteutettiin Node.js-sovellukehyksen päälle REST-arkkitehtuurista tyypillisellä toimintaperiaatteella. Tietokantoina sovelluksessa käytettiin MySQL- ja MongoDB-tietokantoja. Ennen tiedon viemistä MongoDB-tietokantaan tietoa tarvitsi esikäsitellä, jota varten kirjoitettiin node.js:llä suoritettavia pieniä sovelluksia. MongoDB-tietokantoja varten luotiin useampi tietokanta, jotta pystyttiin vertailemaan yksittäisen ja hajautetut toteutuksen nopeuseroja.

Toisessa luvussa tutustutaan Big Data -ilmiöön, mikä on syy yllämainittujen teknologioiden kiinnostavuuteen. Luvussa kolme käsitellään nykyaikaisia asiakas- ja palvelinpuolen teknologioita sekä niiden skaalautumista. Neljännessä luvussa käsitellään eri NoSQL-toteutuksia, joista perehdytään erityisesti MongoDB-tietokantaan. Luvussa viisi tutustutaan Hadoop sovelluskehykseen ja sen ympärille rakennettuihin sovelluskehyksiin ja ohjelmointimalleihin. Kuudennessa luvussa testataan palvelinpuolen toteutuksia ja niiden skaalautuvuutta sekä MongoDB:n eri tietokantaratkaisuja ja viimeisenä tässä luvussa käsitellään asiakasohjelman testausta. Luvussa seitsemän käydään läpi toteutetun sovelluksen arkkitehtuuria sekä millaisia ongelmia havaittiin käytännössä. Luku kahdeksan sisältää yhteenvedon.

2 Big Data

Työssä käsiteltävän TwitterSensor-palvelun tieto on peräisin Twitter-palvelun tarjoamasta tietolähteestä. Twitter tarjoaa avoimen rajapinnan kautta noin 1,0 % kaikista palveluun tulleista viesteistä. Palveluun lähetetään noin 5700 viestiä sekunnissa [6.], ja yhden viestin sisältämä tieto on keskimäärin noin 22 kilotavua, keskiarvo laskettu käytössä olevasta datasta. Tämä tarkoittaa 4,3 miljoonaa viestiä päivässä, jotka vievät noin 10 gigatavua tilaa. Tiedonlähteitä voi olla useita joten laskettaessa todellisia lukuja aikaisemmat kappale ja tilamäärät voidaan kertoa kolmella tai neljällä.

Big Datasta puhutaan useasti, mutta yleisesti on epäselvää, mitä termillä tarkoitetaan. Big Data-termillä voidaan viitata monenlaiseen dataan. Ei ole aivan yksiselitteistä tapaa määrittää, mitä on Big Data: On aika yrityskohtaista, mitä missäkin yrityksessä luokitel-

laan Big Dataksi. IBM:n mukaan käsitteellä Big Data viitataan dataan, jota ei pystytä prosessoimaan tai analysoimaan normaaleilla käytössä olevilla menetelmillä. Dataa joka yleisesti luokitellaan Big Dataksi luonnehditaan sen volyymin, vauhdin ja variaation perusteella. [2, s. 3.]

2.1 Ominaisuudet

2.1.1 Volyymi

Dataa syntyy ja sitä kerätään jatkuvasti enemmän. Twitter tuottaa yli 7 teratavua (TB) dataa päivässä, ja Facebook tuottaa vastaavasti päivässä yli 10 teratavua. [2, s. 4-5.] Volyymin kasvaessa sen käsittely ja hallinta vaikeutuvat perinteisten relaatiotietokantojen avulla. Volyymin kasvaessa siitä saatavat tiedot ovat entistä tarkempia ja tästä tiedosta pystytään saamaan tarkempia johtopäätöksiä. Esimerkiksi vakuutusyhtiöt suorittavat jatkuvaa analysointia, miten kannattavaa asiakkaille on tarjota tietynlaisia vakuutuksia. Mitä enemmän heillä on käytettävissä tietoa muista vastaavanlaisista asiakkaista, sitä paremmin he pystyvät arvioimaan kannattaako vakuutuksen hakijalle myöntää vakuutusta eli onko se hyvä sijoitus yhtiön kannalta. Samankaltaiset tilanteet, joissa datan määrästä on hyötyä johtavat tilanteisiin, jossa tiedon talletuksesta aiheutuu ongelma. Tämänlaista tilannetta on lähdetty ratkaisemaan jakamalla data pienempiin osiin ja sijoittamalla se eri palvelimille, joista jokainen käsittelee murto-osan datasta.

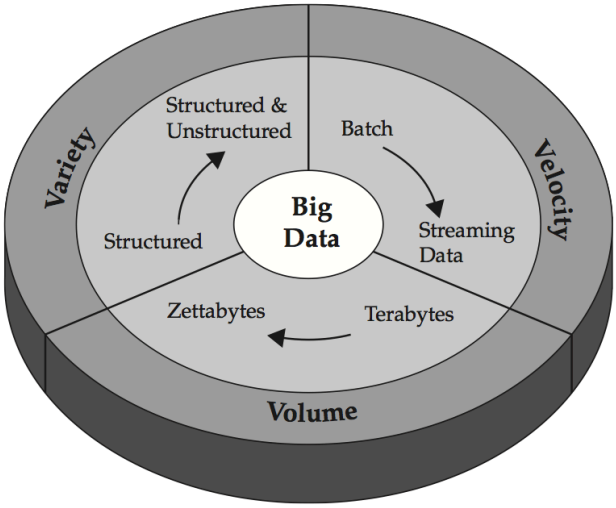
2.1.2 Vauhti

Vauhdilla tarkoitetaan ilmiötä, kuinka nopeasti dataa syntyy. Sitä koskee samanlainen ilmiö kuin volyymia. Nykyään suurta osaa järjestelmistä pyritään automatisoimaan ja kehittämään eteenpäin, mikä tarkoittaa RFID (Radio Frequency IDentification) -sensoreiden ja muiden tietolähteiden lisääntymistä. Voidaan puhua niin sanotusta Teollisesta internetistä, jossa järjestelmät pystyvät viestimään keskenään internetin välityksellä. [6.] Järjestelmien kehittymisen myötä ollaan tilanteessa, jossa uutta dataa tuotetaan sellaisella vauhdilla, että sen hallitseminen normaaleilla järjestelmillä on mahdotonta. [2, s. 8.]

2.1.3 Variaatio

Palveluiden, laitteiden ja sensoreiden lisääntyessä saadaan yhä enemmän erilaista tietoa. Tiedon jäsentäminen ei ole enää niin yksinkertaista, koska tieto voi olla raakadataa, semistrukturoitua, strukturoimatonta, web-sivuja, lokitiedostoja, sensoridataa tai vastaavaa. Tietoa on siis hyvin monessa eri muodossa, mistä johtuen sen tallennus ja hyödyntäminen tavanomaisilla tietokannoilla ja järjestelmillä ei ole mahdollista. Perinteinen relaatiotietokanta on hyvä vaihtoehto tallentaa ja hakea järjestettyä tietoa, joka on usein yksiselitteisesti numero- tai merkkimuotoista. Tietolähteiden lisääntyessä tiedon sisältö ja tallennusmuoto voivat vaihdella huomattavasti, mitä relaatiotietokanta ei pysty käsittelemään. [2, s. 7.]

Suurin osa Big Datan kanssa tekemisissä olevista yrityksistä käyttää kolmea yllämainittua ominaisuutta määrittelemään, luokitellaanko tieto Big Dataksi. Ominaisuuksia on hahmotettu kuvassa 1, joka kuvaa tiedon monimuotoisuutta, vauhtia ja volyymin kasvua.



.

Kuva 1. Volyymi, Vauhti, Variaatio. [2.]

Big Data ei ole vain yksi asia eikä siihen myöskään ole vain yhtä oikeaa ratkaisua. Markkinoilla on ohjelmistoja ja yrityksiä, jotka pyrkivät ratkaisemaan ongelman. Seuraavaksi lähdetään tutustumaan, mitä eri vaihtoehtoja on olemassa nykyisille relaatiotietokannoille, mitä ne tarjoavat ja miten ne soveltuvat big datan asettamiin haasteisiin. Ongelma ei aiheudu relaatiotietokannan kyvystä skaalautua tai siitä, etteikö tietoa voitaisiin esikäsitellä muotoon jolla se on saatavissa relaatiotietokannasta.

Ongelma on enemmän siinä, kuinka kauan tiedon esikäsittely vie resursseja ennen kuin tieto on hyödyllisessä muodossa. Tästä seuraa myös ilmiö, jossa käsitelty tieto on hyödyllistä vain yhdessä tarkoituksessa. Mikäli myöhemmin ollaan kiinnostuneita tutkimaan tietoa eri näkökulmasta, se vaatii taas tiedon esikäsittelyn ja tallentamisen uudestaan hyödyllisessä muodossa. Tämä ei lisää ainoastaan tiedon käsittelyyn kuluvaa aikaa, mutta lopputuloksena aiheutuu tiedon monistuvuutta. Järjestelmässä on sama tietoa useaan kertaan hieman erilaisena, ja se vie turhaan resursseja.

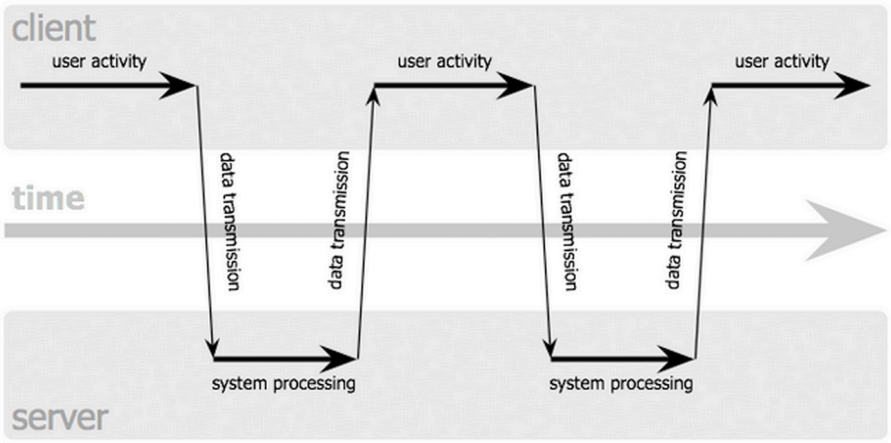
3 Modernit web-teknologiat

Luvussa 1 mainittiin Googlen ja Microsoftin tekemästä tutkimuksesta miten käyttäjien kokemat latausajat vaikuttivat negatiivisesti käyttökokemukseen. Jakob Nielsenin mukaan 0,1 sekuntia on vasteaika jolloin käyttäjä kokee verkkosivun vastaavan toimintoihin reaaliaikaisesti. Vasteajan ollessa 0,2-1,0 sekuntia käyttäjä tuntee pystyvänsä käyttämään sivua vaivattomasti, mutta huomaavansa käytössä viivettä. Vasteajan ylittäessä 10,0 sekuntia käyttäjä ei enää keskity verkkosivuun vaan rupeaa ajattelemaan muuta. [8.] Vuonna 2012 verkkosivun keskimääräinen latausaika oli tietokoneella 6,9 sekuntia ja mobiililaitteella 10,0 sekuntia. [9.] Huolimatta päätelaitteiden kehityksestä myös palvelut, joita laitteilla käytetään, ovat kehittyneet. Siitä johtuen sivustojen vasteajat pysyneet samana tai jopa kasvaneet. Radwaren heinäkuussa 2014 julkaiseman tutkimuksen mukaan mediaanin verkkokaupan latausaika on kasvanut noin 49 % (+3,5 sekuntia) ja keskimääräinen vasteaika on kasvanut 27 % (+1,3 sekuntia). [10.]

Tässä luvussa käsitellään teknologioita, joilla on mahdollista pienentää vasteaikoja. AngularJS on sovelluskehys, jolla on mahdollista vähentää käyttäjän kokemaa viivettä verkkosovelluksessa. Lisäksi alaluvussa 3.5 käsitellään Node.js-sovelluskehystä, joka mahdollistaa asynkronisen palvelinpuolen toteutuksen, jolla on mahdollista vähentää tietokantakyselyiden prosessoimiseen ja asiakasohjelman pyyntöihin vastaamiseen kuluvaa aikaa. Alaluvussa 3.6 käsitellään JavaScript-kirjastoja joilla voidaan toteuttaa erilaisia kaavioita ja karttoja, joilla visualisoida ja selkeyttää käyttäjälle palautettavaa tietoa ja mahdollisesti vähentää käyttäjän kokemaa vasteaikaa asiakasohjelman käytössä.

Perinteiset web-sovellukset perustuvat HTTP-malliin, jossa käyttäjän aiheuttama toiminta saa aikaan palvelupyynnön palvelimelle. Palvelin käsittelee pyynnön, johon vastauksena lähetetään HTML-muotoinen sivu. Mikäli sivu sisältää viittauksia muihin tie-

dostoihin kuten CSS-tyylitiedostoihin, niin selain lähettää uudelleen palvelupyynnön näiden tiedostojen lataamiseksi. Aina käyttäjän siirtyessä sivulta toiselle sama toimenpide toistetaan. Kyseistä toimintomallia on havainnollistettu kuvassa 2, missä käyttäjän aiheuttama toimintaa suorittaa palvelimelle pyynnön. Palvelimen käsiteltyä pyynnön se palautetaan käyttäjälle. [11.]

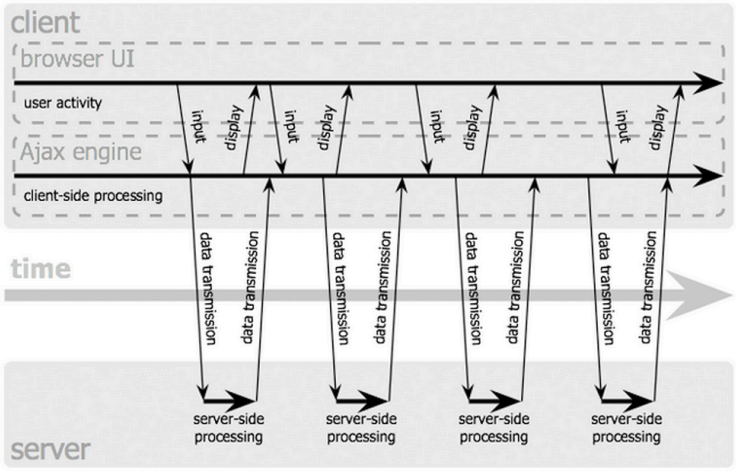


Kuva 2. Perinteisen web-sovelluksen toimintamalli. [11.]

Jokainen käyttäjän toiminta aiheuttaa palvelinpyynnön. Pyyntö saattaa olla sivun latauspyyntö tai pyyntö palvelimelle hakea tietoa tietokannasta. Tietokannasta haettua tietoa saatetaan joutua käsittelemään välissä ennen kuin se palautetaan asiakasohjelmalle ja näytetään käyttäjälle. Tämän tapaiset pyynnöt saattavat olla raskaita palvelimelle, mikäli niitä suoritetaan paljon tai tietokannasta haetun tiedon määrä on suuri. Tietokannasta haettaessa suuria määriä tietoa vastaavasti palvelimelta asiakasohjelmalle palautettavan tiedon määrä on suuri.

Perinteisessä verkkosovellusmallissa asiakasohjelma joutuu odottamaan palvelimen vastausta ennen kuin vastauksena saatu verkkosivu voidaan näyttää käyttäjälle. Tämä tarkoittaa sitä, että kyseisenä aikana käyttäjän ei ole mahdollista suorittaa minkäänlaisia toimintoja verkkosivulla.

Ratkaisuna perinteisen web-sovelluksen ongelmiin on kehitetty malli, jossa reagoidaan käyttäjän tekemiin toimintoihin päivittämällä sellaista osaa sivusta johon käyttäjän toiminta on pelkästään vaikuttanut. Lisäksi osa palautetusta vastauksesta käsitellään asiakasohjelman päässä sen sijasta, että palvelin hoitaisi kaiken työn ja palauttaisi valmiin tuloksen asiakasohjelmalle. [11.] Tämän kaltaista toimintamallia on havainnollistettu kuvassa 3.



Kuva 3. AJAX-toimintamalli. [11.]

Ongelmaa lähdettiin ratkaisemaan vaihtamalla aikaisemmin käytössä olleet tekniikat nykypäivänä suosituksi tulleisiin ja tehokkaiksi havaittuihin JavaScript-tekniikoihin, sekä muuttamalla palvelun tietokantaliitännän toteutusta sellaiseen muotoon, että se pystyy hallitsemaan kasvavat tietovarastot.

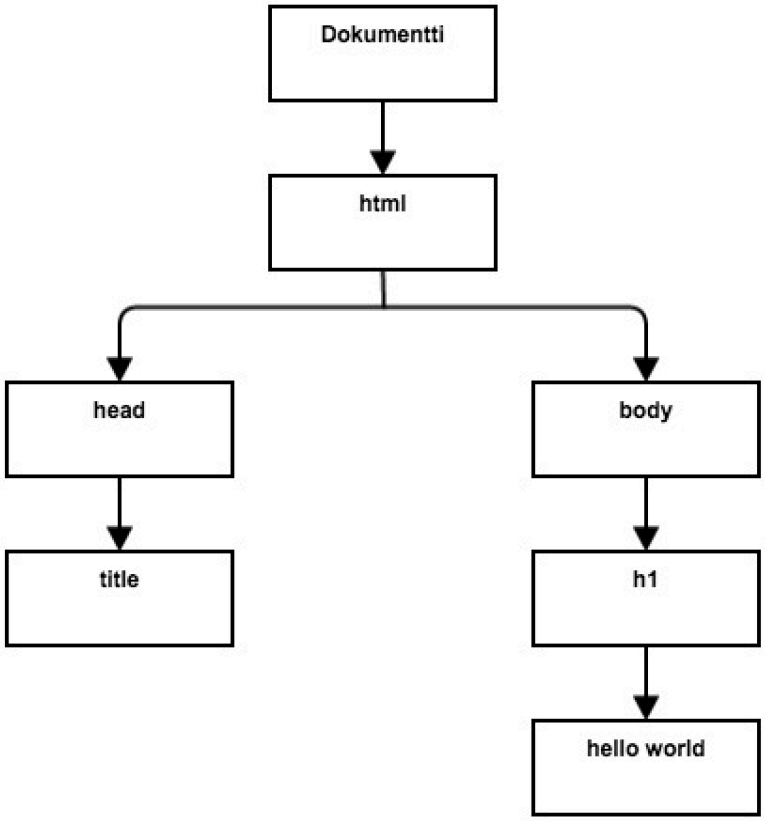
Document-object Model

Selaimen tulkitessa HTML-muotoista dokumenttia se muodostaa dokumentin rakenteesta mallin, joka sisältää tiedon siitä, miten eri elementit on jäsennetty ja hyödyntää tätä mallia, kun tieto esitetään selaimessa. JavaScript ymmärtää DOM-mallia mikä mahdollistaa elementtien lisäämisen, muokkaamisen ja poistamisen. HTML-dokumentin hierarkkisesta rakenteesta johtuen siitä muodostettava DOM noudattaa samaa rakennetta, jota useasti kuvataan puu-mallilla. Kuvassa 4 havainnollistetaan ”hello world” -sivun toteutusta HTML-merkkikielellä, missä määritellään elementit joista sivu koostuu. [12, s. 231.]

```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1>hello world</h1>
  </body>
</html>
```

Kuva 4. Hello world HTML-merkkikielellä

Kuvassa 5 havainnollistetaan kuvasta 4 muodostuvaa DOM-rakennetta. Tarkastelemalla kuviota 5 voidaan havaita <html>-aloituselementin kuuluvan ylempään kokonaisuuteen jota selain käsittelee dokumenttina. Kuvassa 5 puurakennetta seuraamalla havaitaan vasemman puoleisen puun ensimmäisen lehden muodostuvan <head>-elementistä, jonka lehtenä on <title>-elementti. Vastaavalla tavalla dokumentin seuraavasta tunnisteesta joka on <head>-elementin kanssa samalla syvyydellä muodostetaan puun oikean puoleinen lehti. Oikean puoleisen lehden sisältämät lehdet muodostetaan siinä järjestyksessä miten syvällä HTML-merkkikielen rakenteessa ne sijaitsevat.



