Jani Saareks

Tehokkaan tekstihaun toteuttaminen käyttöoikeudet huomioiden

Tietotekniikan Pro gradu -tutkielma

6. kesäkuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Jani Saareks

Yhteystiedot: jani.j.saareks@student.jyu.fi

Ohjaaja: Vesa Lappalainen, Antti-Juhani Kaijanaho

Työn nimi: Tehokkaan tekstihaun toteuttaminen käyttöoikeudet huomioiden

Title in English: Implementing efficient full text search with access rights

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 66+7

Tiivistelmä: Tekstihaulla etsitään vastaavaa sisältöä indeksoiduista dokumenteista. Dokumentteja voivat olla mitkä tahansa tiedostot tai erilaiset tekstiä sisältävät www-sivut. Tässä tutkielmassa esitellään erilaisia toteutustapoja tekstihaun luomiselle. Lisäksi käsitellään erilaisia ominaisuuksia, jotka parantavat hakujen toiminnallisuutta. Tutkielman käytännön osassa toteutetaan tekstihaku, joka huomioi käyttöoikeudet käyttäen Elasticsearch-nimistä hakumoottoria. Johtopäätöksenä tutkimuksesta saatiin, että Elasticsearch on tehokas tapa toteuttaa tekstihaku.

Avainsanat: Tekstihaku, Elasticsearch, Apache Lucene

Abstract: Full text search tries to find matches from indexed documents. There are multiple different choices to choose full text engine from and in this thesis we give overview of different tools. We also do some research what properties these tools have and how they implement different methods which help user to search from the index. In practical part of the thesis we create full text search using Elasticsearch and evaluate its performance. Conclusion is that Elasticsearch indeed is a very powerful tool for creating the search and it performs very well.

Keywords: Full-text search, Elasticsearch, Apache Lucene

Termiluettelo

Boolen haku	Haku, missä monipuolisemmat haut on mahdollistettu Boolen operaattoreiden ja, tai, ei yhdistelmien avulla.
CSE	Custom Search Engine, Googlen sisältöhaun toteuttava moduuli.
CSV	Tiedostomuoto, missä taulukkomuotoinen data on erotettu toisistaan pilkuilla ja rivinvaihdoilla.
InnoDB	Tietokantamoottori MySQL:n päälle. Luotiin korvaamaan MyISAM.
JSON	Avoimen standardin tiedostomuoto, jossa tieto on ihmisten luettavassa muodossa. JSON on nykyisin pitkälti korvannut XML:n.
MyISAM	Tietokantamoottori MySQL:n päälle. Paremmat ominaisuudet kuin InnoDB:ssä ja tukee transaktioita toisin kuin InnoDB.
MySQL	Relaatiotietokantaohjelmisto, joka on helppokäyttöinen ja nopea.
N-grammi	N-gram on n-merkin mittainen jakso. Käsiteltävästä merkkijonosta muodostetaan n-merkin mittaiset peräkkäisistä merkeistä koostuvat jonot.
OmaTila	Rekisteröityneen käyttäjän henkilökohtainen profiili Peda.net-palvelussa. Käyttäjän on mahdollista muokata tila mieleisekseen, ja tilassa on käytössä samat sisällöntuottamisen työvälineet kuin organisaation oppimisympäristössä.
PostgreSQL	Relaatiotietokantaohjelmisto, kilpailija MySQL:lle, ja sitä mainostetaan toimintavarmempana kuin MySQL:ää.
PR	PageRank, Google haun käyttämä web-sivustojen pisteytys. Suuremman PR-arvon omaava sivu näytetään ensin hakutuloksissa.
Restful API	Ohjelmointirajapinta, joka toteuttaa REST-arkkitehtuurin GET, PUT, POST ja DELETE pyynnöt.
SEO	Search Engine Optimization, Hakukoneoptimointi.

SQL	Kyselykieli, jolla voidaan tehdä relaatiotietokantaan kyselyitä.
Sulkusanat	Stopwords, Kielestä riippuen artikkelit, prepositiot ja konjunktiot, ovat sulkusanoja jotka eivät anna mitään arvoa haulle.
Säännöllinen lauseke	RegEx, määrittelee säännöllisen kielen. Mahdollistaa esimerkiksi käyttäjän antamien syötteiden muuntamisen toiseksi, jos syöte vastaa säännölliseen lausekkeeseen.
TFIDFSimilarity	Määrittele Apache Lucenen pisteytyksen komponentit. Ylikirjoittamalla komponentit voidaan vaikuttaa Lucenen pisteytykseen.
Vektoriavaruusmalli	Kahden dokumentin vastaavuus on niiden haku- ja indeksitermien yhteys esitettynä vektorina. Tämä mahdollistaa tilanteen, missä dokumentti täyttää hakutermin vain osittain.
XML	Rakenteellinen kuvauskieli, jonka avulla tietoa voidaan jäsentää.

Kuviot

Taulukot

Sisältö

1 Johdanto

Useat ihmiset käyttävät jotain hakukonetta päivittäin. Tämä tekee hakualgoritmeista ja erilaisista hakumoottoreista tärkeitä, vaikka ne eivät ole teknologian uusinta kärkeä (Lewandowski 2015). Hakukoneita ovat esimerkiksi Google tai johonkin palveluun toteutettu sisältöhaku. Tehokkaiden hakujen merkitys tulee vain korostumaan, koska datamäärät jatkavat kasvua ja usean tiedoston seasta manuaalisesti etsien aikaa menee hukkaan, joten tehokkaat toteutusratkaisut ovat tärkeitä.

Tutkimuksen tarkoituksena on selvittää erilaisia keinoja toteuttaa haku, joka on tehokas ja toimii suurilla tietomäärillä. Haku toteutetaan Peda.net-palveluun, jossa tällä hetkellä ei ole käytössä toimivaa hakua, koska tarpeeksi tehokasta ratkaisua ei ole löytynyt. Haun toteuttamisessa Peda.net-palveluun ovat ongelmana myös erilaiset käyttäjät ja käyttöoikeudet: kaikki sisältö ei ole avointa kaikille kirjautuneille tai kirjautumattomille käyttäjille. Erilaiset käyttöoikeudet joudutaan huomioimaan hakutuloksissa ja tämä tekee haun toteutuksesta vaikeamman kuin perinteisen tekstihaun toteutuksesta palveluun, jossa kaikki sisältö on jokaisen saatavilla. Käyttöoikeuden laskenta voi myös tehdä hausta erittäin raskaan. Käyttöoikeuksien laskenta johtaa siihen, että heti alusta alkaen suunnittelussa täytyy ottaa huomioon, miten käyttöoikeudet saadaan järkevästi laskettua mahdollisimman pienellä viiveellä.

Tässä tutkielmassa keskitytään avoimeen lähdekoodiin perustuviin hakukoneisiin, joiden avulla voidaan toteuttaa tekstihaku. Avoin lähdekoodi on tärkeässä asemassa siksi, että avoimeen lähdekoodiin perustuvia työkaluja voidaan ylläpitää vielä senkin jälkeen, jos niiden kehitys lopetetaan tai hidastuu merkittävästi. Peda.netin kannalta avoimen lähdekoodin ratkaisut ovat siksi paras vaihtoehto.

Tutkimusmenetelmänä työssä käytetään design scienceä, jossa luodaan artefakti jota kehitetään iteratiivisesti eteenpäin käyttäen vaatimusmäärittelyä sekä työkavereilta saatuja kommentteja virheiden korjaamiseen (Hevner ym. 2004). Tutkielman tarkoituksena on luoda teorian pohjalta toimiva ja tehokas sisältöhaku Peda.net-palveluun. Teoriaosuuden avulla perustellaan käytettävä toteutustapa sekä sovelletaan sitä käytännössä. Samalla teoriaosuuden avulla kartoitetaan mahdollisia vaihtoehtoja joiden avulla voidaan toteuttaa tehokas tekstiha-

ku.

Aluksi tutkielmassa käsitellään erilaisia toteutusratkaisuja, joiden avulla voidaan toteuttaa tekstihaku. Luvussa 3 käsitellään erilaisia ominaisuuksia, mitä eri toteutusratkaisut toteuttavat. Samalla käydään läpi miten nämä ominaisuudet voidaan toteuttaa ja mitä ne tarkoittavat. Neljännessä luvussa käsitellään tekstihaun toteuttamista. Siinä esitellään tutkimuksen kohde sekä miksi sitä lähdetään kehittämään. Viidennessä luvussa käydään läpi tutkimuksen tulokset ja pohditaan niiden mielekkyyttä. Lopuksi on yhteenveto koko Pro gradu -tutkielmasta.

2 Tekstihaun toteutustapoja

Tekstihaulla (engl. full-text search) tarkoitetaan hakua, jossa käyttäjä voi etsiä dokumentteja suoraan tietokannasta käyttäen avainsanoja tai niiden yhdistelmiä. Tekstihaussa tarkoituksena on löytää paras mahdollinen vastaavuus tietokannan taulun tai taulujen sisällöstä käyttäjän antamaan syötteeseen. Siksi suurin osa erilaisista keinoista toteuttaa tekstihaku pisteyttävät jollain tavalla haun tulokset, jotta ne voidaan järjestää osuvuuden mukaan (Kuc ja Rogozinski 2014). Tässä luvussa esitellään erilaisia keinoja joiden avulla voidaan toteuttaa tekstihaku palveluun.

2.1 Yksinkertainen tekstihaku

Yksinkertaisimmillaan tekstihaku voidaan toteuttaa käyttäen valitun SQL-tietokannan sisäänrakennettua ominaisuutta, kuten esimerkiksi MySQL:n tai PostgreSQL:n tekstihakua. Tekstihaku voi myös kohdistua tekstitiedostoihin, jotka toimivat haettavana aineistona ja niistä voidaan hakea esimerkiksi *grep*-komennon avulla. Tekstihaku voidaan kytkeä päälle tietokantaan, jonka jälkeen tekstisarakkeista voidaan etsiä vastaavuuksia annettuun SQL-kyselyyn. Käytännössä tämä tarkoittaa, että luodaan käänteishakemisto (engl. inverted index. Lisää luvussa 3.1) ja Boolen haku tai vektoriavaruusmalli (engl. vector space model) otetaan käyttöön. (Manning, Raghavan ja Schütze 2008; “MySQL Documentation” 2019; “PostgreSQL Documentation” 2019)

PostgreSQL:ssä tekstihaku käyttää tsvektoreita, jotka ovat järjestettyjä listoja erilaisia lekseemejä. Tässä tapauksessa kyseessä ovat normalisoidut sanat, jotka ovat järjestetty aakkosjärjestykseen pituuden mukaan (“PostgreSQL Documentation” 2019). Seuraavaksi esimerkki, mitä edellä mainittu käytännössä tarkoittaa:

```
SELECT 'a sad cat sat at rat'::tsvector;
```

Muodostaa seuraavanlaisen listan:

```
          tsvektori
------------------------------
```

```
'a', 'at', 'cat', 'sad', 'sat', 'rat'
```

PostgreSQL:llä voidaan myös yhdistää erilaisia lekseemejä Boolen operaattoreiden avulla (“PostgreSQL Documentation” 2019). Esimerkiksi:

```
SELECT 'cat & sat'::tsquery;
```

Saadaan luotua seuraavanlainen yhdistelmä:

```
    tsquery
--------------
 'cat' & 'sat'
```

mikä vastaa aina, jos molemmat sekä ‘cat’ ja ‘sat’ esiintyvät haussa. Myös Boolen tai(|)- ja ei(!)-operaattorit ovat mahdollisia käyttää.

MySQL tekstihaku on hieman erilainen kuin PostgreSQL:n. Tärkeää on huomata, että MySQL tekstihaku toimii vain InnoDB tai MyISAM taulujen kanssa. MySQL mahdollistaa kolmen erilaisen tekstihaun käyttämisen: luonnollinen kieli, Boolen haku ja kyselyn laajennushaku (engl. Query expansion), joka oikeastaan on vain luonnollisen kielen haun muunnos. (“MySQL Documentation” 2019).

Luonnollinen kielihaku ei käytä mitään operaattoreita ja on oletustekstihaku MySQL:ssä, jos käyttäjä ei anna mitään parametreja (“MySQL Documentation” 2019). Taulukossa 1 näkyy esimerkkitaulu, jossa tauluun on luotuna sarakkeet id,title ja body. Fulltext-parametri on annettu sarakkeille: title ja body. Kysely kyseiseen tauluun voidaan tehdä seuraavasti:

```
SELECT id, MATCH (title,body) AGAINST ('Test' IN NATURAL LANGUAGE MODE)
       FROM articles;
```

Taulukko 1: Esimerkkitaulu.

id	title	body
1	Test	Test article 1.
2	Article	Another text article

Käyttäessä Boolen hakua voidaan erikseen valita, mitä sanoja halutaan jättää pois hausta ja, mitä sanoja tuloksen pitää vähintään sisältää. MySQL:ssä Boolen operaattorit ovat erilaisia kuin PostgreSQL:ssä. Tässä + vastaa JA, - EI ja ilman operaattoria on TAI (“MySQL Documentation” 2019). Edellä esitettyä luonnollisen kielen hakua vastaava haku toteutettaisiin Boolen haulla seuraavasti:

```
SELECT id FROM articles WHERE MATCH (title,body)
    AGAINST ('+Test' IN BOOLEAN MODE)
```

Yksinkertaiset tekstihaut ovat kuitenkin yleensä huonoja vaihtoehtoja. Toisaalta pienissä palveluissa se voi olla tarpeeksi tehokas. Suuremmissa palveluissa SQL:n oman tekstihaun rajat kuitenkin tulevat nopeasti vastaan. Yksinkertainen tekstihaku ei skaalaudu tarpeeksi hyvin suurelle datamäärälle ja haun muokkaaminen ei ole helppoa. (Kuc ja Rogozinski 2014). Erityisesti jos tietokanta sisältää useita tauluja ja jokaisessa on hieman erilaista dataa, joudutaan tekstihaun toteuttavia sarakkeita luomaan useita ja luomaan myös monimutkaisia SQL-kyselyitä. Tämä tekee kyselyiden muokkaamisesta hidasta ja vaikeuttaa niiden ylläpitoa.

Yllämainittujen ongelmien ratkaisemiseksi on kehitetty erilaisia keinoja, joista yksi on Apache Lucene.

2.2 Apache Lucene

Apache Lucene on Java-pohjainen kirjasto tekstihaun luomiseen eli oikeastaan hakukonekirjasto. Ensimmäinen versio siitä on julkaistu vuonna 1999 ja viimeisin versio (7.7.0) helmikuussa 2019 (“Apache Lucene” 2019). Se on nopea ja skaalautuu hyvin. Itsessään Lucenella ei suoraan pysty luomaan tekstihakua vaan se tarjoaa lähinnä rajapinnan, jonka avulla voidaan luoda hakemistot (engl. indexes) ja toteuttaa haku palveluun (Cui ym. 2011). Eli pelkästään Lucenea asentamalla ei voida luoda toimivaa hakua.

Lucene koostuu seitsemästä moduulista, jotka ovat lueteltuina taulukossa 2. Niistä neljä vaadittua on merkittynä taulukossa tähdellä(*). Näiden neljän luokan toteuttaminen mahdollistaa Lucenen käyttämisen tekstihaun toteuttamiseen. Lucenen käyttö menee yksinkertaistettuna seuraavasti: ensimmäisenä luodaan tiedosto (engl. document), joka koostuu nimetyistä kentistä (engl. fields). IndexWriter-luokka luo hakemiston (engl. index), johon tiedostot

tallennetaan. Se vastaa indeksin luonnista ja hallinnasta. Seuraavaksi tarvitaan QueryParser-luokkaa, jonka avulla luodaan kysely halutulla merkkijonolla. IndexSearcher-luokka ottaa kyselyn vastaan ja välittää sen itse hakumetodille. (Kuc ja Rogozinski 2014; "Apache Lucene" 2019).

Taulukko 2: Lucenen moduulit

Moduulin nimi	Toiminto
org.apache.lucene.analysis*	Toteuttaa Analysoijan, mikä jakaa dokumentin.
org.apache.lucene.codecs	Indeksirakenteen koodaus ja dekoodaus.
org.apache.lucene.document*	Hallitsee dokumentteja, toteuttaa kentät.
org.apache.lucene.index*	Dokumenttien lisäys indeksiin ja indeksin luku.
org.apache.lucene.search*	Haun toiminnallisuudet.
org.apache.lucene.store	Pysyvien tietojen tallentaminen.
org.apache.lucene.util	Sisältää hyödyllisiä tietorakenteita ja luokkia.

*Merkityt ovat vaadittuja.

