TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Boriss Zahharov 185988IAIB

ELASTICSEARCHI JA POSTGRESQLI SOBIVUSE VÕRDLEMINE TÄISTEKSTOTSINGU REALISEERIMISEKS KONKREETSE SÜSTEEMI NÄITEL

Bakalaureusetöö

Juhendaja: Erki Eessaar
PhD

Tallinn 2024

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Boriss Zahharov

10.01.2024

Annotatsioon

Täistekstotsing on tänapäevase inforohkuse kontekstis tähtis teema. See on kasutusel suures hulgas infosüsteemides ning ajapikku on selle tarbeks loodud mitmeid lahendusi ja tööriistu. Täistekstotsignu funktsionaalsust pakuvad paljud otsingumootorid ja andmebaasisüsteemid. Ühed populaarsemad neist on Elasticsearch ja PostgreSQL. Mainitud tööriistad on oma olemuselt erinevad - Elasticsearch on täistekstotsingule spetsialiseeritud otsingumootor, PostgreSQL on SQL andmebaasikeelt kasutada võimaldav üldotstarbeline andmebaasisüsteem, mis pakub ka täistekstotsingu realiseerimise võimalusi. Käesolevas töös võrreldakse nende tööriistade sobivust täistekstotsingu realiseerimiseks konkreetse süsteemi näitel ja analüüsitakse võimalusi selle funktsionaalsuse loomiseks mõlemas süsteemis. Süsteemi jaoks parima täistekstotsingu lahenduse leidmiseks kasutatakse analüütiliste hierarhiate meetodit. Kuigi selles süsteemis oli juba enne käesoleva töö algust realiseeritud täistekstotsing Elasticsearchi kasutades, tuli uuringust välja, et parem lahendus olnuks kasutada PostgreSQLi, kus on loodud *tsvector* tüüpi veergudega tabelid.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 57 leheküljel, 9 peatükki, 51 joonist, 22 tabelit.

Abstract

Comparing Suitability of Elasticsearch and PostgreSQL for Implementing Full-text Search in a Specific System

Full-text search is an important topic in the context of the abundance of information. It is used in a large number of systems. Over time many solutions and tools have been created for it. Full-text search functionality is provided by many search engines and database management systems. Elasticsearch and PostgreSQL are well-known examples of such tools. However, the tools are different in nature. On the one hand, Elasticsearch is a search engine that is specialised for full-text search. On the other hand PostgreSQL is a general-purpose database management system that among other things offers possibilities to implement full-text search. This thesis compares the suitability of those tools for the implementation of full-text search in case of a specific system. The system is used to provide access to statistics and a part of it is searching data about data elements, i.e., metadata. The author participated in its implementation. The system gets a JSON document that has a complicated structure as an input. The system was implemented by using Elasticsearch. However, the author decided to investigate as to whether it was the best choice or perhaps PostgreSQL that is used in the other parts of the system could be used instead. The thesis analysed different ways to implement full-text search functionality in both these tools. In the thesis different experiments were made based on the Elasticsearch design and various different designs in PostgreSQL. The experiments were made based on Elasticsearch 8.11 and PostgreSQL 16. The code that was used in the experiment is available in: `https://github.com/borzah/full-text-search-comparison`. In the end the best solution (design) was selected by using analytic hierarchy process. The results showed that in case of the particular system the best solution would be the use of PostgreSQL and tables with *tsvector* data type. In case of existing similar comparisons of PostgreSQL and Elasticsearch the amount of data is much bigger and Elasticsearch shows better results than PostgreSQL in terms of query performance. The experiments also showed that in case of the particular system, saving JSON documents to a column with JSONB data type in PostgreSQL did not provide good query performance.

The thesis is written in Estonian and is 57 pages long, including 9 chapters, 51 figures and 22 tables.

Lühendite ja mõistete sõnastik

AHM	Analüütiliste hierarhiate meetod
AHP	Analytic Hierarchy Process, analüütiliste hierarhiate meetod
API	Application Programming Interface
Base64	Meetod binaarandmete e kahendformaadis andmete (nt failid) kodeerimiseks ASCII tekstiks
DSL	Domain Specific Language
GIN	Generalized Inverted Index
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSONB	Süsteemi-defineeritud andmetüüp PostgreSQLis, et saaks salvestada JSON andmeid kahendformaadis.
POST meetod	REST meetod uue ressursi loomiseks
PUT meetod	REST meetod uue ressursi lisamiseks või olemasoleva uuendamiseks
REST	Representational State Transfer
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol/Internet Protocol
Tsvector	Sorteeritud loend erinevatest lekseemidest - sõnadest, mis on normaliseeritud, et ühendada sama sõna erinevad variandid
URL	Uniform Resource Locator
XML	Extensible Markup Language
UUID	Universally Unique Identifier

Sisukord

1 Sissejuhatus **11**
1.1 Probleem ja töö kirjeldus 11
1.2 Töö edasine struktuur 11

2 Süsteemi hetkeolukord **12**

3 Teoreetiline taust **15**
3.1 Täisteksekstotsing 15
3.1.1 Teksti indekseerimine 15
3.1.2 Võtmesõnade kaalumine 15
3.1.3 Otsingualgoritmid 15
3.1.4 Hägus otsing 16
3.2 Täistekstotsingut võimaldavad süsteemid 16
3.2.1 Apache Lucene 17
3.2.2 Solr 17
3.2.3 Elasticsearch 18
3.2.4 PostgreSQL 18
3.3 Analüütiliste hierarhiate meetod 19

4 Realisatsioon Elasticsearchis **21**
4.1 Andmete hoiustamine 21
4.2 Indeksi loomine 22
4.3 Andmete otsing 23

5 Realisatsioon PostgreSQLis **26**
5.1 Testandmete genereerimine 27
5.2 Baastabel iga olemitüübi kohta 27
5.3 Materialiseeritud vaade 31
5.4 Eeltöödeldud andmeid sisaldavad baastabelid 34
5.5 JSONB tüüpi veerg 36

6 Eksperimentide kirjeldus **43**
6.1 Testandmete hulk 43
6.2 Otsingute kiirus 44
6.3 Andmete salvestamise kiirus 45
6.4 Koormustestimine 45

6.5 Edasiarendamise lihtsus 46

7 Eksperimentide tulemused 47
7.1 Otsingute kiirus 47
7.2 Salvestamise kiirus 50
7.3 Koormustestimine 50
7.4 Edasiarendamise lihtsus 51
7.4.1 Hägus otsing Elasticsearchis ja PostgreSQLis 52
7.5 Andmemahud 57
7.6 Võrdlus olemasolevate tulemustega 57

8 Parima alternatiivi valimine 59
8.1 Kriteeriumite valik 59
8.2 Kriteeriumite suhteline olulisus 60
8.3 Alternatiivide omavaheline võrdlus 61
8.3.1 Otsingute kiirus 61
8.3.2 Realiseerimise lihtsus 61
8.3.3 Edasiarenduse lihtsus 61
8.3.4 Koormustaluvus 62
8.3.5 Andmete salvestamise kiirus 62
8.4 Tulemuste analüüs 62
8.5 Tundlikkuse analüüs 64
8.6 Soovitused olemasoleva lahenduse parandamiseks 66

9 Kokkuvõte 67

Kasutatud kirjandus 68

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 72

Lisa 2 – Metaandmete JSON struktuur 73

Lisa 3 – Baastabel iga olemitüübi kohta - tabelite struktuur 74

Lisa 4 - Elasticsearch ja PostgreSQL otsingule kulunud aja geomeetriline keskmine (millisekundites) 75

Lisa 5 - Elasticsearch ja PostgreSQL otsingule kulunud aja hälve (millisekundites) 76

Lisa 6 - Elasticsearch ja PostgreSQL otsingule kulunud aja mediaan (millisekundites) 77

Lisa 7 - Elasticsearch ja PostgreSQL salvestamisele kulunud aja geomeetriline keskmine (millisekundites) **78**

Lisa 8 - Elasticsearch ja PostgreSQL salvestamisele kulunud aja hälve (millisekundites) **79**

Lisa 9 - Elasticsearch ja PostgreSQL salvestamisele kulunud aja mediaan (millisekundites) **80**

Jooniste loetelu

1 Metaandmete struktuur. 12
2 Süsteemi ülevaade. 13

3 Elasticsearch klaster. 21
4 Indeksi loomine Elasticsearchis. 22
5 Dokumendi objekti loomine Spring rakenduses. 23
6 Elasticsearch repositooriumi kasutamine Spring rakenduses. 23
7 *Elasticsearchoperations* abil päringu tegemine Spring rakenduses. 24
8 Elasticsearchi otsingupäring. 24
9 Otsingupäringu esitus JSON formaadis. 25

10 Tekstivektori kasutamine PostgreSQLis. 26
11 Tekstivektori kasutamisel põhinev päring PostgreSQLis. 28
12 Tekstivektori veeru loomine PostgreSQLis. 28
13 Päring tabelite veergudes hoitud tesktivektorite puhul. 29
14 Tekstivektori vastete leidmine. 30
15 Testkivektori päringus olulisuse arvutamine. 30
16 Testkivektori päringus otsingusõne esiletoomine. 30
17 Testkivektori päringus muutujate andmete agregeerimine. 31
18 Materialiseeritud vaate ja seotud indeksite loomine. 32
19 Päring materialiseeritud vaatest. 33
20 Otsingu tarbeks kohandatud baastabelite loomine. 34
21 Andmete sisestamine otsingu tarbeks kohandatud baastabelitesse. 35
22 Päring otsingu tarbeks kohandatud baastabelitest. 35
23 Otsingu tarbeks kohandatud baastabelitest otsingu vastete arvu küsimine. . 36
24 JSONB veeruga baastabel andmete salvestamiseks uuringute kaupa. . . . 37
25 Päring uuringute kaupa andmetega JSONB tüüpi veeruga tabelist. 37
26 JSONB veeruga baastabel sissetulnud JSON dokumendi töötlemata kujul salvestamiseks. 38
27 Päring töötlemata kujul JSON dokumendiga tabelist. 38
28 Trigram indeksite loomine töötlemata kujul JSON dokumendiga tabelile. . 39
29 Töötlemata kujul JSON dokumendiga tabelist tehtud päringu täitmisplaan. 40
30 Lihtsustatud päring JSONB tüüpi andmete põhjal. 40
31 Trigram indeksi loomine JSONB veerus json väljale. 40
32 Lihtsa JSONB tüüpi väärtuse põhjal tehtud päringu täitmisplaan. 41
33 Kontseptide JSONB tüüpi väärtustena salvestamiseks mõeldud tabel. . . . 41

34 Trigram indeks JSONB tüüpi veerus olevale json väljale. 41
35 Kontsepti leidmiseks mõeldud päring. 42
36 Kontsepti päringu täitmisplaan ilma trigram indeksita. 42
37 Kontsepti päringu täitmisplaan koos trigram indeksiga. 42