Kuva 5. Hello world - DOM rakenne

3.1 Representational State Transfer (REST)

Representational State Transfer (REST) on HTTP-protokollaan perustuva arkkitehtuurimalli, joka voidaan määrittää kuuden eri vaatimuksen perusteella. Vaatimuksia ovat asiakas-palvelin-malli, tilattomuus, välimuisti, kerroksittainen järjestelmä, yhdenmukai-

nen rajapinta ja code-on-demand-toteutus. Näistä viimeinen ei ole välttämättömyys, jotta toteutus voidaan määritellä REST-arkkitehtuurin mukaiseksi. [13.]

REST-arkkitehtuurimallissa erotetaan asiakas- ja palvelinohjelman toiminta toisistaan, mikä mahdollistaa molempien itsenäisen toteutuksen sekä tekee palvelusta alusta- ja ympäristöriippumattoman. Asiakasohjelman ei tarvitse tietää millaisella teknologialla sen kutsuihin vastaava palvelinohjelma on toteutettu eikä palvelinohjelman tarvitse tietää miten kutsuja lähettävä asiakasohjelma on toteutettu. Sama palvelin pystyy vastaamaan kutsuihin, jotka lähetetään esimerkiksi verkkoselaimessa tai älypuhelimessa toimivan sovelluksen kautta. [13.]

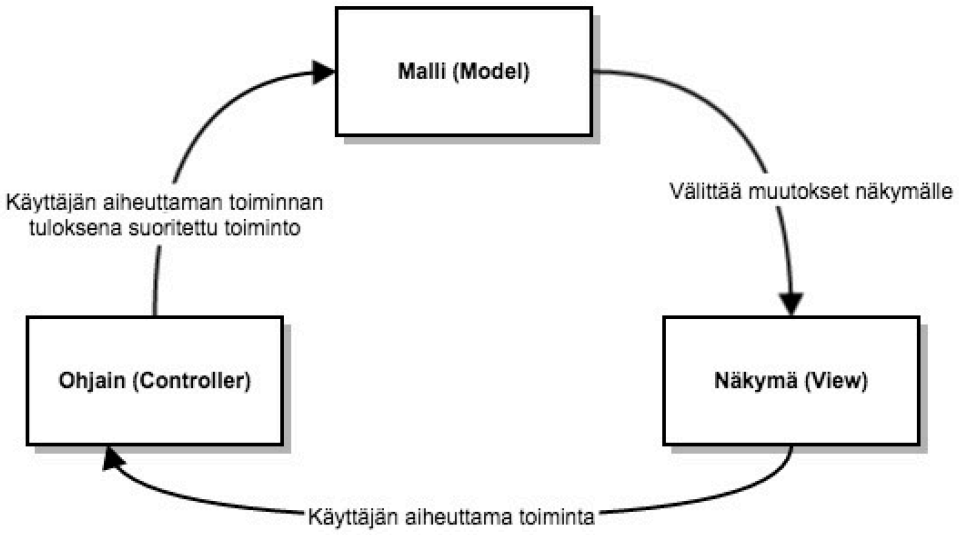
Tilattomuudesta johtuen asiakas-palvelin-mallissa asiakasohjelman lähettämän kutsun täytyy sisältää kaikki mahdollinen tieto jota palvelin tarvitsee pystyäkseen käsittelemään pyynnön oikein. Tästä syystä palvelin ei pysty lähettämään esikäsiteltyä tietoa asiakasohjelmalle. Arkkitehtuurimallista johtuen asiakasohjelma on ainut, joka tietää palvelun tilan, mistä johtuen palvelimen ei tarvitse vastata kuin yksittäisiin kutsuihin huolehtimatta missä tilassa ollaan milläkin hetkellä, mikä lisää palvelimen skaalautuvuutta, kykyä vapauttaa resursseja uusien kutsujen myötä ja mahdollisuutta reagoida vikatilanteisiin, koska vikatilanne on aina kutsukohtainen. [13.]

Välimuistin käytöllä REST-arkkitehtuurissa tarkoitetaan palvelimen kykyä varastoida asiakasohjelmalle lähetettäviä vastauksia välimuistiin siten, että niitä ei haeta joka kerta palvelimelta uudestaan. Käytettäessä välimuistia palvelimelle lähetettävän kutsun täytyy olla yksiselitteinen siitä, haetaanko tieto suoraan palvelimelta vai välimuistista. Jossain tapauksissa välimuistin käyttö ei ole mahdollista, mikäli kutsun vastauksena lähettämä tieto on riippuvainen kutsun sisältämästä muuttujasta.

REST-arkkitehtuurin tilattomuudesta johtuen se saattaa lisätä verkon välistä liikennettä, koska oletuksena jokainen kutsu palvellaan suoraan palvelimelta. Tästä johtuen useasti toistuvien samanlaisten kutsujen palvelemisen välimuistista voi vähentää verkon välistä liikennettä ja palvelimen kuormaa, koska kutsua ei tarvitse käsitellä palvelimella. REST-arkkitehtuurin käytöllä on useita hyötyjä se mahdollistaa rajapintojen siirrettävyyden, skaalautuvuuden, yksinkertaisemman toteutuksen ja verkon suorituskyvyn paranemisen. [13.]

3.2 Model-View-Controller (MVC) -arkkitehtuuri

Model-View-Controller arkkitehtuurissa pyritään erottamaan sovelluksen rakenne selkeästi kolmeen eri alueeseen, jolla pyritään helpottamaan ohjelman toteutusta, kun tiedetään selkeästi mitä kunkin alueen kuuluu toteuttaa. MVC-arkkitehtuuria kuvataan usein myös suunnittelumallina jossa malli (Model) sisältää ja hallitsee tietoa, näkymä (View) on vastuussa tiedon esittämisestä ja ohjain (Controller) sisältää tiedon ohjelman logiikasta ja näin ollen pyytää tietoa mallilta. MVC-suunnittelumalli kehitettiin 1970-luvulla ja on siitä lähtien ollut laajalti käytössä riippumatta ohjelmointikielestä. [14, s. 3.] Kuvassa 6 havainnollistetaan MVC-mallin toimintaa, missä ohjain käsittelee käyttäjän toiminnon, jonka seurauksena ohjain muokkaa, tallentaa tai poistaa tietoa mallilta ja mallin muutokset päivittyvät takaisin näkymälle.



Kuva 6. MVC-suunnittelumallin esimerkkitoiminta.

3.3 AngularJS

AngularJS on Googlen kehittämä täysin JavaScriptillä toteutettu asiakaspuolen ohjelmistokehys, joka on tarkoitettu dynaamisten web-sovellusten toteuttamiseen. Sen tarkoituksena on helpottaa ja luoda joustavuutta nykyaikaisten web-sovellusten toteutukseen lisäämällä HTML-merkkikieleen uusia toimintatapoja käsitellä merkkikieltä ja tuomalla muualta toimivaksi todetut ohjelmointimallit asiakaspuolen ohjelmistoihin.

TwitterSensor-palvelussa asiakasohjelmisto toteutettiin käyttäen AngularJS-sovelluskehystä, joka mahdollistaa asiakasohjelmiston toteuttamisen Model-View-Whatever (MVW) -suunnittelumallilla. Suunnittelumallin käyttö helpottaa ohjelmistojen ylläpitoa ja lisätoimintojen toteuttamista jo olemassa olevalle sivustolle. Lisäksi AngularJS tuo web-sovelluskehitykseen modulaarisuutta.

AngularJS mahdollistaa osan sovelluslogiikan toiminnallisuuden toteutuksesta asiakasohjelman päässä, joka vähentää palvelimelle kohdistuvaa kuormaa. Samalla pystytään välttämään saman tiedon välittämistä asiakasohjelmaan jokaisella sivulatauksella, joka vähentää käyttäjän kokemaa viivettä esimerkiksi sivua vaihdettaessa. Perinteisten MVC-suunnittelumallin mukaan AngularJS:llä kehitetty web-sovellus sisältää mallin, näkymän ja ohjaimen, minkä lisäksi ohjelma voi sisältää palveluita (Services), direktiivejä (Directives) ja suodattimia (Filters). [15, s. 9.]

3.3.1 Model-View-Whatever (MVW) -arkkitehtuuri

Kehittäjien keskuudessa oli pitkään epäselvyys asiasta, millaisen suunnittelumallin AngularJS tarjoaa. Ehdotuksia olivat esimerkiksi Model-View-Controller (MVC) ja Model-View-View-Model (MVVM). Lopulta päädyttiin virallisesti Model-View-Whatever (MVW) -arkkitehtuuriin. Sen tarkoituksena on selventää ohjelmiston rakennetta ilman halua lokeroitua mihinkään tiettyyn arkkitehtuurimalliin. Tämän takia ”Whatever” tarkoittaa kokonaisuudessaan ”Whatever works for you” eli mitä tahansa ratkaisua, joka vastaa käyttäjän tarpeita. Tällä pyritään selkeyttämään ohjelman rakennetta, tuomaan ohjelmaan modulaarisuutta ja joustavuutta ja mahdollistamalla ohjelman helpomman testauksen.[15, s. 9.]

3.3.2 Malli (Model)

AngularJS:n malli (Model) voi olla yksinkertaisuudessaan merkkijono tai numero. Ei ole olemassa tarkkaa määritystä sille, missä luokassa mallit täytyisi toteuttaa tai mitä metodeja jokaisen mallin täytyisi toteuttaa. Yleensä mallit ovat JavaScript-objekteja, jotka kirjoitetaan funktioiksi. Myöhemmin kirjoitetaan logiikka ja käytössä olevat metodit prototyyppimäärittelyn avulla. Mallin määrittelyn jälkeen malli saadaan AngularJS:n komponenttien käytettäväksi injektoimalla se angular.module-value-metodin avulla. Mallin luomista ja injektointia on havainnollistettu kuvassa 7. Esimerkissä luodaan Person-malli, jolla on muutama muuttuja. Tämän jälkeen prototyyppimäärittelyllä toteutetaan

osalle muuttujista toteutetaan metodit, jonka jälkeen malli injektoidaan käytettäväksi viimeisellä rivillä. [15, s. 72.]

```
// AngularJS Malli
// Esimerkki henkilö ilman tietoja
var Person = function() {
  var self = this;
  self.firstname = '',
  self.lastname = '',
  self.address = '',
  self.creditcardBalance = 0.0
};

// Henkilölle funktiot millä voidaan asettaa
// etunimi, sukunimi ja osoite¢
Person.prototype = {
  setFirstname: function(firstname) {
    this.firsname = firsname;

  },
  setLastname: function(lastname) {
    this.lastname = lastname;

  },
  setAddress: function(address) {
    this.address = address;
  }
};

var module = angular.module('personModel',[]);
module.value('Person',Person);
```

Kuva 7. AngularJS-malliesimerkki

3.3.3 Näkymä (View)

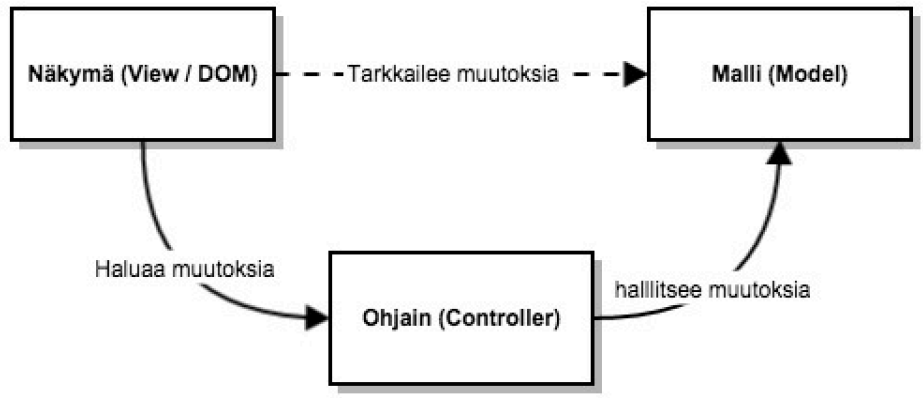
Useissa ohjelmistokehyksissä näkymät toteutetaan erillisellä merkkikielellä joka lopulta käännetään HTML-merkkikielelle, jotta selain pystyy tulkitsemaan näkymän. AngularJS:n näkymät toteutetaan suoraan HTML-merkkikielellä johon AngularJS tuo uusia attribuutteja, jotka mahdollistavat kontrollerissa sijaitsevan tiedon välittämisen asiakasohjelmalle. Tällä pyritään helpottamaan näkymien toteutusta, sillä useimmille HTML-merkkikieli on ennestään tuttu, joten uuden merkkikielen opetteluun ei ole tarvetta. Näkymät perustuvat HTML-merkkikielen lisäksi direktiiveistä, jonka vastuulla on DOM-

olioiden muokkaus ja hallinnointi. Kuvassa 8 havainnollistetaan AngularJS-näkymien toteutusta HTML-merkkikielellä ja AngularJS-direktiivien avulla.

```
<!-- AngularJS template example -->
<div class="person">
  <h1>Etunimi: {{ person.firstname }}</h1>
  <h1>Sukunimi: {{ person.lastname }}</h1>
</div>
```

Kuva 8. AngularJS näkymä HTML-merkkikielen kanssa.

Kuvassa 9 kuvataan AngularJS-sovelluksen tapaa välittää tietoa. Kuviossa näkymä ilmoittaa tarvitsevansa tietoa ohjaimelta, jolloin ohjain suorittaa tarpeelliset toimenpiteet näkymän pyyntöön liittyen ja tarvittaessa päivittää muutokset myös malliin. Kuviossa on hyvä kiinnittää huomiota siihen, että kaikki nuolet näyttävät poispäin näkymästä, joka tarkoittaa ettei ohjelman tarvitse alkaa näkymästä. Ohjain ja malli voidaan alustaa toisistaan ja näkymästä riippumatta, joka lisää testattavuutta huomattavasti. [16.]



Kuva 9. AngularJS-näkymän, ohjaimen ja mallin suhde. [16.]

3.3.4 Ohjain (Whatever)

Ohjain pitää huolen näkymän ja mallin välisestä toiminnasta. Ohjaimessa yleensä alustetaan näkymässä näytettävät tiedot sekä ohjaimeen toteutetaan sovelluslogiikka. AngularJS:ssa ohjaimen sijasta sovelluksen tilasta voi huolehtia esimerkiksi näkymä. Tässä tapauksessa ohjain on paikka jossa toteutetaan funktioita ja hallitaan malleja, mutta sovelluksen ohjauksesta vastaa näkymä.

3.3.5 Direktiivit, palvelut ja suodattimet

Direktiivit (Directives)

Direktiivit ovat tapa merkitä DOM-elementit AngularJS:n HTML-kääntäjää varten. Tällä tavalla HTML-kääntäjä osaa liittää tietyn toiminnallisuuden oikeisiin DOM-elementteihin, joita ovat esimerkiksi elementtien nimet, attribuutit, luokkien nimet sekä kommentit. AngularJS:ssä on sisäänrakennettu tuki ngBind-, ngModel-, ja ngClass-direktiiveille. [17.]

Palvelut (Services)

Palveluilla tarkoitetaan Singleton-suunnittelumallin mukaisia olioita, joita on olemassa vain yksi kappale. Palvelun käyttöön on sovelluksen laajuinen pääsy ja sen elinkaarta säätelee ohjelmistokehys. AngularJS mahdollistaa palveluiden toteuttamisen käyttämällä tehdas-suunnittelumallia, jonka tehtävänä on luoda halutunlaisia olioita. Palveluita voidaan myös luoda palveluina milloin sen luominen tapahtuu New-operaattorilla. Kuvassa 10 havainnollistetaan tehdas-suunnittelumallin ja palvelun eroa. Tehdas-suunnittelumallissa palautetaan tehdas-funktio return-lauseessa, mutta palvelussa luodaan uusi palvelu. [18, s. 69-74.]

```
// Factory example
test.factory("testService", function() {
  var _doSomething = function(obj) {
    // set obj values
  };

  // return obj
  return {
    doSomething: _doSomething
  };
});

// Service example
test.service("testService2", function() {
  this.doSomething = function(obj) {
    // set obj values
  };
});
```

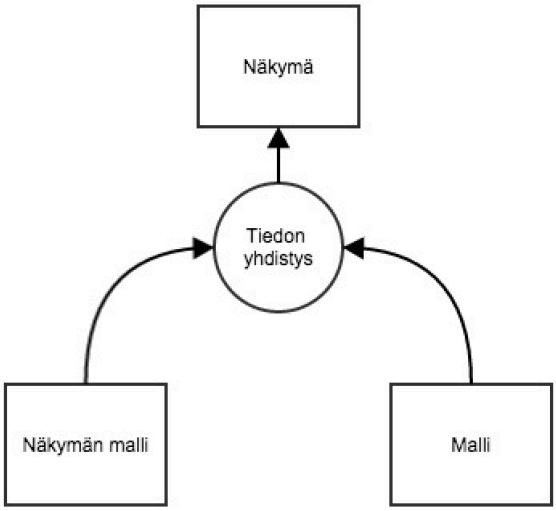
Kuva 10. Tehdas-suunnittelumallin ja palvelun ero.

Suodattimet (Filters)

Suodattimet toimivat yhteistyössä esimerkiksi direktiivien kanssa. Suodattimet mahdollistavat minkä tahansa tiedon muuttamisen ja muuntamisen. Suotimien käsittelemä tieto ei rajoitu pelkästään näkymän sisältämän muuttujien arvoihin. Suotimilla voidaan muokata myös injektoitujen komponenttien kuten ohjainten ja palveluiden sisältämää tietoa. [18, s. 55.]

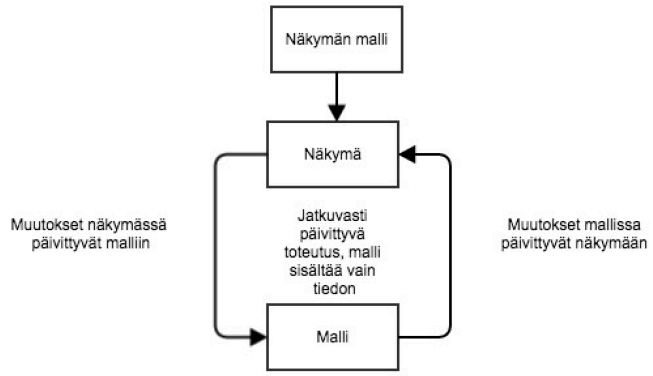
3.3.6 Kaksisuuntainen tiedon sitomismalli

Perinteisissä web-sovelluksissa on totuttu näkemään yksisuuntainen tiedon sitomismalli. Tällä tarkoitetaan toteutusta jossa näkymän sisältämä tieto saadaan mallilta, jonka jälkeen malliin tapahtuvat muutokset eivät päivity enää näkymälle. Aikaisemmin tätä on pystytty muuttamaan kolmannen osapuolen tarjoamilla JavaScript-kirjastoilla kuten esimerkiksi jQuerylla. Yksisuuntaista tiedon sidontaa on havainnollistettu kuvassa 11, jossa mallin ja näkymän mallit yhdistetään kerran ennen kuin malli välitetään asiakasohjelmalle.



Kuva 11. Yhdensuuntainen tiedonsidonta. [17.]

Kaksisuuntaisella toteutuksella tarkoitetaan toteutusta, jossa mallin ja näkymän välinen tieto on aina ajan tasalla. Kahdensuuntaista tiedon sidontaa on havainnollistettu kuvassa 12. Aluksi näkymän malli käännetään, jolloin siihen muodostetaan direktiivien väliset liitokset. Tämän jälkeen näkymässä aiheutuvat muutokset muutetaan suoraan malliin ja vastaavasti malliin tapahtuvat muutokset päivittyvät suoraan näkymälle. Näkymää voidaan pitää heijastuksena mallista. [17.]



Kuva 12. Kaksisuuntainen tiedon sidonta. [17.]

3.3.7 Reititys

AngularJS:llä kehitetyt ohjelmat ovat lähes aina Single-Page-Application (SPA) -toteutuksia eli sivuston päänäkymä ladataan kerran, jonka jälkeen päivitetään pelkästään DOM:n sisältöä, minkä avulla saadaan toteutettua eri sivunäkymät asiakasohjelmalle. Tästä johtuen asiakasohjelman puolella täytyy toteuttaa reititys, joka tapahtuu $routeProvider-palvelun kautta. [14, s. 38.]