Lucene tallentaa kaikki sille annetut tiedot käänteishakemistoon. Lucene ei siis etsi alkuperäisestä tietokannasta mitään vaan haku suoritetaan käänteishakemiston kautta ja tulokset saadaan sieltä. Tällöin ongelmaksi voi muodostua se, että haku ei välttämättä löydä täysin oikeita dokumentteja, jos indeksi ei ole päivitettynä tai jotain tärkeää sisältöä ei ole lisätty indeksiin. Mitä enemmän tekstihakuun lisättäviä sarakkeita tietokannassa on, sitä useampi käänteishakemisto muodostetaan. Esimerkiksi, jos taulussa olisi sarakkeet otsikko, sisältö ja tiivistelmä, luotaisiin siitä kolme käänteishakemistoa. Tällöin olisi mahdollista myös lisätä hakemistoon tunniste, siitä missä kohdassa tekstiä kyseinen esiintymä on. Tunnisteiden käyttö mahdollistaa hakemisen painotuksilla niin, että tekstissä esiintyvät osumat saavat enemmän painoarvoa kuin otsikossa esiintyvät. (Manning, Raghavan ja Schütze 2008; Kuc ja Rogozinski 2014)

Lucene pisteyttää tulokset luokalla TFIDFSimilarity. Luokan käyttämät metodit voidaan kuitenkin ylikirjoittaa, jos pisteytystä halutaan muokata. Pisteytys tapahtuu käyttämällä Boolen

mallia (engl. Boolean model) ja vektorimallia (engl. vector space model). Boolen mallissa hakutermeillä on binääriset painoarvot ja painoarvoja yhdistelemällä Boolen operaattoreiden avulla saadaan selvitettyä osuuko dokumentti hakuun vai ei. Vektorimallissa hakutermin ja indeksin yhteys on esitetty vektorina, mikä mahdollistaa sen, että dokumentin sisällön ei tarvitse vastata täysin hakutermiä vaan myös osittaisia osumia voidaan sisällyttää tuloksiin. Sen avulla tulokset voidaan myös lajitella samankaltaisuuden mukaan. (“Apache Lucene” 2019; Manning, Raghavan ja Schütze 2008).

Lucene karsii ensin tuloksia Boolen mallin avulla, jonka jälkeen vektorimallilla pisteytetään hyväksytyt tulokset. Lucenen pisteytys käyttää seuraavaa kaavaa:

$$score(q,d) = \sum_{t\,in\,q}(tf(t\,in\,d) * idf(t)^2 * t.getBoost() * norm(t,d))$$

missä,

- $tf(t\,in\,d)$ on termin (t) esiintyvyys dokumentissa (d). Oletuksena $tf(t\,in\,d) = frekvenssi^{1/2}$.
- $idf(t)$ on käänteinen dokumenttien määrä missä termi (t) esiintyy. Se lasketaan $idf(t) = 1 + \log(\frac{dLukumaara + 1}{dFrekvenssi + 1})$
- $t.getBoost()$ on kyselyssä määritetty korotus. Se lasketaan kertomalla dokumentin pisteet hakutermin (q) korotuksen kanssa.
- $norm(t,d)$ on indeksointiajasta riippuva muuttuja, joka on riippuvainen merkkien määrästä dokumentissa ja mitä lyhyempi kenttä, sitä enemmän pisteitä se saa.

(“Apache Lucene” 2019).

Lucene taipuu moneen, mutta suoraan sen avulla ei pystytä rakentamaan toimivaa tekstihakua vaan se vaatii aina huolellisen suunnittelun ja toteutuksen. Samalla Lucenen dokumentaatio täytyy hallita hyvin, jos sen avulla halutaan rakentaa suorituskykyinen ja toimiva hakukone.

2.3 Apache Solr

Apache Solr (lausutaan Solar) on avoimeen lähdekoodiin perustuva hakukone, joka on rakennettu Apache Lucenen päälle (“Apache Solr” 2019). Apache Solr on mahdollista ot-

taa käyttöön myös muokattuihin Lucene-pohjaisiin hakuihin muokkaamalla solrconfig.xml-tiedostoa, mikä sisältää Apache Solr:in vaatimat määritykset (Vijay Karambelkar 2014). Solrconfig sisältää määritykset esimerkiksi web-hallintapaneelille, käsittelijöille, jotka hoitavat dokumentin lisäyksen indeksiin sekä kuuntelijoiden määrittelyt, jotka kuuntelevat hakukyselyihin liittyviä tapahtumia (“Apache Solr” 2019). Solr aloitettiin omana projektinaan, mutta nykyisin sitä kehitetään Lucenen yhteydessä, mikä takaa sen, että Lucenen uusien ominaisuuksien pitäisi olla suoraan mukana Solr:ssa (Shahi 2016).

Apache Solr on kirjoitettu Javalla ja sitä ajetaan omana instanssina. Solr tulee täysin toimivana ladattaessa, ja käyttäjän ei tarvitse tietää ennalta mitään ohjelmoinnista, jos hakukonetta ei tarvitse räätälöidä erikseen kyseiseen palveluun sopivaksi. Ainoa vaatimus, että Lucene ja Solr pyörivät on Java. Solr vaatii vähintään JRE (Java Runtime Environment) version 1.8 tai uudemman (“Apache Solr” 2019). Solr:n säätäminen vaatii Java-osaamista, koska se tehdään periyttämällä alkuperäisiä luokkia. (Shahi 2016).

Apache Solr:n avulla voidaan tallentaa dataa suoraan indeksiin, tai sille voidaan vaihtoehtoisesti antaa valmis malli (engl. schema), jonka mukaan data halutaan tallentaa indeksiin. Solr pystyy myös automaattisesti luomaan mallin datan perusteella, jos käyttäjä ei halua erikseen antaa mallia. Malli voidaan antaa schema.xml tiedostona, joka määrittelee sen, mistä kentistä dokumentti koostuu. (“Apache Solr” 2019; Shahi 2016).

Solr antaa käyttäjälle tehokkaan työkalun, jonka avulla on mahdollista luoda hakuja käyttäen fraaseja (engl. phrase search), korvausmerkkejä (engl. Wildcards), säännöllisiä lausekkeita (engl. Regular Expression) tai Boolen operaattoreita sekä erilaisia rajauksia. Hakutuloksia voidaan rajata esimerkiksi päivämäärän mukaan tai rajata kaikki tietyn kirjoittajan julkaistut pois tuloksista. (Vijay Karambelkar 2014).

Apache Solr mahdollistaa XML, CSV tai JSON tiedostomuotojen käyttämisen, mutta sen antamia hakutuloksia ei voida muokata omilla skripteillä. Jos on tarvetta muokata haunpisteytystä reaaliajassa, niin Solr ei ole silloin oikea ratkaisu ongelmaan, koska siinä ei kyseistä ominaisuutta ole (“Apache Solr” 2019). Hyvänä puolena Solr:ssa voidaan pitää web-hallintapaneelia, joka mahdollistaa indeksin tietojen katsomisen suoraan selaimesta. Samalla hallintapaneelin kautta voidaan suorittaa kyselyitä indeksiin ja analysoida tiedostojen kenttiä,

jotta indeksistä saadaan tehtyä tehokkaampi. Hallintapaneeliin myös kirjataan kaikki mahdolliset indeksissä tapahtuvat virheet, jotka voidaan jälkikäteen katsoa läpi. (“Apache Solr” 2019; Shahi 2016)

Solr toimii hyvin tekstipitoiselle sisällölle, mutta jos palvelu sisältää myös paljon muuta dataa kuin tekstiä—esimerkiksi kuvia tai tiedostoja—ei Apache Solr välttämättä ole paras mahdollinen tapa toteuttaa tekstihaku palveluun (Oliver 2017). Apache Solr:lla on myös haastajia, kuten Elasticsearch (Shahi 2016).

2.4 Elasticsearch

Elasticsearch on Apache Lucenen päälle rakennettu avoimeen lähdekoodiin perustuva hakukone, jonka avulla voidaan toteuttaa skaalautuva ja monipuolinen tekstihaku erilaisiin palveluihin. Elasticsearchin omistaa Elastic N.V, joka on hollantilainen yritys. Elasticsearchissa on lähes reaaliaikainen indeksointi eli käyttäjän ei tarvitse odottaa kauaa, että tiedosto indeksoidaan ja se on nopea suurilla tietomäärillä. Elasticilla on myös muita palveluita, jotka mahdollistavat Elasticsearchin monipuolisemman käytön, kuten Kibana, jonka avulla Elasticsearchin indeksin sisältämää dataa voidaan visualisoida. (“Elasticsearch Documentation” 2019; Gormley ja Tong 2015)

2.4.1 Käyttö

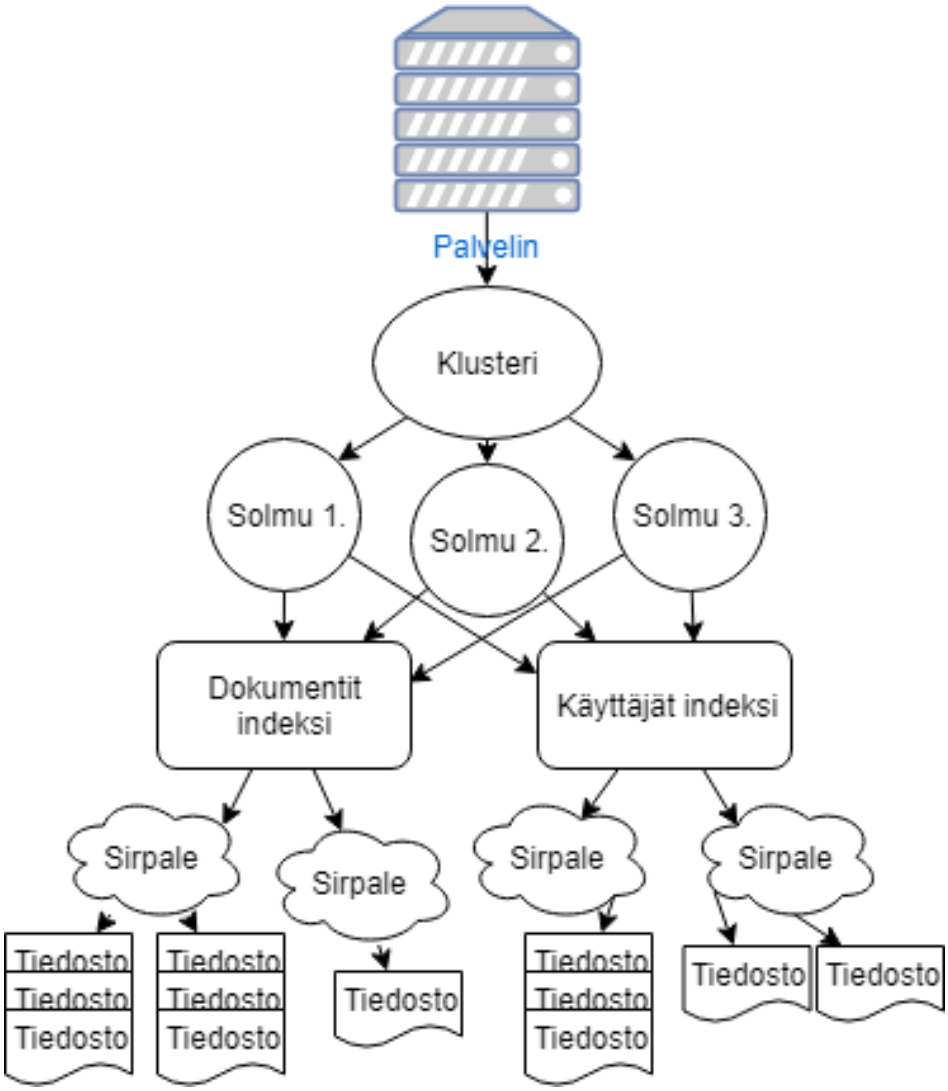
Elasticsearchiä käytetään RESTful APIn avulla ja tiedostomuotona toimii ainoastaan JSON. Esimerkiksi Apache Solr:n tukemaa XML- tiedostomuotoa ei voi käyttää Elasticsearchin kanssa (“Elasticsearch Documentation” 2019). Elasticsearch tarjoaa asiakasrajapinnat Java-, Python-, C#-, Perl-, Ruby- ja PHP-ohjelmointikieliin (“Elasticsearch Documentation” 2019). Elasticsearchin tarjoamat rajapinnat ovat hyvin yksinkertaisia, ja ne toimivat vain hyvänä pohjana toteuttaa oma kirjasto, minkä avulla käyttää Elasticsearchia.

2.4.2 Elasticsearchin osat

Elasticsearch koostuu seuraavista osista:

1. Klusterista (engl. Cluster),
2. Solmuista (engl. Node),
3. Indeksistä / hakemistosta (engl. Index),
4. Tiedostosta / dokumentista (engl. Document).

Klusteri on kokoelma solmuja, joka muodostaa perustan Elasticsearchin indeksille. Solmut ovat yksittäisiä palvelimia, jotka ovat osa klusteria. Klusteri voi koostua vain yhdestä solmusta, mutta yleensä niitä on useampia. Indeksi on kokoelma dokumentteja, jotka sisältävät samanlaista dataa. Elasticsearch indeksi koostuu sirpaleista (engl. Shards), jotka ovat täysin itsenäisiä indeksejä, jotka toimivat millä tahansa solmulla. Sirpaleiden määrä voidaan määrittää uutta indeksiä luodessa. Jokainen sirpale on oikeasti Apache Lucenen indeksi, mikä rajoittaa tiedostojen maksimimäärän sirpaleelle *2 147 483 519*:aan dokumenttiin. Elasticsearchin osien välistä suhdetta on kuvattu tarkemmin Kuviossa 1. (“Elasticsearch Documentation” 2019).



Kuvio 1. Elasticsearchin osat (“Elasticsearch Documentation” 2019)

Indeksiin tallennetut tiedostot/dokumentit sisältävät kenttiä (engl. fields), joissa on kaikki tiettyyn dokumenttiin liittyvä tieto. Elasticseachissa voidaan myös määrittää erikseen tietyn indeksin dokumenttien sisältämät kentät (“Elasticsearch Documentation” 2019). Toistaiseksi indeksit voivat sisältää erilaisia tyyppejä, mutta tyypit tullaan poistamaan kokonaan Elasticsearching versiossa 8 (“Elasticsearch Documentation” 2019). Eri tyypeillä on omat kenttänsä, joista dokumentin tiedot koostuvat. Esimerkiksi oppilas ja opettaja tyyppien rakenteen määritys voisi olla muotoa:

```
{
```

```
  "mappings": {
    "oppilas": {
      "properties": {
        "nimi": { "type": "text" },
        "kayttajatunnus": { "type": "keyword" },
        "sahkoposti": { "type": "keyword" }
      }
    },
    "opettaja": {
      "properties": {
        "nimi": { "type": "text" },
        "kayttajatunnus": { "type": "keyword" },
        "sahkoposti": { "type": "keyword" },
        "koulutus": { "type": "keyword" }
      }
    }
  }
}
```

Elasticsearchin versiossa 5 olisi molemmat sekä oppilaat, että opettajat voitu tallentaa samaan indeksiin, mutta molemmilla täytyy olla sama rakenne eli ylläolevan esimerkin pitäisi oikeasti olla muotoa:

```
{
  "mappings": {
    "oppilas": {
      "properties": {
        "nimi": { "type": "text" },
        "kayttajatunnus": { "type": "keyword" },
        "sahkoposti": { "type": "keyword" },
        "koulutus": { "type": "keyword" }
      }
    },
    "opettaja": {
      "properties": {
        "nimi": { "type": "text" },
        "kayttajatunnus": { "type": "keyword" },
        "sahkoposti": { "type": "keyword" },
```

```
        "koulutus": { "type": "keyword" }
      }
    }
  }
}
```

Samassa indeksissä sijaitsevat kaksi erilaista tyypitystä kuitenkin liittyvät toisiinsa (“Elasticsearch Documentation” 2019), vaikka käyttäjä ei olisi niin alunperin ajatellutkaan. Oppilaalla oleva koulutus-kenttä ei ole oleellinen indeksoitavan tiedon kannalta ja siksi se olisi loogisinta jättää tyhjäksi. Oikeasti kenttien jättäminen tyhjäksi ei ole kannattavaa, koska Apache Lucene toimii Elasticsearchin taustalla. Tästä johtuen tyhjien kenttien tallentaminen samaan indeksiin sellaisten tyyppien kanssa, joissa kaikissa kentissä on sisältöä, ei ole tehokasta (Berman 2018).

Elasticsearchin tulevissa versioissa opettajat ja oppilaat tulee jakaa eri indekseihin, jolloin indeksin sisällöstä tulee loogisempaa ja vältytään tyhjien kenttien aiheuttamilta ongelmilta. Samalla Elasticsearchin pisteytyksen pitäisi parantua, koska sen ei tarvitse pisteyttää tyhjiä kenttiä. (“Elasticsearch Documentation” 2019)

Perinteisiin tietokantoihin verrattuna Elasticsearchin osat menevät seuraavasti: indeksi vastaa taulua, tiedosto vastaa tietokannan riviä ja kentät ovat sarakkeita (Gormley ja Tong 2015). Eli samoin kuin tavallisissa relaatiotietokannoissa erilaiset asiat tallennetaan eri tauluihin, niin myös toimitaan Elasticsearchin kanssa, eli opettajat ja oppilaat olisivat eri indekseissä Elasticsearchin klusterissa.

2.4.3 Hakutulosten järjestäminen

Elasticsearch järjestää oletuksena haun antamat tulokset pisteiden mukaan. Pisteytyksen hoitaa Apache Lucene. (Smith 2018). Järjestys voidaan myös tehdä käyttäjän haluamien tietojen mukaan esimerkiksi, jos dokumentin määritelmässä on kenttä

```
{"pvm":{"type":"date"}}
```

voidaan tulokset järjestää päivämäärä kentän mukaan nousevaan tai laskevaan järjestykseen. Elasticsearchin *sort* parametriksi voidaan antaa useiden kenttien nimet, jolloin kenttien jär-

jestyksellä on väliä. Ylin kenttä saa suurimman painoarvon, kun taas alimmainen eli viimeisenä oleva kenttä merkitsee vähiten tulosten järjestyksen kannalta (“Elasticsearch Documentation” 2019).