38 Elasticsearh realisatsiooni päringu täitmiseks kuluva aja jaotus. 48
39 PostgreSQL realisatsiooni päringu täitmiseks kuluva aja jaotus. 48
40 Sõneosa järgi otsing Elasticsearchis. 52
41 Hägus otsing Elasticsearchis. 53
42 Tabelite loomine PostgreSQLis hägusa otsingu jaoks. 54
43 Hägusa otsingu võimalus PostgreSQLis. 55
44 Vigadega kirjutatud otsingusõnele kõige lähedasema vaste leidmine tekstivektori päringus kasutamiseks. 56
45 Elasticsearchi päring, mis tagastab statistikat salvestatud indeksite kohta. . 57
46 PostgreSQL päring, mis tagastab kui palju salvestusruumi kasutab konkreetne andmebaasi skeem. 57

47 Otsingu kiiruse kriteeriumi tundlikkuse analüüs. 64
48 Realiseerimise lihtsuse kriteeriumi tundlikkuse analüüs. 64
49 Edasiarenduse lihtsuse kriteeriumi tundlikkuse analüüs. 65
50 Koormustaluvuse kriteeriumi tundlikkuse analüüs. 65
51 Andmete salvestamise kiiruse kriteeriumi tundlikkuse analüüs. 66

Tabelite loetelu

1 Saaty skaala. 20

2 Esialgseks testimiseks andmete arv. 27

3 Arvuti riistvara näitajad. 43
4 Andmete arv testimiseks. 43
5 Testimiseks valitud test-andmetes sisalduvad sõnad. 44

6 Otsingu keskmised kiirused. 47
7 Päringu täitmise kiiruste geomeetrilised keskmised. 48
8 Salvestamise keskmised kiirused. 50
9 Elasticsearch realisatsiooni koormustestimise tulemused. 50
10 PostgreSQL + tsvector realisatsiooni koormustestimise tulemused. 51
11 PostgreSQL + JSONB realisatsiooni koormustestimise tulemused. 51
12 Koormustestimise koondtulemused. 51
13 Salvestusruumi kasutus andmete hoiustamiseks. 57

14 Kriteeriumite suhteline olulisus. 60
15 Kriteeriumite kaalud. 60
16 Alternatiivide võrdlus otsingu kiiruse suhtes. 61
17 Alternatiivide võrdlus realiseerimise lihtsuse suhtes. 61
18 Alternatiivide võrdlus edasiarenduse lihtsuse suhtes. 62
19 Alternatiivide võrdlus koormustaluvuse suhtes. 62
20 Alternatiivide võrdlus andmete salvestamise kiiruse suhtes. 62
21 Alternatiivide kaalud. 63
22 Alternatiivide kaalud kokku. 63

1. Sissejuhatus

Täistekstotsing on oluline infohalduse tegevus, mis võimaldab kasutajatel leida vastuseid oma küsimustele suure hulga tekstilise sisu seast. Antud teema on muutunud eriti oluliseks digitaalsel infoajastul, kus inimeste poolt kogutud ja talletatud andmete hulk pidevalt kasvab. Täistekstotsingu eesmärk on tagada kiire ja tõhus juurdepääs tekstilisele informatsioonile.

1.1 Probleem ja töö kirjeldus

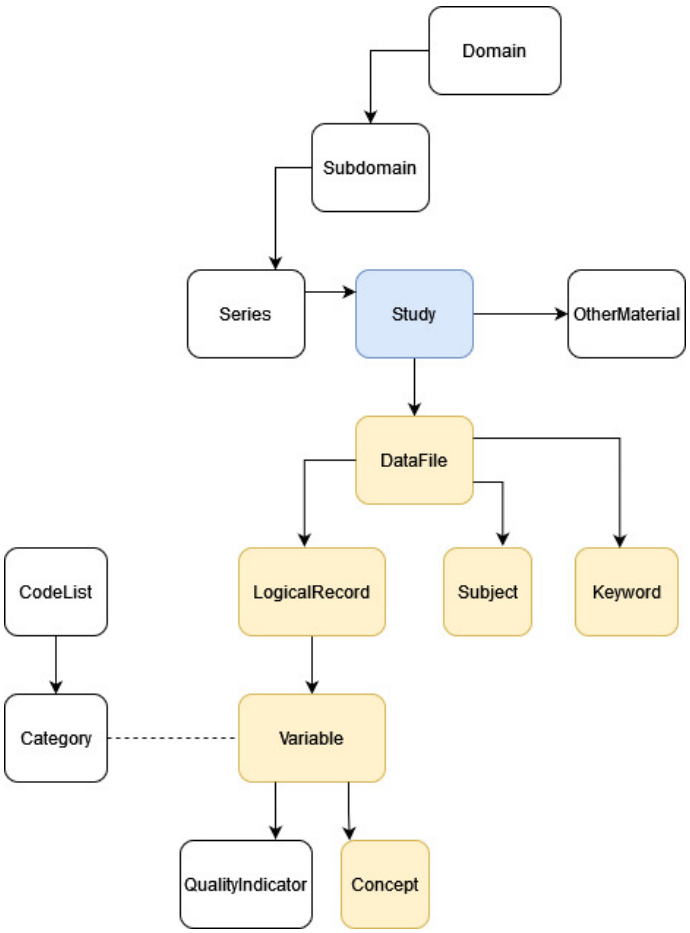
Töö autor töötas meeskonnas, mis loob Statistikaametile tarkvarasüsteemi, mille eesmärk on hõlbustada huvitatud isikute ligipääsu statistikale. Selle süsteemi üheks funktsionaalsuseks on täistekstotsing. Süsteemi arendamise algfaasis kerkis küsimus, milline täistekstotsingu töörist tuleks kasutamiseks valida. Alternatiividеks, mille vahel valida, kujunesid Elasticsearch [1] ja PostgreSQL[2]. 2023. aasta detsembri seisuga on need andmebaasisüsteemide populaarsuse indeksis vastavalt seitsmendal ja neljandal kohal [3]. Antud tehnoloogiaid võrdlevad arutelud on tekkinud ka mujal [4] [5], sest vaatamata Elasticsearchi täistekstotsingule spetsialiseerumisele pakub ka PostgreSQL võimalusi taolise probleemi lahendamiseks [6]. Sellised võrdlused on avalikus veebis olemas ja kättesaadavad [7], kuid nendes võrreldakse süsteeme lihtsate ärinõuete ja lihtsa struktuuriga andmete alusel. Kuigi töö aluseks olevas tarkvaraprojektis valiti ja realiseeriti Elasticsearchi kasutav variant, avaneb antud olukorras hea võimalus viia läbi eelmainitud tööriistade võrdlus keerukama ärijuhtumi alusel.

1.2 Töö edasine struktuur

Peatükis 2 antakse ülevaade süsteemi hetkeolukorrast. Peatükis 3 esitatakse töö teoreetiline taust. Peatükis 4 kirjutatakse, kuidas on realiseeritud Elasticsearchil põhinev lahendus. Peatükis 5 kirjutatakse erinevatest võimalustest, kuidas PostgreSQLis saab täistekstotsingut realiseerida ning tehakse nendega esialgseid katsetusi. Peatükis 6 selgitatakse eksperimente, mida täistekstotsingu võimaluste võrdlemiseks läbi viidi ning peatükis 7 tuuakse välja nende tulemused. Peatükis 8 esitatakse parima alternatiivi valik analüütiliste hierarhiate meetodi abil ning peatükis 9 võetakse töö kokku.

2. Süsteemi hetkeolukord

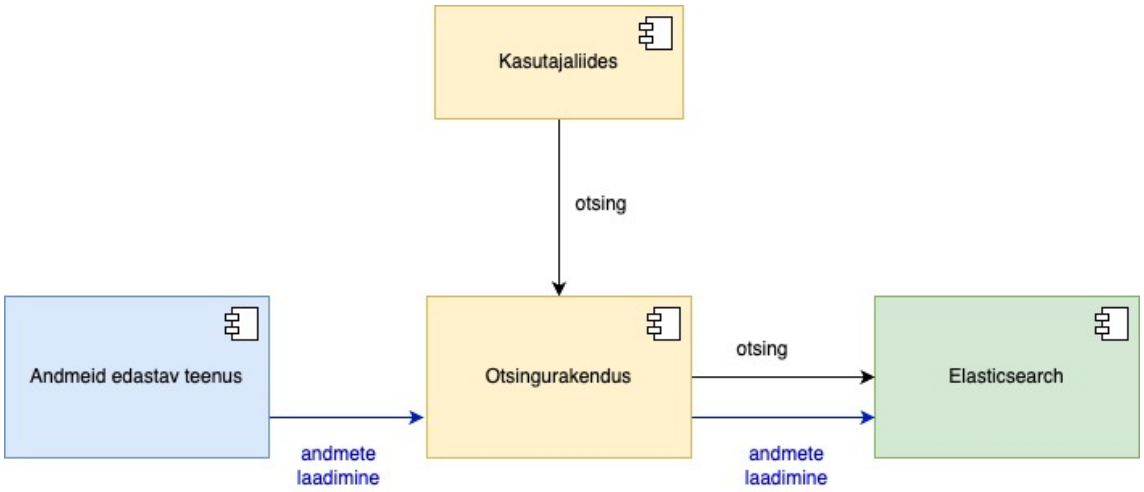
Elasticsearchi ja PostgreSQLi võrdluse aluseks on tarkvara, mille eesmärk on automatiseerida ja lihtsustada huvitatud isikute ligipääsu statistilistele andmetele. Käesoleva töö kirjutamise ajal (2023. aasta sügisel) on käimas selle arenduse esimene etapp, mis algas 2023. aasta sügisel ning mis lõpeb 2024. aasta märtsis. Üks osa süsteemist on üksikandmeid kirjeldavate metaandmete (andmed andmete kohta) otsing. Otsingutulemustest valitud andmed saab lisada tellimusele ning teatud protsesside järel antakse tellijale ligipääs üksikandmetele. Metaandmeid esitav JSON dokument on joonisel 1 esitatud struktuuriga. Uuring on märgitud sinisega ja selle allelemendid, kust peab otinguvasteid otsima, kollasega. Täpsem JSON struktuur on näidatud lisas 2.



Joonis 1. Metaandmete struktuur.

Üheks funktsionaalsuseks on uuringute (*study*) otsing kindlate andmeväljade ning selle alamelementide e alamolemite kindlate andmeväljade järgi. Iga tulemuses oleva uuringu tulemuse kohta tuleb väljastada kuni kümme otsinguvastet saanud muutujat (*variable*). Muutujates tuleb tõsta esile tekstiosad, mis vastavad otsingusõnele. REST rakendusliides (API) peab tagastama tulemusi lehekülgede kaupa (*pagination*) ja otsingutulemused peavad olema järjestatud vastavalt olulisusele. Lisaks täistekstotsingule on vaja erinevate kasutajaliidese detailivaadete tarbeks lugeda üksikute olemite andmeid identifikaatorite järgi.