Selaimen URL-osoitteen muuttuessa Angular lataa sitä vastaavan näkymän, ja samalla ladatun näkymän juurielementtiin liitetään kyseisen URL-reitin alla nimetty ohjain. Mikäli käyttäjä menee sivulle /persons AngularJS lataa, partials/person-list.html näkymän ja liittää näkymän juuri elementtiin PersonListCtrl-nimisen ohjaimen. Valitusta toteutustavasta riippuen voidaan web-sovelluksen ohjaimen toteutus kirjoittaa myös reitittimen yhteyteen sen sijasta, että se haettaisin muualta. Reitittimen toimintaa ja eri ohjainvaihtoehtoja on havainnollistettu kuvassa 13.

```
personsApp.config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
      when('/persons', {
        templateUrl: 'partials/persons-list.html',
        controller: 'PersonListCtrl'
      }).
      when('/person/:personId', {
        templateUrl: 'partials/person.html',
        controller: function () {
          console.log("personId controller");
        }
      }).
      otherwise({
        redirectTo: '/persons'
      });
  }]);
```

Kuva 13. AngularJs - reitityksen ja eri kontrollerien havainnollistaminen.

Vaihtoehtoisesti reititys voidaan toteuttaa kolmannen osapuolen kehittämällä AngularUI Router -sovelluskehyksellä, josta ei ole vielä olemassa vakaata julkaisua. AngularUI muuttaa reitityksen käyttämään tilakone-suunnittelumallin kaltaista toteutusta. Tämä mahdollistaa näkymien välillä siirtymisen muullakin kuin pelkän url-osoitteen perusteella.

3.4 Node.js

JavaScript kehitettiin alun perin käytettäväksi selainpuolen ohjelmoinnissa, jossa se sai paljon huomiota osakseen ja node.js:stä tuli suosittu kehittäjien keskuudessa. Ennen node.js:n kehitystä oli olemassa muutamia aikaisempia yrityksiä hyödyntää JavaScriptiä palvelinpuolen ohjelmoinnissa. Yrityksistä huolimatta se ei noussut suureen suosioon, koska sitä ei pidetty vakavasti otettavana ohjelmointikielenä. Lisäksi JavaScript oli hidas verrattuna muihin ohjelmointikieliin. JavaScript oli kuitenkin ainut ohjelmointikieli, jota kaikki selaimet tukivat. Samalla kun eri valmistajien kilpailu selainmarkkinoilla kasvoi, kasvoi myös heidän kiinnostuksensa panostaa JavaScript-teknologiaan enemmän. Lopputuloksena tästä on Googlen Chrome-selaimesta tuttu V8-virtuaalikone, mikä vei JavaScriptin nopeuden aivan uudelle tasolle. [19, s. 1.]

Node.js on C++:lla ja JavaScriptillä toteutettu tapahtumavetoinen asynkroninen ohjelmistokehys, mikä käyttää hyväkseen Googlen V8-virtuaalikonetta. Perinteiset websovellukset ja palvelinratkaisut ovat I/O-sidonnaisia. Kun suoritetaan esimerkiksi levyltä lukutoimintoja, niin sillä aikaa ei pystytä tekemään mitään muuta. Ratkaisuna tähän perinteiset palvelinohjelmistot kuten Apache käyttää useampaa säiettä, mutta säikeiden ylläpitäminen on raskasta, mikä ei mahdollista suurta skaalautuvuutta. Node.jsinstanssi päinvastoin on käynnissä vain yhdessä säikeessä. Mikäli tämä säie jää odottamaan, niin silloin koko palvelin pysähtyy. Tästä syystä Node käyttää I/O-mallia, jossa se ei jää odottamaan, että tietyt asiat tapahtuvat ennen kuin se pystyy jatkamaan ohjelman suorittamista. [19, s. 1.]

Noden I/O-toteutus ja asynkronisuus tarjoavat erittäin hyvän skaalautuvuuden palvelinpuolella minkä ovat huomanneet monet suuret yritykset kuten Microsoft, Yahoo!, LinkedIn sekä Walmart. Esimerkiksi vaihtamalla node.js:n LinkedIn, jolla oli aikaisemmin käytössä 15 palvelinta, missä jokaisella oli toiminnassa 15 virtuaalikonetta, sai vaihdettua malliin jossa 15 virtuaalikoneen sijasta jokaisella palvelimella on toiminnassa vain 4 virtuaalikonetta. [19, s. 1.] Skaalautuvuuden lisäksi PayPal on havainnut node.js:n kaksinkertaistavan kyselyiden määrän, mitä palvelin pystyy käsittelemään, vähentämään vasteaikaa jopa 35 % ja nopeuttamalla sovelluskehitystä kokonaisuudessaan. [20.]

Tässä vaiheessa on hyvä palauttaa mieleen, että lopputyössä on tarkoituksena tutkia eri teknologioita sellaisesta näkökulmasta, että niillä on mahdollista parantaa Twitter-Sensor-palvelun nopeutta ja skaalautuvuutta.

Palvelun parantaminen käytännössä tarkoittaa nopeampaa tiedon välittämistä asiakasohjelmalle ja mahdollisimman monen yhtäaikaisen asiakasohjelman palvelemista. Tästä johtuen node.js:n toimintaa käydään pintapuolisesti läpi sellaisilta osa-alueilta, joita testausvaiheessa kokeiltiin ja joista oltiin kiinnostuneita.

Node.js:lle on olemassa paljon eri kirjastoja (module), jotka helpottavat ja selkeyttävät esimerkiksi perinteisen web-palvelimen toteutusta. Kuvassa 14 esitetään kuinka otetaan käyttöön http-kirjasto rivillä, jonka jälkeen luodaan http-kirjaston sisältämällä createServer-metodilla uusi http-palvelininstanssi, joka kuuntelee porttia 8000. Tämän jälkeen selaimella voidaan mennä osoitteeseen localhost:8000 jolloin selaimeen tulostuu teksti ”node.js is running on port 8000”.

```
var http = require("http"),
    port = 8000;
http.createServer(function(req, res) {
  res.writeHead(200);
  res.end("node.js is running on port " + port);
}).listen(port);
```

Kuva 14. Http-palvelin node.js:llä

3.4.1 REST-arkkitehtuurin mukainen reititys

Asiakasohjelman reititys tapahtuu AngularJS:n toimesta. Tästä johtuen palvelinpuolen toiminta on mahdollista toteuttaa hyvin pitkälti REST-tyylisellä toteutuksella. Tällä tarkoitetaan sitä, että palvelin vastaa ja käsittelee kutsuja vain silloin, kun ne liittyvät tietokantaan tapahtuviin toimintoihin. Tällaisia toimintoja ovat tiedon lisäys, haku, muokkaus ja poisto. Kuvassa 14 on havainnollistettu esimerkkikoodilla, miten node.js:sä voidaan toteuttaa REST-rajapinnan mukaisiin toimintoihin vastaaminen.

```
// router example
var Person = require('../models')
module.exports = function (router) {

  router.get('/', function (req, res) {
    res.render('index');
  });

  // add a person
  router.post('/api/example/person/:person', function (req, res) {
    // insert a person into a database
    // return ok if successful
    // otherwise return error
  });

  // get a person
  router.get('/api/example/person/:person', function (req, res) {
    // get a person from database and return data
    // return error if not found
  });

  // update a person
  router.put('/api/example/person/:person', function (req, res) {
    // update person data on database and return ok
    // return error if not found or invalid data

  });

  // delete a person
  router.delete('/api/example/person/:person', function (req, res) {
    // delete a person from database.
    // if found and deleted return ok
    // if not found or not deleted return error

  });

};
```

Kuva 15. Node.js - REST-rajapinta toiminnot

Router.post (Create)

Post-toiminto käsittelee asiakasohjelmalta tulevat lomakkeenlähetyspyynnöt. Sen käyttötarkoituksena on luoda uusia olioita. Esimerkkinä sivulla voisi olla lomake, jolla luodaan uusia käyttäjiä. Kun käyttäjä on täyttänyt tiedot hän painaa lähetä-nappia, jolloin post-osoite voisi näyttää palvelimelle ’/api/example/person/5’, missä luku 5 tarkoittaa käyttäjä-id-tietoa. Tässä tapauksessa tietokantaan voitaisiin luoda uusi käyttäjä, jonka tunniste on viisi.

Router.get (Read)

Get käsittelee pyynnöt joissa asiakasohjelma pyytää palvelimelta tietoa. Se on tarkoitettu tiedon hakemiseen palvelimelta. Esimerkkinä voidaan pitää tilannetta jossa käyttäjä on kirjautunut palveluun ja menee tarkastelemaan omia tietojaan, jolloin selain lähettää palvelimelle get-pyynnön, joka sisältää id-tiedon, vastaavalla tavalla kuin post-esimerkissä. Id:n perusteella tietokannasta voidaan hakea käyttäjän tiedot ja palauttaa ne asiakasohjelmalle.

Router.put (Update)

Put on toiminto joka on tarkoitettu olemassa olevan tiedon päivitykseen esimerkiksi, kun käyttäjä haluaa päivittää omia tietojaan. Tässä tapauksessa asiakasohjelman lähettämä osoite pysyy samana kuin aikaisemmin, mutta lähetettävän pyynnön tyyppi on put jolloin se käsitellään oikeassa paikassa palvelimella ja tarvittavien tietojen päivitys voidaan suorittaa.

Router.delete (Delete)

Delete on tarkoitettu tietojen poistamiseen. Vastaavalla tavalla kuin aikaisemmissa kohdissa palvelimelle lähetettävä osoite pysyy samana, mutta lähetettävän pyynnön tyyppi on tässä tapauksessa delete. Esimerkiksi pääkäyttäjä voisi olla halukas poistamaan tunnuksen jota ei enää käytetä. Tässä tapauksessa deletessä voitaisiin käsitellä asiakasohjelman lähettämä pyyntö ja poistaa käyttäjätunnus.

3.4.2 Skaalautuvuus

Node.js:llä skaalautuvien sovellusten toteuttaminen ei ole aivan itsestään selvää, koska JavaScript on toteutettu toimimaan yhdessä säikeessä. Tämä estää hyödyntämästä nykyaikaisia usean ytimen suorittimia. Sovelluksen toteutusvaiheessa voidaan miettiä haetaanko staattinen sisältö esimerkiksi erilliseltä palvelimelta, joka tarkoitettu pelkästään sisällön säilyttämiseen josta se voidaan noutaa sovellukseen (CDN) jättäen sovel-

lus huolehtimaan vain dynaamisen sisällön tarjoamisesta asiakasohjelmalle. Vaihtoehtoisesti toteutusta voidaan suorittaa useammassa säikeessä käyttäen esimerkiksi cluster-moduulia. Tämä mahdollistaa usean samanaikaisen node.js-prosessin käynnistämisen jotka jakavat yhteiset resurssit. [19, s. 249.]

Usean samanaikaisen prosessin käynnistäminen yhdellä tietokoneella tuo rajallisen hyödyn, koska jossain vaiheessa tietokoneen resurssit tulevat vastaan. Yhden tietokoneen sijasta kuorma on mahdollista hajauttaa useammalle tietokoneelle. Usean tietokoneen käyttäminen vaatii erillisen http-proxy-moduulin käyttöä.

3.4.3 Skaalautuvuus yhdellä tietokoneella

Cluster-moduuli mahdollistaa yksittäisen sovelluksen suorittamisen useaan kertaan saman aikaisesti. Sovellukset ovat toisistaan riippumattomia, mutta toimivat samassa portissa joka mahdollistaa kuorman jakamisen sovellusten kesken. Cluster-moduulia käytettäessä voidaan käynnistää niin monta yksittäistä sovellusta kuin käyttäjä määrittää, mutta suorituskyvyn kannalta ei ole suositeltavaa käynnistää kuin käytettävissä olevien ytimien verran sovelluksia. [19, s. 249.]

Aikaisemmin luvussa 3.5, kuvassa 14 esitettiin yksinkertainen http-palvelimen toteutus. Tätä toteutusta voidaan jatkaa ja käynnistää se toimimaan useampana sovelluksena cluster-moduulin avulla. Kuvassa 16 jatketaan aikaisempaa koodiesimerkkiä. Kuvassa 16 otetaan käyttöön cluster-moduuli ja selvitetään järjestelmän käytössä olevien ytimien määrä. Tämän jälkeen fork()-metodilla käynnistetään ytimien määrää vastaava määrä rinnakkaisia node.js-sovelluksia.

```
var http = require("http"),
    port = 8000,
    cluster = require("cluster"),
    cores = require("os").cpus().length;

if(cluster.isMaster) {
  for(var i = 0; i < cores; i++) {
    console.log("started process " + i);
    cluster.fork();
  }
}else {
  http.createServer(function(req, res) {
    console.log("node.js is running on port " + port);
    }).listen(port);
}
```

Kuva 16. Node.js cluster toteutus

Käytössäni on tietokone jossa on 4 ydintä. Kuvassa 17 suoritetaan aikaisempi koodi-esimerkki. Kuvassa 17 havaitaan kuvan 16 toistosilmukan suorittavan yhteensä 4 säiettä 0 – 3.

```
markuss-mbp-2:examples markus$ node cluster.js
started process 0
started process 1
started process 2
started process 3
```

Kuva 17. Node.js cluster-esimerkin suoritus.

Varmistetaan vielä, että sovelluksia on järjestelmässä suoritettavana 4 kappaletta, jota on havainnollistettu kuvassa 18. Kuvaa tutkimalla havaitaan node.js:n (/usr/local/bin/node) suorittavan cluster.js-tiedostoa neljällä eri prosessi-idellä 4460-4463.

```
501  4460 ttys003    0:00.10 /usr/local/bin/node /Users/markus/nodejs/examples/cluster.js
501  4461 ttys003    0:00.10 /usr/local/bin/node /Users/markus/nodejs/examples/cluster.js
501  4462 ttys003    0:00.11 /usr/local/bin/node /Users/markus/nodejs/examples/cluster.js
501  4463 ttys003    0:00.11 /usr/local/bin/node /Users/markus/nodejs/examples/cluster.js
```

Kuva 18. suoritettavien sovellusten varmistus

3.5 Modernit visualisointikirjastot

Luvussa 1 käsiteltiin, miten tietoa kerätään kasvavissa määrin sekä siitä, miten suuria tietomääriä on vaikea hallita ja ymmärtää. Tiedon visualisoinnilla pyritään esittämään tietoa tavalla jota on helpompi ymmärtää. Tiedon visualisointi ei ole yksinkertainen työväline vaan hyödyllisen tiedon visualisoinnin takia on tarpeellista ymmärtää tietoa, mistä visualisointi on tuotettu ja millä tapaa visualisointi on toteutettu. Tiedon keräämiseen yleensä liittyy tutkimustyötä, matematiikkaa ja tilastotiedettä. [21, s. 10.]

Johdannossa mainittiin TwitterSensor-palvelun olevan tutkimusalusta jonka tarkoituksena on helpottaa Big Dataan liittyvää tutkimustyötä. Tietoa halutaan visualisoida ja samalla käyttökokemuksen halutaan pysyvän mahdollisimman hyvänä. Visualisointiin on tästä syystä käytetty nykyaikaisia javascript-kirjastoja. Tällä saadaan siirrettyä palvelimen tekemää työtä asiakasohjelmalle, mikä mahdollistaa paremman käyttökokemuksen.

3.5.1 D3

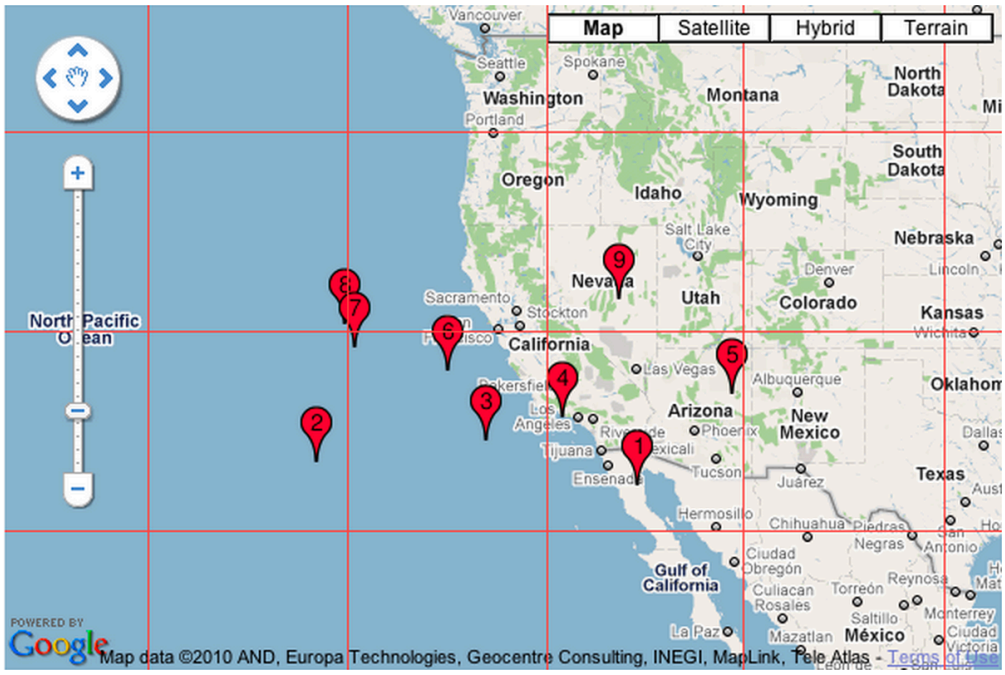
D3 on JavaScript-kirjasto, joka mahdollistaa HTML-,SVG- ja Canvas-elementtien manipuloinnin. Suurimpana hyötynä voidaan pitää toteutustapaa joka mahdollistaa olemassa olevien teknologioiden käytön ilman tarvetta opetella tapaa, jolla voidaan luoda kaavioita. [22.]

D3 ei ole tarkoitettu tarjoamaan kaikkia mahdollisia ratkaisua valmiina, vaan se tarjoaa ratkaisun tehokkaaseen DOM-elementtien muokkaukseen ja tarjoamalla kehittäjälle vapauden muokata kaavioita käyttämällä standardisoituja HTML5-, CSS3- ja SVG-tekniikoita. [22.]

3.5.2 Google Maps-API

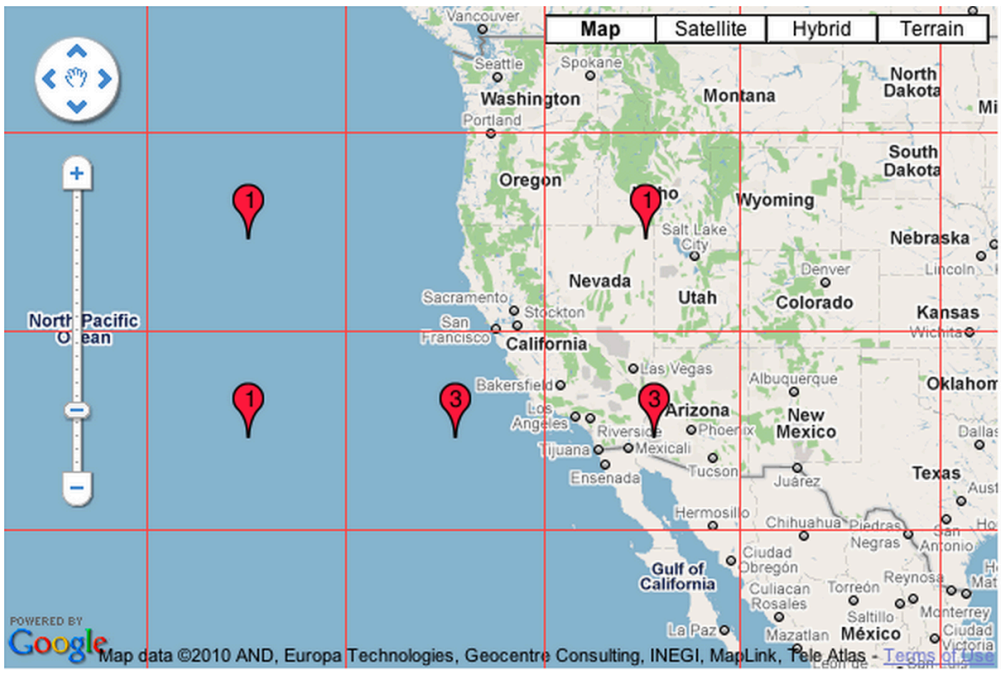
Google tarjoaa useita eri rajapintoja kehittäjien käyttöön, joista eräs on Maps-API, joka tarjoaa Google Mapsista tutut ominaisuudet kehittäjien käyttöön. Yksi mielenkiintoinen Googlen tarjoamista ratkaisuista on ruudukointiin perustuva ryvästys. Tällä tarkoitetaan ilmiötä jossa kartta jaetaan useaan neliöön, joiden sisällä olevat kohteet kauempaa katsottaessa kasataan yhteen ja käyttäjälle näytetään kohteiden yhteenlaskettu luku-

määrä. Kuvassa 19 on havainnollistettu näkymää ilman ryvästystä, jolloin kartalla havaitaan 9 erillistä kohdetta. [23.]