2.5 Google

Google tarjoaa sivustoille lisättävän hakukoneen, Google-täsmähaun (engl. Google Custom Search Engine, CSE). CSE toimii samalla tavoin kuin Googlen normaalihaku, mutta sitä pystytään muokkaamaan hieman. Käyttäjä pystyy esimerkiksi muokkaamaan hakulaatikon tyyliä niin, että se sopii sivuston ulkoasuun. (“Google custom search” 2019).

Google tarjoaa valmiin työkalun, jonka avulla hakukone voidaan rakentaa. Käyttäjän tehtäväksi jää tämän jälkeen vain lisätä se palveluun (“Google custom search” 2019). Käyttöönoton kynnys on siis erittäin pieni. Hakukone ei pelkästään toimi sisältöhaun toteuttavana komponenttina, vaan sillä voidaan nimen mukaisesti tehdä kustomoitu hakukone, joka hakee Googlen hakuindeksistä sopivia osumia.

Googlen toteutus ei ole mainosvapaa eikä se pysty indeksoimaan yksityisiä sivustoja. Hausta saa mainosvapaan maksamalla, ja yksityisen indeksin teko myös onnistuu yritysasiakkaille, mutta se ei ole ilmainen. Googlen täsmähaku määritellään käyttäen XML-tiedostoa, joka kertoo, minkä niminen hakukone on kyseessä, sekä mistä urleista se etsii (“Google custom search” 2019).

Indeksoinnissa oikeisiin tuloksiin pääseminen vaatii sen, että sivuille on lisätty oikeat tagit, mitä hyvä SEO myös vaatii. Eli sivun meta-tagien pitää olla kunnossa, niissä täytyy mainita avainsanat ja sivun lyhyt tiivistelmä, jotta se indeksoidaan oikein ja haku toimii hyvin (“Google custom search” 2019). Googlen haku ei siis sovellu hyvin palveluun, jossa sivut eivät välttämättä ole tasalaatuisia ja osasta puuttuu hyvän tavan mukaiset meta-kentät. Tämä voidaan toki kiertää säätämällä CSE:tä, mutta toisilla vaihtoehtoisilla toteutuksilla sisällön indeksoiminen voi olla kannattavampaa.

2.5.1 PageRank

PageRank (PR) on Googlen kehittämä ja käyttämä algoritmi, jolla pisteytetään hakutuloksia Googlen haussa. PageRank on nimetty Googlen toisen perustajan Larry Pagen mukaan. PageRank pisteytys toimii sivukohtaisten viittausten mukaan, mitä enemmän tietylle sivulle linkitetään muilta sivustoilta sitä tärkeämpi se on ja sitä korkeamman aseman se saa pisteytyksessä (Rogers 2019).

Ensimmäisellä laskukierroksella pisteitä ei voida täysin tietää, jos kaikilla sivuun linkittyvillä sivustoilla ei ole olemassa olevaa PageRankkia. Tällöin PR on vain arvio, joka tarkentuu kun matriisia lasketaan edelleen. (Rogers 2019). Ennen vuotta 2016 sivuston PR oli mahdollista nähdä Googlen Toolbarin avulla, mutta Google poisti sen ominaisuuden (Schwartz 2016).

PR:n yksi ongelma on se, että tarkan matriisin laskeminen vaatii aikaa ja useita iteraatioita, jotta kaikki linkitykset saadaan painotettua oikein ja jotta lopullinen PR voidaan muodostaa (Rogers 2019). Matriisin koko on *NxN*, kun indeksoitavia www-sivuja on *N* kappaletta. Esimerkiksi koko webin indeksoiminen kestää kauan vaikka algoritmi olisikin optimoitu. Google ei myöskään tue lähes reaaliaikaista indeksointia (“Google custom search” 2019), mikä vaikuttaa uuden sisällön indeksoimisen kestoon.

Googlen PageRank algoritmi ei kuitenkaan ole avointa koodia, joten kukaan ei tarkasti tiedä miten se toimii. PageRank on silti yksi Googlen tunnetuimmista algoritmeista, joskaan se ei ole ainut algoritmi, jota Google käyttää hakutulosten pisteytykseen (Rogers 2019).

3 Hakumoottoreiden ominaisuuksia

Tässä luvussa käydään läpi hakumoottoreiden yleisiä ominaisuuksia kuten indeksointia ja hakumoottoreiden vaatimuksia. Samalla luvussa käsitellään, miten nämä ominaisuudet voidaan toteuttaa.

3.1 Indeksointi

Indeksoinnilla voidaan tarkoittaa eri asioita, mutta tiedonhakuun liittyen käänteistiedosto / käänteishakemisto on tärkein indeksointimenetelmä (Manning, Raghavan ja Schütze 2008). Indeksoinnissa haluttu sisältö luetaan levyltä, ja siitä muodostetaan hakemisto avainsanojen tai metatietojen mukaan. Indeksoinnin avulla voidaan merkittävästi lyhentää vaadittuja hakuaikoja.

Käänteishakemisto toimii siten, että ensin muodostetaan sanakirjatiedosto, joka koostuu esimerkiksi indeksoitavasta merkkijonosta löytyvistä erinäisistä termeistä ja sanojen esiintymien lukumäärästä kyseisessä merkkijonossa. Tämän jälkeen luodaan itse käänteishakemisto, johon tallennetaan termi sekä toiseen sarakkeeseen kaikkien alkuperäisessä tietokannassa olevien rivien id, joissa kyseinen termi esiintyy. (Manning, Raghavan ja Schütze 2008). Alkuperäisessä taulussa (taulukko 3) on haluttu sisältö, mistä muodostetaan käänteishakemisto.

Taulukko 3: Alkuperäinen taulu

id	sisältö
1	Apache Lucene API
2	Elasticsearch API
3	Apache Lucene ja Elasticsearch
4	Viimenen esimerkki

Taulukossa 4 on esimerkki sanakirjatiedostosta. Sanakirjatiedostossa on käänteishakemiston termit eli hakuavaimet ja esiintymien lukumäärä.

Taulukko 4: Sanakirjatiedosto

termi	esiintymien lukumäärä
Apache	2
API	2
Elasticsearch	2
esimerkki	1
ja	1
Lucene	2
Viimeinen	1

Käänteishakemistossa on termit, lisäksi siinä on aina alkuperäisestä taulusta löytyvän esiintymän id tai vastaava tunniste, jolla voidaan suoraan valita hakutermiä vastaava tiedosto. Esimerkki käänteishakemistosta on taulukossa 5.

Taulukko 5: Käänteishakemisto

termi	id
Apache	1,3
API	1,2
Elasticsearch	2,3
esimerkki	4
ja	3
Lucene	1,3
Viimeinen	4

Yllä olevassa käänteishakemistossa on myös indeksoituna *ja*, mutta yleensä sulkusanat (engl. stopwords) jätetään pois indeksistä, koska ne ovat merkityksettömiä haun kannalta. Toisaalta sulkusanojen pois jättämiseen ei nykyisillä tallennustiloilla enää ole merkitystä ja sulkusanojen tiputtaminen indeksistä voi jopa huonontaa hakua. Sulkusanojen tiputtaminen voi johtaa

siihen, että hakuosumien tarkkuus laskee. Esimerkiksi hakutermi *President of the United States* on huomattavasti tarkempi sulkusanojen kanssa kuin ilman niitä vertaa: *President United States* (Manning, Raghavan ja Schütze 2008, 27).

3.2 Haku kirjoittaessa

Termeillä haku kirjoittaessa (engl. search as you type) tai pikahaku (engl. instant search) tarkoitetaan hakutoteutusta, missä käyttäjän kirjoittaessa palvelu osaa antaa tuloksia tai täydennyksiä annettuun hakutermiin, vaikka termi ei olisi kokonaan kirjoitettu. Pikahaku on yleinen ominaisuus kaikissa suosituimmissa hakukoneissa ja sosiaalisen median sovelluksissa (Venkataraman ym. 2016).

Molemmat sekä Apache Solr ja Elasticsearch tarjoavat mahdollisuutta kyselyiden tekemiseen samalla kun käyttäjä kirjoittaa hakutermiä. Apache Solr:ssa *Suggester* hoitaa automaattiset täydennykset hakutermiin. Käyttöönottoa varten solrconfig.xml:ään pitää lisätä `<searchComponent name="suggest" class="solr.SuggestComponent">` ja antaa sille määritykset, mistä kentistä halutaan luoda automaattisia täydennyksiä (“Apache Solr” 2019).

Elasticsearchissa on tällä hetkellä neljä erilaista keinoa, joiden avulla voidaan toteuttaa automaattinen täydennys. Kuten Apache Solr:ssa myös Elasticsearchissa se kulkee nimellä *Suggester*. Elasticsearchin vaihtoehtoina ovat *term suggester*, *phrase suggester*, *completion suggester* ja *context suggester* (“Elasticsearch Documentation” 2019). Jotta käyttäjälle olisi hyötyä toiminnallisuudesta, joka hakee tuloksia samalla kuin hakutermiä kirjoitetaan, tulisi vasteajan olla mahdollisimman pieni, ja tulosten pitäisi olla lähes reaaliajassa käyttäjän nähtävillä.

Term - ja *phrase suggester* ovat melko samanlaisia, mutta siinä missä Term suggester tekee asioita vain termikohtaisesti phrase suggester huomioi koko hakulauseen. Molemmat ehdottavat vaihtoehtoja suoraan indeksoidusta datasta määritysten mukaisesti. Tärkeimmät määritykset ovat mistä kentistä haetaan, mitä analysoijaa käytetään ja kuinka paljon *korjauksia* tarjotaan kerralla. (“Elasticsearch Documentation” 2019). Käytännössä ne eivät siis itsessään toteuta pikahakua vaan toimivat enemmänkin oikeinkirjoituksen tarkistajina. Nämä

kaksi toteutusta eivät vaadi lisätietoja indeksiin kuten seuraavat kaksi toteutustapaa. Esimerkiksi *context suggester*, vaatii aina kontekstin lisäämisen indeksoitaessa, ja hakiessa tietoa indeksistä.

Completion suggester antaa mahdollisuuden luoda automaattisesti täydentyvän tai pikahaku tyylisen toiminnallisuuden indeksille. Tarkoituksena on ohjata käyttäjää relevantteihin tuloksiin samalla, kun käyttäjä kirjoittaa hakulauseketta. Sitä ei ole tarkoitettu samalla tavalla korjaamaan käyttäjän kirjoitusvirheitä kuin *term suggesteria* tai *phrase suggesteria*. *Completion suggester* on optimoitu nopeutta ajatellen ja se käyttää tietorakenteita, jotka mahdollistavat nopean haun. Huonoina puolina näissä tietorakenteissa ovat seuraavat asiat: ne ovat raskaita rakentaa ja samalla ne ovat tallennettuina vain muistiin, jonka seurauksena tietorakenteet joudutaan rakentamaan aina uudestaan, jos palvelin irrotetaan verkkovirrasta ja palvelin sammuu (“Elasticsearch Documentation” 2019).

Context suggester mahdollistaa kontekstin antamisen kyselyissä, millä voidaan mahdollistaa esimerkiksi maakohtaisen sisällön näyttämisen käyttäjän kohdemaan perusteella. Tämäkin vaihtoehto kuten *Completion suggester* vaatii sen, että indeksiin lisätään lisäkenttiä, joiden perusteella tuloksia rajataan (“Elasticsearch Documentation” 2019). Tämä ei ole kannattava keino, jos indeksin koko halutaan pitää pienenä.

Elasticsearchissa *Suggester* otetaan käyttöön antamalla indeksin mappings osiossa:

```
"properties" : {
                "suggest" : {
                    "type" : "completion"
                }
```

missä *type* kertoo mitä *suggester* tyyppiä halutaan käyttää.

3.3 Sumea haku

Sumea haku (engl. Fuzzy Search, approximate string matching) termiä käytetään silloin, kun viitataan merkkijonojen vertailuun sallien kirjoitusvirheet vertailtavissa merkkijonoissa. Käyttäjä voi esimerkiksi kirjoittaa hakutermin *Jyväskylö*, vaikka oikeasti olisi tarkoituksena

kirjoittaa *Jyväskylä*, mutta huolimatta kirjoitusvirheestä, hakukone osaa silti tarjota osumia, joissa sana Jyväskylä esiintyy (Manning, Raghavan ja Schütze 2008, 56).

Sumeassa haussa on tarkoituksena löytää hakua vastaavia sanoja indeksistä. Vastaavien sanojen määrittämisessä käytetään yleensä Levensteinin etäisyyttä (engl. Levenshtein distance). Yleensä kuitenkin puhutaan pelkästään editointietäisyydestä. Levensteinin etäisyys ilmoittaa pienimmän määrän operaatioita, joilla merkkijono str1 voidaan muuttaa merkkijonoksi str2 (Manning, Raghavan ja Schütze 2008). Merkkijonojen *Jyväskylö* ja *Jyväskylä* editointietäisyys on 1, koska vaaditaan vain yksi operaatio, jotta merkkijonoista saadaan vastaavat. Tässä tapauksessa ö vaihtuu ä:ksi. Sallittuja operaatioita merkkijonoille ovat merkin lisäys, poisto ja korvaus (Navarro 2001).

Editointietäisyyden laskevan algoritmin pseudokoodi:

```
editDistance(str1, str2)
{
    int c
    int e[i,j] = 0

    for i from 1 to length(str1)
        e[i,0] = i
    for j from 1 to length(str2)
        e[0,j] = j

    for i from 1 to length(str1)
        for j from 1 to length(str2)
        {
            # Jos merkit ovat samat etäisyys on 0, muuten 1
            str1[i] == str2[j] ? c = 0 : c = 1
            e[i,j] = min(
                        e[i-1, j] + 1, # Merkin poisto
                        e[i, j-1] + 1, # Merkin lisäys
                        e[i-1, j-1] + c # Merkin korvaus
                        )
         }
         return e[length(str1), length(str2)]
}
```

Algoritmi laskee kahden merkkijonon välisen etäisyyden vertailemalla niiden yksittäisiä merkkejä keskenään. Aluksi muodostetaan taulukko, jossa rivit ja sarakkeet määräytyvät vertailtavien sanojen mukaan. Jos merkit ovat samat on etäisyys 0. Algoritmi laskee, kuinka "kallista" merkin lisääminen, poistaminen tai korvaaminen on ja niistä pienin arvo on merkkien välinen etäisyys. Merkkijonojen välinen etäisyys löytyy siten taulukon viimeisen rivin viimeisestä sarakkeesta (Taulukko 6).

Taulukko 6: Editointietäisyys

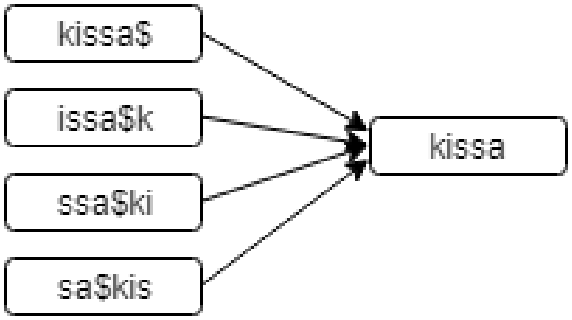
		K	o	i	r	a
	0	1	2	3	4	5
P	1	**1**	2	3	4	5
o	2	2	**1**	2	3	4
i	3	3	2	**1**	2	3
k	4	4	3	2	**2**	3
a	5	5	4	3	3	**2**

3.4 Korvausmerkit

Korvausmerkit eli jokerimerkit (engl. Wildcards) ovat tärkeitä tiedonhaussa. Niiden avulla voidaan etsiä osumia indeksistä, vaikka termin tarkkaa kirjoitustapaa ei tiedettäisi, esimerkiksi *color* tai *colour*. Korvausmerkkejä ovat * ja ?, tähdellä viitataan yhteen tai useampaan merkkiin ja kysymysmerkillä viitataan aina vain yhteen merkiin. Korvausmerkki voi olla joko termin alussa ?olor, keskellä col*r tai lopussa col* (Manning, Raghavan ja Schütze 2008, 52).

Korvausmerkki kyselyitä voidaan käsitellä permutaatiohakemiston (engl. permuterm index) tai k-grammi indeksin avulla (Manning, Raghavan ja Schütze 2008). Permutaatiohakemistoissa otetaan käyttöön *$*, millä viitataan termin lopetukseen eli hakutermi *color* muuttuu muotoon *color$*. Permutaatiohakemistoon tallennetaan kaikki termin rotaatiot (Kuvio 2.), minkä jälkeen ne liitetään alkuperäiseen termiin. Tarkoituksena on aina saada hakutermi sellaiseen muotoon, että korvausmerkki löytyy hakutermin lopusta, jolloin kyselystä *col*r*

saataisiin *r$col**. Tämän avulla saadaan aina termin käännökset ja oikeat sanamuodot, joita voidaan etsiä käänteishakemistosta, minkä avulla löydetään hakua vastaavat dokumentit (Manning, Raghavan ja Schütze 2008). Permutaatiohakemiston ongelmana on indeksin koko, koska kaikkien termien rotaatioiden tallentaminen vie tilaa.