Süsteemi ülevaade on näidatud joonisel 2.



Joonis 2. Süsteemi ülevaade.

Ülesande lahendamiseks tuleb otsida eelnevalt määratud väljadest teksti täisvasteid. See tähendab, et hägus otsing (*fuzzy search*) ei ole antud juhul vajalik. Otsitavates andmetes esitatud olemihulkade teadaolevad ligikaudsed suurusjärgud on järgmised:

1. *studies* (uuringud) ~100,
2. *dataFiles* (andmefailid) ~200,
3. *logicalRecords* (loogilised kirjed)~5000,
4. *variables* (muutujad) ~100 000.

Andmeid saadab hoidlasse väline teenus üks kord ööpäevas, mille järel kirjutatakse olemasolevad andmed üle. Seega ei pea antud kontekstis arvestama tiheda andmete uuendamisega ning andmete kirjutamise kiirus pole oluline valikukriteerium. Sissetulevad andmed kirjutavad üle hoidlas olevad andmed. Seega pole ka andmemahud väga suured ja andmemaht pole oluline valikukriteerium. Uuringute arv on suhteliselt väike - räägime ligikaudu sajast olemist. Küll aga sisaldub uuringutes hulganisti alamobjekte, kust peab otsingu korral vasteid leidma, mis teeb ülesande keerukamaks.

Projekti algfaasis seisime valiku ees - kas realiseerida täistekstotsing PostgreSQLis, mis on ka antud tarkvara teistest osades kasutusel, või kasutada täistekstotsingu-spetsiifilist lahendust pakkuvat Elasticsearchi. Valituks osutus Elasticsearch ning selle abil realiseeriti metaandmete otsingu funktsionaalsus. Valik langetati selle pärast, et Elasticsearch oli 2023. aasta lõpu seisuga kõige populaarsem täisteksti otsingumootor [8], mis võimaldab kiirelt otsingupäringuid ülesehitada ja ärinõutete muutumise korral neid täiendada ja laiendada. Otsinguga seotud tegevuste vahendamiseks (andmete vastuvõtt, otsingu- ja detailide päringute rakendusliidese pakkumine, teatud päringute turvamine ning logide kirjutamine) on loodud Java rakendus Spring raamistikku kasutades.

Elasticsearchiga suhtlemise jaoks on kasutusel teek Spring Data Elasticseach [9]. Elasticsearchi puhul kasutatakse versiooni 8.11.

3. Teoreetiline taust

Selles peatükis esitatakse lühidalt täistekstotsingu teoreetiline taust ning selgitatakse lühidalt analüütiliste hierarhiate meetodit ehk Saaty meetodit.

3.1 Täisteksekstotsing

Täistekstotsing on informatsiooni otsimise meetod, mida kasutatakse tekstipõhise sisu leidmiseks enamasti suurest tekstihulgast. See tehnika võimaldab kasutajatel tõhusalt leida spetsiifilist informatsiooni, olgu see siis veebilehtedel, dokumentides, artiklites või muudes tekstilistes ressurssides. Täistekstotsingu võimaldamine ja tegemine sisaldab mitmeid tegevusi, alates teksti indekseerimisest kuni päringute hindamiseni.

3.1.1 Teksti indekseerimine

Teksti indekseerimine on protsess, kus teksti analüüsitakse ja struktureeritakse, et otsing oleks kiire ja tõhus. Selle käigus eraldatakse tekstidest võtmesõnad ja nende asukohad salvestatakse indeksisse, mis võimaldab vastavalt otsinguparameetritele kiiret juurdepääsu otsitavale informatsioonile.

3.1.2 Võtmesõnade kaalumine

Iga sõna olulisus tekstis võib olla erinev, sõltuvalt sõna sagedusest, asukohast ja muudest teguritest. Kaalud aitavad määrata, kui olulised on tekstis esinevad sõnad konkreetse otsingu kontekstis.

3.1.3 Otsingualgoritmid

Täistekstotsingu realiseerimiseks on olemas erinevad lähenemised nagu näiteks andmete puu-struktuurina indekseerimine ja läbikäimine [10]. Algoritmid võivad olla lihtsad, kuid tänapäeval kasutatakse sageli keerulisemaid masinõppe algoritme, mis suudavad õppida ja kohanduda kasutajate eelistustega [11]. Täistekstotsingus kasutatavad tingimused võivad sisaldada ka loogilisi operatsioone nagu "ja", "või" ja "mitte", mis aitavad täpsustada otsingutulemusi vastavalt kasutaja konkreetsetele vajadustele. Pärast otsingutulemuste saamist tuleb neid hinnata vastavalt nende olulisusele. See hõlmab sageli tulemuste

järjestamist vastavalt olulisuse indeksile, et esimesena väljastatavad tulemused oleksid kasutajale kõige informatiivsemad.

3.1.4 Hägus otsing

Hägus otsing (*fuzzy search*) on tehnika, mida kasutatakse andmehulgast otsimisel, et leida päringule ligikaudsed vasted. Erinevalt "traditsioonilisest" täpsest täistekstotsingust võetakse hägusa otsingu korral arvesse otsinguparameetrite väärtuste (argumentide) variatsioone ja nendes esinevaid kirjavigu. Seega sobib hägus otsing olukordades, kus kasutaja ei tea täpseid võtmesõnu, mille järgi andmetest tulemusi leida, kuid võib teha seda sõnade fragmentide järgi otsides. Hägusa otsingu ülesande lahendamiseks on olemas erinevaid võimalusi.

1. Üks levinud lähenemine hägusa otsingu realiseerimiseks hõlmab Levenshteini kauguse (*Levenshtein distance*) arvutamist kahe sõne vahel. Levenshteini kaugus on minimaalne ühe tähemärgi haaval tehtud muudatuste (sisestuste, kustutamiste või asenduste) arv, mis on vajalik ühe sõne teisendamiseks teiseks sõneks. Võrreldes Levenshteini kaugust päringu tingimuses kasutatud väärtuse ja võimalike vastete vahel, saab tuvastada ligikaudsed vasted [12].
2. Abiks võivad olla foneetilised algoritmid, mis on loodud sõnade kodeerimiseks nende häälduse põhjal. Soundexi [13] ja Metaphone [14] algoritmide abil saab leida sarnaselt kõlavaid sõnu, isegi kui need on erinevalt kirjutatud. See on eriti kasulik, kui käsitletakse nimesid või sõnu, mida võib hääldada sarnaselt, kuid millel on erinev kirjapilt.
3. Osaliste vastete leidmisel on kasulikud N-grammid [15]. N-gram algoritmid jagavad sõnesid jadadeks, mis koosnevad järjestikulistest n-tähelistest sõnaosadest. Näiteks trigrami puhul on sõna "otsing" esitatav jadana [ots, tsi, sin, ing]. Hägusa otsingu puhul võrreldakse päringus esitatud otsingusõne põhjal koostatud n-gramme sihtandmetes sisalduvate n-grammidega. See lähenemine on vastupidav kirjavigade esinemisele ja suudab tuvastada osalisi vasteid.

3.2 Täistekstotsingut võimaldavad süsteemid

Tänapäeval on olemas suur hulk erinevaid süsteeme, mis võimaldavad täistekstotsingut realiseerida [16]. Antud jaotises tutvustatakse mõningaid nendest.

3.2.1 Apache Lucene

Apache Lucene [17] on avatud lähtekoodiga teabeotsingu teek, mida kasutatakse laialdaselt täisteksti indekseerimiseks ja otsimiseks. See on osa Apache Software Foundationist [18] ja on kirjutatud Java keeles. Lucene pakub funktsionaalsust otsinguvõimaluste lisamiseks rakendustesse.

1. Lucene võimaldab luua dokumentidest indeksi - andmestruktuuri, mis võimaldab kiiret ja tõhusat otsingut. Indeksi koostamisel salvestatakse ja märgistatakse dokumentides sisalduvad sõnad, mis võimaldab kiiresti leida konkreetseid sõnu sisaldavaid dokumente. Indekseerimise tõhustamiseks sisaldab otsingumootor analüsaatorite komplekti, mis suudavad teksti eeltöödelda ja indekseerimiseks sobivaks muuta. Analüsaatorid viivad sõnu baaskujule, eemaldavad stoppsõnu ning salvestavad neid tõstutundetult.
2. Teek pakub paindlikku päringukeelt (*Query DSL*), mis võimaldab indekseeritud andmete otsimiseks koostada keerukaid päringuid. See toetab otsimist nii sõnade täisvastete kui ka sõnaosade järgi. Lucene kasutab otsingutulemuste asjakohasuse alusel järjestamiseks hindamisalgoritmi. Hindamisalgoritm võtab arvesse terminite ja dokumendide sagedust ning väljade pikkuse normaliseerimist.
3. Lucene sisaldab kõrg- ja madala taseme rakendusliideseid ning on oma olemuselt laiendatav, mis toetab kasutamist erinevates rakendustes.

3.2.2 Solr

Apache Solr [19] on avatud lähtekoodiga otsinguplatvorm, mille on välja töötanud Apache Software Foundation. See on üles ehitatud Apache Lucene-le. Solr on loodud skaleeritavana ja kõrge tõrketaluvusega ning on kergesti muude rakendustega integreeritav. See pakub täistekstiotsingut ja filtreerimist, on võimeline töötama arvutite klastril ning pakub andmebaaside integreerimist. 2023. aasta detsembri seisuga oli Solr andmebaasisüsteemide populaarsuse indeksi kohaselt populaarsuselt kolmas täistekstotsingu tööriist [8].

1. Solr võimeline teostama nii täistekstotsingud kui toetab ka filtrite kasutamist - näiteks filtreerimine kategooriate või ajalise kaetuse järgi. Lisaks tavapärasele JSON dokumentide indekseerimisele suudab see vahend indekseerida ka XML dokumente ja binaarvorminguid. Seda kasutatakse sageli PDF ja Word dokumentide indekseerimiseks ja sealt otsingu teostamiseks [20].
2. Pakub REST APIt mis võimaldub teenusega suhelda ning on võimeline importima andmeid erinevatest allikatest (näiteks andmebaasidest) ja erinevas formaatis

(näiteks JSON või XML). Solr sisaldab georuumilise otsingu tuge, võimaldades asukohapõhiste andmete indekseerimist ja otsimist [21].

3.2.3 Elasticsearch

Elasticsearch on avatud lähtekoodiga otsingu- ja analüüsimootor, mis on ehitatud Apache Lucene peale. Elasticsearch pakub skaleeritavat otsingu- ja analüüsilahendust paljude kasutusjuhtude jaoks - alates lihtsatest otsingurakendustest kuni keerukate suuremahuliste süsteemideni, mis tegelevad suurtest andmemassiividest otsinguga ja analüüsiga. Näiteks kasutab seda populaarne logide töötlemise tööriist graylog [22]. Elasticsearch oli 2023. aasta detsembri seisuga kõige populaarsem täistekstotsingu tööriist [8].