Kuva 19. Google maps – ryvästys ei käytössä. [23.]

Kuvassa 20 havainnollistetaan kuvaa 19, jolloin siihen on lisätty ryvästys käyttöön. Tarkastelemalla kuvaa 20 voidaan huomata miten tietyillä neliönmuotoisilla alueilla sijaitsevat punaiset merkit on kasattu yhteen. Esimerkkinä voidaan ottaa kuvan 19 merkit 1, 4 ja 5 jolloin kuvassa 19 näkyy vain yksi punainen merkki, jossa on arvona 3. Tämä tarkoittaa, että kyseisen neliön alueelta on yhdistetty 3 merkkiä.



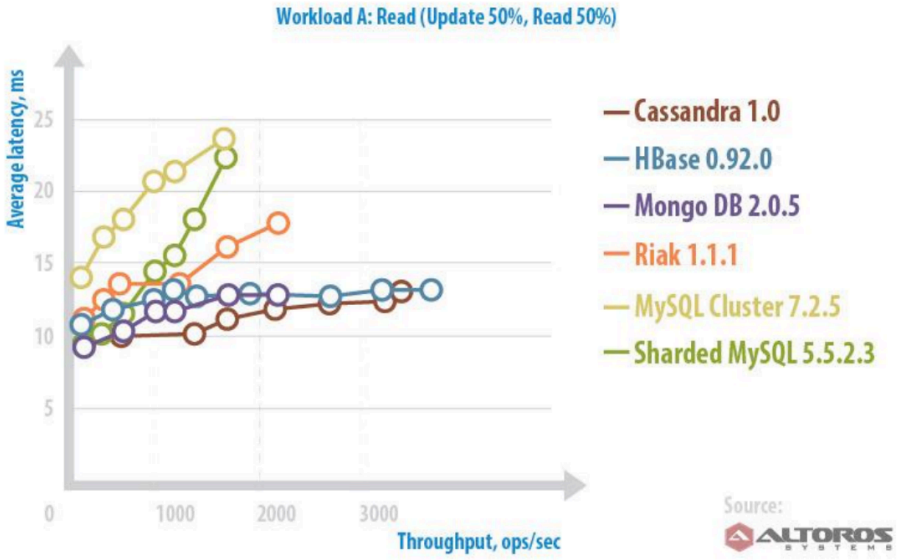
Kuva 20. Google maps - ryvästys käytössä. [23.]

Google Maps-API:a on mahdollista käyttää sekä palvelin- että asiakasohjelman toteutukseen. Käytettävä toteutustapa kannattaa valita sen mukaan miten paljon asiakasohjelmalle halutaan näyttää kohteita kartalla. Huolimatta v3 JavaScript API:n nopeudesta jossain tapauksessa tiedot kannattaa käsitellä palvelimella, josta ne voidaan tarjota asiakasohjelmalle. [23.]

4 Modernit tietokantaratkaisut

Aikaisemmin mainittiin kasvavan tiedon määrän ja monimuotoisuuden aiheuttavan ongelmia tiedon tallentamisessa ja käsittelyssä. Tavallinen relaatiotietokanta soveltuu erittäin hyvin järjestetyn ja yksiselitteisen tiedon, kuten asiakastietokantojen, lukemiseen. Relaatiotietokannat pystyvät skaalautumaan vastaamaan todella suurta tarvetta. Esimerkiksi Facebookilla on 1,32 miljardia käyttäjää [24.] ja osa sen toiminnallisuudesta on toteutettu relaatiotietokannoilla. [25.]

Web-sovellusten kasvaessa ja toimintojen lisääntyessä huomattiin, ettei perinteinen relaatiotietokanta pysty tarjoamaan riittävää suorituskykyä uusien sovellusten tarpeisiin. Tietokantakyselyiden sisältäessä muitakin kuin luku-operaatioita voidaan huomata kuvaa 21 katsomalla relaatiotietokantojen aiheuttaman viiveen lähes kaksinkertaistuvan yhtäaikaisten kyselyiden ollessa noin 2000 kyselyä / sekunti.



Kuva 21. MySQL ja NoSQL päivitys / kirjoituksen aiheuttama viive. [26.]

Nykyään halutaan tallentaa jäsentämätöntä ja sekalaista tietoa. Esimerkkeinä voidaan mainita sosiaalisen median tuottama data, web-sivujen sisältö ja vastaavat. Relaatiotietokantaa ei ole suunniteltu tiedolle, josta nykyään ollaan kasvavassa määrin kiinnostuttu. Tästä johtuen on kehitetty vaihtoehtoisia tietokantoja, joiden tarkoitus on vastata muuttuvaan tarpeeseen. Vaihtoehtoiset tietokannat tallentavat datan eri muodossa kuin tavalliset relaatiotietokannat ja myöhemmin tämänlaisia tietokantoja alettiin kutsua nimellä NoSQL-tietokanta.

4.1 NoSQL-tietokannat

NoSQL-tietokannan rakenne on suunniteltu yksinkertaisemmaksi kuin relaatiotietokannan. Markkinoilla on useita eri NoSQL-toteutuksia, joiden tiedontallennusmallit eroavat toisistaan ja jotka soveltuvat eri käyttötarkoituksiin. NoSQL:llä ei tarkoiteta vain yhtä ohjelmaa tai teknologiaa vaan se voi tarkoittaa teknologioita jotka yhdessä mahdollistavat tiedon tallennuksen ja muokkaamisen. [27, s. 4.]

Dokumenttitietokannat

Jokaisella avaimella on parina monimutkainen tietorakenne, jota kutsutaan dokumentiksi. Dokumentit voivat sisältää useita eri avain-arvo-pareja, avain-taulu-pareja tai sisäkkäisiä dokumentteja. [28.]

Avain-arvo-varastot

Jokaisella tietokantaan tallennetulla oliolla on tarkennus tai ”avain” olion arvon yhteydessä. [28.]

4.1.1 Laajat sarakevarastot

Laajat sarakevarastot voivat sisältää useita arvoja samassa solussa eikä sarakevarastoja tallenneta taulukohtaisesti vaan sarakevarasto kerrallaan. Tämä mahdollistaa yhden taulun jakamisen useammalle tietokoneelle. Sarakevarastoissa käytetään rivikohtaista avainta, joka identifioi rivit sarakevarastossa. [27, s. 77.]

4.1.2 Skaalautuvuus

Tietorakenteen ansiosta NoSQL-tietokannat skaalautuvat yleensä horisontaalisesti, mikäli tietokanta rakentuu useammasta palvelimesta. Horisontaalisesti skaalautuvissa järjestelmissä kuorma yleensä jaetaan lisäämällä järjestelmään uusi palvelin hallitsemaan lisääntynyttä kuormaa. Kuorman lisääntyessä se pyritään jakamaan ennemmin uudelle tietokannalle kuin kuormittamalla käytössä olevia resursseja lisää. Tiedon jakaminen usealle eri koneelle on huomattavasti kustannustehokkaampaa kuin lisätä yksittäisen tietokoneen resursseja käsitellä pituussuunnassa kasvavaa tietokantaa. Tiedon ollessa jaettuna usealle eri koneelle sen käsitteleminen onnistuu rinnakkain, mikä tekee tietokannan käsittelystä nopeampaa, koska yksittäisen koneen ei tarvitse välittää muiden koneiden käsittelemästä datasta. NoSQL-tietokannat ovat löytäneet aseman suurten yritysten kuten Googlen, Facebookin, ja eBayn palvelinjärjestelmissä, jotka voivat kattaa tuhansia tai jopa satoja tuhansia palvelimia. [27, s. 9.]

4.2 MongoDB

MongoDB on korkean suorituskyvyn omaava helposti skaalautuva dokumenttitietokanta. MongoDB sisältää Aggregation-ohjelmistokehyksen mikä on tarkoitettu prosessoimaan tietoa ennen kuin tulokset palautetaan käyttäjälle. Lisäksi MongoDB:ssa on mahdollista käyttää MapReduce-ohjelmointimallia prosessoimaan tietoa ennen kuin se palautetaan käyttäjälle tai asiakasohjelmalle. [28.]

4.2.1 Aggregation-ohjelmistokehys

MongoDB tarjoaa valmiiksi toteutettuja työkaluja, minkä tarkoituksena on mahdollistaa monimutkaisempien kyselyiden toteutus. Aggregationin avulla kyselyn tuloksia voidaan prosessoida ennen kuin ne palautetaan asiakasohjelmalle. Yksinkertaisia esimerkkejä aggregation-ohjelmistokehyksen tarjoamista työkaluista ovat count ja distinct. Count palauttaa kokoelmassa olevien dokumenttien määrän ja distinct palauttaa valitun avaimen perusteella massasta eroavat tietueet. [29, s. 81.]

Kehittyneempiä aggregation-työkaluja ovat esimerkiksi group ja finalizer. Group mahdollistaa monimutkaisempien hakujen toteuttamisen, missä välissä voidaan luoda uusia ryhmiä itse valitun avaimen perusteella ja tähän ryhmään voidaan suorittaa aggerationin tarjoamia operaatioita. Finalizer-työkalua voidaan käyttää minimoimaan tietokannasta käyttäjälle lähetettävän tiedon määrää, mikä on hyödyllinen esimerkiksi group-työkalun jäljiltä olevan vastauksen minimoimiseen ennen asiakasohjelmalle lähettämistä.[29, s. 82-84.]

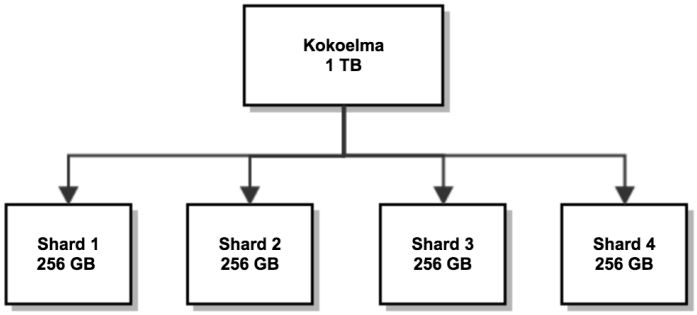
4.2.2 MapReduce-ohjelmointimalli

MapReduce-ohjelmointimalli on tarkoitettu suurien data määrien prosessointiin, jonka tuloksena saadaan tietoa johon on nopea suorittaa kyselyitä. MapRecude on hidas verrattuna aggregation-ohjelmistokehyksen tarjoamiin työkaluihin, mutta sen suorittama työ on helposti hajautettavissa useammalle tietokoneelle. MapReducea ei ole tarkoitettu reaaliaikaisten kyselyiden suorittamiseen siihen kuluvan ajan takia. MongoDB mahdollistaa useiden eri funktioiden yhdistämisen MapReducen kanssa, tehden siitä hyvin käytännöllisen työkalun esimerkiksi tiedon esikäsittelyyn. MapReduce-ohjelmointimalliin yleiseen toimintaperiaatteeseen tutustutaan myöhemmin luvussa 5.1.3 enemmän. [29, s. 86.]

4.2.3 Skaalautuvuus

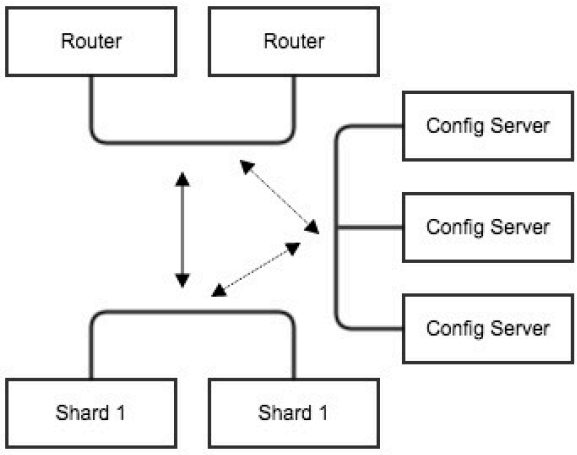
Johdannossa käsiteltiin sitä, miten kasvava tiedon määrä aiheuttaa ongelmia. MongoDB:lla ongelmaa voidaan lähestyä jakamalla kuorma useamman palvelimen kesken, mitä kutsutaan termillä Sharding, joka vähentää yksittäisen jaetun kokoelman kokoa ja siihen kohdistuvan työn määrää kun tietokannasta haetaan tuloksia. Shard-termillä tarkoitetaan yhtä tai useampaa palvelinta, jotka kuuluvat osaksi suurempaa kokonaisuutta. Mikäli käytössä on useampi palvelin, jokainen palvelin sisältää osan datasta.

Tuotantokäytössä shard on yleensä kopio muualla sijaitsevasta tiedosta, joten tiedosta on aina olemassa kopio, jolloin mahdollisen virhetilanteen sattuessa tietokannassa oleva tieto ei vääristy. Kokoelman hajauttamista on havainnollistettu kuvassa 21, jossa yksi tietokanta tai kokoelma jaetaan neljään pienempään osioon.



Kuva 22. MongoDB-kokoelman hajautus. [28.]

Hajautetun järjestelmän toteuttamiseen tarvitaan yksi tai useampi reititin (Router), jotka ovat mongos-instansseja, jotka sisältävät tiedon tietokannan asetuksista ja käsittelevät asiakasohjelmalta saadut tietokantakyselyt. Lisäksi tarvitaan kolme kappaletta kokoonpanopalvelimia (Config Server) ja kaksi tai useampaa Shardia. Hajautetun järjestelmän eri komponenttien välisiä suhteita havainnollistetaan kuvassa 22. Reitittimet suorittavat luku- ja kirjoitusoperaatiot tietokantoihin. Kokoonpanopalvelimet sisältävät hajautetun järjestelmän meta-tiedot ja siksi niiden on erittäin tärkeää pysyä toimintakuntoisena. Lisäksi kokoonpanopalvelinten tiedon on pysyttävä yhtenäisenä, minkä vuoksi niistä on hyvä pitää varmuuskopioita. Shardit sisältävät osan hajautetun tietokannan tiedosta. [28.]



Kuva 23. Hajautetun toteutuksen komponenttien suhteet. [28.]

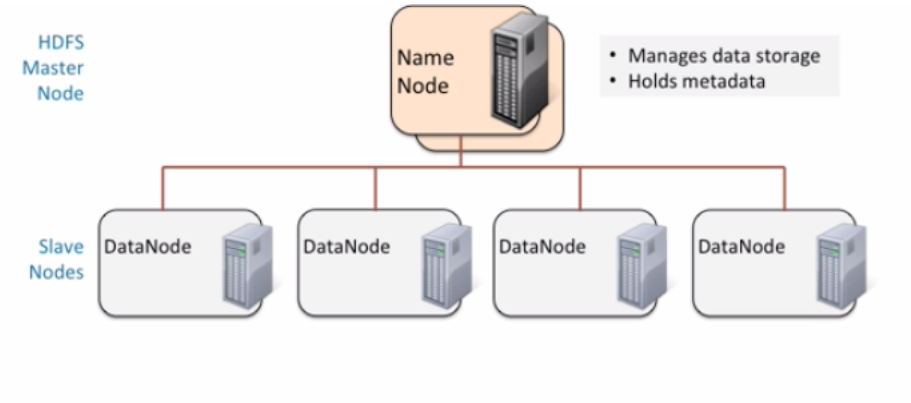
5 Hadoop

Google kohtasi aikanaan ongelman, koska sen hetkinen laskentateho ei riittänyt suorittamaan toimintoja, jotka olivat tarpeellisia heidän toiminnan kannalta. Tämän tuloksena Google alkoi kehittämään teknologiaa, joka mahdollistaisi tiedon ja kuorman jakamisen useampaan pieneen osaan ja lopputuloksena oli Google File System (GFS) [30.] sekä MapReduce-ohjelmointimalli. [31.] Julkaisuista huolimatta Google ei ole halukas julkaisemaan ohjelmistoja open source -lisenssin alaisena eikä se ole ollut missään vaiheessa halukas paljastamaan enempää yksityiskohtia kummastakaan teknologiasta. Myöhemmin näiden julkaisujen pohjalta kehitettiin open source -projektina Apache Hadoop. [32, s. 9-10.]

TwitterSensor-palvelussa ollaan kiinnostuneita hajautetun levyjärjestelmän ja hajautetun tiedonprosessoinnin hyödyntämisestä. Lisäksi ollaan kiinnostuneita mahdollisuudesta prosessoida tietoa reaaliaikaisesti sekä Hadoop-ohjelmistokehyksen tarjoamista joustavista tietokantakyselyistä.

5.1 Apache Hadoop

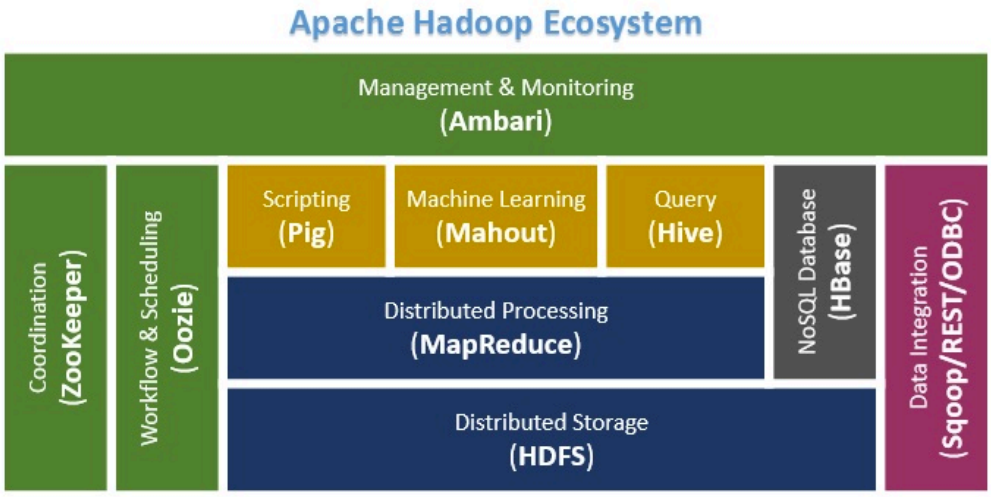
Apache Hadoop on kokoelma open source -ohjelmistoja, jotka yhdessä mahdollistavat suurten datamassojen hajautetun prosessoinnin ja hallitsemisen yksinkertaisten ohjelmointimallien avulla. Keskeisimpiä ohjelmiston osia on Hadoop Distributed File System (HDFS) sekä Hadoop MapReduce. Hadoop on suunniteltu skaalautumaan aina yhdestä palvelimesta jopa tuhansiin palvelimiin. Kuvassa 24 on havainnollistettu Hadoop isäntä-orja-arkkitehtuuria (master/slave), jossa isäntäsolmu pitää huolen siitä, miten tiedostojärjestelmän tiedostot ja tehtävät jaetaan orjien kesken. Suurimmassa osassa Apachen sovelluksista on aina isäntäsolmu, jolla on orjia. Kuvassa 24 isäntäsolmussa on toiminnassa HDFS- sekä MapReduce-sovelluskehykset, jotka koordinoivat orjissa käynnissä olevia saman nimisiä sovelluksia.



Kuva 24. Hadoop – Isäntä-orja-arkkitehtuuri. [33.]