Kuvio 2. Vajaa permutaatiohakemisto termistä kissa(Manning, Raghavan ja Schütze 2008, 54)

K-grammi on toinen keino toteuttaa jokerikysely. K-grammi on k:n peräkkäisen merkin jakso ja siinä termeihin liitetään alku- ja loppumerkki *$*. Termistä *castle* muodostuu siis *$ca, cas, ast, stl, tle, le$* 3-grammit (Manning, Raghavan ja Schütze 2008, 54). K-grammi indeksissä kaikki k-grammit ja k-grammin esiintymät osoittavat termiin, mikä sisältää kyseisen k-grammin. Tämä aiheuttaa ongelmaksi sen, että kaikki haut eivät enää tarkoita samaa kuin mitä käyttäjä on alunperin halunnut. Esimerkiksi termi *red** vastaa termiin *retired*, koska se sisältää *$re* ja *red*. Siksi k-grammi indeksin kanssa tarvitaan jälkisuodatusta, joka lisää kyselyn prosessointiaikaa (Manning, Raghavan ja Schütze 2008). K-grammi indeksit kuitenkin vaativat vähemmän tilaa kuin permutaatioindeksit, mutta haut voivat helposti muodostua raskaiksi, jos hakumoottorin tarvitsee yhdistellä suuria kokonaisuuksia.

3.5 Tekstin normalisointi

Erilaiset merkistökoodaukset aiheuttavat ongelmia hakukoneissa, koska käyttäjän mielestä visuaalisesti vastaavat kaksi merkkiä voivat olla samanlaisia, mutta oikeasti ne voivat koostua täysin eri tavuista, jolloin niitä ei käsitellä samoina merkkeinä (Manning, Raghavan ja Schütze 2008, 28). Siksi tekstin normalisointi on tärkeää, koska elämme unicode maailmas-

sa. Normalisoinnilla tarkoitetaan tekstin muokkaamista tietyn toimintamallin mukaiseksi, esimerkiksi suurilla kirjaimilla kirjoitettu teksti muutetaan aina pieniksi kirjaimiksi ja tämä tehdään joka kerta.

Normalisointi suoritetaan aina tekstialkiokohtaisesti (engl. Token) (Manning, Raghavan ja Schütze 2008, 28). Hakuterminä voisi olla *Jyväskylän koulut*, jolloin tekstialkioita on kaksi *Jyväskylän* ja *koulut*. Yleensä normalisoinnissa kannattaa muuntaa tekstialkiot pieniksi kirjaimiksi, jotta lauseen keskellä olevat sanat vastaavat myös haluttuun termiin. (Manning, Raghavan ja Schütze 2008, 28-32).

Tekstiä normalisoidessa kannattaa huomioida Unicode-normalisointi. Unicode-normalisoituja merkkijonoja voidaan verrata keskenään, jolloin saadaan tieto siitä, ovatko ne vastaavia. Vastaavuus voi olla joko kanoninen- tai yhteensopivuusvastaavuus (engl. compatibility equivalence) riippuen normalisointitavasta (Whistler 2019). Kanoninen vastaavuus tarkoittaa sitä, että esimerkiksi pieni e (U+0065) on kanonisesti vastaava merkin è (hajoitettuna U+0065 U+0300) kanssa. Yhteensopivuusvastaavuus tarkoittaa sitä, että merkit näyttävät samalta joten voidaan ajatella, että ne tarkoittavat samaa. Esimerkiksi fi-ligatuuri (U+FB01) on yhteensopiva merkkien fi (U+0066 U+0069) kanssa. Merkit, jotka ovat kanonisesti vastaavia ovat myös yhteensopivia, mutta yhteensopivat merkit eivät aina ole kanonisesti vastaavia, koska yhteensopivuusvastaavuus on heikompi näistä kahdesta (Whistler 2019).

Unicode normalisointi muotoja on neljä:

1. Normalization Form D (NFD),
2. Normalization Form C(NFC),
3. Normalization Form KD (NFKD) ja
4. Normalization Form KC (NFKC).

NFD ja NFC ovat kanonisia, sillä erolla, että NFD hajottaa merkit kanonisen vastaavuuden avulla, kun NFC:ssä merkit hajotetaan ja sitten kootaan kanonisen vastaavuuden avulla. NFKD ja NFKC käyttävät vastaavuuskriteerinä yhteensopivuutta. NFKD:ssa merkit hajoavat yhteensopivuuden mukaan ja NFKC:ssa merkit hajoavat, mutta ne kootaan sen jälkeen kanonisen vastaavuuden avulla. (Whistler 2019). Taulukossa 7 on esimerkki normalisointi muodoista.

Taulukko 7: Tekstin unicode normalisointi fi-ligatuurille (U+FB01)

Normalisointi muoto	hex
NFD	fb01
NFC	fb01
NFKD	66 69
NFKC	66 69

NFD ja NFC eivät erota f ja i erikseen, koska unicode merkkinä ne on sulautettu yhteen niin, että i:ssä ei ole pistettä päällä. NFKD ja NFKC taas poistavat muotoilun ja tuloksena tulee kaksi eri merkkiä pieni f ja pieni i. Whistlerin (2019) mukaan NFKD:ta ja NFKC:ta ei saa käyttää sokeasti tekstiin, koska ne poistavat muotoilun eroja ja voivat poistaa tekstin semantiikalle tärkeitä eroja.

3.6 Osumien korostus

Osa hakukoneista pystyy korostamaan (engl. Highlight) hakutermien osumat haettavasta dokumentista. Esimerkiki Google haku korostaa käyttäjän etsimät sanat hakutulosten joukosta, jolloin käyttäjä näkee suoraan tekstinkohdan, missä haettu sana on esiintynyt. Samalla korostuksella saadaan annettua visuaalinen vahvistus käyttäjälle siitä, miksi kyseinen dokumentti on tullut hakutuloksiin.

Lucene osaa korostaa sanoja tuloksista (“Apache Lucene” 2019). Tämän seurauksena molemmat sekä Solr, että Elasticsearch pystyvät näyttämään korostuksella hakuun osuneet sanat. Osumien korostamisen hyödylle ei löytynyt tieteellistä näyttöä, mutta monet käyttäjäkokemukseen eli UX(User Experience) keskittyneet sivut listasivat korostuksen hyödylliseksi keinoksi, joka tulisi olla haussa mukana mahdollisuuksien mukaan (Babich 2017; Tsech 2018).

4 Sisältöhaun suunnittelu ja toteutus

Tässä luvussa aluksi esitellään ongelma, jonka jälkeen käsitellään, millä alustalla työ toteutettiin sekä miksi tietty toteutustapa valittiin käytettäväksi. Luvussa käsitellään lisäksi, miten valittu toteutustapa toimii. Lopuksi tarkastellaan työn kanssa kohdattuja ongelmia.

4.1 Ongelman kuvaus

Tutkimuksen sovelluskohteena on Peda.net. Peda.net on yhteisöllinen oppimisympäristö, jota käytetään selaimella. Palvelu on tarkoitettu ensisijaisesti oppilaitosten ja yksittäisten oppilaiden sekä opettajien käyttöön. Palvelua voivat käyttää kuitenkin myös yhdistykset, yritykset ja yksityishenkilöt.

Peda.netin alkuperäinen haku on hyvin rajoittunut, sillä se on rajattu vain käyttäjän OmaTilaan, missä se löytää vain julkisen sisällön. Tämän seurauksena hakua eivät käytä opettajat, oppilaat eikä kukaan muukaan käyttäjä. Alkuperäinen haku on toteutettu käyttäen Google-hakua, ja sen antamat tulokset eivät auta käyttäjiä löytämään oikeaa sisältöä palvelusta. Lisäksi haku ohjaa pois Peda.net-palvelusta. Käyttäjällä ei ole mitään keinoa etsiä sisältöä koulujen sivuilta, koska haku on rajattu niin tiukasti. Esimerkiksi, jos käyttäjä haluaa löytää tietyn kurssisivun koulun sivuilta, ei hänellä ole mitään mahdollisuutta etsiä sitä, vaan sivu täytyy etsiä manuaalisesti palvelusta tai käyttämällä Googlea.

Tässä tutkimuksessa toteutetaan tekstihaku ja sen vaatima indeksointi palveluun. Näin vanha haku voidaan korvata uudella. Uuden haun tarkoitus on helpottaa palvelun käyttöä ja parantaa käyttäjien tyytyväisyyttä.

Tutkimuksessa joudutaan huomioimaan käyttäjien sivukohtaiset käyttöoikeudet. Sama käyttäjä voi olla samaan aikaan sekä ylläpitäjä että lukija. Tämä lisää työn kompleksisuutta ja voi vaikuttaa siihen, kuinka tehokas ohjelmasta saadaan.

4.2 Kehitysympäristön kuvaus

Kehitysympäristönä työssä käytettiin Ubuntua (KDE) ja IDEnä toimi PhpStorm. Ohjelmointikielenä käytettiin PHP:ta hyödyntäen Peda.netin omaa sovelluskehystä. Palvelimena toimi Ubuntu Server, jossa Elasticsearchia ajettiin paikallisesti.

Suurin osa alkuvaiheen kyselyistä ja Elasticsearchin asetuksista tehtiin käyttäen curlia, joka on komentorivityökalu tiedonsiirtoon eri protokollien mukaan. Alkumäärittelyiden jälkeen suurin osa testikyselyistä toteutettiin käyttäen Dejavu-nimistä lisäosaa Chromessa, joka mahdollistaa kyselyiden muokkaamisen ja suorittamisen selaimesta Elasticsearch indeksiin. Lopulliset kyselyt ja testaukset toteutettiin testipalvelimella, käyttäen haun graafista käyttöliittymää.

4.3 Valittu toteutustapa

Perinteinen tekstihaku käyttäen PostgreSQL:n omaa tekstihakua olisi ollut todella työlästä johtuen siitä, että kaikki data ei ole tallennettuna samaan tauluun. Tämä olisi vaatinut tekstihaun päälle kytkemisen moneen eri tauluun. Lisäksi tämä toteutustapa olisi johtanut todella pitkiin SQL-kyselyihin, jotka olisivat aina olleet dokumenttiriippuvaisia. Tämän takia toteutustavan skaalautuvuus olisi ollut erittäin heikko. Tekstihaun vaatimien indeksien luominen tietokantaan olisi ollut raskas operaatio, johtuen tietokannan suuresta koosta. Tässä toteutuksessa olisi myös pitänyt huomioida tuki erikoistapauksille liittyen merkkien normalisointiin, mikä olisi entisestään kasvattanut ylläpidollisia toimia. Indeksien luominen tuotantoversioon olisi myös vaatinut palvelun alasajoa, eikä sitä olisi pystytty toteuttamaan samalla tavalla kuin tässä työssä esitettyä Elasticsearchin indeksointia tausta-ajona, mitä voidaan suorittaa samalla kun palvelu on muuten toiminnassa. Joten erillistä huoltokatkoa ei tarvita siihen, että haku saadaan käyttöön.

Toteutustavaksi valikoitui Elasticsearch. Valintaan vaikutti se, että Elasticsearch on avoimeen lähdekoodiin perustuva, ja Peda.netissä on jo henkilöhaku toteutettu käyttäen Elasticsearchia. Tuntui luontevalle jatkaa saman hakumoottorin päällä ja kasvattaa klusteria vain uudella indeksillä dokumentteja varten. Elasticsearch vastaa myös hyvin Peda.netin tarpeisiin, koska se on helposti muokattavissa sopivaksi kyseiseen palveluun. Elasticsearchin tarjoama laa-

ja tuki erilaisille ilmaisille analysointityökaluille sekä Elasticsearchin oma lisäosan hallinta vaikuttivat myös lopulliseen valintaan.

Elasticsearch on tällä hetkellä suosituin hakukone (“DB-Engines Ranking of Search Engines” 2019), joten tuen saaminen ja jatkokehitys on taattua ainakin seuraaviksi vuosiksi. Myös suuret toimijat, kuten Wikipedia (Horohoe 2014) käyttävät Elasticsearchia, joten se todistetusti skaalautuu erilaiselle datalle ja on nopea suurillakin datamäärillä.

Apache Solr tippui sen seurauksena pois vaihtoehdoista, koska tarkoituksena ei ole indeksoida pelkästään tekstimuotoista dataa. Sisältöhaun tulee löytää myös muun tyyppistä tietoa tulevaisuudessa, esimerkiksi tiedostoja. Apache Solrin säätäminen olisi vienyt aikaa pois varsinaiselta suunnittelulta. Taulukossa 8 esitellään teoriaosuudessa esiteltyjen toteusratkaisuiden ominaisuuksia.

Taulukko 8: Tekstihakutyökalujen ominaisuudet.

	PostgreSQL	Apache Solr	Elasticsearch	Google CSE
Avoin lähdekoodi	x	x	x	
Web-hallintapaneeli		x	x*	
Lähes reaaliaikainen indeksointi	x	x	x	
Hakutulosten korostus		x	x	x
Skaalautuvuus		x	x	x**
Jokerikyselyt		x	x	x
Haku kirjoittaessa		x	x	

*Lisäosa ** Premium, joka maksaa.

4.4 Indeksin rakenne ja asetukset

Alkuperäinen indeksin rakenne oli muotoa `/pedanet/documents/document/id`, missä *pedanet* on klusterin nimi, *documents* on indeksin nimi ja *document* on tyyppi. Ensimmäinen indeksi sisälsi kentät *parentname, authorname, title* ja *data* ja relaatiotietona *parent*,

jossa relaatio oli *parentname* kenttään. Kentille ei oltu annettu ennakkoon mallia, millaista dataa ne sisältävät, vaan Elasticsearch sai itse päättää minkä muotoista dataa se niihin hyväksyy ensimmäisen dokumentin indeksoinnin jälkeen. Ongelmaksi muodostui heti se, että indeksistä puuttui title-kenttä, ja kaikki data oli tarkoitus pistää yhteen data kenttään, mistä tietoa ei olisi pystynyt erottelemaan kunnolla. Samalla selvisi, että relaatioiden käyttö Elasticsearchin indeksissä on huono ajatus, sillä se voi hidastaa indeksiä merkittävästi. Tämä tarkoitti siitä luopumista ja vaihtoehtoisen toteutustavan pohdintaa.

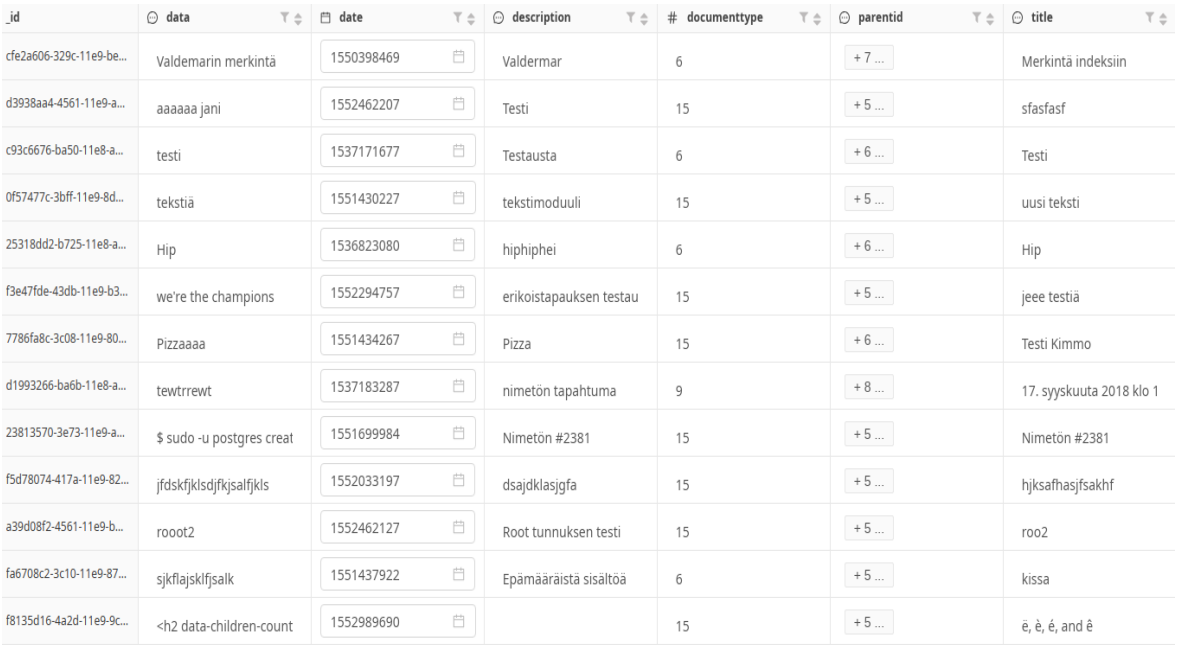
Seuraavassa versiossa data oli muuttunut muotoon:

```
"data":{
    "properties":{
        "title" :
        "content":
    }
}
```

missä ongelmaksi muodostui se, että indeksoitavasta datasta muodostettiin objekti, jossa oli eri kenttiä. Kenttien sisältöä ei oltu määrätty etukäteen ja ensimmäisessä testidokumentissa, joka lisättiin indeksiin, sattui olemaan *20-09-2018* otsikkona, jolloin Elasticsearch automaattisesti asetti kentälle *date* tyyppisen datatyypin. Tämä estää kaiken muun tyyppisen datan indeksoinnin kyseiseen kenttään. Toista testidokumenttia, jossa *title* kenttä sisälsi pelkästään tekstiä ei pystytty lisäämään indeksiin. Tämän seurauksena luovuttiin ajatuksesta, että Elasticsearchin annettaisiin luoda automaattisesti indeksin määritykset.

Seuraavassa versiossa *author* kenttä tiputettiin pois, koska selvisi, että kaikilla dokumenteilla ei ole selkeää tekijätietoa saatavilla. Koska tyhjien kenttien indeksoiminen ei ole tehokasta niin, paras ratkaisu oli jättää kenttä kokonaan pois.