1. Elasticsearch on kavandatud toimima hajussüsteemina, mis võimaldab süsteemi horisontaalselt skaleerida, lisades arvutite klastrisse rohkem sõlmi (*node*). Tarkvara jaotab andmed automaatselt sõlmede vahel, pakkudes kõrget kättesaadavust ja tõrketaluvust.
2. Andmed salvestatakse JSON-dokumentidena. Dokument on olem, mida saab indekseerida ja otsida. Dokumendid on organiseeritud indeksiteks, mis on loogilised dokumentide kogumid.
3. Pakub REST API-t mille kaudu rakenduse poole pöörduda ning kasutab JSON-põhist päringukeelt (Query DSL), mille abil saab keerukamaid päringuid koostada.
4. On tihti kasutatud Elastic Stack tarkvarakogumi ühe osana, mille moodustavad veel:
    - Logstash - andmete kogumiseks ja teisendamiseks,
    - Kibana - veebipõhine kasutajaliides Elasticsearchis sisalduvate andmete otsimiseks ja visualiseerimiseks,
    - Beats - andmete edastamiseks Elasticsearchi.

Elasticsearchi kasutatakse laialdaselt erinevates tarkvarasüsteemides, sealhulgas otsingumootorites, logianalüüsides, ärianalüüsides ja seires [23].

3.2.4 PostgreSQL

PostgreSQL, lühidalt "Postgres", on avatud lähtekoodiga SQL-andmebaasisüsteem, kus saab kasutada SQL andmebaasikeelt. Selle ajalugu algas 1980-ndatel aastatel California ülikoolis Berkeleys ning 2023. aasta detsembri seisuga oli see andmebaasisüsteemide seas populaarsuse järgi neljandal kohal [3]. PostgreSQL-il on suur ja aktiivne arendajate ja kasutajate kogukond, kes panustavad selle arendamisse ja pakuvad tuge. Lisaks on selle funktsionaalsuse täiustamiseks saadaval ulatuslik laienduste ja tööriistade ökosüsteem.

PostgreSQL leiab kasutust nii väiksema mahuga projektidest kui ka suurtes tarkvarasüsteemides. Selle funktsioonide, laiendatavuse ja avatud lähtekoodiga olemuse kombinatsioon teeb selle paljude arendajate ja organisatsioonide jaoks populaarseks valikuks [24].

3.3 Analüütiliste hierarhiate meetod

Analüütiliste hierarhiate meetodi (AHM) e Saaty meetodi töötas välja matemaatik Thomas L. Saaty. See on otsustusmeetod, mis on abiks alternatiivide vahel valiku tegemisel, kui valik põhineb mitmel kriteeriumil.[25] See meetod pakub struktureeritud lähenemisviisi otsuste tegemisele, jagades probleemi eesmärgi, kriteeriumite ja alternatiivide hierarhiliseks struktuuriks. Meetod võimaldab teha valiku nii, et see oleks minimaalselt subjektiivne. Meetodit kasutatakse laialdaselt erinevates valdkondades ja ka antud töös on see kasutusele võetud.

Meetodi põhisammud on järgmised.

1. Eesmärgi sõnastamine.
2. Alternatiivide määratlemine.
3. Kriteeriumite valimine.
4. Kriteeriumite paariviisiline võrdlemine ja igale kriteeriumile “kaalu” arvutamine. Kokkuvõttes leitakse kriteeriumite suhteline olulisus.
5. Alternatiivide paariviisiline võrdlemine iga kriteeriumi suhtes, et leida nende suhteline headus antud kriteeriumi suhtes.
6. Tulemuse arvutamine vastavalt kriteeriumite kaaludele, mille tulemusel saab iga alternatiiv arvulise “indeksi”.
7. Suurima “indeksiga” alternatiiv osutub eesmärgi mõttes parimaks valikuks.
8. Tundlikkuse analüüs, veendumaks, et väike kaalude muutmine ei tingi suurt muudatust tulemuses.

Alternatiivide paarikaupa võrdluseks kasutatakse Saaty skaalat (vt Tabel 1).

Tabel 1. Saaty skaala.

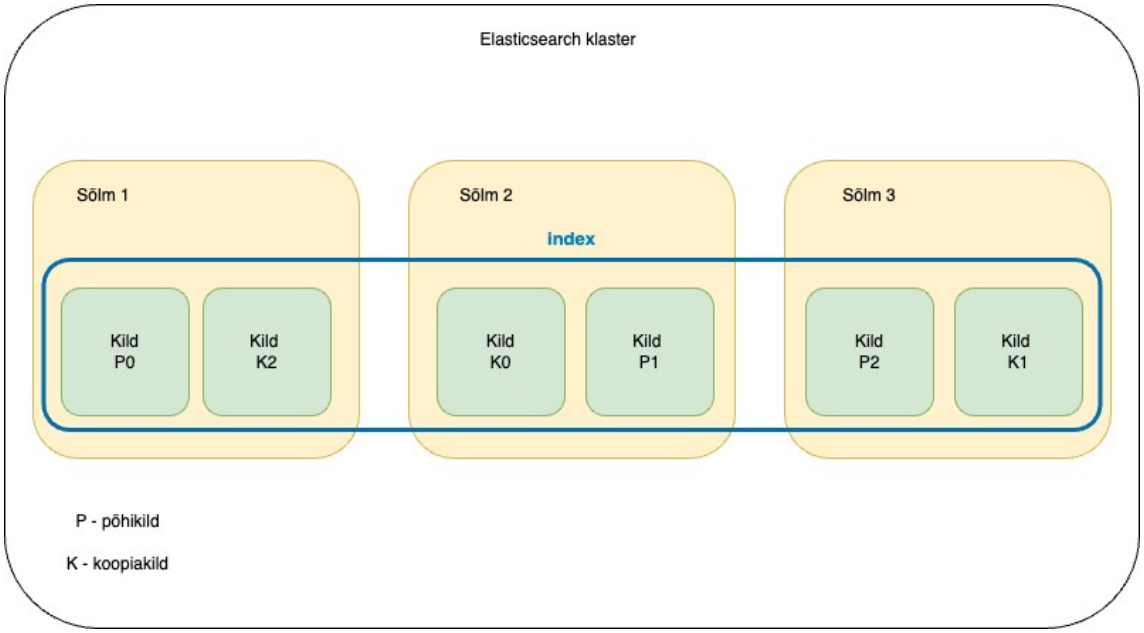
Intensiivsus	**Definitsioon**	**Selgitus**
1	Võrdselt tähtis	Panus on võrdne eesmärgi saavutamisel
2	Nõrk paremus või tähtsus	
3	Mõõdukas paremus või tähtsus	Kogemuse ja hinnangu põhjal eelistatakse kergelt ühte teisele
4	Mõõdukas-suur paremus või tähtsus	
5	Suur paremus või tähtsus	Kogemuse ja hinnangu põhjal eelistatakse tugevalt ühte teisele
6	Üsna suur paremus või tähtsus	
7	Väga suur paremus või tähtsus	Praktika näitab, et eelistatakse väga tugevalt ühte teisele
8	Väga-väga suur paremus või tähtsus	
9	Ekstreemne paremus või tähtsus	Tõendid kinnitavad suurimat võimalikku eelistust teise ees

4. Realisatsioon Elasticsearchis

Selles peatükis antakse ülevaade täistekstotsingu realiseerimisest Elasticsearchi abil nii nagu see on tehtud 2023. aasta detsembri seisuga realiseeritud süsteemi versioonis.

4.1 Andmete hoiustamine

Kui puutume kokku Elasticsearchiga, siis tavaliselt on tegemist klastriga - mitme koos töötava rakenduse eksemplariga. Andmeid hoitakse JSON dokumentidena, dokumendi kogumi nimi on indeks. Iga indeks on jaotatud mööda klastrit laiali. Andmete hoiustamisel on kasutusel killustamine e lõigustamine (*sharding*), kus iga kild on Apache Lucene indeks (andmete kogum). Killud võivad olla põhikillud (*primary shard*) või koopiakillud (*replica shard*). Põhikillud on kasutusel andmete otsinguks ja koopiakillud andmete taastamiseks, kui üks klastri sõlm (*node*) lakkab töötamast. Ühte andmeosa hoidvad põhi- ja koopiakild peavad asuma erinevates sõlmedes. Ehk siis kui meil on indeks üheksast dokumendist ja klaster koosneb kolmest sõlmest, kus igaühes on üks põhi- ja üks koopiakild, siis indeksi salvestamisel hoitakse igas sõlmes kolm dokumenti põhikillus ja kolm dokumenti koopiakillus. Piltliku esitust on näidatud joonisel 3.



Joonis 3. Elasticsearch klaster.

4.2 Indeksi loomine

Indeksi loomine käib Elasticsearchis REST API kaudu. Selle jaoks on olemas PUT meetod, kus URLis antakse ette indeksi nimetus, ning päringu kehas kirjeldatakse indeksi seadistusi [26]. Indeksi loomise päring on näidatud joonisel 4.

```
PUT /index-name
{
  "settings": {
    "number_of_shards": 1
    "number_of_replicas" : 1
  },
  "mappings": {
    "properties": {
      "id": { "type": "keyword" }
      "field1": { "type": "text" },
      "nestedField": {
        "type": "nested",
        "properties": {
          "field_name": {"type": "text"}
        }
      }
    }
  }
}
```

Joonis 4. Indeksi loomine Elasticsearchis.

Töös käsitletud süsteemis on suhtlus Elasticearchiga lahendatud Java rakenduses Spring Data Elasticsearch teegi abil [9]. See lubab seada igale dokumendile vastavusse Java klassi (vt. Joonis 5) ning salvestada dokumendid spetsiaalse liidese meetodi abil (vt. Joonis 6), mis lahendab ise REST päringu koostamist ja saatmist.

```
@Document(indexName = "study")
@Setting(settingPath = "/elastic_settings.json")
@Getter
@Setter
@JsonInclude(NON_NULL)
public class StudyIndex {

    @Id
    @Field(type = FieldType.Keyword)
    private String id;

    @Field(type = FieldType.Boolean, index = false)
    private Boolean isAdminData;

    @Field(type = FieldType.Text)
    private String title;

    @Field(type = FieldType.Text)
    private String summary;

    @Field(type = FieldType.Nested)
    private List<DataFileNested> dataFiles;
}
```

Joonis 5. Dokumendi objekti loomine Spring rakenduses.

```
public interface StudyRepository extends ElasticsearchRepository
   <StudyIndex, String> {}

...

List<StudyIndex> studyIndices = getStudyIndicies();
studyRepository.saveAll(studyIndices);
```

Joonis 6. Elasticsearch repositooriumi kasutamine Spring rakenduses.