5.1.1 Hadoop-ekosysteemi

Ekosysteemin kaksi ydinosaa ovat aikaisemmin mainitut HDFS sekä MapReduce. Näiden osien ympärille on luotu muita projekteja, minkä tarkoitus on parantaa sekä helpottaa skaalautuvien ja hajautettujen toteutusten hallintaa. Liitännäisten ohjelmointiympäristöjen kokoelmaa on havainnollistettu kuvassa 25, jossa sinisellä alueella HDFS ja MapReduce muodostavat ytimen, jonka ympärille muut osat rakentuvat. Kuvassa 25 vihreät sovelluskehykset on tarkoitettu järjestelmän ylläpitoon, työtehtävien valvomiseen ja suorittamiseen. Oranssit osat ovat sovelluskehyksiä jolla voidaan hakea ohjelmallisesti prosessoida ja varastoida tietoa. Harmaan alueen HBase on korkean suorituskyvyn omaava NoSQL-tietokanta ja punaiset alueen sovelluskehykset on tarkoitettu tarjoamaan asiakasohjelmille pääsy järjestelmän sisältämään dataan.



Kuva 25. Apache Hadoop -ekosysteemi. [34.]

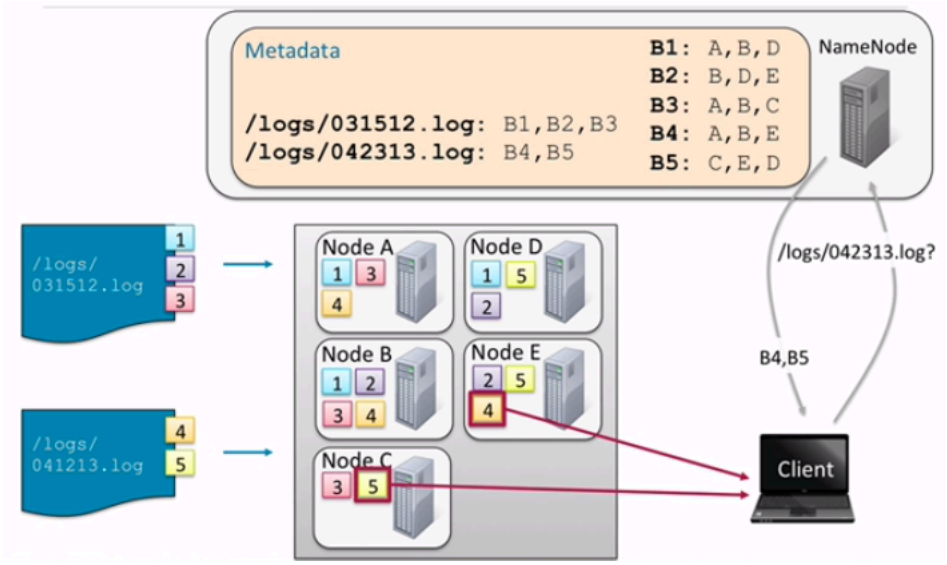
5.1.2 Distributed File System (HDFS)

Hajautettu tiedostojärjestelmä perustuu isäntä-orja-arkkitehtuuriin, jossa järjestelmän nimittäjäsolmu hallitsee tiedostojärjestelmää ja asiakasohjelmien pääsyä tiedostojärjestelmään. Lisäksi järjestelmään kuuluu orjia, joiden tarkoitus on tarjota levytilaa ja laskentatehoa järjestelmään. HDFS-sovellusten tehokkuus perustuu ”kirjoita kerran, lue useasti”-malliin. Tiedosto kirjoitetaan järjestelmään kerran, jonka jälkeen siihen ei ole tarkoitus tehdä muutoksia.

HDFS:n tarjoama hajautettu levytila muodostetaan eri palvelimilla sijaitsevasta levytilasta, joka näyttää käyttäjälle normaalilta tiedostojärjestelmältä. Levytilaan voidaan suorittaa kirjoitus- ja lukuoperaatioita samalla tavalla kuin mihin tahansa normaalille levyosiolle. Levyjärjestelmä mahdollistaa suurien tiedostojen kirjoitus- ja lukutoiminnot viipaloimalla datan yleensä 64, 128 tai 256 megatavuun ja samalla data tallennetaan useaan paikkaan. Viipaloidun datan koko on suuri verrattuna levyn viipaleen kokoon, tällä tavalla pystytään vähentämään verkon välistä liikennettä.

Datan tallennus useaan paikkaan tekee järjestelmästä tehokkaan, koska hajautettavan datan siirtämiseen ei tarvitse käyttää resursseja vaan resurssit voidaan kohdistaa suoraan suositettavaan toimenpiteeseen. Vastaavasta syystä järjestelmä on myös vikasietoinen ja pystyy palautumaan vikatilanteista ilman käyttäjän välitöntä toimenpidettä. Vikatilanteen sattuessa datasta löytyy aina kopio muualta, eli mikäli yksi järjestelmän osa hajoaa, se voidaan korvata uudella vastaavalla ja järjestelmä osaa kopioida datan uudelle palvelimelle. [35, s. 3.]

HDFS-arkkitehtuuria on havainnollistettu kuvassa 26, jossa nähdään oikealla asiakasohjelma (Client application), joka kommunikoi nimittäjäsolmun (NameNode) ja tietosolmujen (DataNode) kanssa. Nimittäjäsolmulla on tieto siitä missä tietosolmuissa yksittäisen tiedoston osat on hajautettu. Tästä johtuen asiakasohjelma kommunikoi nimittäjäsolmun kanssa saadakseen hajautetun tiedon sijainnin, minkä jälkeen tieto pystytään hakemaan tietosolmuista. Kuvaa 26 katsomalla voidaan havaita asiakasohjelman haluavan tiedoston joka koostuu osista 4 ja 5. Asiakasohjelma kysyy tiedon nimittäjäsolmulta, jonka jälkeen tiedosto pystytään hakemaan levyltä.

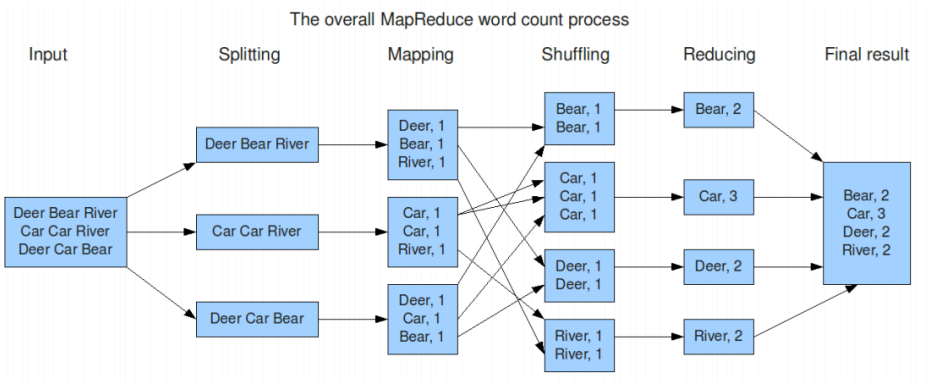


Kuva 26. HDFS-arkkitehtuurin toimintamalli. [33.]

5.1.3 MapReduce

Ohjelmointikehys / ohjelmointimalli mahdollistaa tiedostojen tai datan hajautetun käsittelyn käyttäen operaation toteuttamiseen useita palvelimia. MapReduce toimii yhdessä HDFS:n kanssa, eli se perustuu myös isäntä-orja-arkkitehtuuriin ja on suunniteltu suurien tietomäärien työstämiseen, missä normaaleilla tietokoneilla menee jopa päiviä. Ohjelmointimalli helpottaa hajautettua prosessointia tarjoamalla korkeamman tason abstraktiota sen toteuttamiseen, joten käyttäjän ei tarvitse tietää yksityiskohtia siitä miten tai minne käsiteltävä data hajautetaan järjestelmässä.

MapReducen tarkoituksena on jakaa käsiteltävä aineisto useampaan paikalliseen kopioon ilman, että alkuperäiseen aineistoon tehdään muutoksia. Kun jokainen solmu on käsitellyt ja järjestänyt sille annetun aineiston, tulos lähetetään pääsolmulle. Kun pääsolmu on saanut tulokset yksittäisiltä solmuilta, se yhdistää aineistot yhdeksi uudeksi tulokseksi. MapReduce-operaation toimintaa on havainnollistettu kuvassa 27. Kuvaa katsomalla huomataan miten ensimmäisessä vaiheessa aineisto jaetaan useaan eri osaan, jonka jälkeen aineisto käsitellään ja järjestetään, jonka jälkeen poistetaan ylimääräiset tiedot ja pääsolmu palauttaa tuloksen asiakasohjelmalle.



Kuva 27. MapReduce-esimerkki. [36.]

5.1.4 MapReduce ohjelmointimalli

Nimensä mukaisesti kyseinen toiminto pystytään jakamaan kahteen ylemmän tason toimintaan. Map-toiminta yleisesti lataa, jäsentää, muuntaa ja suodattaa dataa. Reduce-toiminta pitää huolen Mapperilta saaduista osajoukoista keräämällä datan yhteen ja jäsentämällä tiedon uudelleen. [32, s. 6.]

5.1.5 Hive

Hive on tiedon varastointiin tarkoitettu ohjelmointikehys, jonka on kehittänyt Facebook. Hive tarjoaa käyttäjälle relaatiotietokannoista tutun SQL-kyselykielen kaltaisen kielen, HQL:n. Käyttäjälle ei tarjota aivan korkean tason toteutusta, joten käyttäjän täytyy pitää itse huoli kyselyiden oikeasta rakenteesta. On hyvä huomioida, että Hive kuten muutkin Hadoopin osat on suunniteltu suurien tietomäärien hallinnoimiseen, minkä takia yksinkertaisessa kyselyssä Hive-kyselykielellä voi mennä huomattavasti pidempi aika kuin relaatiotietokannassa vastaavalla kyselyllä, eikä sitä ole tarkoitettu korvaamaan jo käytössä olevia relaatiotietokantoja. [32, s. 12.]

5.1.6 Pig

Pig on Yahoon kehittämä korkeamman tason skriptikieli, jonka tarkoituksena on vähentää käyttäjän kuluttamaa aikaa MapReduce-ohjelmien toteuttamiseen. Tämä jättää käyttäjälle enemmän aikaa datan tulosten analysoimiseen ja hyödyntämiseen. Pig rakentuu kahdesta komponentista, joista toinen on PigLatin-ohjelmointikieli ja toinen on ympäristö, jossa PigLatin-ohjelmia ajetaan. Näiden välinen suhde on saman tyyppinen

kuin Java-ohjelmointikielellä ja Java Virtuaalikoneella (JVM), jossa Java-ohjelmat suoritetaan. PigLatin ohjelmat käännetään suoritusympäristössä ryhmäksi MapReduce-tehtäviä ennen suorittamista. [32, s. 66.]

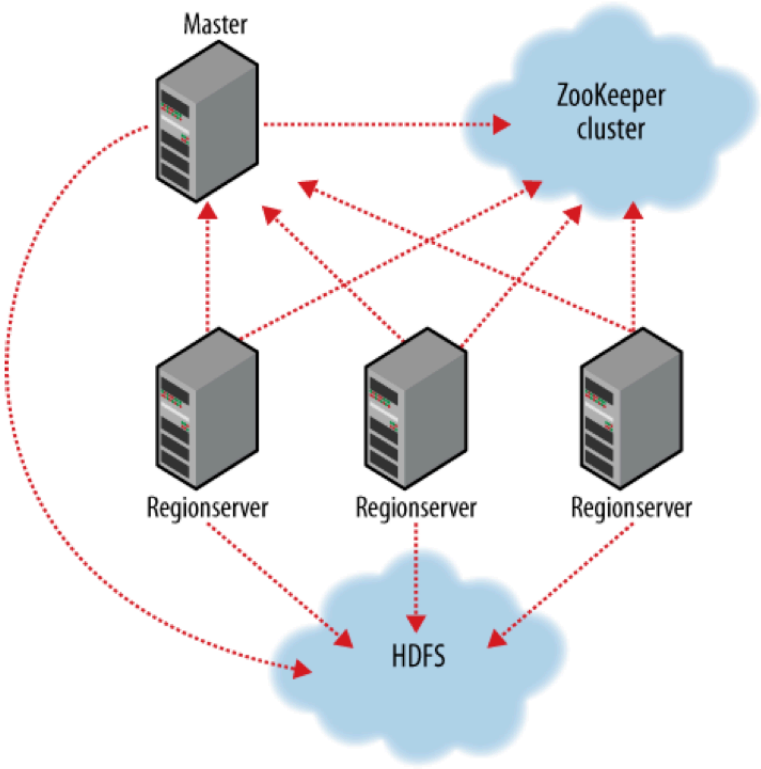
5.1.7 Zookeeper

Toisin kuin normaaleissa tiedostojärjestelmissä, hajautetuissa järjestelmissä tarvitaan tieto siitä, mikä järjestelmän osa lukee tai kirjoittaa mihinkin tiedostoon ja milloin. Tätä varten on olemassa Zookeeper-ohjelmistokehys, jonka tehtävänä on toimia koordinaattorina ja valvoa, ettei järjestelmässä tapahdu samanaikaisia samojen tiedostojen lukua tai kirjoitusta. Lisäksi Zookeeper huolehtii järjestelmän asetusten tiedostoista sekä hajautettujen tiedostojen pitämisestä ajan tasalla keskenään. Zookeeperin kehitti Yahoo! helpottamaan oman järjestelmänsä ylläpitoa. [37.]

5.1.8 HBase NoSQL-tietokanta

HBase on laaja sarakevarasto, jota voidaan käyttää yksittäisenä järjestelmänä ilman HDFS-tiedostojärjestelmää. Mikäli HBasea käytetään ilman HDFS:ää tietokanta huolehtii itse rakenteestaan ja siitä, mihin tietoa sijoitetaan. HBase on kuitenkin toteutettu Google BigTablen pohjalta ja tästä syystä se pystyy sisältämään todella suuren määrän rivejä ja sarakkeita ja soveltuu hajautettuihin järjestelmiin. Mikäli järjestelmä asennetaan HDFS:n päälle HBase käyttää tällöin samaa arkkitehtuuria, mikä tekee tietokannasta hajautetun ja skaalautuvan.

HBase on suunniteltu tarjoamaan käyttäjälle reaaliaikainen pääsy satunnaiseen tietoon. Mikäli HBasea käytetään osana Hadoop-järjestelmää käyttäjälle ei tarjota suoraa pääsyä kantaan vaan pyynnöt kulkevat Apache ZooKeeper-sovelluskehyksen kautta.[28, 83-85] HBase kuten monet muutkin Hadoopin kanssa toimivat sovelluskehykset käyttävät isäntä-orja-arkkitehtuuria. Zookeeperia käytettäessä tämä tarkoittaa sitä, että Zookeeperin pyynnöt menevät isäntäsolmulle, joka kertoo, missä haluttu tieto sijaitsee, jonka jälkeen Zookeeper pystyy hakemaan halutun tiedon. [32, s. 458-461.] Kyseistä toimintamallia on havainnollistettu kuvassa 28.



Kuva 28. Zookeeper ja HBase. [32.]

5.1.9 Oozie

Oozie on järjestelmä, jonka tehtävänä on valvoa Hadoopissa tapahtuvien töiden kulkua. Ooziella pystytään ajastamaan tehtäviä perustuen tiettyyn kellonaikaan tai siihen, milloin saadaan jonkun toisen prosessin päätyttyä uutta dataa käsiteltäväksi. Oozie rakentuu kahdesta pääjärjestelmästä, joista ensimmäinen on työnkulkua valvova järjestelmä, jonka tehtävänä on tallentaa ja suorittaa ennalta määrättyjä töitä, sekä vahtia töiden kulkua. Toinen on koordinaattori-järjestelmä, joka osaa odottaa sen tarvitseman syötteen valmistumista toiselta prosessilta.

Yksi järjestelmän suurista hyödyistä on, ettei sen tarvitse odottaa syötteen valmistumista ennen tehtävän aloittamista. Tehtävä voidaan aloittaa ja suorittaa niin pitkälle, kunnes tiettyä syötettä tarvitaan, jolloin ohjelma jää odottamaan, mikäli syöte ei ole valmistunut. Mahdollisuuksien mukaan odottaessa suoritetaan prosessin muita osia, jotka eivät riipu syötteestä tai syötteen tuloksesta jota odotetaan toiselta prosessilta. Tällä tavalla tehtävään käytettävä aika pystytään optimoimaan, kun syötteestä riippumattomat osat suoritetaan ilman odottamista. Jokainen työnkulku sisältää alku- ja päätöspis-

teen. Mikäli työ ei saavuta päätöspistettä se tulkitaan epäonnistuneeksi. [32, s. 182-183.]

5.2 Kaupalliset ratkaisut

Apache Hadoop on open source -versio Googlen käyttämästä arkkitehtuurista. Viime vuosina on huomattu, että tiedon tallennuksen ja käsittelyn tarve ei ole vähentymässä, mikä on luonut alalle myös uusia yrityksiä kuten Cloudera, Hortonworks ja R. Näiden yritysten lähtökohtana on parantaa olemassa olevaa Apache Hadoop -ohjelmistokehystä luomalla siihen omia lisäosia tai lisätoiminnallisuutta. Yleisin ja helpoiten erotettava ominaisuus joka vaihtelee eri toteutusten kesken on hallintapaneeli, joka Apache Hadoopissa on Ambari. Yritykset ovat yleensä lisänneet toteutuksiin oman hallintapaneelin, mikä helpottaa kokonaisuuden hallintaa ja valitusta ohjelmistokehyksestä riippuen erilaisuuksia voidaan löytää käytettävästä NoSQL-tietokantaratkaisusta ja valmiiksi asennetuista lisäosista.

6 Teknologioiden testausta

Sovelluksen skaalautuminen riippuu suurimmaksi osaksi palvelin- ja tietokantaratkaisuista. Testausvaiheessa haluttiin demonstroida millaisia hyötyjä eri palvelinpuolen ratkaisuilla ja eri tietokantaratkaisuilla on mahdollista saavuttaa.

6.1 Palvelinpuolen toteutus

Yleinen palvelinpuolen ratkaisu on ollut käyttää Apache http-palvelinohjelmaa, jonka kanssa on käytetty php-ohjelmointikieltä luomaan dynaamisia verkkosivuja asiakasohjelmalle. Testiympäristössä haluttiin vertailla, millaisia eroja voidaan saavuttaa korvaamalla Apache nginx-http- ja proxy-ohjelmistolla ja vaihtamalla php nodejs -toteutukseen.

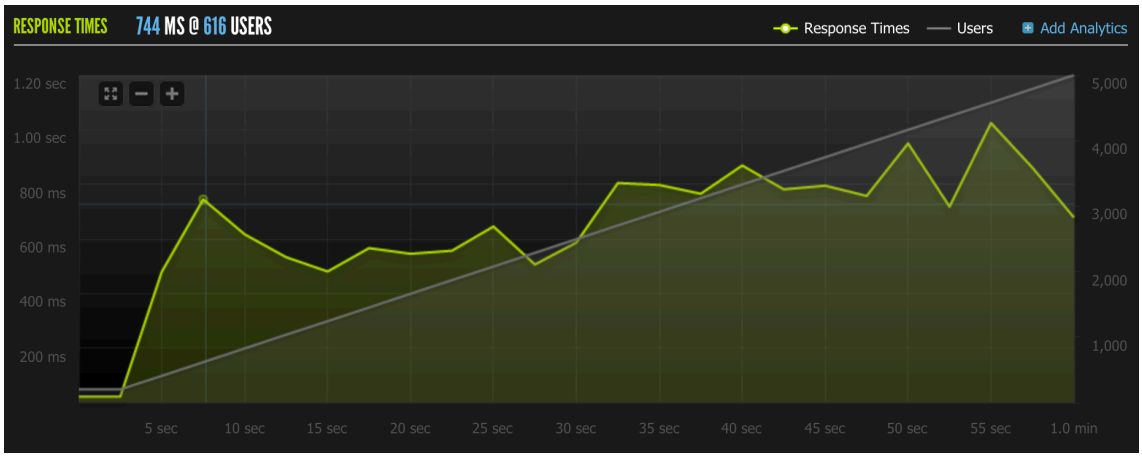
Testit 6.1.1 ja 6.1.2 toteutettiin identtisillä virtuaalipalvelinympäristöillä joissa käytettiin yhtä suorittimen ydintä ja 512 megatavun keskusmuistia. Testissä 6.1.3 käytettiin vastaavaa laiteympäristöä, suorittimen ydinten lukumäärän noustessa kahteen kappaleeseen ja keskusmuistin määrän kasvaessa kahteen gigatavuun.