Indeksin viimeisimmässä versiossa indeksin rakenne on tarkasti määritelty ja se poikkeaa ensimmäisestä indeksistä todella paljon. Siinä rakenne on muotoa `/pedanet/documents/_doc/id` ja *_doc* koostuu kentistä *data, date, description, documenttype, parentid ja title* (Kuvio 3). Data-kenttä sisältää kaikkien kyseisen dokumentin indeksoitavaksi halutun sisällön, eli käytännössä sitä voitaisiin ajatella vastaavana kuin, mitä alussa *content* oli.



_id	data	date	description	documenttype	parentid	title
cfe2a606-329c-11e9-be...	Valdemarin merkintä	1550398469	Valdermar	6	+ 7 ...	Merkintä indeksiin
d3938aa4-4561-11e9-a...	aaaaaa jani	1552462207	Testi	15	+ 5 ...	sfasfasf
c93c6676-ba50-11e8-a...	testi	1537171677	Testausta	6	+ 6 ...	Testi
0f57477c-3bff-11e9-8d...	tekstiä	1551430227	tekstimoduuli	15	+ 5 ...	uusi teksti
25318dd2-b725-11e8-a...	Hip	1536823080	hiphiphei	6	+ 6 ...	Hip
f3e47fde-43db-11e9-b3...	we're the champions	1552294757	erikoistapauksen testau	15	+ 5 ...	jeee testiä
7786fa8c-3c08-11e9-80...	Pizzaaaa	1551434267	Pizza	15	+ 6 ...	Testi Kimmo
d1993266-ba6b-11e8-a...	tewtrrewt	1537183287	nimetön tapahtuma	9	+ 8 ...	17. syyskuuta 2018 klo 1
23813570-3e73-11e9-a...	$ sudo -u postgres creat	1551699984	Nimetön #2381	15	+ 5 ...	Nimetön #2381
f5d78074-417a-11e9-82...	jfdskfjklsdjfkjsalfjkls	1552033197	dsajdklasjgfa	15	+ 5 ...	hjksafhasjfsakhf
a39d08f2-4561-11e9-b...	rooot2	1552462127	Root tunnuksen testi	15	+ 5 ...	roo2
fa6708c2-3c10-11e9-87...	sjkflajsklfjsalk	1551437922	Epämääräistä sisältöä	6	+ 5 ...	kissa
f8135d16-4a2d-11e9-9c...	<h2 data-children-count	1552989690		15	+ 5 ...	ë, è, é, and ê

Kuvio 3. Kuva indeksin rakenteesta dejavu selainlisäosalla

Date-kenttä on pakotettu hyväksymään vain aikaleimat, jotka vastaavat formaattia *epoch* millisekunnit. Tämä siksi, koska tietokannasta saatavaa tietoa ei tarvitse käsitellä mitenkään. Description-kenttään tallennetaan kyseisen dokumentin kuvaus, joka voidaan näyttää hakutuloksissa tunnistetietoina. Documenttype kertoo järjestelmän sisäisen moduulin tyypin. Tämän avulla voidaan painottaa tiettyjä moduuleita haussa. Esimerkiksi tekstimoduuli saa suuremman painoarvon kuin tapahtuma. Parentid-kentässä on taulukkona kaikkien vanhempien id:t, jotka kyseisellä dokumentilla ovat. Tämä kenttä on ratkaisevassa asemassa, kun haku halutaan kohdistaa vain johonkin tiettyyn polkuun (luku 4.8). Title-kentässä on kyseisen dokumentin otsikko. Otsikko on erotettu itse sisällöstä, koska otsikko-kentällä voi tietyissä tapauksissa olla enemmän painoarvoa kuin data-kentän sisällöllä ja siksi se tallennetaan eri kenttään. Kaikissa tekstityyppisissä kentissä on käytössä analysoija “nfkd_analyzer”, josta tarkemmin myöhemmin. Indeksin rakenne saadaan komennolla:

```
$ curl -X GET "localhost:9200/documents/_mapping/_doc?pretty=true
```

Mikä palauttaa alla olevan rakenteen.

```
"mappings" : {
```

```
"_doc" : {
  "properties" : {
    "data" : {
      "type" : "text",
      "fields" : {
        "keyword" : {
          "type" : "keyword",
          "ignore_above" : 256
        }
      },
      "analyzer" : "nfkd_analyzer"
    },
    "date" : {
      "type" : "date",
      "format" : "epoch_millis"
    },
    "description" : {
      "type" : "text",
      "fields" : {
        "keyword" : {
          "type" : "keyword",
          "ignore_above" : 256
        }
      },
      "analyzer" : "nfkd_analyzer"
    },
    "documenttype" : {
      "type" : "long"
    },
    "parentid" : {
      "type" : "text",
      "fields" : {
        "keyword" : {
          "type" : "keyword",
          "ignore_above" : 256
        }
      }
```

```
      },
      "title" : {
        "type" : "text",
        "fields" : {
          "keyword" : {
            "type" : "keyword",
            "ignore_above" : 256
          }
        },
        "analyzer" : "nfkd_analyzer"
      }
    }
  }
}
```

Kuten ylläolevasta tyypityksestä huomataan, kentissä on käytössä *analyzer*, joka on itse määritelty analysointisuodatin monipuolisempien hakutulosten saamiseksi. Indeksin asetukset saadaan selville komennolla:

```
$ curl -X GET "localhost:9200/documents/_settings?pretty=true"
```

missä parametri pretty=true palauttaa vastauksen luettavassa muodossa siististi sisennettynä.

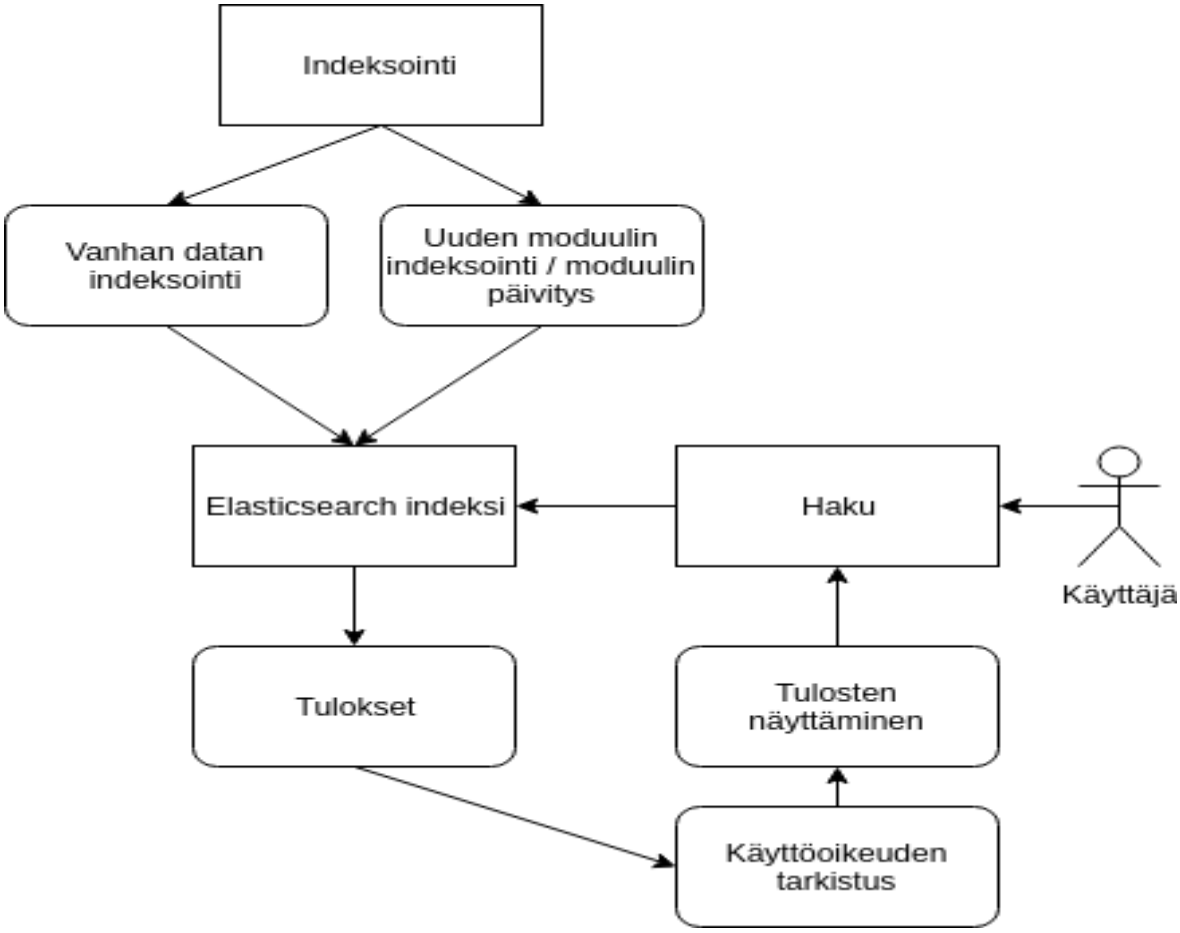
```
"documents" : {
    "settings" : {
      "index" : {
        "number_of_shards" : "5",
        "provided_name" : "documents",
        "creation_date" : "1554112656954",
        "analysis" : {
          "filter" : {
            "nfkd_normalized" : {
              "mode" : "decompose",
              "name" : "nfkc",
              "type" : "icu_normalizer"
            }
          },
          "analyzer" : {
```

```
      "nfkd_analyzer" : {
        "filter" : [
          "nfkd_normalized",
          "lowercase",
          "icu_folding"
        ],
        "tokenizer" : "icu_tokenizer"
      }
    }
  }
```

Oikeasti asetukset palauttaa paljon enemmän tietoa indeksistä, mutta ylläolevasta tulosteesta on osa tiedoista karsittu pois. Indeksiin on lisätty oma analysointi, joka käyttää omaa suodatinta *nfkd_normalized*, mikä on icu_normalizer tyyppiä ja käyttää unicode-normalisointimallia NFKD (luku 3.5). Lisäksi käytetään suodattimia *lowercase* ja *icu_folding*. *Lowercase* muuttaa kaikki hakutermit pieneksi, mikä mahdollistaa termin *Auto* löytämisen myös lauseen keskeltä, koska suodattimen avulla siitä saadaan *auto* ja *icu_folding* muuntaa è => e, ä => a, ó => o. Tämä mahdollistaa hakutuloksia, vaikka käyttäjä ei osaisi kirjoittaa hakutermia juuri oikealla merkillä (luku 4.7).

4.5 Ohjelman rakenne

Ohjelma koostuu kolmesta eri komponentista: Elasticsearch indeksistä, datan indeksoijasta sekä haun käyttöliittymästä (Kuvio 4). Datan indeksointi voidaan jakaa kahteen osaan: uuden datan indeksointiin ja vanhan datan indeksointiin (Luku 4.6). PostgreSQL tietokanta voidaan myös laskea osaksi ohjelmaa, koska sieltä ladataan vanhat datat indeksoitavaksi.



Kuvio 4. Ohjelman rakenne

Indeksointi on liitetty kahteen tapahtumaan palvelun käytössä. Ensiksikin siihen, kun käyttäjä luo uuden moduulin tai sivun, ja toiseksi siihen, kun sivua tai moduulia päivitetään. Kummassakin tapauksessa kutsutaan samaa metodia, joka yrittää lisätä dokumentin Elasticsearch-indeksiin. Mikäli indeksiin lisääminen kestää yli kaksi sekuntia tai se epäonnistuu jollain tavalla, luodaan datan lisäämisestä uusi työ työjonoon. Työtä yritetään tehdä heti, kun jollain palvelimella on aikaa. Jos työ epäonnistuu edelleen, merkataan kyseinen rivi epäonnistuneeksi ja kirjataan virhe lokiin.

Palvelusta (tässä tapauksessa sivu/moduuli) saadaan tarvittava data indeksiä varten metodeilla, jotka ylikirjoittavat pääluokassa määritetyt metodit. Tämä on mahdollista, koska kaikki moduulit periytyvät samasta pääluokasta. Moduulin saaminen indeksoitavaksi vaatii kahden metodin toteutusta: `isIndexable()` ja `getContentstoIndex()`. Oletuksena pää-

luokassa määritetään, että mikään ei ole indeksoitavaa, joten aliluokassa joudutaan ylikirjoituksella kertomaan, että kyseinen moduuli halutaan indeksoitavaksi. Jotta indeksoitavassa tiedossa olisi jotain järkeä, tulee luokan myös toteuttaa toinen metodi, millä kerrotaan, mitä dataa kyseisestä moduulista halutaan lisättäväksi indeksin datakenttään. Kaikkiin muihin kenttiin tiedot saadaan oletuksena.

4.6 Vanhan datan indeksointi

Vanhan datan indeksointi toteutetaan *cronjobin* (komento Linux-ympäristössä, millä voidaan aikatauluttaa töitä) avulla, ja kaikki vanha data indeksoidaan tausta-ajona. Indeksoinnin toteuttavan metodin koodi löytyy liitteestä 4. Joka minuutti järjestelmä hakee *y* määrän rivejä, jotka yritetään lisätä indeksiin. Koska työjonoa ajetaan rinnan, se vaatii PostgreSQL-tietokanta rivien lukitsemista, jotta eri prosessit eivät ota samoja rivejä käsittelyyn. Tämä aiheuttaa ongelmaksi sen, että jos samaa riviä muokataan jonkin toisen kutsun kautta, niin lukko estää kyseisen rivin muokkaamisen, ja kaikki muut kutsut odottavat indeksoinnin valmistumista. Rivien enimmäismäärä on rajattu ja yhdellä kutsulla lukitaan tietty määrä rivejä, jotta tutkimuksessa voidaan mitata suorituskykyä erilaisilla luvuilla. Lopullinen luku tulee määräytymään tutkimuksen tuloksien mukaan. SQL-lause on muotoa:

```
SELECT x from xxx FOR UPDATE SKIP LOCKED LIMIT y
```

`FOR UPDATE` lukitsee rivit ja `SKIP LOCKED` hyppää yli riveistä, jotka on merkitty päivitystä varten. `LIMIT` on käytössä vain tulosten rajaamisen vuoksi. Rivien määrä on rajattu kyselyssä, koska kaikkien tietokannan rivien valitseminen olisi liian raskasta ja muisti loppuisi kesken viimeistään siinä vaiheessa kun olioita alettaisiin käynnistämään (luku 5.1).

Indeksiin tarvittavien tietojen saaminen olisi myös mahdollista pelkästään SQL-kyselyiden avulla, mutta suuresta määrästä erilaisia tauluja ja niiden välisistä relaatioista johtuen eri palvelut vaatisivat eri kyselyt ja puhdasta SQL:ää joutuisi kirjoittamaan paljon enemmän. Tämä vaihtoehto olisi huono päivitettävyyden ja ylläpidon kannalta, koska mahdollisia muutoksia tehdessä joutuisi aina huomioimaan tämän luokan kyselyt.

4.7 Indeksin ominaisuudet ja niiden testaus

Tässä työssä kehitetty ohjelma mahdollistaa kaikkien Elasticsearchin tukemien ominaisuuksien käytön haussa, ominaisuudet täytyy vain ottaa käyttöön Elasticsearchin asetuksista. Oletuksena merkkijono normalisoidaan ja ajetaan eri suodattimien läpi, jotta saadaan enemmän osumia.

Elasticsearchiin saadaan asennettua lisäosia Elasticsearchin omalla lisäosan hallinnalla. Sitä voidaan käyttää Debian ympäristössä seuraavasti:

```
$ /usr/share/elasticsearch/bin/elasticsearch-plugin install analysis-icu
```

Tässä asennetaan analysis-icu-lisäosa indeksiin, mikä on Lucenen virallinen ICU moduli, joka lisää Unicode-tuen indeksiin. Indeksi joudutaan ensin sulkemaan, jotta muutoksia voidaan tehdä.

```
$ curl -XPOST localhost:9200/documents/_close
```

Yllä olevalla komennolla voidaan sulkea määrätty indeksi, jotta sitä voidaan muokata halutulla tavalla. Kuten lisätä halutut analysoijat indeksin eri kenttiin. Vastaavasti _open avaa indeksin taas uusille tiedostoille.