4.3 Andmete otsing

Java rakenduses on päringute tegemiseks kasutatud *Spring Data Elasticsearch* teegi poolt pakutud klassi *ElasticsearchOperations*, mis teeb andmevahetuse mugavamaks (vt. Joonis 7). Andmete otsingu jaoks on kasutusel REST API päring kujul POST /{indeksi-nimi}/_search [27]. Päring on näidatud joonisel 8. Otsingupäringu sisu on base64 kodeeritud JSON, mille sisu on näidatud joonisel 9.

```
    Pageable pageable = getSearchQueryPageable(searchQueryParams
       .getPage());
    Query query = new StudySearchQuery(searchQueryParams,
       pageable).getQuery();
    SearchHits<StudyIndex> searchHits = operations.search(query,
        StudyIndex.class);
```

Joonis 7. *Elasticsearchoperations* abil päringu tegemine Spring rakenduses.

```
POST /study/_search?search_type=query_then_fetch&typed_keys=true
{
  "_source": {
    "includes": [
      "title",
      "referenceArea",
      "timeCoverage",
      "universeLabel",
      "seriesTitle",
      "purpose",
      "dataFiles.logicalRecords.id",
      "dataFiles.logicalRecords.variables.id"
    ]
  },
  "from": 0,
  "query": {
    "wrapper": {
      "query": "{base64EncodedJsonQuery}"
    }
  },
  "size": 10,
  "track_scores": false,
  "version": true
}
```

Joonis 8. Elasticsearchi otsingupäring.

```
{"bool": {"must": [
      {
        "bool": {
          "should": [
            {
              "multi_match": {
                "query": "search",
                "fields": [
                  "seriesTitle",
                  "universeLabel",
                  "title",
                  "summary",
                  "purpose",
                  "studyCode",
                  "sectorCoverage",
                  "otherDissemination",
                  "documentationOnMethodology",
                  "geographicalComparability",
                  "comparabilityOverTime",
                  "sourceData",
                  "frequencyOfDataCollection",
                  "dataCollection",
                  "dataValidation",
                  "dataCompilation"]}},
            {
              "nested": {
                "path": "dataFiles",
                "query": {
                  "match": {
                    "dataFiles.title": {
                      "query": "search"}}}}},
            {
              "nested": {
                "path": "dataFiles.keywords",
                "query": {
                  "match": {
                    "dataFiles.keywords.name": {
                      "query": "search"}}}}},
            {
              "nested": {
                "path": "dataFiles.subjects",
                "query": {
                  "match": {
                    "dataFiles.subjects.name": {
                      "query": "search"}}}}},
            {
              "nested": {
                "path": "dataFiles.logicalRecords",
                "query": {
                  "multi_match": {
                    "query": "search",
                    "fields": [
                      "dataFiles.logicalRecords.name",
                      "dataFiles.logicalRecords.label",
                      "dataFiles.logicalRecords.description"]}}}},
            {
              "nested": {
                "path": "dataFiles.logicalRecords.variables",
                "query": {
                  "multi_match": {
                    "query": "search",
                    "fields": [
                      "dataFiles.logicalRecords.variables.name",
                      "dataFiles.logicalRecords.variables.label",
                      "dataFiles.logicalRecords.variables.description",
                      "dataFiles.logicalRecords.variables.representedVariableLabel",
                      "dataFiles.logicalRecords.variables.conceptualVariableLabel"
                    ]}},
                "inner_hits": {
                  "size": 1000,
                  "highlight": {
                    "number_of_fragments": 0,
                    "fields": {
                      "dataFiles.logicalRecords.variables.label": {}}}}}},
            {
              "nested": {
                "path": "dataFiles.logicalRecords.variables.concepts",
                "query": {
                  "match": {
                    "dataFiles.logicalRecords.variables.concepts.label": {
                      "query": "search"}}}}}]}}]}}
```

Joonis 9. Otsingupäringu esitus JSON formaadis.

5. Realisatsioon PostgreSQLis

Esimene samm, võrdlemaks Elasticsearchi ja PostgreSQLi täistekstotsingu võimekust, on PostgreSQLis vastava realisatsiooni loomine. PostgreSQL pakub mitmeid võimalusi täistekstotsingu realiseerimiseks. Põhilised tööriistad selleks on *tsvector* ja *tsquery*. *Tsvector* on andmetüüp, millesse kuuluv väärtus hoiab dokumenti või tekstiosa täistekstotsinguks optimeeritud kujul. See on sisuliselt erinevate sõnade (lekseemide) sorteeritud loend koos informatsiooniga nende asukoha kohta dokumendis. *Tsquery* on andmetüüp, mida kasutatakse päringu esitamiseks täistekstotsinguks sobival kujul. *websearch_to_tsquery* funktsiooni abil saab sõne teisendada täistekstotsinguks sobivaks *tsquery*'ks. Lihtne näide tesktivektori kasutusest on näidatud joonisel 10.

```
CREATE TABLE test_table (
    id UUID PRIMARY KEY,
    content TSVECTOR NOT NULL
);

INSERT INTO test_table(id, content)
VALUES (
'649e811a-c5f1-4822-8026-a6b546b7123a'::UUID,
to_tsvector('King Arthur is a legendary king of Britain')
);

Tsvector: "'arthur':2 'britain':8 'king':1,6 'legendari':5'"

SELECT * FROM test_table WHERE content @@ to_tsquery('king')
```

Joonis 10. Tekstivektori kasutamine PostgreSQLis.

Järgnevas analüüsis ning läbiviidud eksperimentides kasutatakse samasuguse struktuuriga JSON dokumente nagu tegelikus süsteemis. PostgreSQL puhul kasutatakse PostgreSQL versiooni 16.

5.1 Testandmete genereerimine

Lahenduse testimiseks kasutatakse genereeritud testandmeid. Testandmete genereerimiseks nii esialgsel testimisel kui ka peatükis 5 kirjutatud eksperimentides on kasutatud projektimeeskonna liikme poolt loodud Node.js rakendust. Rakendus küsib, kui palju oodatakse andmeid erinevates hierarhia tasemetes ja siis genereerib juhuslikest sõnedest koosnevaid fraase sisaldavad JSON dokumendid. Andmete hulk on esialgse testimise eesmärkide saavutamiseks väiksem kui on oodatud reaalses kasutuses.

Tabel 2. Esialgseks testimiseks andmete arv.

Andmetabel	**Esialgse testimise tarbeks ridade arv**
metadata_domain	3
sub_domain	9
series	27
study	135
other_material	405
data_file	405
keyword	1215
subject	1215
logical_record	1215
variable	8505
concept	25515
quality_indicator	29160
code_list	2
category	4

5.2 Baastabel iga olemitüübi kohta

Lahendamaks täistekstotsingu probleemi, soovitab PostgreSQL dokumentatsioon lisada igasse otsingus kasutatavasse baastabelisse e tabelisse *tsvector* tüüpi veeru, mis hakkab hoidma täistesktotsinguks vaja olevat tekstivektorit. Seejärel saab otsingutulemusi leida kasutades päringu WHERE klauslis *websearch_to_tsquery* funktsiooni, nii nagu on näidatud joonisel 11. Otsingu kiirendamiseks soovitatakse kasutada GIN indeksit. PostgreSQLis on GIN-indeks (*Generalized Inverted Index*) andmebaasi siseskeemi kuuluv andmestruktuur, mis on loodud liitväärtuste (näiteks dokumentide) tõhusaks käsitlemiseks, et võimaldada kiiret otsingut liitväärtusesse kuuluvate väärtuste (näiteks dokumendis olevad sõnad) põhjal [28].

```
SELECT name FROM department WHERE to_tsvector(search)
@@ websearch_to_tsquery('english', 'search');
```

Joonis 11. Tekstivektori kasutamisel põhinev päring PostgreSQLis.

Seega esimene lahendusvariant, millega alustada on järgmine.

1. Salvestada JSON dokumendist loetud olemite andmed eraldi tabelitesse - iga olemitüübi kohta on eraldi tabel ja iga olemi kohta on tabelis eraldi rida.
2. Lisaks automaatselt loodud indeksitele (primaarvõtme ja unikaalsuse kitsenduste alusel) luua indeksid ka välisvõtme veergudele, et kiirendada ühendamisoperatsioonide läbiviimist.
3. Lisada tabelisse *tsvector* tüüpi veerg arvutatava veeruna, milles olev väärtus arvutatakse teiste samas reas olevate andmete põhjal. Alternatiiviks oleks kasutada *tsvector* tüüpi veergu väärtuste arvutamiseks tabelitega seotud reataseme INSERT ja UPDATE trigereid.
4. Lisada igale *tsvector* tüüpi veerule indeks.
5. Sooritada tesktiotsing *websearch_to_tsquery* abil.

Tekstivektori veeru loomine ja sellele indeksi lisamine on näidatud joonisel 12.

```
ALTER TABLE study ADD COLUMN study_search TSVECTOR GENERATED ALWAYS AS
    (to_tsvector('english',
     coalesce(series_title, '') || ' ' ||
     coalesce(universe_label, '') || ' ' ||
     coalesce(title, '') || ' ' ||
     coalesce(summary, '') || ' ' ||
     coalesce(purpose, '') || ' ' ||
     coalesce(study_code, '') || ' ' ||
     coalesce(sector_coverage, '') || ' ' ||
     coalesce(other_dissemination, '') || ' ' ||
     coalesce(documentation_on_methodology, '') || ' ' ||
     coalesce(geographical_comparability, '') || ' ' ||
     coalesce(comparability_over_time, '') || ' ' ||
     coalesce(source_data, '') || ' ' ||
     coalesce(frequency_of_data_collection, '') || ' ' ||
     coalesce(data_collection, '') || ' ' ||
     coalesce(data_validation, '') || ' ' ||
     coalesce(data_compilation, ''))
    ) STORED NOT NULL;

 CREATE INDEX idx_study_table_search ON study USING gin(study_search);
```

Joonis 12. Tekstivektori veeru loomine PostgreSQLis.