Palvelimen toimintaa haluttiin tutkia käyttäjien määrää kasvattamalla. Käyttäjien maksimimääräksi asetettiin 5000, jonka jälkeen tutkittiin palvelimen kykyä vastata kutsuihin käyttäjämäärän kasvaessa 1:stä 5000:een käyttäjään yhden minuutin aikana.

6.1.1 Apache ja PHP-ympäristö

Kuvassa 29 havainnollistetaan palvelimen vasteaikaa käyttäjien määrän lisääntyessä. Kuvassa vasemmalla y-akselilla on ilmoitettu palvelimen vasteaika ja oikealla y-akselilla yhtäaikaisten käyttäjien määrä. X-akseli kuvaa kulunutta aikaa ja vihreä kuvaaja kuvaa palvelimen vasteaikaa suhteessa käyttäjien määrään.

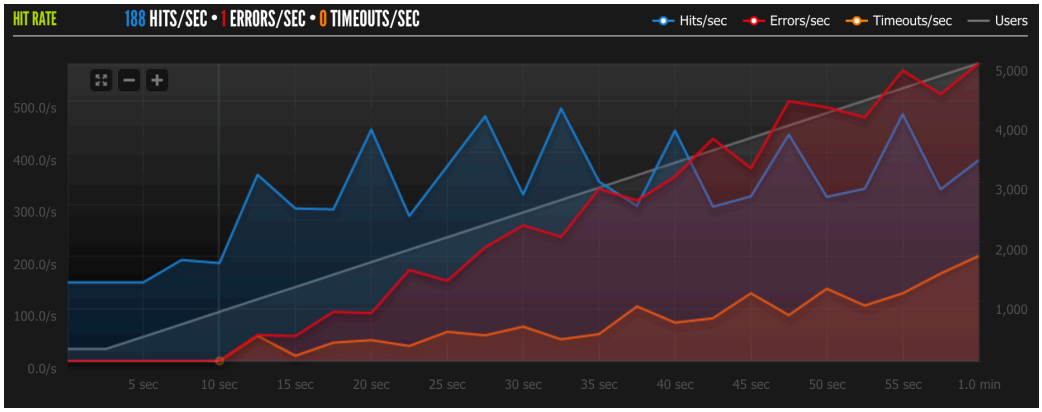
Lähemmästä tarkastelusta voidaan huomata palvelimen vasteajan kasvavan yli 700 millisekuntiin käyttäjien määrän ollessa hieman yli 600. On syytä kiinnittää huomiota siihen, että palvelimen vasteaika on pienempi myöhemmin noin 27 sekunnin kohdalla jolloin yhtäaikaisia käyttäjiä on hieman alle 2300.



Kuva 29. Apache ja PHP - Vasteaika käyttäjien määrän kasvaessa

Kuvassa 30 havainnollistetaan palvelimelle kohdistuvien pyyntöjen määrää käyttäjien lukumärän kasvaessa. Vasen y-akseli kuvaa yhtäaikaisia tapahtumia per sekunti ja oikea y-akseli yhtäaikaisten käyttäjien määrää. X-akseli kuvaa ajanjaksoa 0 – 60 sekuntia ja harmaa väri kuvaa käyttäjien määrän kasvua. Sininen kuvaa yhtäaikaisten pyyntöjen määrää palvelimelle. Punainen kuvaa tapahtuneiden virheiden määrää ja oranssi kuvaa aikakatkaisujen määrää.

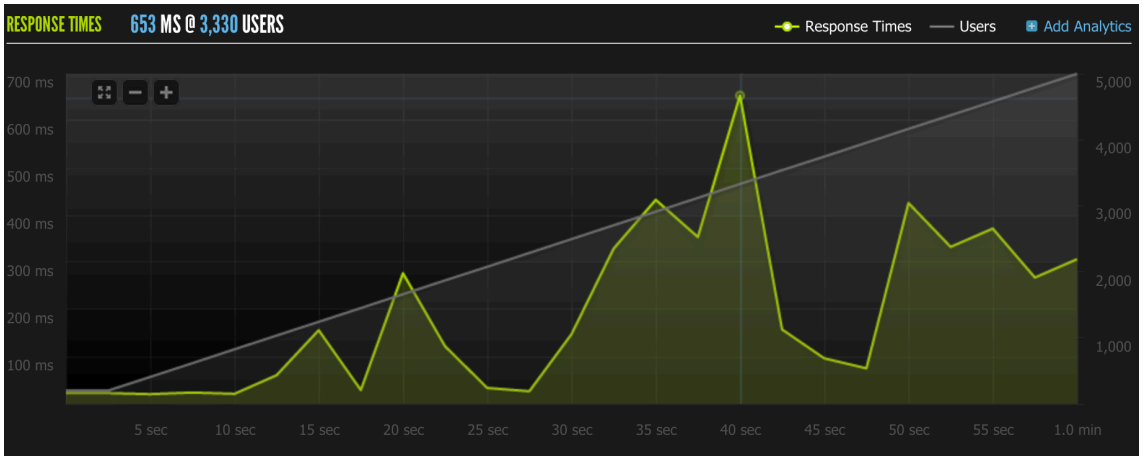
Kuvasta pystytään havaitsemaan punaista viivaa katsomalla, että ensimmäinen asiakasohjelman suorittama pyyntö jää palvelematta käyttäjien määrän ollessa 826, joka nähdään harmaasta viivasta, jos kuvaajaa lähennetään. Ensimmäisen virhetilanteen sattuessa yhtäaikaisten pyyntöjen lukumäärä on 188 kappaletta, joka havaitaan sinisestä viivasta. Täytyy kuitenkin huomioida, että pyyntöjen lukumäärä oli hieman korkeampi ennen ensimmäisen virhetilanteen sattumista, jolloin yhtäaikaisia pyyntöjä oli 194. Voidaan todeta testauksessa käytetyn Apache + PHP-toteutuksen pystyvän palvelemaan hieman alle 200 samanaikaista palvelinkutsua ilman virhetilanteita.



Kuva 30. Apache ja PHP - Pyyntöjen ja virhetilanteiden määrä.

6.1.2 Nginx ja Node.js -ympäristö

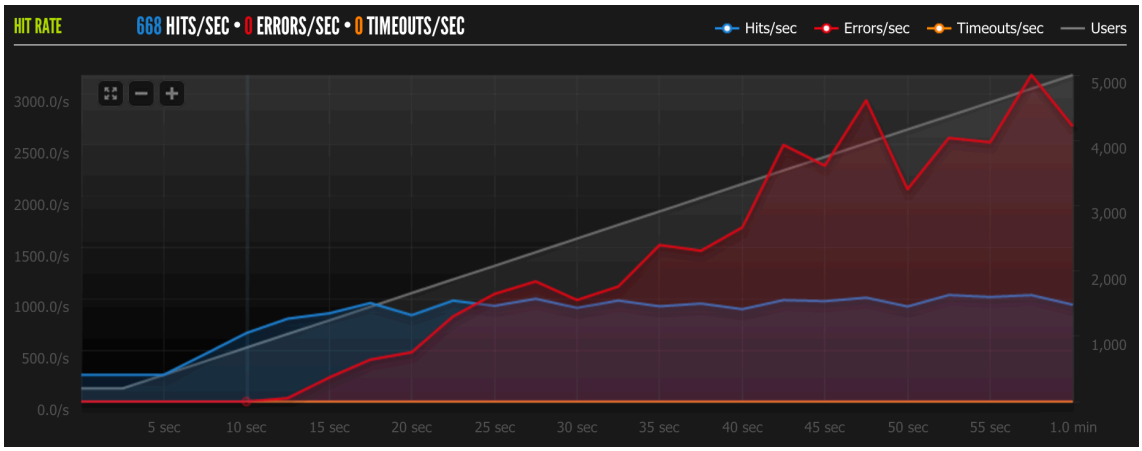
Kuvassa 31 havainnollistetaan Nginx ja Node.js -toteutuksen kykyä vastata asiakasohjelman kutsuihin. Suurimmaksi osaksi vasteaika pysyy alle 500 millisekunnin, mutta nousee hetkellisesti hieman alle 700 millisekuntiin yhtäaikaisten käyttäjien ollessa 3300 kappaletta.



Kuva 31. Nginx + node.js vasteaika käyttäjien määrän kasvaessa.

Kuvassa 32 havainnollistetaan vastaavasti, kuinka paljon yhtäaikaisia asiakasohjelman suorittamia pyyntöjä pystytään palvelemaan käyttäjien määrän lisääntyessä. Yhtäai-

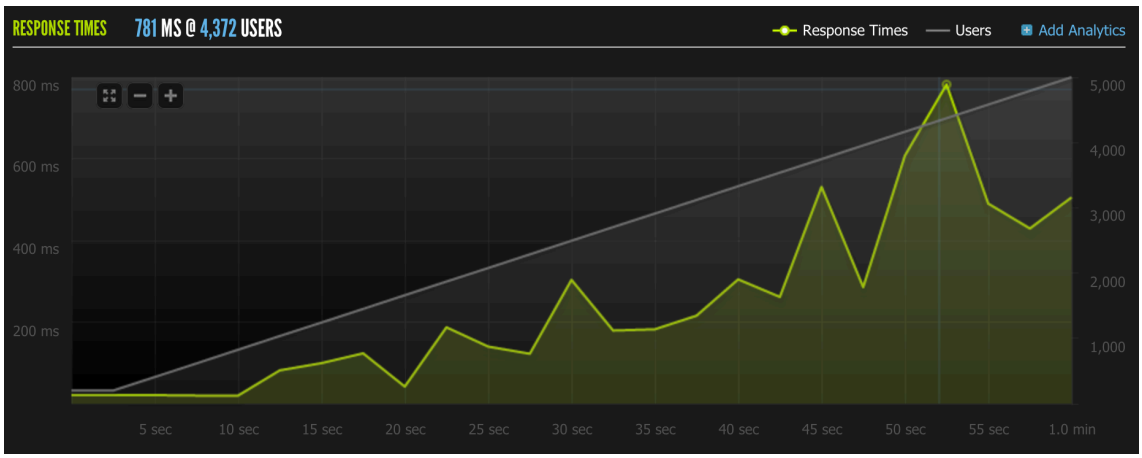
kaisten pyyntöjen ollessa 668 kappaletta ei ole sattunut vielä yhtään virhetilannetta. Ensimmäiset virhetilanteet sattuvat yhtäaikaisten pyyntöjen määrän noustessa 808 kappaleeseen. Näin ollen voidaan todeta Nginx ja Node.js toteutuksen pystyvän palvelemaan hieman alle 800 yhtäaikaista pyyntöä ongelmitta.



Kuva 32. Nginx ja Node.js - Pyyntöjen ja virhetilanteiden määrä

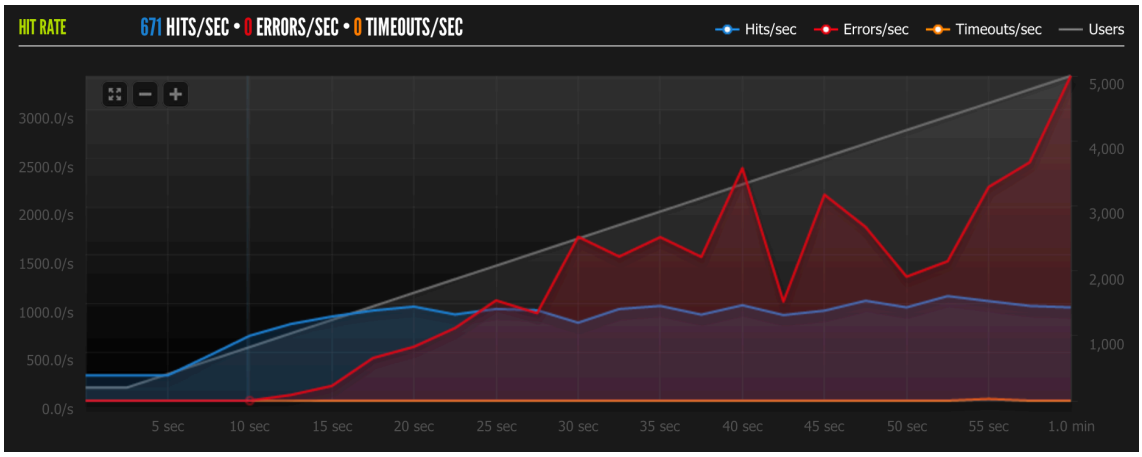
6.1.3 Node.js-skaalautuvuus

Aikaisemmassa luvussa huomattiin, miten potentiaalinen vaihtoehto Node.js on käyttää palvelinpuolen toteutuksessa. Aikaisemmin luvussa 3.5 käsiteltiin Node.js-mahdollisuutta skaalautua, mikäli käytettävissä oleva suoritin tukee useampaa säiettä. Tässä vaiheessa on syytä huomioida, että palvelimen ytimien määrä on tuplaantunut ja muistin määrä nelinkertaistunut aikaisempaan node.js-toteutukseen verrattuna. Tarkastelemalla kuvaa 33 voidaan huomata, ettei Nginx- ja Nodejs-toteutus ole kovin riippuvainen palvelimen resursseista, kun tarkastellaan käyttäjän palvelinkutsuihin vastaamista.



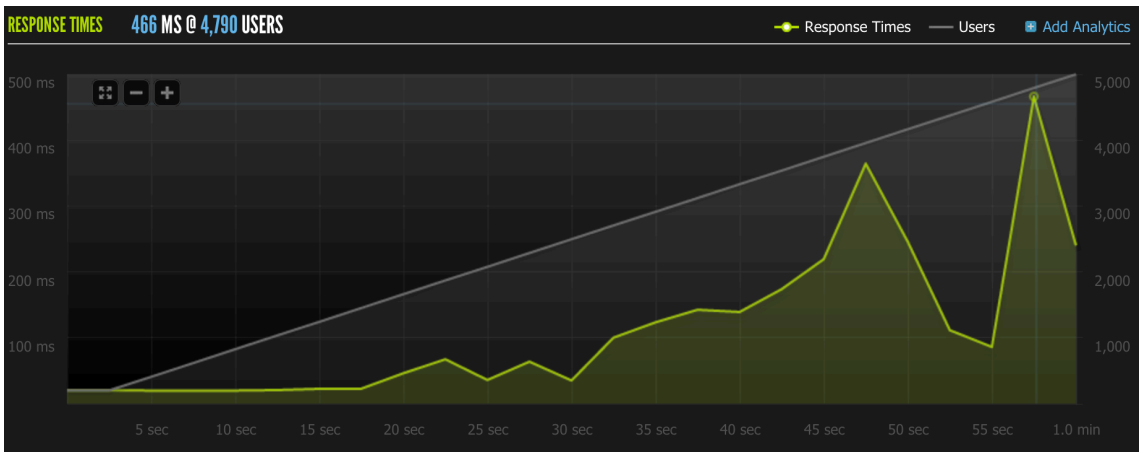
Kuva 33. ginx ja Nodejs - Tehokkaampi palvelin, vasteaika käyttäjien määrän kasvaessa

Kuvassa 34 vastaavasti havainnollistetaan tehokkaamman palvelimen kykyä vastata saapuviin pyyntöihin. Kaaviota tarkkailemalla pystytään havaitsemaan sama ilmiö kuin vasteajan kanssa eli palvelimen kyky vastata kutsuihin pysyy lähes samana.



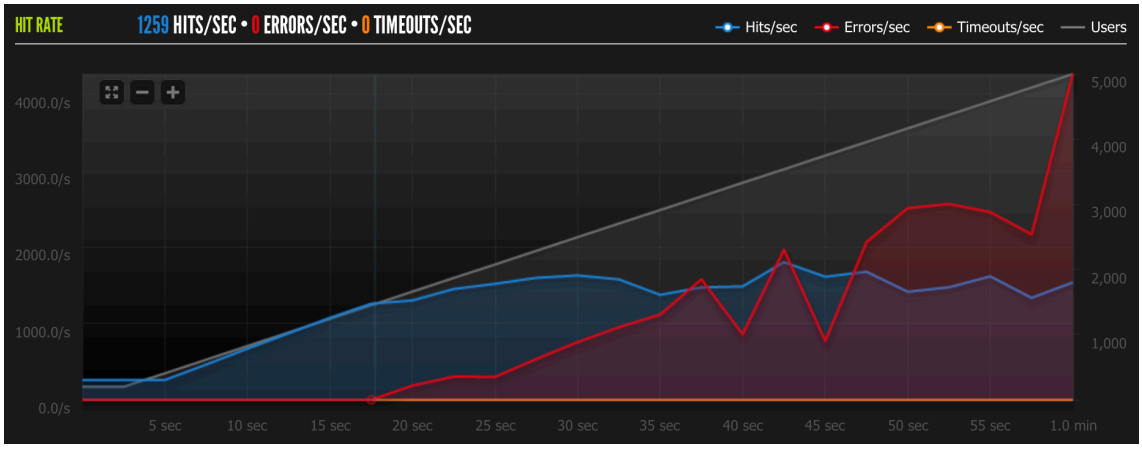
Kuva 34. Nginx ja Nodejs – Tehokkaampi palvelin, kyselyiden ja virheiden määrä

Aikaisemmissa testeissä käytetty node.js sovellus muutettiin käyttämään useampaa ydintä. Kuvassa 35 havainnollistetaan kahden ytimen node.js-sovelluksen vasteaikaa käyttäjien määrän kasvaessa. Kuviota tarkastelemalla voidaan huomata toisen ytimen vaikuttavan huomattavasti siihen, kuinka nopeasti käyttäjien pyyntöihin pystytään vastaamaan. Verrattuna aikaisempaan node.js-toteutukseen vasteaika on pahimmassakin tapauksessa alle 500 millisekuntia vaikka yhtäaikaisten käyttäjien lukumäärä lähenee 5000:ta.



Kuva 35. Nginx ja Node.js -2 ytimen toteutus

Kuvassa 36 havainnollistetaan useamman ytimen vaikutusta pyyntöjen määrään, johon pystytään vastaamaan ongelmitta. Tarkastelemalla kaaviota huomataan onnistuneiden pyyntöjen määrän melkein kaksinkertaistuneen aikaisempaan node.js-toteutukseen verrattuna. Ytimien määrän ja käynnissä olevien nodejs-instanssien tuplaantuessa vaikutus pystytään näkemään suoraan.



Kuva 36. Nginx ja Nodejs – Palveltujen pyyntöjen määrä kahdella ytimellä

Palvelimella, jossa oli 2 ydintä ja 2 gigatavua välimuistia, kokeiltiin vielä Apachen skaalautuvuutta käytössä olevan palvelimen resurssien kasvaessa. Tulokset eivät olleet sellaisia jotka saisivat harkitsemaan Apachen käyttöä Node.js:n sijasta. Suurin asiakasohjelman kokema viive oli 3300 käyttäjän kohdalla 1023 millisekuntia ja yhtäaikaisten palveltujen pyyntöjen määrä vikatilanteen sattuessa oli 315 pyyntöä / sekunti.

Tuloksia tarkastellessa on syytä muistaa testien perustuvan yksinkertaiseen asiakasohjelman pyyntöön saada palvelimelta vastaus. Testi ei huomioi mahdollisten muiden

osa-alueiden vaikutusta asiakasohjelman ja palvelimen väliseen viiveeseen. Verkkosivuilla ja verkkosovelluksissa viivettä saattavat aiheuttaa tietokantakyselyt sekä tiedon prosessointi. Tällä testillä haluttiin demonstroida Node.js ja Apachen kykyä skaalautua kasvavan kysynnän mukaan.

6.2 Tietokantaratkaisu

Olemme olleet pitkään tietoisia siitä, ettei nykyinen käytössä oleva MySQL-tietokanta pysty vastaamaan käytettävissä olevan tiedon määrään, koska tietokannan taulut kasvavat liian suureksi, jolloin tietokannasta hakemisesta tulee hidasta eikä tietokannan tehokas käyttäminen ole mahdollista. Tarkoituksena ei ollut itse lähteä vertailemaan tietokantaratkaisuiden nopeutta perinpohjaisesti, koska järjestelmä haluttiin joka tapauksessa hajauttaa johtuen sen suuruudesta. Olimme tietoisia MongoDB:sta, joka saattaisi tarjota ratkaisun ongelmaan. On olemassa näyttöä MongoDB:n nopeudesta MySQLään verrattuna, missä MongoDB on 1,3-2 kertaa MySQL-tietokantaa nopeampi, mikäli käytössä ei ole tiedon hakua välimuistista (testauksessa käytettiin 200 miljoonaa tietuetta). Kyseinen testi antaa hyvän lähtökohdan sille millaisia hyötyjä MongoDB:n käyttöönottamisella voidaan saavuttaa. Kyseisessä testissä ei oltu kuitenkaan huomioitu MongoDB:n muita mahdollisia hyötyjä. Mikäli käytetään useampaa palvelinta, saadaan käyttöön enemmän prosessointitehoa ja välimuistia. [38.]