Asetusten muuttaminen tapahtuu seuraavasti curlin avulla:

```
$ curl -X PUT "localhost:9200/documents" -H 'Content-Type: application/json' -d'{
  "settings": {
        "analysis" : {
            "analyzer" : {
                "nfkd_analyzer" : {
                    "tokenizer" : "icu_tokenizer",
                    "filter" : ["nfkd_normalized", "lowercase", "icu_folding"]
                }
            },
            "filter" : {
                "nfkd_normalized" : {
                    "type" : "icu_normalizer",
                    "name" : "nfkc",
                    "mode" : "decompose"
```

```
                }
            }
        }
    }}'
```

Tämän jälkeen voidaan testata eri suodattimien toimintaa käyttämällä _analyze-komentoa. Koska kaikki suodattimet on asetettu omaan analysoijaan voidaan siihen viitata suoraan *analyzer* kohdassa. Parametrilla *explain* saadaan tarkemmat tiedot, joiden avulla merkkijonoja voidaan analysoida paremmin. Alla olevalla esimerkillä myös saadaan testattua, miten ligatuureja käsitellään haussa.

```
$ curl -X GET "localhost:9200/documents/_analyze?pretty=true"
-H 'Content-Type:application/json' -d'
{
  "analyzer" : "nfkd_analyzer",
  "explain": "true",
  "text":     "Is this déjà vu fi?"
}
'
```

Huomataan, että *icu-tokenizer* osaa pilkkoa hakutermin oikeista paikoista eli välilyönneistä. Toinen tärkeä havainto on, että kysymysmerkki häviää pois analysoitavasta hakutermistä. Merkit kuten piste tai kysymysmerkki eivät anna lisäarvoa haulle ja siksi ne voidaan tiputtaa pois luodessa tekstialkiot. Tässä voitaisiin myös käyttää *icu-tokenizerin* tilalla Elasticsearchin vakiona tulevaa *standard tokenizer:a*, mutta ICU antaa paremman tuen eri kielille, ja koska sitä käytetään jo Unicode merkkien takia niin ei ole syytä olla käyttämättä sitä. Tämän jälkeen itse tekstialkiot (tokenit) ajetaan suodattimien läpi.

```
"detail" : {
    "custom_analyzer" : true,
    "charfilters" : [ ],
    "tokenizer" : {
      "name" : "icu_tokenizer",
      "tokens" : [
        {
          "token" : "Is",
```

```
  "start_offset" : 0,
  "end_offset" : 2,
  "type" : "<ALPHANUM>",
  "position" : 0,
  "bytes" : "[49 73]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "this",
  "start_offset" : 3,
  "end_offset" : 7,
  "type" : "<ALPHANUM>",
  "position" : 1,
  "bytes" : "[74 68 69 73]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "déjà",
  "start_offset" : 8,
  "end_offset" : 12,
  "type" : "<ALPHANUM>",
  "position" : 2,
  "bytes" : "[64 c3 a9 6a c3 a0]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "vu",
  "start_offset" : 13,
  "end_offset" : 15,
  "type" : "<ALPHANUM>",
  "position" : 3,
```

```
        "bytes" : "[76 75]",
        "positionLength" : 1,
        "script" : "Latin",
        "termFrequency" : 1
      },
      {
        "token" : "fi",
        "start_offset" : 16,
        "end_offset" : 17,
        "type" : "<ALPHANUM>",
        "position" : 4,
        "bytes" : "[ef ac 81]",
        "positionLength" : 1,
        "script" : "Latin",
        "termFrequency" : 1
      }
    ]
  },
```


NFKD suodatin tekee juuri sen mitä sen halutaankin tekevän. Se muuntaa Unicode-merkit erillisiksi merkeiksi. Tämä voidaan todeta alla olevasta esimerkistä kohdasta *bytes*, tokenin *fi* kanssa.


```
"tokenfilters" : [
      {
        "name" : "nfkd_normalized",
        "tokens" : [
          {
            "token" : "Is",
            "start_offset" : 0,
            "end_offset" : 2,
            "type" : "<ALPHANUM>",
            "position" : 0,
            "bytes" : "[49 73]",
            "positionLength" : 1,
            "script" : "Latin",
            "termFrequency" : 1
          },
```

```
{
  "token" : "this",
  "start_offset" : 3,
  "end_offset" : 7,
  "type" : "<ALPHANUM>",
  "position" : 1,
  "bytes" : "[74 68 69 73]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "déjà",
  "start_offset" : 8,
  "end_offset" : 12,
  "type" : "<ALPHANUM>",
  "position" : 2,
  "bytes" : "[64 c3 a9 6a c3 a0]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "vu",
  "start_offset" : 13,
  "end_offset" : 15,
  "type" : "<ALPHANUM>",
  "position" : 3,
  "bytes" : "[76 75]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "fi",
  "start_offset" : 16,
  "end_offset" : 17,
```

```
        "type" : "<ALPHANUM>",
        "position" : 4,
        "bytes" : "[66 69]",
        "positionLength" : 1,
        "script" : "Latin",
        "termFrequency" : 1
      }

    ]
  },
```


Lowercase tekee nimensä mukaisesti kirjainten muunnoksen pieniksi kirjaimiksi, jotta sanat pystytään löytämään myös lauseiden keskeltä. Tässä on tärkeää huomata, että fi-ligatuuri on edelleen ensimmäisen suodattimen muuntama, joten lowercase-suodattimen ei tarvitse käsitellä alkuperäistä tokenia.


```
{
    "name" : "lowercase",
    "tokens" : [
      {
        "token" : "is",
        "start_offset" : 0,
        "end_offset" : 2,
        "type" : "<ALPHANUM>",
        "position" : 0,
        "bytes" : "[69 73]",
        "positionLength" : 1,
        "script" : "Latin",
        "termFrequency" : 1
      },
      {
        "token" : "this",
        "start_offset" : 3,
        "end_offset" : 7,
        "type" : "<ALPHANUM>",
        "position" : 1,
        "bytes" : "[74 68 69 73]",
```

```
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "déjà",
  "start_offset" : 8,
  "end_offset" : 12,
  "type" : "<ALPHANUM>",
  "position" : 2,
  "bytes" : "[64 c3 a9 6a c3 a0]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "vu",
  "start_offset" : 13,
  "end_offset" : 15,
  "type" : "<ALPHANUM>",
  "position" : 3,
  "bytes" : "[76 75]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
},
{
  "token" : "fi",
  "start_offset" : 16,
  "end_offset" : 17,
  "type" : "<ALPHANUM>",
  "position" : 4,
  "bytes" : "[66 69]",
  "positionLength" : 1,
  "script" : "Latin",
  "termFrequency" : 1
}
```

```
    ]
  },
```

Viimeisenä suodattimena oleva *icu_folding* toiminta tulee hyvin selville esimerkkikoodin kohdasta *deja* (token), missä voidaan nähdä, että erikoismerkit on muutettu tavallisiksi e ja a, minkä seurauksena termi vastaa kaikkia versioita siitä, myös väärinkirjoitettuja.

```
{
  "name" : "icu_folding",
  "tokens" : [
    {
      "token" : "is",
      "start_offset" : 0,
      "end_offset" : 2,
      "type" : "<ALPHANUM>",
      "position" : 0,
      "bytes" : "[69 73]",
      "positionLength" : 1,
      "script" : "Latin",
      "termFrequency" : 1
    },
    {
      "token" : "this",
      "start_offset" : 3,
      "end_offset" : 7,
      "type" : "<ALPHANUM>",
      "position" : 1,
      "bytes" : "[74 68 69 73]",
      "positionLength" : 1,
      "script" : "Latin",
      "termFrequency" : 1
    },
    {
      "token" : "deja",
      "start_offset" : 8,
      "end_offset" : 12,
      "type" : "<ALPHANUM>",
      "position" : 2,
```

```
            "bytes" : "[64 65 6a 61]",
            "positionLength" : 1,
            "script" : "Latin",
            "termFrequency" : 1
          },
          {
            "token" : "vu",
            "start_offset" : 13,
            "end_offset" : 15,
            "type" : "<ALPHANUM>",
            "position" : 3,
            "bytes" : "[76 75]",
            "positionLength" : 1,
            "script" : "Latin",
            "termFrequency" : 1
          },
          {
            "token" : "fi",
            "start_offset" : 16,
            "end_offset" : 17,
            "type" : "<ALPHANUM>",
            "position" : 4,
            "bytes" : "[66 69]",
            "positionLength" : 1,
            "script" : "Latin",
            "termFrequency" : 1
          }
        ]
      }
    ]
  }
}
```


Tekstin ja merkkien tunnistamisen ominaisuudet ovat siis lähes erinomaiset kyseisillä testitapauksilla. Toki kaikkia mahdollisia variaatioita ei pystytä testaamaan, ja niihin voidaan puuttua vasta ongelmien ilmentyessä. Esimerkki kuitenkin osoittaa, että suodattimet toimivat suunnitellusti ja tämän seurauksena ne parantavat osumien tarkkuutta. Jolloin oikeinkirjoit-

tamisen vastuuta siirretään käyttäjältä järjestelmälle.

4.8 Haun rakenne ja toiminta

Haku koostuu kahdesta eri hakukyselystä Elasticsearchin indeksiin, koska hauilla on selkeästi eri tavoitteet. Toisella halutaan löytää sisältöä koko palvelusta ja toisella vain tarkasti määritetystä polusta, esimerkiksi kunnan Jyväskylä alta. Siksi toinen haku on yhdistelmä Boolen kyselyitä, jossa on asetettu tietty määrä vastaavia ehtoja, joiden täytyy toteutua. Hakukyselyt löytyvät tutkielman liitteistä.

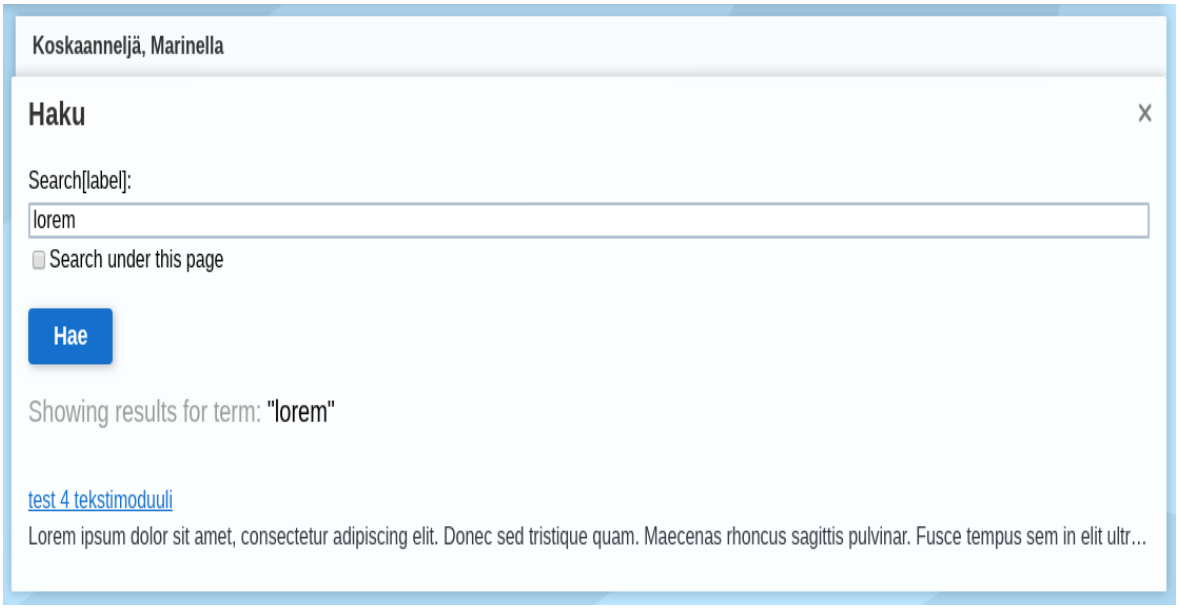
Haku, joka etsii tietyn polun alta olevia dokumentteja, on toteutettu käyttäen Boolen hakua ja jokerimerkin käytön mahdollistavaa hakutermin käsittelyä. Ensimmäinen Boolen haku tehdään dokumentin id:tä vasten, koska jokaisella indeksoidulla dokumentilla on koko polku tallennettuna *parentid*-kenttään, täytyy kyseisestä kentästä löytyä vastaavuus dokumentin omaan id:seen, mistä käsin haku on aktivoitu. Tämän jälkeen etsitään itse hakutermiä vastaavia kohteita niistä tuloksista, jotka ovat löytyneet ensimmäisellä haulla. Elasticsearch mahdollistaa sisäkkäisten hakujen toteuttamisen.

Toinen koko palvelusta etsivä kysely on muodostettu käyttäen Elasticsearchin *query_string*-tyyppistä kyselyä. Tämän kyselyn kanssa ei ollut samalla tavalla rajoittavia tekijöitä kuten aiemmin mainitun tietystä polusta etsivän kyselyn kanssa. Siksi tämä toteutus eroaa hieman toisesta toteutuksesta.

```
"query:
    {
    "query_string":
        {
        "default_field":"*",
        "analyzer":"nfkd_analyzer",
        "analyze_wildcard":"true",
        "query":"%term%",
        "default_operator":"and"
        }
    }
```

Yllä olevassa kyselyssä mahdollistetaan kaikista indeksistä olevista kentistä etsiminen, siinä analysoijaksi määrätään itse tehty analysoija, *analyze_wildcard* mahdollistaa jokerimerkkien käyttämisen haussa ja viimeiseksi operaattori muutetaan *OR => AND*. Kysely käyttäen *query_string* mahdollistaa sellaisten kyselyiden tekemisen, joissa voidaan etsiä esimerkiksi hakutermillä *(koulu) OR (Jyväskylä)*. Tällöin ylikirjoitetaan asetettu *and*-operaattori ja Elasticsearch palauttaa tuloksia, jotka vastaavat joko termiin koulu tai Jyväskylä. Oletuksena tässä toteutustavassa on myös aloittavien jokerimerkkien tuki päällä, joten hakutermit voidaan aloittaa esimerkiksi **lu*.

Haussa on mahdollista valita, halutaanko haettavan koko indeksin kaikista dokumenteista vai tietyn polun alla olevista dokumenteista (Kuvio 5). Tämä auttaa käyttäjää rajaamaan tuloksia ja voi auttaa saamaan tarkempia osumia.



Kuvio 5. Kuvakaappaus hausta

4.9 Ongelmat toteutuksen kanssa

Tutkimuksessa suurimmaksi ongelmaksi muodostui vanhan jo olemassa olevan datan indeksoiminen. Tästä johtuen tässä työssä ei vielä toteutettu hakua, joka osaa etsiä myös tiedostoista sopivia osumia. Pohjalla oleva tietokanta koostuu monesta taulusta ja vanhojen moduulien kanssa täytyi olla äärimmäisen varovainen, koska siellä on saatettu toteuttaa joskus

metodi, mikä nykyään ylikirjoittaa jonkun toisen metodin. Näin kävi tämän työn kanssa.

Ei indeksoitavassa moduulissa oli ylikirjoitettu metodi, joka aiheutti 65 000 rivin indeksoinnin epäonnistumisen 80 kertaa. Käytännössä tämä ei tarkoittanut indeksoinnin pysähtymistä 80 minuutiksi vaan kyseessä oli usean tunnin mittainen pysäytys, koska aina tehtävän epäonnistuessa sen uudelleen aloittamisen aika kasvaa eksponentiaalisesti. Toteutustavasta johtuen uutta työtä ei voitu ottaa käsittelyyn ennen kuin tämä työ oli saatu valmiiksi. Tämän aikana mitään tietoa ei saatu indeksoitua, vaan koko prosessi oli jumissa ja virheen selvittämisessä kesti lähes päivä. Tämä hidasti merkittävästi testi-indeksin luomista.

5 Tulokset ja pohdinta

Tässä luvussa käsitellään tutkimuksen tuloksia ja pohditaan niiden mielekkyyttä. Aluksi käsitellään toteutuksen tehokkuutta, jonka jälkeen tarkastellaan miten hyvin hakutulokset vastaavat hakutermiä. Lopuksi arvioidaan toteutettua artefaktia.

5.1 Tehokkuus

Koodin tehokkuutta mitattiin seuraamalla, miten kauan dokumenttien (Peda.netin eri sivut/työvälineet) indeksoinnissa kestää työjonon kautta sekä miten nopeasti Elasticsearch pystyy löytämään oikeita tuloksia indeksistä. Testattavan indeksin koko oli 24 438 625 dokumenttia, mikä on noin viidesosa itse palvelun kaikista dokumenteista. Elasticsearch palauttaa JSON-muodossa *took:value*-parina sen, miten kauan Elasticsearchilta meni kyselyn prosessoimiseen. Siinä ei ole mukana pyynnön - tai vastauksen lähetysviivettä ("Query timing: took value and what I'm measuring" 2012). Indeksiin lisätyt dokumentit olivat eri mittaisia ja sisällöltään erikokoisia sekä sisällössä eri dokumenttien välillä oli vaihtelua. Indeksin koko levyllä oli 35,5 gb ja se oli jakautunut viidelle solmulle (solmun koko levyllä 7,1 gb) ja indeksi koostui kymmenestä sirpaleesta.

PHP:ssa indeksoinnin tehokkuutta mitattiin `microtime()` avulla siten, että metodin alussa asetettiin aloitusaika ja lopuksi kun kaikki toiminnot oli suoritettu otettiin uudestaan aika. Kokonaisaika saatiin vähentämällä lopetusaika aloituksesta. Lisäksi palvelinkonetta seurattiin Linuxin `top`-komennon avulla.

Suurin osa haun testaamisesta toteutettiin käyttäen tunnusta, jolla oli oikeudet kaikkeen sisältöön, ja joissain tapauksissa hakua testattiin tunnuksella, jolla oli rajoitettu oikeus sisältöön. Tämä siksi, jotta voitiin testata vaikuttaako käyttöoikeuksien laskenta haun nopeuteen.

5.1.1 Indeksoinnin tehokkuus

Indeksointi osoittautui yllättävän nopeaksi. Ensioletus oli, että indeksointi olisi ollut paljon hitaampaa kuin mitä tuloksissa saatiin selville. Tämä johtuu siitä, että PHP-olioiden pitämi-

nen muistissa on raskasta ja testipalvelin koostuu vanhoista komponenteista, mikä vaikuttaa sen suorituskykyyn negatiivisesti. Tulokset ovat kuitenkin suuntaa antavia, vaikka oikeat palvelimet ovat paljon tehokkaampia kuin testauksessa käytetty palvelin.