Päring, mis kasutab testivektoreid, on näidatud joonisel 13. Kuna otsinguvasteid peab otsima uuringu ja tema allelementide seast, siis on vaja antud päringu korral teostada suure arvu tabelite liitmist.

```
SELECT * FROM (
     SELECT DISTINCT ON
         (s.study_id)  s.study_id as study_id,
          s.series_title as series_title,
          s.universe_label as universe_label,
          s.title as title,
          s.purpose as purpose,
          s.reference_area as reference_area,
          s.time_coverage as time_coverage,
          ts_rank(s.study_search ||
                  df.data_file_search ||
                  kw.keyword_search ||
                  sb.subject_search ||
                  lr.logical_record_search ||
                  v.variable_search ||
                  c.concept_search, websearch_to_tsquery(:search)) as rank
          FROM concept c
                   JOIN variable v USING (variable_id)
                   JOIN logical_record lr USING (logical_record_id)
                   JOIN data_file df USING (data_file_id)
                   JOIN keyword kw USING (data_file_id)
                   JOIN subject sb USING (data_file_id)
                   JOIN study s USING (study_id)
                   JOIN series sr ON sr.series_id = s.series_id
                   JOIN subdomain sd ON sd.subdomain_id = s.subdomain_id
                   JOIN metadata_domain md ON md.metadata_domain_id = s.
                       metadata_domain_id
          WHERE s.study_search ||
                df.data_file_search ||
                kw.keyword_search ||
                sb.subject_search ||
                lr.logical_record_search ||
                v.variable_search ||
                c.concept_search @@ websearch_to_tsquery(:search)
          GROUP BY s.study_id, s.series_title, s.universe_label, s.title, s.purpose,
               s.reference_area, s.time_coverage,
                  ts_rank(s.study_search ||
                          df.data_file_search ||
                          kw.keyword_search ||
                          sb.subject_search ||
                          lr.logical_record_search ||
                          v.variable_search ||
                          c.concept_search, websearch_to_tsquery(:search))
                     ORDER BY study_id, rank DESC
              ) sub
ORDER BY rank DESC
OFFSET 0
FETCH NEXT 10 ROWS ONLY;
```

Joonis 13. Päring tabelite veergudes hoitud tesktivektorite puhul.

Täistekstotsingu vastete leidmiseks saab kasutada joonisel 14 esitatud WHERE klauslis olevat otsingutingimust.

```
WHERE tsvector_column @@ websearch_to_tsquery('search')
```

Joonis 14. Tekstivektori vastete leidmine.

Järjestamaks tulemused olulisuse järgi on abiks funktsioon *ts_rank*, mis tagastab tulemuse olulisust näitava arvulise väärtuse. Mida suurem arv, seda olulisem on tulemus (vt. Joonis 15). Relevantsuse välja arvutamiseks võtab funktsioon arvesse kui palju vasteid ühe rea sees on otsingusõnega leitud ning kui palju on konkreetses reas vasteid võrreldes teiste vastet saanud ridadega.

```
ts_rank(tsvector_column, websearch_to_tsquery('search'))
```

Joonis 15. Testkivektori päringus olulisuse arvutamine.

Otsingusõne esiletõstmiseks kasutame funktsiooni *ts_headline* (vt. Joonis 16). Vaikimisi asetab see esiletõstetud tesktiosad <b> märgendite vahele. Vajadusel saab funktsiooni argumendina määrata teistusguse märgendi. Tulemust näitav HTML leht saab need sõne osad seeläbi esile tuua.

```
ts_headline(field, websearch_to_tsquery('search'))
```

Joonis 16. Testkivektori päringus otsingusõne esiletoomine.

Otsinguvasteid sisaldavaid muutujaid agregeeritakse iga *study* kohta kasutades *json_agg* kokkuvõttefunktsiooni (vt. Joonis 17). See võimaldab esitada igale uuringule vastavad andmed JSON kujul. Selle tulemusena sisaldub igas otsingutulemuse reas vastet saanud muutujate olemasolul JSON formaadis nimekiri nendest muutujatest. *json_agg* funktsiooniga on võimalik kasutada ka ORDER BY ja WHERE klauslit.

```
json_agg(jsonb_build_object(
            'variable', variable_id,
            'logicalRecordId', logical_record_id,
            'variableLabel', variable_label,
            'variableName', variable_name,
            'representationType', representation_type,
            'variableRank', variable_rank) ORDER BY
               variable_rank )
            FILTER ( WHERE variable_rank > 0 ) AS variables
```

Joonis 17. Testkivektori päringus muutujate andmete agregeerimine.

Otsing antud lahenduse puhul võtab 5-5.5 sekundit, mis ei ole piisavalt kiire. Aeglase päringu täitmise põhjuseks on tabelite päringutes ühendamine, kuna SELECT algab kontseptide (*concept*) tabelis (kus on hetkel 25515 rida) ja ühendab tabelid kuni uuringute (*study*) tabelini välja, saades tulemuseks suure ridade hulga. Tabelite struktuuri pilt on nähtav lisas 3.

5.3 Materialiseeritud vaade

Järgnev lahendus täiendab baastabel olemitüübi kohta lahendust. Kuna andmed muutuvad harva, siis antud kontekstis on mõttekam teha otsingu jaoks eraldi hoidla, mis koondab juba eelnevalt kokku otsingu jaoks vajaliku info. Esimene selleks mõttesse tulev lahendus on materialiseeritud vaate (*materialized view)* e hetktõmmise loomine, mida uuendatakse iga kord kui uuendatakse andmeid (kord ööpäevas). Sellise vaate korral on alampäringu tulemus ette välja arvutatud ja eraldi salvestatud. Tehes päringut sellise tuletatud tabeli põhjal ei kuluta andmebaasisüsteem liigselt aega ühendamise operatsioonile, kuna ühendamine on juba eelnevalt tehtud. Samas suurendab see andmete salvestamiseks kuluvat aega, sest materialiseeritud vaadet peab värskendama. Selleks saab kasutada REFRESH MATERIALIZED VIEW lauset. Kasutatud PostgreSQLi versioonis tähendab see materialiseeritud vaate nullist uuesti loomist (st ei toimu osalist värskendamist). Andmete uuendamise väike sagedus tähendab, et värskendamisest tingitud täiendav ajakulu on aktsepteeritav. Kuna uus saadav andmete komplekt kirjutab vanad andmed üle, siis on ka materialiseeritud vaate nullist uuesti loomine igati sobiv lahendus.

Ülesande lahendamiseks luuakse andmebaasi materialiseeritud vaade, mis sisaldab päringu tulemuses väljastatavaid veerge *study* tabelist ning kahte tekstivektori veergu - üks uuringute ja teine muutujate otsimiseks. Ka nendele tekstivektori veergudele luuakse indeks.

Materialiseeritud vaate ja seotud indeksite loomine on näidatud joonisel 18:

```
CREATE MATERIALIZED VIEW study_search_index AS
    SELECT s.study_id AS study_id, s.series_title AS series_title, s.universe_label AS
        universe_label,
           s.title AS title, s.purpose AS purpose, s.reference_area AS reference_area,
           s.time_coverage AS time_coverage,
           v.variable_id as variable_id, v.label as variable_label, v.name as
               variable_name,
           v.representation_type as representation_type, c.label as concept_label,
           to_tsvector(coalesce(s.series_title, '')) || '' ||
           to_tsvector(coalesce(s.universe_label, '')) || '' ||
           to_tsvector(coalesce(s.title, '')) || '' ||
           to_tsvector(coalesce(s.summary, '')) || '' ||
           to_tsvector(coalesce(s.purpose, '')) || '' ||
           to_tsvector(coalesce(s.study_code, '')) || '' ||
           to_tsvector(coalesce(s.sector_coverage, '')) || '' ||
           to_tsvector(coalesce(s.other_dissemination, '')) || '' ||
           to_tsvector(coalesce(s.documentation_on_methodology, '')) || '' ||
           to_tsvector(coalesce(s.geographical_comparability, '')) || '' ||
           to_tsvector(coalesce(s.comparability_over_time, '')) || '' ||
           to_tsvector(coalesce(s.source_data, '')) || '' ||
           to_tsvector(coalesce(s.frequency_of_data_collection, '')) || '' ||
           to_tsvector(coalesce(s.data_collection, '')) || '' ||
           to_tsvector(coalesce(s.data_validation, '')) || '' ||
           to_tsvector(coalesce(s.data_compilation, '')) || '' ||
           to_tsvector(coalesce(df.title, '')) || '' ||
           to_tsvector(coalesce(kw.name, '')) || '' ||
           to_tsvector(coalesce(sb.name, '')) || '' ||
           to_tsvector(coalesce(lr.name, '')) || '' ||
           to_tsvector(coalesce(lr.label, '')) || '' ||
           to_tsvector(coalesce(lr.description, '')) || '' ||
           to_tsvector(coalesce(v.name, '')) || '' ||
           to_tsvector(coalesce(v.label, '')) || '' ||
           to_tsvector(coalesce(v.description, '')) || '' ||
           to_tsvector(coalesce(v.represented_variable_label, '')) || '' ||
           to_tsvector(coalesce(v.conceptual_variable_label, '')) || '' ||
           to_tsvector(coalesce(c.label, '')) AS document,
           to_tsvector(coalesce(v.name, '')) || '' ||
           to_tsvector(coalesce(v.label, '')) || '' ||
           to_tsvector(coalesce(v.description, '')) || '' ||
           to_tsvector(coalesce(v.represented_variable_label, '')) || '' ||
           to_tsvector(coalesce(v.conceptual_variable_label, '')) AS variable_document
    FROM concept c
             JOIN variable v USING (variable_id)
             JOIN logical_record lr USING (logical_record_id)
             JOIN data_file df USING (data_file_id)
             JOIN keyword kw USING (data_file_id)
             JOIN subject sb USING (data_file_id)
             JOIN study s USING (study_id);

    CREATE INDEX idx_study_search_document ON study_search_index USING gin(document);
    CREATE INDEX idx_study_search_variable_document ON study_search_index USING gin(
        variable_document);
```

Joonis 18. Materialiseeritud vaate ja seotud indeksite loomine.

Materialiseeritud vaatest otsimine on näidatud joonisel 19 .

```
SELECT * FROM (
    SELECT DISTINCT ON
        (study_id) study_id,
        series_title AS series_title,
        universe_label AS universe_label,
        title AS title,
        purpose AS purpose,
        reference_area AS reference_area,
        time_coverage AS time_coverage,
        json_agg(DISTINCT jsonb_build_object(
            'variable', variable_id,
            'logicalRecordId', logical_record_id,
            'variableLabel', ts_headline(variable_label,
               websearch_to_tsquery(:search)),
            'variableName', ts_headline(variable_name,
               websearch_to_tsquery(:search)),
            'representationType', representation_type,
            'variableRank', ts_rank(variable_document,
               websearch_to_tsquery(:search)))
            ) FILTER (WHERE variable_document @@
               websearch_to_tsquery(:search)) AS variables,
        ts_rank(document, websearch_to_tsquery(:search)) AS rank
    FROM study_search_index
    WHERE document @@ websearch_to_tsquery(:search)
    GROUP BY study_id, series_title, universe_label, title,
       purpose, reference_area, time_coverage, ts_rank(document,
        websearch_to_tsquery(:search))
    ORDER BY study_id, rank DESC) sub
ORDER BY rank DESC;
```

Joonis 19. Päring materialiseeritud vaatest.

Uuendatud lahenduse korral täidetakse päring kiiremalt: 1000-1500 millisekundiga. Eelmise lahendusega võrreldes on see parem, kuid soov oleks saada tulemus vähem kui 1000 millisekundiga. Üks põhjus, miks sellise lähenemisega päringu täitmist piisavalt kiireks ei saa, seisneb selles, et materialiseeritud vaates on ühe uuringu kohta keskmiselt 220 000 rida. Uuringu tabelis on sellel ajal kõigest 135 rida.