Kokeilimme itse, millaista nopeushyötyä hajauttamisella on mahdollista saada. Testauksessa käytetty data sisälsi noin 1,3 miljoonaa tietuetta, joihin suoritettiin kyselyitä käyttäen tarkoituksella säännöllistä lauseketta, jolla nähdään, miten nopea MongoDB on haettaessa tietoa, joka ei ole suoraan indeksoidussa muodossa.

Hakuja suoritettiin 100 kappaletta kahdella eri tavalla, tietokantaan, jota ei ollut ladattu välimuistiin ja tietokantaan, joka oli ladattu välimuistiin. Taulukossa 1 havainnollistetaan tiedon keskimääräistä hakunopeutta, kun tietoa ei haeta välimuistista.

Taulukko 1. Tietokanta haut ilman välimuistia.

Hakusana	Yksittäinen MongoDB-tietokanta	Hajautettu MongoDB-tietokanta
Google	1,304 s	0,449 s
Amazon	1,318 s	0,402 s
Ebay	1,396 s	0,361 s

Microsoft	1,599 s	0,354 s
Twitter	0,963 s	0,682 s

Taulukossa 2 havainnollistetaan hakuaikoja haettaessa samoilla hakusanoilla välimuistista.

Taulukko 2. Tietokantahaut välimuistista.

Hakusana	Yksittäinen MongoDB-tietokanta	Hajautettu MongoDB-tietokanta
Google	1.482 s	0,390 s
Amazon	1,469 s	0,370 s
Ebay	1,419 s	0.357 s
Microsoft	1,364 s	0,368 s
Twitter	0,704 s	0,535 s

Tarkastelemalla taulukkoa 1 voidaan huomata, ettei hajautetusta tietokannasta hakeminen ole välttämättä aina nopeampaa kuin yksittäisestä tietokannasta. Hakutuloksissa huomiota herättää Twitter-hakusana, koska hajautetusta taulusta haettaessa tuloksen saamisessa kesti melkein 8 kertaa niin kauan kuin vastaavassa haussa yksittäisestä tietokannasta. Palvelin, jolla testejä suoritettiin, on kuitenkin aktiivisessa käytössä joten on hyvin mahdollista, että joku toinen prosessi on käyttänyt juuri kyseisellä hetkellä paljon resursseja, mikä voisi selittää ilmiön.

6.3 Asiakasohjelman testausta

PHP-ohjelmointikieli on ollut todella suosittu verkkosivujen ja web-sovellusten kehityksessä aikaisemmin. Aikaisemmin mainittiin tämän tapaisten toteutusten kuitenkin suorittavan sivujen vaihdossa ja hakulomakkeita käytettäessä aina kokonaisen sivun uudelleen hakemisen. Halusimme katsoa miten AngularJS muuttaa asiakasohjelman toimintaa, kun verkkosovellus toteutetaan niin sanottuna Single-Page-Applikaationa. Sovelluksesta oli aikaisemmin olemassa PHP-toteutus, jossa yksittäinen sivulataus vei 1-3 sekuntia, kun huomioidaan, että yleisesti käytössä olevan kaavion mukaan asiakkaan kutsuihin pitäisi pystyä reagoimaan alle sekunnin ja 5-10 sekuntia on kipuraja, minkä jälkeen sovelluksen kanssa luovutetaan.

Yhden sivun sovellukset lataavat ensimmäisellä sivun latauskerralla koko sivuston välimuistiin, minkä jälkeen palvelimelta noudetaan vain muutettavan sivukomponentin toteuttamiseen tarvittavat tiedot.

Aikaisemmin on myös huomioitu, että sovelluksen toiminnassa saavutettu nopeus ei kokonaisuudessaan johdu vain yhdestä osa-alueesta vaan siihen vaikuttavat kokonaisuudessaan palvelinalusta, tietokantatoteutus ja asiakasohjelman toteutus. Testiä varten pidimme tiedon samassa MySQL-tietokannassa, jota oltiin käytetty myös aikaisemmassa toteutuksessa. Asiakasohjelman päässä vaihdettiin siis ohjelmointiympäristö JavaScriptiin ja aikaisemman JPG-kuvan sijasta asiakkaalle näytettävät kaaviot luodaan dynaamisesti JavaScriptillä käyttäen D3js-kirjastoa.

Ensimmäisen sivulatauksen havaittiin kestävän kokonaisuudessaan 3,16 sekuntia. Tämän jälkeen yksittäiset sivunvaihdot eivät suorittaneet ollenkaan hakuja palvelimelle. Seuraavaksi palvelimelta haettiin muutamia yksittäisiä hakusanoja. Tiedon hakemiseen ja kaavion piirtämiseen asiakasohjelmalle kului keskimäärin 100 – 300 millisekuntia.

7 TwitterSensor-palvelun sovellusarkkitehtuurin toteutus

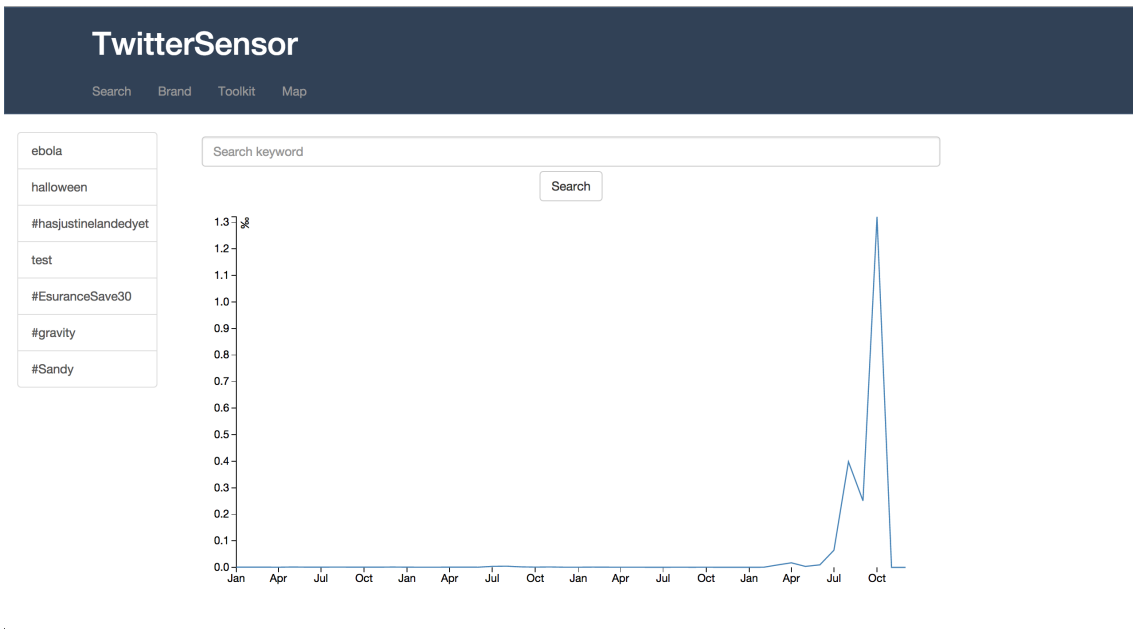
Palvelu oli aikaisemmin toteutettu yleisesti käytössä olevalla tavalla toteuttaa web-palvelin, joka pystyy suorittamaan dynaamisia www-sivuja. Käyttöjärjestelmänä on Linux jossa on käytössä Apache-palvelinohjelmisto. Tietokantaympäristönä toimi MySQL-relaatiotietokanta ja dynaamisten sivujen toteutukseen käytettiin PHP-ohjelmointikieltä.

Lähdin toteuttamaan TwitterSensor-palvelun uutta versiota teoriaosuudessa käsiteltyjen teknologioiden pohjalta eli MEAN-ohjelmistopinolla (MongoDB, Express, AngularJS, Node.js). Kuvassa 37 havainnollistetaan missä palvelun kerroksessa mitäkin teknologioita on hyödynnetty.

Kuva 37. Eri kerroksissa käytetyt teknologiat. Ylimpänä näkymäkerros, keskellä palvelinohjelmakerros ja alimpana tietokantakerros.

Asiakaspuolen ohjelman toteutuksessa käytettiin aikaisemmin käsiteltyä AngularJS-ohjelmointikehystä ja sivuston rakenne toteutettiin modulaarisena. Asiakasohjelma oli jaettu neljään eri näkymään, jotka tarjoavat käyttäjälle erilaisia hakumahdollisuuksia. Sivuston staattinen näkymä, joka kattaa yläpalkin ja sivuston rakenteen, muodostetaan palvelimen puolella, josta se tarjotaan asiakasohjelmalle siten, että asiakasohjelman tehtäväksi jää vaihtaa dynaamista näkymää, joka kattaa eri näkymissä näytettävät tiedot.

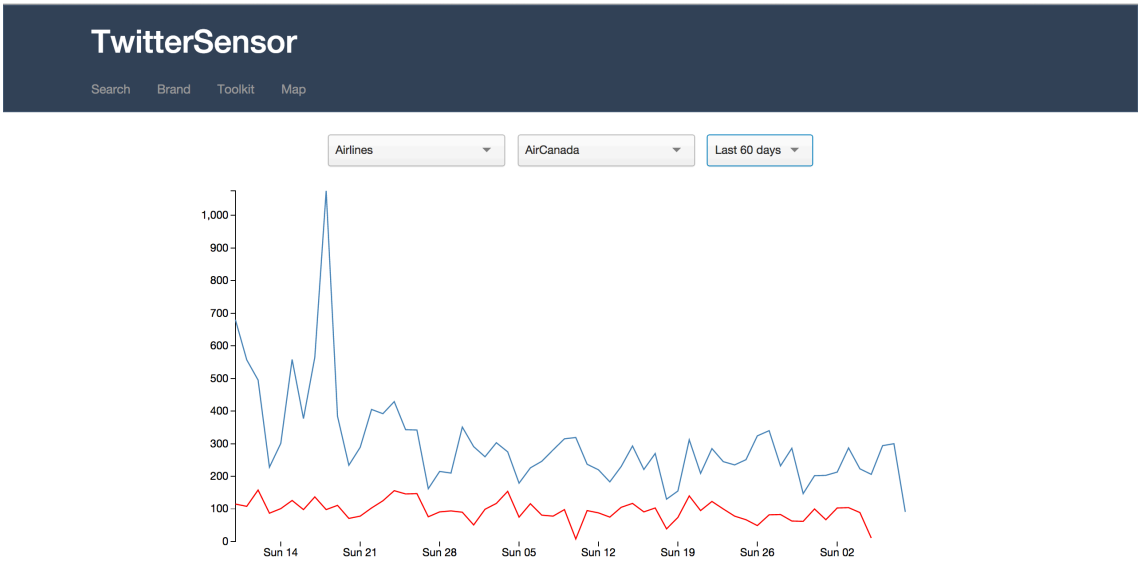
Search-näkymässä käyttäjälle näytetään lista eniten käytetyistä hakusanoista ja hakukenttä jolla voidaan etsiä eri hakusanoja. Hakusanojen tuloksista käyttäjälle näytetään kaavio. Search-näkymän haetuimmat hakusanat sekä käyttäjälle näytettävän kaavion arvojen noutaminen tietokannasta oli toteutettu palvelinsovellukseen toteutettua REST-rajapintaa käyttämällä. Search-näkymää on havainnollistettu kuvassa 38.



Kuva 38. Search-näkymä.

Search-näkymässä hakupainikkeen tai hakusanan painaminen vasemmalta käsitellään ohjaimessa, joka päivittää käyttäjälle näytettävän kaavion, mistä johtuen sivun ainoa päivitettävä osuus on kaavio, jonka tiedot haetaan palvelimelta ja käsitellään asiakasohjelman puolella. Näin minimoidaan yksittäiseen hakuun käytetty aika sekä palvelimelta vastauksena saadun tiedon määrä.

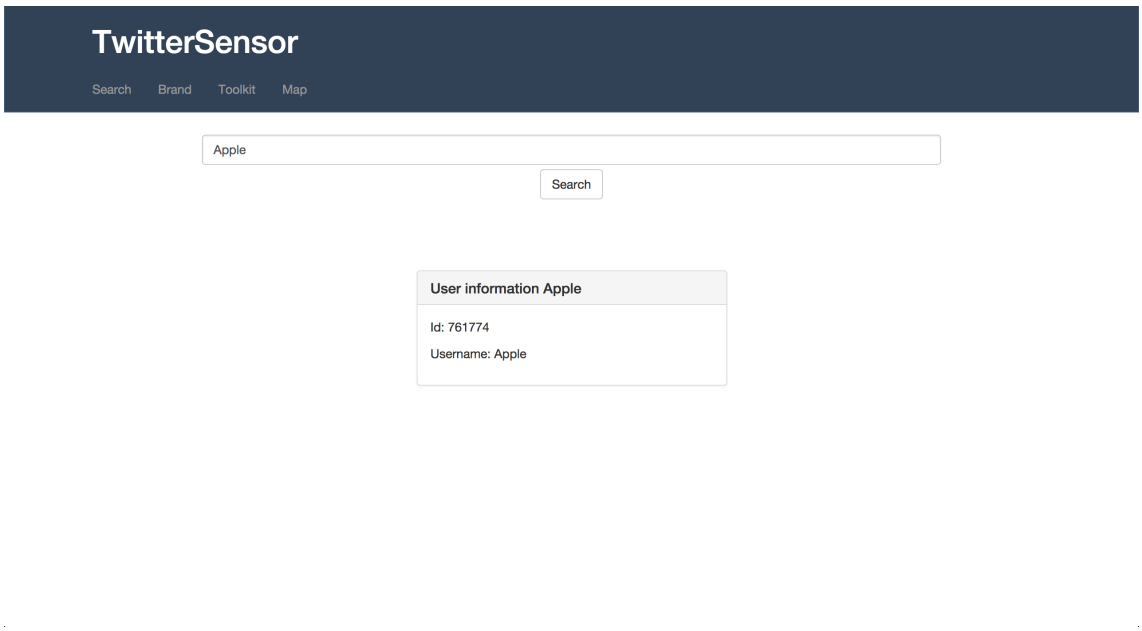
Brand-näkymä koostuu kahdesta näkymästä joista ensimmäisessä on kolme valintasarakketta vasemmalta: Teollisuudenala (Industry), Yritys (Company) ja Ajanjakso (Period). Toisessa näkymässä luodaan valituilla arvoilla käyttäjälle kaavio. Brand-näkymää on havainnollistettu kuvassa 39.



Kuva 39. Brand-näkymä.

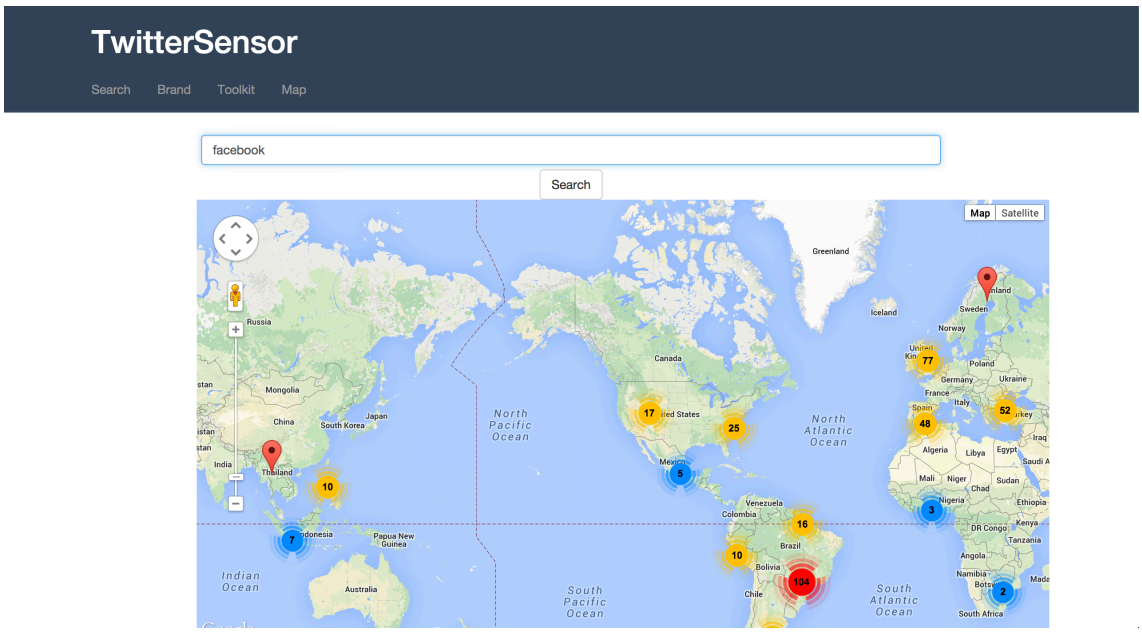
Käyttäjän saapuessa Brand-sivulle ladataan ensimmäiseksi Industry-valikon sisältämät vaihtoehdot. Käyttäjän valitessa Industry-arvon, kuten kuvassa 39 käytetty ”Airlines”, suoritetaan haku, josta saadaan tähän valintaan liittyvät tuotemerkit. Esimerkkinä kuvassa 39 käytetty ”AirCanada”. Tämän jälkeen Period-valintaa vaihtamalla suoritetaan tietokantaan haku, joka palauttaa kyseistä ajanjaksoa sisältävät tiedot asiakasohjelmalle, minkä perusteella asiakasohjelma muodostaa tiedosta käyttäjälle näytettävän kaavion.

Toolkit-näkymässä käyttäjä pystyy hakemaan käyttäjätunnus-käyttäjä-id-pareja. Näkymässä Search-painikkeen toiminta on yhdistetty ohjaimeen, joka suorittaa tietokantahaun ja päivittää tiedot asiakasohjelman näkymään. Jos hakusanaa vastaavaa käyttäjätunnusta tai käyttäjä-id:tä ei löydy, siitä ilmoitetaan käyttäjälle viestillä. Kuvassa 40 on havainnollistettu Toolkit-näkymää, kun hakusanaa vastaava käyttäjänimi on löytynyt.



Kuva 40. Toolkitnäkymä – löydetty käyttäjätunnus.

Map-näkymässä käyttäjän saapuessa sivulle hänelle näytetään hakukenttä ja hakupainike. Hakukenttä ja kartta ovat itsenäisiä näkymiä ja haettaessa hakusanalla kartta päivitetään. Map-näkymää hakutuloksen jälkeen on havainnollistettu kuvassa 41.



Kuva 41. Mapnäkymä - Hakusanan visualisointi.

Map-näkymässä palvelimelta haetaan tietokannasta hakusanan perusteella tietoa jossa ehtona käytetään säännöllistä lauseketta (regular expression, RegExp). Vastauksena asiakasohjelmalle lähetetään malli, joka sisältää hakusanaa vastaavat leveys- ja pituuskoordinaatit, joiden perusteella asiakasohjelman puolella toteutetaan kuvan 41 tyylinen kartta.

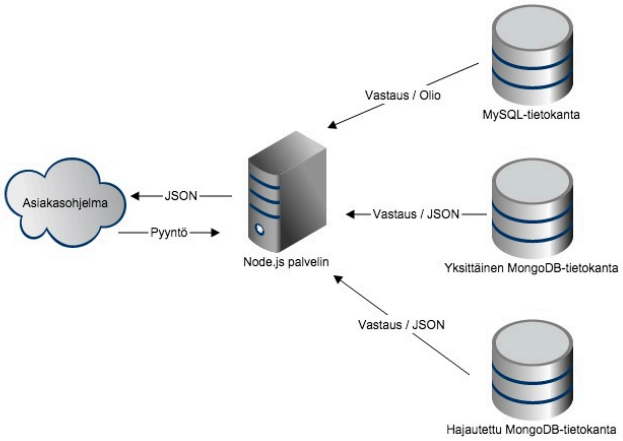
7.1 Palvelin ja tietokanta arkkitehtuurin toteutus

Toteutin http-palvelinohjelman käyttäen nodejs:n express-sovelluskehystä ja kraken.js-sovelluskerrosta luomaan palvelintoteutuksella selkeän ja helposti hallittavan rakenteen. Express-sovelluskehys on tarkoitettu joustavien http-palvelinohjelmien toteutukseen ja kraken.js on PayPal-yhtiön kehittämä liitännäinen, joka tuo lisätoiminnallisuutta express-sovelluskehyksen normaaliin toteutukseen helpottamalla suurien sovellusarkkitehtuurien hallintaa.