Tässä luvussa palvelut ja moduulit viittaavat Peda.netin sisällä käytettyihin työkaluihin. Esimerkiksi OmaTilaan lisättävä tekstityöväline on yksi palvelu tai moduuli samoin kuin uusi sivu. Alle 5000:en moduulin indeksointi oli melko triviaalia: 100:n moduulin indeksointi kesti 198 ms, tuhannen 1,4 sekuntia ja 5000 moduulia saatiin indeksoitua 7,5 sekuntiin (Kuvio 6). Indeksoitavien moduulien määrän noustessa 20 000 alkoi indeksointinopeus selkeästi hidastua ja 30 000 moduulin indeksoinnissa kesti jo lähes minuutti. Niiden indeksointi kesti keskimäärin 50-57 sekuntia. Merkittävä hidastuminen saatiin kuitenkin aikaiseksi nostaessa moduulien määrää 35 000:een. Tällöin olioiden käynnistämisen raskaus alkoi selkeästi näkyä, ja indeksointiaika oli 61-160 sekunnin välillä. Tähän vaikutti se, montako valituista riveistä jouduttiin lisäämään indeksiin ja moneenko riitti vain indeksoinnin tilaa kuvaavan sarakkeen arvon muuttaminen -1, mikä tarkoittaa, että moduulille ei ole vielä toteutettu indeksoinnin mahdollistavia metodeja. Sarakkeen arvon muuttaminen tehdään suoraan SQL-lauseella, ilman olion metodeja, joten se on nopeaa.

Merkittävää nousua indeksointiajassa ei huomattu valittaessa 35 000-50 000 riviä tietokannasta. Seuraava suuri hyppy tuli vasta valittaessa kyselyllä 70 000 riviä. Tämän moduulimäärän indeksoiminen kesti keskimäärin 283 sekuntia, mikä on huomattavan hidasta ottaen huomioon, että nykyisillä järjestelmillä 100 000 ei ole erityisen suuri luku. Valittujen moduulienmäärän noustessa yli 100 000:een indeksoinnin hitaus alkoi jo selkeästi näkyä, keskiarvo eri indeksointi kertojen välillä oli 444 sekuntia, valittaessa kerralla 100 000 riviä tietokannasta. Nostaessa valittujen tietokantarivien lukumäärää indeksointi hidastui entisestään. Kuten kuviosta 6 nähdään 150 000:en moduulin indeksoiminen kesti jo 600 sekuntia ja määrän noustessa 200 000:een indeksointiaika oli 712 sekuntia. Tuotannossa 712 sekunnin viive on äärimmäisen huono, koska se tarkoittaa sitä, että mikään muu kutsu ei pysty muokkaamaan lukittua riviä ennen kuin indeksointi on tehty. Tämän takia näin raskasta indeksointia ei voida julkaista tuotantoon, koska se aiheuttaisi merkittäviä katkoja käyttäjien toimissa.

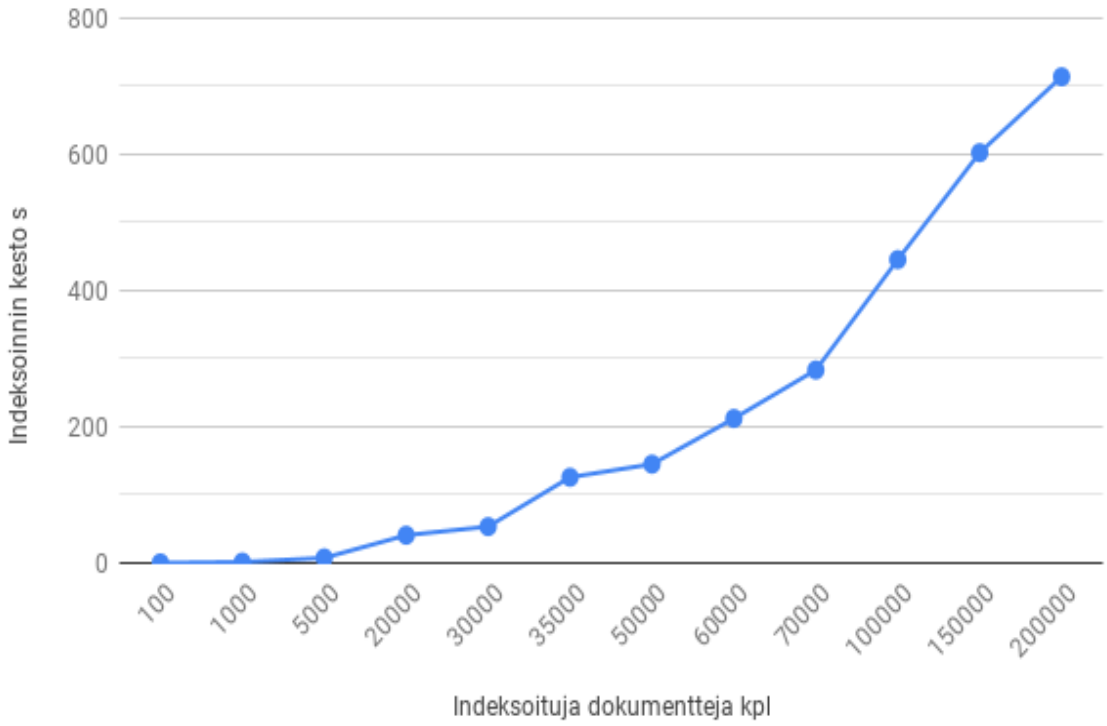
Näiden lukujen perusteella ei ollut enää järkevää nostaa valittuja rivejä yli 200 000, koska palvelin alkoi selkeästi hidastumaan, ja testipalvelimen muisti loppui kesken jo 70 000 mo-

duulin kohdalla. Selvästi 16 GB RAM ei ole riittävästi, jos halutaan indeksoida paljon moduuleja siten, että niitä käsitellään PHP-olioina. Itse prosessorin tehokkuus ei tullut vastaan testeissä vaan ongelmana oli jokaisessa tapauksessa muisti ja sen vähyys.

Kaikki yli minuutin kestävät työt ovat huonoja, koska ne hidastavat indeksointia. Uutta indeksointipyyntöä ei saada tehtyä yhdellä koneella, jos edellinen on vielä kesken. Joten paras ratkaisu on käyttää lukua väliltä 30 000-35 000, jotta indeksointiaika saadaan pidettyä mahdollisimman lähellä tai alle minuutin. Jos halutaan olla varmoja, että indeksointi ei tule kestämään yli minuuttia, on varminta olisi valita korkeitaan 30 000 moduulia kerralla.

Toinen ratkaisu olisi merkittävästi pienentää indeksoitavien moduulien lukumäärää esimerkiksi valitsemalla yhdellä kutsulla vain tuhat tietokantariviä. Tällöin voitaisiin kutsua uudelleen kyseistä indeksointimetodia, jos aikaa on kulunut alle minuutti. Valittaessa tuhat riviä kerrallaan saataisiin indeksoitua tutkimuksen tulosten perusteella lähes 42 000 moduulia minuutissa, mikä on suurempi luku kuin kerralla valittaessa suuri määrä moduuleja indeksoitavaksi. Tuloksen mukaan 40 000 moduulin indeksoiminen kerralla kestäisi yli kaksi minuuttia, koska 35 000 moduulin indeksoiminen kerralla kesti keskimäärin 120 sekuntia.

Tuhannen rivin lukitseminen kerralla tarkoittaisi myös lähes olematonta hieman yli sekunnin viivettä käyttäjille, koska niiden käsittely on niin nopeaa. Tämän seurauksena palvelun käyttö ei keskeytyisi aina kun työjonosta otetaan indeksointi käsiteltäväksi. Siksi pienen rivimäärän käsittely kerralla on suositeltu ratkaisuvaihtoehto. Tämä toteutus myös rasittaa palvelimia vähemmän, koska muistia ei varata niin paljoa kerralla. Testien mukaan, jos valittujen tietokantarivien määrä pidetään alhaisena niin suorituskyky on parempi.



Kuvio 6. Indeksoinnin tehokkuus

Moduulien indeksointi, joilla ei ollut indeksoinnin mahdollistavia metodeja kesti lähes poikkeuksetta 50 sekuntia. Moduulien määrää vaihdeltiin välillä 35 000-500 000 ja kaikki antoivat tuloksena 49-53 sekuntia, mikä oli hieman ristiriitaista muiden tulosten kanssa. Mitään poikkeavaa ei löydetty järjestelmästä, mikä selittäisi nämä tulokset. Osittain tähän saattoi vaikuttaa se, että tietokantaan kirjoitettiin puhtaalla sql:llä sen sijaan, että olisi käytetty olioita sarakkeen arvon muuttamiseen.

Indeksoinnin aikana Elasticsearch-indeksi ei osoittanut hidastumista vaan hidastumisen syynä oli aina PHP:n muistinvaraus. Siksi Elasticsearchin avulla voidaan indeksoida varmasti enemmänkin dokumentteja kerralla kuin, mitä tässä testattiin, mikäli tietojen syöttö Elasticsearch-indeksiin saadaan tehokkaammaksi.

5.1.2 Haun tehokkuus

Elasticsearchin tulokset *took*-kentästä katsoen ovat erittäin hyviä. Koko indeksistä etsittäessä hakutermillä *Elasticsearch* osumia saadaan seitsemän ja kyselyssä kesti Elasticsearchin puolella 1056 ms. Rajatessa haku testihenkilön OmaTilaan, missä yhdessä dokumentissa esiintyy kyseinen hakutermi, kesti kyselyssä 167 ms, ja osumia tuli vain yksi. Mikä tarkoittaa sitä, että polun rajaus toimii.

Tietystä polusta etsiminen on siis nopeampaa Boolen haulla kuin kaikista mahdollisista dokumenteista hakutermin etsiminen. Tämän johtuu siitä, että itse hakutermin etsiminen tehdään pienemmästä määrästä dokumentteja, jolloin tuloksen kuuluukin olla parempi.

Mielenkiintoista oli huomata, että hakiessa uudestaan samalla hakutermillä tulokset tulivat nopeammin, joten Elasticsearchin oma kyselyn välimuistiin tallentaminen toimii erinomaisesti. Aiemmin mainittu hakutermi *Elasticsearch* uudelleen etsittynä antoi kahden millisekuntin viiveen mitä voidaan pitää huomattavana parannuksena ensimmäiseen yli sekuntin kestäneeseen kyselyyn.

Toinen testattava termi oli *koulu* mikä on yleinen sana Peda.netissä. Kyselyssä kesti Elasticsearchilla 2143 ms ja osumia tuli indeksistä 145 187. Toisella kysely kerralla kun hakutermi oli tallennettuna välimuistiin, viive oli 21 ms.

Jokerikyselyiden kanssa suorituskyky oli erittäin hyvä. Järjestelmän laajuudella etsiessä hakutermillä *op** kyselyn viive oli vain 2269 ms, vaikka tuloksia saatiin 6 225 917. Elasticsearchin suorituskyky osoittautui testeissä todella hyväksi. Jokerikysely rajattuna tiettyyn polkuun ei ollut merkittävästi nopeampi verrattuna koko järjestelmästä etsivään jokerikyselyyn. Esimerkkikuntaan rajattuna hakutermi *op** kyselyn viive oli 2011 ms ja osumia tuli 7380. Viivessä oli vain 258 ms ero verrattuna koko järjestelmästä etsineeseen kyselyyn.

Indeksi on tarkoitus pitää vapaana tiedosta, millä ei ole haun kannalta merkitystä, joten käyttäjän oikeus nähdä löydetyt dokumentit lasketaan vasta, kun hakutermiä vastaavat dokumentit on löydetty indeksistä. Tämä lisää hieman viivettä näkymän renderöinnissä käyttäjälle ja siten hidastaa hakua.

Käyttöoikeuksien laskenta ei osoittautunut niin raskaaksi kuin aluksi pelättiin. Koko järjes-

telmästä etsittäessä hakutermillä *op**, käyttöoikeuksien laskenta vei 233 ms ja ottaen huomioon, että tuloksia saatiin 6,2 miljoonaa, voidaan suorituskykyä pitää hyvänä. Koska 200 ms ei ole merkittävä viive tulosten näyttämisen kannalta. Tosin järjestelmän ei tarvitse laskea käyttöoikeutta kaikkiin osuman saaneisiin dokumentteihin, koska Elasticsearchin tulokset on rajattu sataan kappaleeseen. Käyttöoikeudet lasketaan aina vain sadalle palvelulle kerralla, mikä pienentää viivettä.

5.2 Hakutulosten osuvuus

Tässä työssä indeksissä ei ole käytössä n-grammi ominaisuutta, vaikka sen avulla mahdollisesti voitaisiin saada tarkempia tuloksia. Tämä sen takia, koska työssä pyrittiin painottamaan tehokkuutta. N-grammin käyttö erään tutkimuksen mukaan vaatii enemmän levytilaa ja on muutenkin raskaampaa palvelimelle. Siksi sitä ei suositella toteutustavaksi, jos datamäärät ovat suuria (Thacker, Pandey ja Rautaray 2016).

Haun osuvuus pyritään varmistamaan tukemalla haussa jokerimerkkejä sekä haun rajauksella vain tiettyyn polkuun. Rajauksen avulla pystytään välttämään se, että oppilas Jyväskylästä löytäisi sisältöä muista kunnista. Rajaus myös toimii tehokkaasti, joten Boolen haku on hyvä keino rajata tuloksia.

Sisäkkäisillä Boolen hauilla toteutettuna tietystä polusta etsiminen osoittautui huonoksi vaihtoehdoksi, koska osumien tarkkuus laski huomattavasti. Testitapauksessa indeksissä oli yksi esiintymä, missä esiintyi sana *Elasticsearch*, minkä olisi pitänyt löytyä kyseisestä polusta etsittäessä hakutermeillä *Ela** tai *Elasticsearch*, mutta Boolen haku ei löytänyt mitään tuloksia indeksistä kyseisillä hakutermeillä. Vastaavasti Boolen haku yhdistettäessä luvussa 4.8 esitettyyn *query_string* tyyppiseen hakuun löysi kyseisen dokumentin molemmilla hakutermeillä (Kuvio 7). Samalla haku huomioi oikein polun rajauksen.

Kuvio 7. Hakutuloksen osuvuus

Toteutettu haku osaa myös käsitellä emoji-merkkejä oikein. Dokumentti voi sisältää emojeita ja etsittäessä emojilla, joka esiintyy dokumentissa, näyttää järjestelmä vastaavuuden oikein hakutuloksissa, jos käyttäjällä on kyseiseen dokumenttiin lukuoikeus.

Tällä hetkellä haku ei korosta osumia näytetyistä tuloksista, mikä on selkeä kehityskohde tulevaisuudessa. Tämän toteuttamisen ei pitäisi myöskään olla työlästä, koska Elasticsearch tukee toimintoa suoraan. Indeksiin tallennetaan myös tyhjiä kenttiä, koska kaikilla moduuleilla ei ole kuvauskentässä mitään sisältöä, jolloin se laskee osumien tarkkuutta. Tulevaisuudessa tätä voitaisiin parantaa lisäämällä oletuksena sisältöä kuvaus kenttään tai painottamalla käyttäjille, että moduuli ei välttämättä löydy yhtä helposti haulla, jos kuvaus jätetään tyhjäksi.

Hakiessa useammalla hakutermillä esimerkiksi *oppilas kurssi* tai *(oppilas) OR (kurssi)*, huomattiin haun toimivan oletetulla tavalla. Tuloksina saatiin selkeästi eri dokumentit ja pakotettu *AND* (Luku 4.8) antaa osuvimpia tuloksia. Pakottaessa vertailuoperaattori muotoon *OR* laskee dokumenttien relevanttius selkeästi. Tämä johtuu siitä, että operaattorilla *OR* riittää vain, että toinen termeistä esiintyy dokumentissa ja siksi tulosten osuvuus laskee.

Ongelmia kuitenkin esiintyy etsittäessä ilman jokerimerkkiä. Esimerkiksi voidaan etsiä hakutermillä *Jyväskylä*, ja käyttäjä olettaa, että kyseinen termi myös vastaisi lauseeseen, missä esiintyy *Jyväskylässä*. Tämä ei kuitenkaan ole mahdollista nykyisillä asetuksilla, koska jär-

jestelmä ei tue osittaisia osumia ilman jokerimerkin käyttöä. Tämä vaatisi N-grammin käyttöönoton indeksiin. Siitä seuraisi suurempi tallennustilan tarve sekä uudelleen indeksointi voisi tulla todella raskaaksi toiminnoksi. Satojen miljoonien dokumenttien indeksoiminen ei ole kevyttä, vaikka Elasticsearch onkin tehokas.

5.3 Arviointi

Tuloksia voidaan pitää osuvina (Luku 5.2), koska ne noudattavat pitkälti sitä, miten Manning (2008) oli asian esittänyt kirjan *Introduction to Information Retrieval* luvussa kahdeksan. Hakua testatessa huomioitiin se, että aineisto sisältää erilaisia dokumentteja ja osassa esiintyy sanoja, jotka eivät ole oikeasti oleellisia dokumentin muun sisällön kanssa.

Elasticsearchin oma pisteytys tuntui toimivan hyvin ja se automaattisesti karsi pois huonoja tuloksia. Pisteytyksessä korkealla olleet dokumentit vastasivat lähes poikkeuksetta hakutermiin parhaalla mahdollisella tavalla. Toisaalta ongelmana oli se, jos käyttäjällä ei ollut oikeuksia kaikista parhaiten osuviin dokumentteihin, niin haun palauttamat tulokset saattoivat oikeasti olla paljon alempana tuloksissa kuin mitä käyttöliittymä antoi ymmärtää. Tämä voitaisiin toteuttaa niin, että käyttäjälle kerrottaisiin, että tuloksia löytyi x-kappaletta, mutta hänellä on vain oikeus y-kappaleeseen niitä ja siksi hakutulokset voivat tuntua merkityksettömiltä.