5.4 Eeltöödeldud andmeid sisaldavad baastabelid

Järgnev lahendus täiendab baastabel olemitüübi kohta lahendust. Palju efektiivsem oleks teostada otsingut juhul, kui otsingu allikas on tabel kus iga otsitava elemendi kohta on üks rida ning selles reas on kõik tekstiotsingu aluseks olevad infokillud koos. Selleks saab lisaks olemasolevatele baastabelitele, mis detailivaadete päringute jaoks andmeid hoiavad, luua eraldiseisva tabeli, kus on uuringute otsingutulemuste väljastamise jaoks vajalikud veerud ning eraldi tekstivektori veerg, mis on sisaldab kõiki sõnesid mis meid konkreetses uuringus otsingu jaoks huvitavad ja ka kõiki selle allolevatest olemitest tulenevaid sõnesid. Samuti luuakse eraldi otsingutabel muutujate (*variables*) jaoks. Tabeleid saab luua joonisel 20 näidatud viisil.

```
CREATE TABLE study_search_document_store
(
    study_id UUID PRIMARY KEY,
    series_title TEXT,
    universe_label TEXT NOT NULL,
    title TEXT,
    purpose TEXT,
    reference_area TEXT,
    time_coverage TEXT,
    study_search_document TSVECTOR,
    CONSTRAINT study_search_document_store_study_fk FOREIGN KEY (study_id)
        REFERENCES study (study_id)
);

CREATE INDEX idx_study_search_document_store_study_search_document ON
    study_search_document_store USING gin(study_search_document);

CREATE TABLE variable_search_document_store
(
    variable_id UUID PRIMARY KEY,
    study_id UUID NOT NULL,
    logical_record_id UUID NOT NULL,
    name TEXT,
    label TEXT,
    representation_type TEXT,
    variable_search_document TSVECTOR,
    CONSTRAINT variable_search_document_study_fk FOREIGN KEY (study_id) REFERENCES
        study (study_id)
);

CREATE INDEX idx_variable_search_document_store_study_id ON
    variable_search_document_store (study_id);
CREATE INDEX idx_variable_search_document_store_variable_search_document ON
    variable_search_document_store USING gin(variable_search_document);
```

Joonis 20. Otsingu tarbeks kohandatud baastabelite loomine.

Andmete vastuvõtmisel pakendatakse programselt kokku kõik sõned uuringust ja seotud olemitest ning seejärel sisestatakse otsingu-tabelitesse, nagu on näidatud joonisel 21.

```
INSERT INTO study_search_document_store (
study_id, series_title, universe_label,
title, purpose, reference_area, time_coverage,
study_search_document)
VALUES (?, ?, ?, ?, ?, ?, ?, to_tsvector(?::text));

INSERT INTO variable_search_document_store (variable_id, study_id,
logical_record_id, name, label,
representation_type, variable_search_document)
VALUES (?, ?, ?, ?, ?, ?, to_tsvector(?::text));
```

Joonis 21. Andmete sisestamine otsingu tarbeks kohandatud baastabelitesse.

Otsingut tegemiseks kasutatakse Joonisel 22 näidatud päringut.

```
WITH variables_match AS (
        SELECT variable_id AS variable_id,
               study_id AS study_id,
               logical_record_id AS logical_record_id,
               ts_headline(label, websearch_to_tsquery(:search)) AS variable_label,
               ts_headline(name, websearch_to_tsquery(:search)) AS variable_name,
               representation_type AS representation_type,
               ts_rank(variable_search_document, websearch_to_tsquery(:search)) AS variable_rank
        FROM variable_search_document_store
        WHERE variable_search_document @@ websearch_to_tsquery(:search)
    ),
         variables_selected AS (
             SELECT variable_id,
                    study_id,
                    logical_record_id,
                    variable_label,
                    variable_name,
                    representation_type,
                    variable_rank
             FROM variables_match
             ORDER BY variable_rank DESC
         )
    SELECT study_id as study_id,
           series_title AS series_title,
           universe_label AS universe_label,
           title AS title,
           purpose AS purpose,
           reference_area AS reference_area,
           time_coverage AS time_coverage,
           ts_rank(study_search_document, websearch_to_tsquery(:search)) AS study_rank,
           json_agg(jsonb_build_object(
                'variable', variable_id,
                'logicalRecordId', logical_record_id,
                'variableLabel', variable_label,
                'variableName', variable_name,
                'representationType', representation_type,
                'variableRank', variable_rank) ORDER BY variable_rank ) FILTER ( WHERE variable_rank > 0 ) AS
                    variables
    FROM study_search_document_store LEFT JOIN variables_selected USING (study_id)
    WHERE study_search_document @@ websearch_to_tsquery(:search)
    GROUP BY study_id, series_title, universe_label, title, purpose, reference_area, time_coverage,
             ts_rank(study_search_document, websearc<h_to_tsquery(:search))
    ORDER BY study_rank DESC
    OFFSET 0
    FETCH NEXT 10 ROWS ONLY;
```

Joonis 22. Päring otsingu tarbeks kohandatud baastabelitest.

Lehekülgede kaupa tulemuse tagastamiseks on vaja lugeda ka tabeli ridade arvu, milles oli leitud otsinguvaste. Selle abil saab arvutada erinevate leheküljenumbrite puhul nihke ja tagastada päringu vastuses tulemuste koguarvu. Ridade arvu leidmise päring on näidatud joonisel 23.

```
SELECT count(*) AS total FROM study_search_document_store
WHERE study_search_document @@ websearch_to_tsquery(:search);
```

Joonis 23. Otsingu tarbeks kohandatud baastabelitest otsingu vastete arvu küsimine.

Sellise lähenemisega on tulemus märkimisväärselt parem - kokku 10-20 millisekundit olemite otsingu ja kokkulugemise päringute jaoks ning seejärel vaherakenduse poolt töötlemiseks ja vastuse väljastamiseks. Antud lahendus on otsingu kiiruse mõttes parim PostgreSQL realisatsioon, mis vaadeldavas olukorras välja tuli. Seda saab kasutada ühe alternatiivina Elasticsearch-ga võrdluses.

5.5 JSONB tüüpi veerg

Alates PostgreSQL versioonist 9.2, mis tuli välja 2012. aasta septembris, on andmebaasisüsteemis olemas süsteemi-defineeritud e sisseehitatud andmetüüp JSON. Alates PostgreSQL versioonist 9.4, mis tuli välja 2014. aasta detsembris, on andmebaasisüsteemis lisaks kasutusvalmis andmetüüp JSONB [29].

JSONB on andmetüüp, mille korral salvestab süsteem JSONi (JavaScript Object Notation) andmed binaarvormingus e kahendvormingus. "B" tähistab JSONB-s "binaarset" ja see näitab, et JSON-andmed on salvestatud kahendformaadis. See on palju tõhusam võrreldes JSON andmetüübiga, mille korral salvestatakse andmed tavalise tekstina. Dokumentatsiooni kohaselt on JSONB tüüpi väärtuste salvestamine veidi aeglasem kui JSON tüüpi väärtuste salvestamine, sest JSONB puhul tuleb andmete teisendamiseks teha lisatööd. Samas on JSONB tüüpi andmete töötlemine oluliselt kiirem kui JSON tüüpi andmete töötlemine. Samuti saab JSONB tüüpi veergudele luua indekseid.

Üks võimalik lähenemine antud otsinguprobleemi lahendamiseks on kasutada JSONB tüüpi veergu, nagu näidatud joonisel 24. See oleks mugav selle poolest, et välisteenusest pärinevad andmed tulevad süsteemi JSON-formaadis ja seega ei oleks vaja realiseerida nende andmete lahtilõhkumist ja eraldi tabelitesse salvestamist.

Välisest teenusest tuleb sisse üks suur JSON dokument, mis sisaldab infot erinevate uuringute ja nende alamobjektide kohta.

Esimeseks lahenduseks on luua tabel, kus primaarvõtme veerus on uuringu identifikaator (UUID tüüpi väärtus) ja teine veerg sisaldab selle uuringu JSON-formaadis andmeid. Seega oleks tabelis üks rida iga uuringu kohta.

```
CREATE TABLE study_json (
    study_id UUID PRIMARY KEY,
    data JSONB NOT NULL
);
```

Joonis 24. JSONB veeruga baastabel andmete salvestamiseks uuringute kaupa.

Kõigepealt proovitakse otsida JSONB funktsioonidega ja tõstutundetut otsingut võimaldava ILIKE operaatoriga kasutades päringut, mis on näidatud joonisel 25.

```
SELECT DISTINCT studies.data-> 'id' AS studyId,
                    studies.data->'seriesTitle' AS seriesTitle,
                    studies.data->'universeLabel' AS universeLabel,
                    studies.data->'title' AS title,
                    studies.data->'purpose' AS purpose,
                    studies.data->'referenceArea' AS referenceArea,
                    studies.data->'timeCoverage' AS timeCoverage
    FROM study_json AS studies,
         jsonb_array_elements(studies.data->'dataFiles') AS dataFiles,
         jsonb_array_elements(dataFiles->'subjects') AS subjects,
         jsonb_array_elements(dataFiles->'keywords') AS keywords,
         jsonb_array_elements(dataFiles->'logicalRecords') AS logicalRecords,
         jsonb_array_elements(logicalRecords->'variables') AS variables,
         jsonb_array_elements(variables->'concepts') AS concepts
    WHERE
    studies.data->>'seriesTitle' ILIKE :search
    OR studies.data->>'universeLabel' ILIKE :search
    OR studies.data->>'title' ILIKE :search
    OR studies.data->>'summary' ILIKE :search
    OR studies.data->>'purpose' ILIKE :search
    OR studies.data->>'studyCode' ILIKE :search
    OR studies.data->>'sectorCoverage' ILIKE :search
    OR studies.data->>'otherDissemination' ILIKE :search
    OR studies.data->>'documentationOnMethodology' ILIKE :search
    OR studies.data->>'geographicalComparability' ILIKE :search
    OR studies.data->>'comparabilityOverTime' ILIKE :search
    OR studies.data->>'sourceData' ILIKE :search
    OR studies.data->>'frequencyOfDataCollection' ILIKE :search
    OR studies.data->>'dataCollection' ILIKE :search
    OR studies.data->>'dataValidation' ILIKE :search
    OR studies.data->>'dataCompilation' ILIKE :search
    OR dataFiles->>'title' ILIKE :search
    OR keywords->>'name' ILIKE :search
    OR subjects->>'name' ILIKE :search
    OR logicalRecords->>'name' ILIKE :search
    OR logicalRecords->>'label' ILIKE :search
    OR logicalRecords->>'description' ILIKE :search
    OR variables->>'name' ILIKE :search
    OR variables->>'label' ILIKE :search
    OR variables->>'description' ILIKE :search
    OR variables->>'representedVariableLabel' ILIKE :search
    OR variables->>'conceptualVariableLabel' ILIKE :search
    OR concepts->>'label' ILIKE :search;
```

Joonis 25. Päring uuringute kaupa andmetega JSONB tüüpi veeruga tabelist.