Palvelinkutsujen välittämiseen käytettiin REST-arkkitehtuurin mukaisia pyyntöjä, joiden käsittelyyn express.js:ssä on valmiiksi määritetyt toteutukset. Eri näkymien lukuoperaatioille määritettiin oma rajapintatoteutus ja lisäksi palvelinpuolella toteutettiin sivuston päänäkymän ulkoasumalli, mihin AngularJS asiakasohjelman puolella lisää näkymästä riippuvaista sisältöä.

Palvelinpuolella MongoDB-tietokannasta haettujen tietojen käsittely tapahtui Mongoose ODM-kirjastolla, joka tarjoaa mallipohjaisen ratkaisun tietokannasta haetun tiedon esittämiseen. Malleissa voidaan määrittää, mitä tietoa palvelinohjelma pystyy palautettavasta mallista käsittelemään, mikä poistaa tarpeen käsitellä tietokannasta saatua vastausta turhan tiedon vähentämiseksi. Samalla toteutus pitää palvelimen ja tietokannan välisen toteutuksen riittävän joustavana, mikä mahdollistaa lisätiedon palauttamisen kyselyltä ilman tarvetta muokata varsinaisia tietokantakyselyitä vaan tekemällä muutokset mallin kuvaukseen, joka tietokannalta palautetaan.

Palvelintoteutus haluttiin pitää mahdollisimman joustavana tulevaisuuden kannalta, mikä mahdollistaa kyselyt eri tietokantoihin ja eri tyyppiseen dataan. Tällä hetkellä palvelin noutaa tietoa kolmesta eri tietokantaratkaisusta: MySQL-tietokannasta, josta saatu tieto käsitellään ja muunnetaan palvelimen päässä ennen kuin se lähetetään asiakasohjelmalle sellaisessa muodossa kuin sen käsittely on järkevää; yksittäisestä MongoDB-tietokannasta, josta haetaan tietoa perustuen yksittäisiin dokumentin sisältäviin arvoihin sekä hajautetusta MongoDB-tietokannasta, jossa haut perustuvat dokumenttien sisältämään satunnaiseen tietoon. Kuvassa 42 on havainnollistettu palvelimen asiakasohjelmalle lähetettävän tiedon yhtenäisyyttä. Asiakasohjelmalle lähetettävä tieto lähetetään aina json-muodossa huolimatta siitä, mistä tieto on tullut. Kuviossa havainnollistetaan myös palvelimen ja eri tietokantojen välistä yhteyttä.



Kuva 42. Tiedon yhtenäinen palautus asiakasohjelmalle.

7.2 TwitterSensor-palvelun suorituskyky

Molempiin versioihin suoritettiin seuraavat testit: hakukentän kautta haettiin 100 kappaletta hakusanoja ja Brand-haun kautta haettiin 20 lentoyhtiötä joiden tuloksista haettiin kaikkein uusimmat ja koko ajalta olevat tulokset. Hakujen vasteaikoja tutkittiin chrome-selaimen kehittäjätyökalujen avulla. Tulokset kirjattiin ylös ja niistä laskettiin eri toiminnoille keskimääräiset arvot, pienimmät ja suurimmat arvot.

Taulukossa 3 havainnollistetaan uuden TwitterSensor-palvelun vasteaikaa chrome-selaimella käytettynä. Vasteajaksi merkittiin aika jolloin palvelin on lähettänyt vastauksen selaimelle.

Taulukko 3. Vasteajat, uusi versio

Haun tyyppi	Keskiarvo	Nopein	Hitain
100 hakua	194 ms	172 ms	419 ms
Brand uusimmat	345 ms	176 ms	1300ms
Brand kaikki	304 ms	181 ms	813 ms

Taulukossa 4 havainnollistetaan vanhan toteutuksen vasteaikoja. Kyseisessä testissä ei huomioitu koko sivuston päivittämiseen kuluvaa aikaa. Mikäli sivuston koko päivitykseen kulunut aika olisi huomioitu, vasteaika olisi ollut jonkun verran suurempi.

Taulukko 4. Vasteajat, vanha versio.

Haun tyyppi	Keskiarvo	Nopein	Hitain
100 hakua	794 ms	673 ms	1450 ms
Brand uusimmat	715 ms	529 ms	2010 ms
Brand kaikki	973 ms	700 ms	1550 ms

Taulukossa 5 on havainnollistettu uuden version yhden pyynnön aiheuttamaa liikennettä. Käytännössä yksi pyyntö ei aiheuta sivustolla yhtään enempää liikennettä, koska tieto lähetetään asiakasohjelmalle joka hoitaa tiedon visualisoinnin käyttäjälle.

Taulukko 5. Siirretyn tiedon määrä, uusi versio.

Siirretyn tiedon määrä	Keskiarvo	Pienin	Suurin
Brand uusimmat	0,70 Kt	0,48 Kt	1,02 Kt
Brand kaikki	2,97 Kt	1,3 Kt	8,7 Kt

Taulukossa 6 havainnollistetaan vanhan version yhden haun aiheuttamaa liikennettä. Näissä luvuissa ei ole huomioitu sitä, kuinka paljon liikennettä sivun muu päivitys aiheuttaa samalla kuin pyyntö käsitellään. Mikäli huomioidaan sivuston muun osien aiheuttama tiedonsiirto, niin luvut ovat noin 15 % suurempia.

Taulukko 6. Siirretyn tiedon määrä, vanha versio.

Siirretyn tiedon määrä	Keskiarvo	Pienin	Suurin
Brand uusimmat	27,16 Kt	21,1 Kt	31,0 Kt
Brand kaikki	52,38 Kt	34,1 Kt	73,6 Kt

Hakutulosten vasteaikoja vertaamalla voidaan huomata uuden toteutuksen suorittavan hakusanalla hakemisen keskimäärin neljä kertaa nopeammin kuin aikaisempi toteutus. Uuden toteutuksen Brand-haussa taulukoiden 3 ja 4 vertailulla huomattavia eroja. Haettaessa viimeaikaisia tietoja uusi versio on keskimäärin kaksi kertaa nopeampi. Haettaessa kaikkia arvoja uusi versio on keskimääräisesti kolme kertaa nopeampi.

Tutkimalla taulukoita 5 ja 6 havaitaan yksittäisen uuden brand-haun vievän keskimäärin 97 % vähemmän verkkoliikennettä ja kaikkia tuloksia haettaessa uusi haku vie keskimäärin 94 % vähemmän verkkoliikennettä.

Lopuksi luotiin vielä yksittäistä MongoDB-tietokantaa vastaava MySQL-tietokanta. Molemmat tietokannat sisälsivät hieman yli 300 miljoonaa tietuetta ja käyttivät indeksointia sekä välimuistista hakua. Molempiin tietokantoihin suoritettiin 100 hakua. Vasteaikaa asiakasohjelmalle tutkittiin vastaavalla tavalla kuin aikaisemmissa testeissä. Taulukossa 7 havainnollistetaan molempien tietokantojen keskimääräistä vasteaikaa asiakasohjelmalle. Taulukkoa tutkimalla huomataan MongoDB-tietokannan olevan keskimäärin hieman alle kaksi kertaa MySQL-tietokantaa nopeampi.

Taulukko 7. MongoDB vs. MySQL –tietokantahaut.

Tietokannan tyyppi	Keskiarvo	Pienin	Suurin
MongoDB	177 ms	169 ms	194 ms
MySQL	347 ms	329 ms	431 ms

7.3 Testatuissa teknologioissa havaitut ongelmat

TwitterSensor-palvelun toteutuksessa hyödynnetyissä teknologioissa ja toteutustavoissa havaittiin kaksi mahdollista, mutta korjattavissa olevaa ongelmaa, ensimmäisen sivulatauksen hitaus ja asiakasohjelmalle siirrettävän työn suuruus.

Yhden sivun sovellusmallilla (SPA) toteutettujen ohjelmistojen ensimmäinen sivulataus on verrattain hidas, koska koko ohjelma täytyy ensiksi ladata palvelimelta ja alustaa. Suuremmissa sovellusratkaisuissa käyttäjälle tarvitsee tietenkin ladata enemmän sisältöä, mikä tarkoittaa esimerkiksi näkymiä ja kontrollereita, voivat ensimmäiset sivun latausajat kasvaa hyvinkin suureksi. Ratkaisuna tähän on toteuttaa osa sivuston latauksesta niin sanottuna laiskana latauksena. Tällöin ladataan sivuston kannalta vain välttämättömät näkymät ja kontrollerit, jonka jälkeen sivusto saadaan alustettua käyttäjälle, jonka jälkeen voidaan siirtyä lataamaan sivuston muita osia.

Kaikkea työtä ei ole järkevää siirtää asiakasohjelman toteutettavaksi. Suurien tietomäärien visualisointi suoraan asiakasohjelmassa toimii tiettyyn pisteeseen saakka hyvin. Mikäli visualisoinnin toteuttamiseen tarvittavaan lasketaan tarvitaan paljon resursseja, se saattaa aiheuttaa hitautta asiakasohjelmassa. Ratkaisuna tähän on arvioida palvelinohjelmiston puolella, kuinka raskasta pyydetyn tiedon visualisointi asiakasohjelmalle

on. Raskaissa tapauksissa kannattaisi suorittaa osa tiedon käsittelystä palvelimen puolella ennen kuin tiedot lähetetään asiakasohjelmalle tai vaihtoehtoisesti lähettää esikäsitelty kaavio asiakasohjelmalle.

8 Yhteenveto

Opinnäytetyön tavoitteina oli TwitterSensor-palvelun kehittäminen vasteaikojen lyhentämiseksi ja suurten datamäärien hyödyntämiseksi. Lisäksi tavoitteena oli teknologioiden arviointi ja mittaus suorituskyvyn parantamisen näkökulmasta.

TwitterSensor-palvelusta rakennettiin uusi versio käyttäen MongoDB-tietokantaa, Express-, AngularJS- ja Node.js-sovelluskehyksiä (MEAN-pinototeutus).Testatut teknologiat osoittautuivat varsin toimiviksi ratkaisuiksi. TwitterSensor-palvelun vasteaikaa saatiin pienennettyä huomattavasti aikaisempaan toteutukseen verrattuna. Suoritettujen testien perusteella vaihtoehtoiset, tutkitut tietokanta ja arkkitehtuuriratkaisut mahdollistavat TwitterSensor-palvelun entistä monipuolisemman jatkokehityksen. Testatut teknologiat mahdollistavat palveluiden skaalautumisen, mutta eivät sovellu yksinään tarjoamaan ratkaisua tiedon määrän ja sisällön monipuolisuuden kasvaessa.

Mielestäni Hadoop-sovelluskehys on ainut teknologia, jolla Big Datan tarpeisiin pystytään vastaamaan. Big Dataan perustuvat sovellukset ja niiden taustajärjestelmät kannattaa miettiä huolellisesti. Lähtökohtaisesti täytyy osata valita oikeat teknologiat sovelluksen eri kerroksiin. Ei ole vain yhtä ongelmaa, niin ei ole olemassa myöskään vain yhtä ratkaisua.

Big Dataan perustuvat sovellukset eivät ole missään nimessä katoamassa lähitulevaisuudessa, vaan niiden kysyntä tulee päinvastoin kasvamaan. Tarve tämän kaltaiselle osaamiselle ollaan huomattu yritys- ja yliopistomaailmassa, mistä johtuen useat korkeakoulut ovat alkaneet tarjota big dataan, datatieteisiin ja data-analytiikkaan keskittyviä koulutusohjelmia.

Suurimmat Big Datan aiheuttamat ongelmat ovat sen tallentaminen ja käsittely. Tämän ratkaisee lähinnä Hadoop-sovelluskehyksen hajautettu levyjärjestelmä ja hajautettu käsittely esimerkiksi MapReducen avulla. Eräänlainen Big Data -kokoonpano voisi sisältää: Hadoop-sovelluskehyksen, joka hoitaa alkuperäisen tiedon tallentamisen, prosessoinnin ja tietokantaan tallentamisen. Asiakasohjelman tietokantana voisi toimia MongoDB, josta saadaan valmiiksi prosessoidut ja mahdollisimman tehokkaaseen muotoon tallennetut tiedot. Asiakasohjelman päässä näytettävän tiedon käsittely tapahtuisi osaksi palvelimella ja osaksi asiakasohjelmassa, jolloin asiakasohjelma pysyy suorituskykyisenä.

Lähteet

1. Yle Uutiset. 2014. S-ryhmä alkaa kerätä asiakkaista aiempaa enemmän tietoja. [viitattu 10.10.2014]. Saatavissa: http://yle.fi/uutiset/s-ryhma_alkaa_kerata_asiakkaista_aiempaa_enemman_tietoja/7158865. Luettu 10.10.2014.

2. Paul C. Zikopoulos Chris Eaton Dirk deRoos Thomas Deutsch George Lapis. Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data. The McGraw-Hill Companies; 2012. Saatavissa: http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&htmlfid=IML14296USEN. Luettu 10.11.2014.

3. Twitter. 2013. Introducing Keyword Targeting in Timelines. [viitattu 10.11.2014]. Saatavissa: https://blog.twitter.com/2013/introducing-keyword-targeting-in-timelines. Luettu 10.11.2014.

4. O,Reilly Radar. 2009. Bing and Google Agree: Slow Pages Lose Users. [viitattu 10.11.2014]. Saatavissa: http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html. Luettu 10.11.2014.

5. University of Rochester. 2014. Huaxia Rui. [viitattu 10.11.2014]. Saatavissa: http://www.simon.rochester.edu/faculty-and-research/faculty-directory/faculty-profile/index.aspx?Username=Huaxia.rui. Luettu 10.11.2014

6. Twitter. 2013. New Tweets per second record, and how!. [viitattu 15.11.2014]. Saatavissa: https://blog.twitter.com/2013/new-tweets-per-second-record-and-how. Luettu 15.11.2014.

7. Ashton K. 2009. Internet of Things. [viitattu 27.10.2014]. Saatavissa: http://www.rfidjournal.com/articles/view?4986. Luettu 27.10.2014.

8. Nielsen J. 1993. Response Times: The 3 Important Limits. [viitattu 9.11.2014]. Saatavissa: http://www.nngroup.com/articles/response-times-3-important-limits/. Luettu 9.11.2014.

9. Gaille B. 2013. Acceptable Web Page Load Times by Country and Industry. [viitattu 12.11.2014]. Saatavissa: http://brandongaille.com/acceptable-web-page-load-times-by-country-and-industry/. Luettu 12.11.2014.

10. Radware. 2014. New Radware Research Reveals Retailers Still Make Same Web Performance Mistakes. [viitattu 16.11.2014]. Saatavissa: http://www.radware.com/newsevents/pressreleases/summer-sotu2014/. Luettu 16.11.2014.

11. Garrett J. 2005. Ajax: A new approach to web applications. [viitattu 15.11.2014]. Saatavissa: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/. Luettu 15.11.2014

12. Haverbecke M. 2007. Eloquent JavaScript. [viitattu 5.11.2014]. Saatavissa: http://eloquentjavascript.net/. Luettu 5.11.2014.

13. Fielding R. 2002. Principled Design of the Modern Web Architecture [viitattu 6.11.2014]. Saatavissa: http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf. Luettu 6.11.2014

14. Green B. & Seshadri S. 2013. AngularJS. USA: O'Reilly Publishing.

15. Branas R. 2014. AngularJS Essentials. UK: Packt Publishing.

16. Google I/O. 2013. Design Decisions in AngularJS. [viitattu 10.11.2014]. Saatavissa: https://developers.google.com/events/io/sessions/325881193. Katsottu 10.11.2014.

17. AngularJS. 2014. Guide to AngularJS Documentation. [viitattu 14.11.2014]. Saatavissa: https://docs.angularjs.org/guide/. Luettu 14.11.2014.

18. Lavin J. 2014. AngularJS Services. UK: Packt Publishing.

19. Colin J. 2013. Pro Node.js for Developers. USA: Apress.

20. Messerschmidt T. 2014. PayPal KrakenJS [viitattu 10.11.2014]. Saatavissa: http://www.slideshare.net/PayPal/krakenjs. Luettu 10.11.2014.

21. Kirk A. 2012. Data Visualization: a successful design process. UK: Packt Publishing.

22. D3.js. 2014. Documentation. [viitattu 6.11.2014]. Saatavissa: https://github.com/mbostock/d3/wiki. Luettu 6.11.2014.

23. Google. 2014. Google Maps API. [viitattu 12.11.2014]. Saatavissa: https://developers.google.com/maps/articles/toomanymarkers. Luettu 12.11.2014.

24. Daily Mail. 2014. Facebook now has 1.32 billion users [viitattu 14.11.2014]. Saatavissa: http://www.dailymail.co.uk/sciencetech/article-2703440/Theres-no-escape-Facebook-set-record-stock-high-results-beats-expectations-1-32-BILLION-users-30-mobile.html. Luettu 14.11.2014.

25. MySQL. 2014. Customer: Facebook [viitattu 14.11.2014]. Saatavissa: http://www.mysql.com/customers/view/?id=757. Luettu 14.11.2014.

26. Altoros. 2014. A Vendor-independent Comparison of NoSQL Databases. [viitattu 14.11.2014]. Saatavissa: http://www.altoros.com/vendor_independent_comparison_of_nosql_databases.html. Luettu 14.11.2014.

27. Tiwari S. 2011. Professional NoSQL. USA: John Wiley & Sons.

28. MongoDB. 2014. Documentation [viitattu 10.11.2014]. Saatavissa: http://www.mongodb.com/. Luettu 10.11.2014.

29. Chodorow K. & Dirolf M. 2010. MongoDB: The Definitive Guide. USA: O'Reilly Publishing.

30. Ghemawat S. Gobioff H. Leung S. Google. 2013. The Google File System [viitattu 27.10.2014]. Saatavissa: http://static.googleusercontent.com/media/research.google.com/fi//archive/gfs-sosp2003.pdf. Luettu 27.10.2014.

31. Dean J. & Ghemawat S. Google. 2004. MapReduce: Simplified Data Processing on Large Clusters [viitattu 27.10.2014]. Saatavissa: http://static.googleusercontent.com/media/research.google.com/fi//archive/mapreduce-osdi04.pdf. Luettu 27.10.2014.

32. White T. 2012. Hadoop: The Definitive Guide. USA: O'Reilly Publishing.

33. Cloudera. 2014. Dissecting the apache hadoop-stack. [viitattu 16.11.2014]. Saatavissa: http://cloudera.com/content/cloudera/en/resources/library/training/dissecting-the-apache-hadoop-stack-2-of-6.html. Luettu 16.11.2014.

34. MSSQLTips. 2014. Related Apache Projects in Hadoop Ecosystem. [viitattu 16.11.2014]. Saatavissa. www.mssqltips.com/sqlservertip/3262/big-data-basics--part-6--related-apache-projects-in-hadoop-ecosystem/. Luettu 16.11.2014.

35. Prajapati V. 2013. Big Data Analytics with R and Hadoop. UK: Packt Publishing.

36. MacLean D. 2011. A Very Brief Introduction to MapReduce [viitattu 16.11.2014]. Saatavissa: http://hci.stanford.edu/courses/cs448g/a2/files/map_reduce_tutorial.pdf. Luettu 16.11.2914.

37. Apache. 2014. Apache ZooKeeper. [viitattu 24.11.2014]. Saatavissa: http://zookeeper.apache.org/. Luettu 24.11.2014.

38. MoreDevs. 2013. MySql vs MongoDB performance benchmark. [viitattu 16.11.2014]. Saatavissa: www.moredevs.ro/mysql-vs-mongodb-performance-benchmark/. Luettu 16.11.2014.