Myöskään pikahaku mahdollisuutta ei ole toteutettuna, joten sen toteuttaminen on seuraava tärkeä kehityskohde. Pikahaku on huomion arvoinen kehityskohde, koska se on yleinen käytetyissä palveluissa ja käyttäjät ovat tottuneita siihen (Venkataraman ym. 2016). Pikahaku on tärkeä ominaisuus siksi, että käyttäjä pääsee käsiksi hakutuloksiin nopeammin. Toisaalta tämä lisää oikeuksien laskentaa, koska niitä jouduttaisiin laskemaan useammin. Seurauksena saattaisi olla suurempi viive. Samalla indeksin rakennetta voitaisiin joutua muuttamaan, jos haluttaisiin ottaa käyttöön *Completion suggester*-tyyppinen toteutusratkaisu. Tämä ei tosin vaadi uudelleen indeksointia vaan indeksiin pystytään lisäämään tarvittavat sarakkeet ilman, että indeksille tarvitsee tehdä muuta kuin sulkea se hetkellisesti.

Mittaustuloksissa (Luku 5.1.1;Kuvio 6) voi olla epävarmuutta välillä 70 000-200 000, koska mittausta ei voitu tutkimuksen aikana toistaa. Mittauksen toistamisen esti se, että testi-

palvelin, jolla aineisto oli ja indeksiä testattiin ei ollut enää tutkimuksen käytettävissä siinä vaiheessa kun mittausten toistaminen olisi ollut ajankohtaista. Kaikilla pienemmillä arvoilla kuitenkin mittausdataa oli runsaasti ja niitä voidaan siksi pitää luotettavina.

Analysoijan ja eri suodattimien testauksessa (Luku 4.7) esitetty esimerkki ei ottanut kantaa, miten esimerkiksi aasialaiset kirjaimet toimivat suodattimissa ja miten analysoija osaa käsitellä niitä. Ligatuurit eivät kuitenkaan vastaa esimerkiksi kiinalaisia merkkejä, joten lopullista tarkkuutta tässä työssä luodulle analysoijalle ei voida määritellä, koska se olisi vaatinut useampia testitapauksia ja erikoistapausten käsittelyä. Tällöin olisi saatu kattavampi kuva siitä, toimiiko analysoija oikeasti niin, miten se oli tarkoitettu toimivaksi.

Analysoija myös aiheuttaa ongelmia Suomessa, koska se muuntaa ä=>a ja ö=>o, jolloin hakutermi *yö* muutetaan muuotoon *yo*, mikä voi esiintyä esimerkiksi dokumentissa mikä sisältää termin *yo-tutkinto*. Tämä taas on selkeästi väärä osuma, joten edellä mainittujen kirjaimien(ä,ö) poistaminen *icu_folding* säännöistä voi olla tarpeen.

Tuloksien rajaaminen myös käyttäjän käyttöoikeuden mukaan voi osoittautua huonoksi toteutukseksi. Sillä voidaan estää sellaisen sivun löytyminen mihin käyttäjä pystyisi liittymään esimerkiksi liittymisavaimella, mutta käyttäjän sen hetkinen käyttöoikeus kyseiseen sivustoon ei ole riittävä. Tämä taas laskee käyttäjän käyttökokemusta, joten kyseiseen ongelmaan pitäisi löytää ratkaisu. Toinen ongelma tulosten rajaamisessa liittyy näytettävien osumien määrään. Joissain tapauksissa voi käydä niin, että kaikki 100 Elasticsearchin palauttamaa tulosta ovat sellaisia mihin käyttäjällä ei ole oikeuksia, jolloin haku palauttaa käyttäjän näkökulmasta pelkästään tyhjää.

Koko tietokannan uudelleen indeksoiminen voi myös olla liian raskasta. Uudelleen indeksoitaessa tietokantarivejä Elasticsearchiin hakutulosten osuvuus voi laskea erittäin paljon, jos haut kohdistetaan uuteen indeksiin, jossa ei ole vielä valmiina kaikkia dokumentteja. Tämä voi aiheuttaa ongelmia eritoten, jos Elasticsearch päivittyy seuraavaan suureen versioon esimerkiksi siirryttäessä versiosta 6.7 versioon 7.0. Tällöin joudutaan suorittamaan dokumenttien uudelleen indeksointi, koska hakumoottori ja indeksin hyväksytyt rakenteet muuttuvat merkittävästi.

6 Yhteenveto

Tässä tutkimuksessa toteutettiin onnistuneesti haku, joka toimii suurella sisällöllä tehokkaasti. Samalla käyttöoikeuksien laskenta on huomioitu tuloksissa. Tutkielmassa myös käsiteltiin vaihtoehtoisia tapoja toteuttaa tekstihaku, vaikka loppujen lopuksi tekstihaku päädyttiin toteuttamaan vain yhdellä tavalla. Teoriaosuudessa käsiteltiin eri toteutustapoja ja hakumoottoreiden ominaisuuksia sekä miten nämä ominaisuudet toteutetaan hakumoottoreissa.

Viimeisteltyä PHP-koodia tutkimuksen aikana syntyi pääluokkaan 521 riviä. Tähän ei ole laskettu mukaan aliluokissa luotua koodia, jotka vaihtelevat välillä 10-20 riviä luokan mukaan.
Lisäksi eri bash-skriptit, mitä käytetään indeksin luomiseen ja Elasticsearchin asetusten muokkaamiseen ovat yhteensä 283 riviä pitkät. Uudelleen indeksoinnin tekevä skripti ei ole mukana laskuissa, koska sille ei löydetty vielä järkevää toteutustapaa. PHP-osiosta saatiin lyhyt, koska työkaluja datan saamiseen tietokannasta ei tarvinnut kirjoittaa, vaan Peda.net-sovelluskehys sisälsi ne. Lisäksi syntyi 60 riviä tyylitiedostoon (CSS).

Tuloksina saatiin, että Elasticsearch osoittautui nopeaksi ja toimivaksi ratkaisuksi toteuttaa haku ja hyvinä puolina siinä on avoin lähdekoodi sekä aktiivinen kehitys. Datamäärien kasvaessa, myös muita hakumoottoreita, jotka eivät pohjaudu Apache Luceneen olisi tarpeen tutkia, koska muissa toteutustavoissa voi olla tehokkaampia ratkaisuja.

Heikkoutena tutkimuksessa voidaan pitää sitä, että haku toteutettiin käyttäen vain yhtä työkalua ja kattavammat tulokset olisi saanut toteuttamalla haun usealla eri tavalla. Tutkimuksessa olisi myös voinut etsiä sellaisia toteutusratkaisuja, missä käyttöoikeudet olisivat olleet sisäänrakennettuina hakumoottorissa, ja keskittänyt tutkimuksen niihin.

Jatkotutkimuksena voitaisiin tehdä tutkimus, jossa kartoitetaan erilaisia avoimeen lähdekoodiin perustuvia hakumoottoreita, missä käyttöoikeudet sisältyisivät hakumoottoriin. Tällöin käyttöoikeuksia ei tarvitse laskea jokaiselle löydetylle dokumentille erikseen vaan tulokset olisivat aina sellaisia, joihin käyttäjällä on varmasti oikeus päästä käsiksi.

Lähteet

"Apache Lucene". 2019. Viitattu 11. helmikuuta 2019. https://lucene.apache.org/.

"Apache Solr". 2019. Viitattu 13. helmikuuta 2019. https://lucene.apache.org/solr/.

Babich, Nick. 2017. "Best Practices for Search". Viitattu 4. huhtikuuta 2019. https://www.uxbooth.com/articles/best-practices-for-search/.

Berman, Daniel. 2018. "The Removal of Mapping Types in Elasticsearch 6: The Aftermath". Viitattu 4. maaliskuuta 2019. https://logz.io/blog/removal-elasticsearch-mapping-types/.

Cui, W., M. Xu, H. Sun ja H. Shao. 2011. "Research on application of Lucene in medical image retrieval system". Teoksessa *Proceedings of 2011 International Conference on Computer Science and Network Technology,* 2:661–664. doi:10.1109/ICCSNT.2011.6182053.

"DB-Engines Ranking of Search Engines". 2019. Viitattu 7. huhtikuuta 2019. https://db-engines.com/en/ranking/search+engine.

"Elasticsearch Documentation". 2019. Viitattu 8. helmikuuta 2019. https://www.elastic.co/guide/en/elasticsearch/reference/6.6/docs.html.

"Google custom search". 2019. Viitattu 26. maaliskuuta 2019. https://developers.google.com/custom-search/.

Gormley, Clinton, ja Zachary Tong. 2015. *Elasticsearch: The definitive guide: A distributed real-time search and analytics engine.* "O'Reilly Media, Inc.". ISBN: 9781449358549.

Hevner, Alan R., Salvatore T. March, Jinsoo Park ja Sudha Ram. 2004. "DESIGN SCIENCE IN INFORMATION SYSTEMS RESEARCH 1". *MIS Quarterly* 28, numero 1 (maaliskuu): 75–105. doi:10.2307/25148625.

Horohoe, Chad. 2014. “Wikimedia moving to Elasticsearch”. Viitattu 7. huhtikuuta 2019. https://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.

Kuc, Rafal, ja Marek Rogozinski. 2014. *Elasticsearch server.* Packt Publishing Ltd. ISBN: 9781783980529.

Lewandowski, Dirk. 2015. “Evaluating the retrieval effectiveness of web search engines using a representative query sample”. *Journal of the Association for Information Science and Technology* 66 (9): 1763–1775. https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.23304.

Manning, Christopher D., Prabhakar Raghavan ja Hinrich Schütze. 2008. *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press. ISBN: 0521865719, 9780521865715.

“MySQL Documentation”. 2019. Viitattu 8. helmikuuta 2019. https://dev.mysql.com/doc/refman/8.0/en/.

Navarro, Gonzalo. 2001. “A guided tour to approximate string matching”. *ACM computing surveys (CSUR)* 33 (1): 31–88. doi:10.1145/375360.375365.

Oliver, Andrew C. 2017. “Why you should use Apache Solr”. *InfoWorld.com* (heinäkuu). https://www.infoworld.com/article/3209685/why-you-should-use-apache-solr.html.

“PostgreSQL Documentation”. 2019. Viitattu 6. helmikuuta 2019. https://www.postgresql.org/docs/11/index.html.

“Query timing: took value and what I’m measuring”. 2012. Viitattu 4. huhtikuuta 2019. https://discuss.elastic.co/t/query-timing-took-value-and-what-im-measuring/9870/2.

Rogers, Ian. 2019. “The Google PageRank Algorithm and How It Works”. Viitattu 4. huhtikuuta 2019. https://www.cs.princeton.edu/~chazelle/courses/BIB/pagerank.htm.

Schwartz, Barry. 2016. “Google has confirmed it is removing Toolbar PageRank”. Viitattu 4. huhtikuuta 2019. https://searchengineland.com/google-has-confirmed-they-are-removing-toolbar-pagerank-244230.

Shahi, Dikshant. 2016. *Apache Solr.* Springer. doi:10.1007/978-1-4842-1070-3.

Smith, Lisa. 2018. “How scoring works in Elasticsearch”. Viitattu 4. maaliskuuta 2019. https://www.compose.com/articles/how-scoring-works-in-elasticsearch/.

Thacker, U., M. Pandey ja S. S. Rautaray. 2016. “Performance of elasticsearch in cloud environment with nGram and non-nGram indexing”. Teoksessa *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT),* 3624–3628. doi:10.1109/ICEEOT.2016.7755381.

Tsech, Nadya. 2018. “Search interface: 20 things to consider”. Viitattu 4. huhtikuuta 2019. https://uxplanet.org/search-interface-20-things-to-consider-4b1466e98881.

Venkataraman, Ganesh, Abhimanyu Lad, Viet Ha-Thuc ja Dhruv Arya. 2016. “Instant Search: A Hands-on Tutorial”. Teoksessa *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval,* 1211–1214. SIGIR ’16. Pisa, Italy: ACM. ISBN: 978-1-4503-4069-4. doi:10.1145/2911451.2914806.

Whistler, Ken. 2019. “Unicode Normalization Forms”. Viitattu 4. huhtikuuta 2019. https://unicode.org/reports/tr15/.

Vijay Karambelkar, Hrishikesh. 2014. *Scaling Apache Solr.* Packt Publishing. ISBN: 9781783981748.

Liitteet

Liite 1

Indeksin rakentava bash skripti

```
#!/bin/bash
set -e

/usr/share/elasticsearch/bin/elasticsearch-plugin remove analysis-icu || true
/usr/share/elasticsearch/bin/elasticsearch-plugin install analysis-icu
curl -X PUT "localhost:9200/documents?pretty=true" -H 'Content-Type: application/j
{
  "settings": {
        "analysis" : {
            "analyzer" : {
                "nfkd_analyzer" : {
                    "tokenizer" : "icu_tokenizer",
                    "filter" : ["nfkd_normalized", "lowercase", "icu_folding"]
                }
            },
            "filter" : {
                "nfkd_normalized" : {
                    "type" : "icu_normalizer",
                    "name" : "nfkc",
                    "mode" : "decompose"
                }
            }
        }
  },
  "mappings": {
    "_doc": {
      "properties" : {
          "data" : {
            "type" : "text",
            "analyzer" : "nfkd_analyzer",
            "fields" : {
```

```
    "keyword" : {
      "type" : "keyword",
      "ignore_above" : 256
    }
  }
},
"date" : {
  "type" : "date",
  "format": "epoch_millis"
},
"description" : {
  "type" : "text",
  "analyzer" : "nfkd_analyzer",
  "fields" : {
    "keyword" : {
      "type" : "keyword",
      "ignore_above" : 256
    }
  }
},
"documenttype" : {
  "type" : "long"
},
"parentid" : {
  "type" : "text",
  "fields" : {
    "keyword" : {
      "type" : "keyword",
      "ignore_above" : 256
    }
  }
},
"title" : {
  "type" : "text",
  "analyzer" : "nfkd_analyzer",
  "fields" : {
    "keyword" : {
```

```
                "type" : "keyword",
                "ignore_above" : 256
              }
            }
          }
        }
    }
  }
}
'
echo
echo OK
```

Liite 2

Koko indeksistä hakeva Elasticsearch kysely

```
{
    "query":
    {
        "query_string":
        {
            "default_field": "*",
            "analyzer": "nfkd_analyzer",
            "analyze_wildcard": "true",
            "query": "kissa",
            "default_operator": "and"
        }
    },
    "sort": [
    {
        "_score":
        {
            "order": "asc"
        },
        "date":
```

```
        {
            "order": "asc"
        }
    }],
    "from": 0,
    "size": 100
}
```

Liite 3

Tietystä polusta hakeva Elasticsearch kysely

```
{
    "query":
    {
        "bool":
        {
            "must": [
            {
                "match":
                {
                    "parentid": "123"
                }
            },
            {
                "query_string":
                {
                    "default_field": "*",
                    "analyzer": "nfkd_analyzer",
                    "analyze_wildcard": "true",
                    "query": "kissa",
                    "default_operator": "and"
                }
            }]
        }
    },
    "sort": [
```

```
    {
        "_score":
        {
            "order": "asc"
        },
        "date":
        {
            "order": "asc"
        }
    }],
    "from": 0,
    "size": 100
}
```

Liite 4

Tietokantarivien indeksoinnin hoitavat metodit

```
    /**
     * Systemjob for adding services to Elasticsearch index.
     * Crawls database and adds services to index if they haven't been added yet.
     * Column value is 0 if service has never been added to index before.
     *
     * @throws Exception In case of internal error
     */
    private function indexOldEntries()
    {
      $limit = 10000;
      $tr = $this->getSession()->getDatabaseTransaction();
      $tr->verifyReadWriteAccess();
      $q = "SELECT id FROM xxx WHERE indexed = '0' AND deleted = '0'
            AND id !='".NULL_UUID."'
            FOR UPDATE
            SKIP LOCKED
            LIMIT $limit
            ";
      $results = $tr->execute($q);
```

```
  $ids = array();
  while (list($id) = $results->row()) {
      $ids[$id] = $id;
  }
  if(!empty($ids)) {
      $services = \XhtmlRestService::start($this->getSession(), $ids,true);
      foreach ($services as $id => $service) {
          if ($service instanceof \XhtmlRestService) {
              try{
                 $service->collectIndexData();
              }catch (\Exception $e){
                $tr->execute("update xxx set x='".self::INDEXING_FAILED."'
                             where id=".$tr->quote($id));
                $this->getSession()->logException($e, "Indexing failed");
              }
          }
          else{
            $tr->execute("update xxx set
                         x='".self::DO_NOT_INDEX."'
                         where id=".$tr->quote($id));
            $this->getSession()->logWarning("Unexpected instance.
                                             Do not index
                                             $id=".debug_tostring($service));
            }
        }
    }
}

/**
 * collectIndexData calls this function.
 * Should not be called alone.
 *
 * @param $dataArray array given by collectIndexData()
 *
 * @throws Exception In case of internal error
 */
private function indexDocument($dataArray)
```

```
{
    assert('is_array($dataArray) and count($dataArray) > 0');

    try{
        $data = $dataArray;
        $id =$data["id"];
        # remove id after assigning it since we're not needing it in index dat
        unset($data["id"]);
        \Elasticsearch::indexData ("documents", "_doc", $id, $data );
        $this->setLastIndex($this->getSession()->getCommonDatestamp());
        $this->commit();
    } catch (\Exception $e){
        $this->getSession()->logException($e,"Indexing failed");
        $this->setLastIndex(self::INDEXING_FAILED);
        $this->commit();
        #Create job to update search index
        \JobQueueData::queueJob(time(), $this, "updateindexdata",
                                array("esdata"=>$dataArray));
    }
}
```