Tulemus ei ole kõige parem, sest päringu täitmiseks kulub 3-4 sekundit.

Veel üks lahendus on selline, et terve suur JSON dokument, mis andmete salvestamisel sisse tuleb, oleks täies mahus salvestatud ühe tabeli ritta (vt Joonis 26) ja selle põhjal tehakse otsing (vt Joonis 27). Sellisel juhul olekski tabelis ainult üks rida.

```
CREATE TABLE domain_json (
    domain_json_id BIGSERIAL PRIMARY KEY,
    data JSONB NOT NULL
);
```

Joonis 26. JSONB veeruga baastabel sissetulnud JSON dokumendi töötlemata kujul salvestamiseks.

```
SELECT DISTINCT studies->>'id' AS studyId,
                studies->>'seriesTitle' AS seriesTitle,
                studies->>'universeLabel' AS universeLabel,
                studies->>'title' AS title,
                studies->>'purpose' AS purpose,
                studies->>'referenceArea' AS referenceArea,
                studies->>'timeCoverage' AS timeCoverage
        FROM domain_json,
             jsonb_array_elements(data->'domains') AS domains,
             jsonb_array_elements(domains->'subdomains') AS subdomains,
             jsonb_array_elements(subdomains->'series') AS series,
             jsonb_array_elements(series->'studies') AS studies,
             jsonb_array_elements(studies->'dataFiles') AS dataFiles,
             jsonb_array_elements(dataFiles->'subjects') AS subjects,
             jsonb_array_elements(dataFiles->'keywords') AS keywords,
             jsonb_array_elements(dataFiles->'logicalRecords') AS logicalRecords,
             jsonb_array_elements(logicalRecords->'variables') AS variables,
             jsonb_array_elements(variables->'concepts') AS concepts
        WHERE
              studies->>'seriesTitle' ILIKE :search
           OR studies->>'universeLabel' ILIKE :search
           OR studies->>'title' ILIKE :search
           OR studies->>'summary' ILIKE :search
           OR studies->>'purpose' ILIKE :search
           OR studies->>'studyCode' ILIKE :search
           OR studies->>'sectorCoverage' ILIKE :search
           OR studies->>'otherDissemination' ILIKE :search
           OR studies->>'documentationOnMethodology' ILIKE :search
           OR studies->>'geographicalComparability' ILIKE :search
           OR studies->>'comparabilityOverTime' ILIKE :search
           OR studies->>'sourceData' ILIKE :search
           OR studies->>'frequencyOfDataCollection' ILIKE :search
           OR studies->>'dataCollection' ILIKE :search
           OR studies->>'dataValidation' ILIKE :search
           OR studies->>'dataCompilation' ILIKE :search
           OR dataFiles->>'title' ILIKE :search
           OR keywords->>'name' ILIKE :search
           OR subjects->>'name' ILIKE :search
           OR logicalRecords->>'name' ILIKE :search
           OR logicalRecords->>'label' ILIKE :search
           OR logicalRecords->>'description' ILIKE :search
           OR variables->>'name' ILIKE :search
           OR variables->>'label' ILIKE :search
           OR variables->>'description' ILIKE :search
           OR variables->>'representedVariableLabel' ILIKE :search
           OR variables->>'conceptualVariableLabel' ILIKE :search
           OR concepts->>'label' ILIKE :search
OFFSET 0
FETCH NEXT 10 ROWS ONLY;
```

Joonis 27. Päring töötlemata kujul JSON dokumendiga tabelist.

Isegi väiksemas mahus testandmetega on see päring ikkagi aeglane - täitmiseks kulub mitu sekundit. Erinevad õpetused [30] soovitavad kasutada JSONB tüüpi veerust otsingu hõlbustamiseks indekseid. Üks variant on trigram indeks, mis jagab sõned kolmetähelisteks osadeks ja võimaldab leida otsinguvasteid efektiivsemalt.

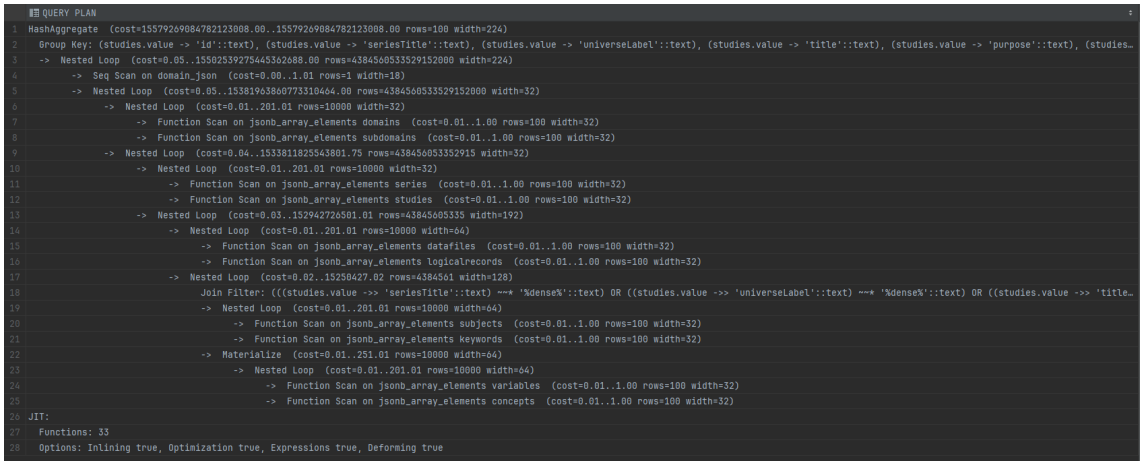
Joonisel 28 luuakse trigram indeksid ja vaadatakse kas tulemus paranes.

```
CREATE EXTENSION pg_trgm;

CREATE INDEX metadata_json_gin_studies_series_title_idx ON domain_json USING
GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'seriesTitle') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_universe_label_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'universeLabel') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_title_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'title') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_summary_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'summary') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_purpose_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'purpose') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_study_code_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'studyCode') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_sector_coverage_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'sectorCoverage') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_other_dissemination_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'otherDissemination') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_documentation_on_methodology_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'documentationOnMethodology') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_geographical_comparability_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'geographicalComparability') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_comparability_over_time_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'comparabilityOverTime') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_source_data_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'sourceData') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_frequency_of_data_collection_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'frequencyOfDataCollection') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_data_collection_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'dataCollection') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_data_validation_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'dataValidation') gin_trgm_ops);
CREATE INDEX metadata_json_gin_studies_data_compilation_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies'->> 'dataCompilation') gin_trgm_ops);
CREATE INDEX metadata_json_gin_data_files_title_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' ->> 'title') gin_trgm_ops);
CREATE INDEX metadata_json_gin_keyword_name_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'keywords'->> 'name') gin_trgm_ops)
        ;
CREATE INDEX metadata_json_gin_subject_name_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'subjects'->> 'name') gin_trgm_ops)
        ;
CREATE INDEX metadata_json_gin_logical_record_name_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords'->> 'name')
        gin_trgm_ops);
CREATE INDEX metadata_json_gin_logical_record_label_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords'->> 'label')
        gin_trgm_ops);
CREATE INDEX metadata_json_gin_logical_record_description_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords'->> 'description')
        gin_trgm_ops);
CREATE INDEX metadata_json_gin_variable_name_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords' -> 'variables' ->>
        'name') gin_trgm_ops);
CREATE INDEX metadata_json_gin_variable_label_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords' -> 'variables' ->>
        'label') gin_trgm_ops);
CREATE INDEX metadata_json_gin_variable_description_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords' -> 'variables' ->>
        'description') gin_trgm_ops);
CREATE INDEX metadata_json_gin_variable_represented_variable_label_idx ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords' -> 'variables' ->>
        'representedVariableLabel') gin_trgm_ops);
CREATE INDEX metadata_json_gin_variable_conceptual_variable_label ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords' -> 'variables' ->>
        'conceptualVariableLabel') gin_trgm_ops);
CREATE INDEX metadata_json_gin_concept_label ON domain_json USING
    GIN ((data->'domains'->'subdomains'->'series'->'studies' -> 'dataFiles' -> 'logicalRecords' -> 'variables' ->
        'concepts' ->> 'label') gin_trgm_ops);
```

Joonis 28. Trigram indeksite loomine töötlemata kujul JSON dokumendiga tabelile.

Pärast trigram indeksite loomist tulemus ei paranenud ning päringu täitmisplaanis (vt. Joonis 29) ei ole näha, et indeksit oleks kasutatud.



Joonis 29. Töötlemata kujul JSON dokumendiga tabelist tehtud päringu täitmisplaan.

Mõistmaks, miks ei anna indeksite kasutamine soovitud tulemust, saab proovida lihtsamaid päringuid ja uurida nende täitmisplaane.

Võtame aluseks lahendusvariandi, kus iga uuring on salvestatud JSONB tüüpi veergu sisaldavasse tabelisse ja teeme päringu ühe välja järgi (vt. joonis 30) millele on tehtud trigram indeks (vt. joonis 31.)

```
SELECT * FROM study_json WHERE data->>'seriesTitle' ILIKE '%
   yahoo%';
```

Joonis 30. Lihtsustatud päring JSONB tüüpi andmete põhjal.

```
CREATE INDEX studies_series_title_idx ON study_json USING GIN ((
   data->>'seriesTitle') gin_trgm_ops);
```

Joonis 31. Trigram indeksi loomine JSONB veerus json väljale.

Lihtsama päringu täitmiseks kasutab andmebaasisüsteem joonisel 32 esitatud tätmisplaani.

```
  Seq Scan on study_json  (cost=0.00..4.03 rows=1 width=34) (
     actual time=84.045..167.037 rows=5 loops=1)
Filter: ((data ->> 'seriesTitle'::text) ~~* '%yahoo%'::text)
Rows Removed by Filter: 130
```

Joonis 32. Lihtsa JSONB tüüpi väärtuse põhjal tehtud päringu täitmisplaan.

Täitmisplaanist on näha, et lause täitmiseks ei kasutatud indeksit. Põhjus võib olla selles, et tabelis ei ole palju ridu (umbes 100), seega andmebaasisüsteem otsustab, et optimaalsem on teha tabeli ridade täielik läbiskaneerimine (*Seq Scan* operatsioon). Indeksi kasutamist saaks proovida suurema hulga andmete peal. Kontseptide arv on uuringute omast palju suurem, seega testimise eesmärgil proovime otsingut ainult nende elementide seast. Kasutame tabelit *Concept*, mis sisaldab JSONB tüüpi veergu (vt. joonis 33). Trigram indeksi loomise lause sellele tabelile on näidatud joonisel 34.

```
CREATE TABLE concept
(
    concept_id UUID PRIMARY KEY,
    concept_data JSONB NOT NULL
);
```

Joonis 33. Kontseptide JSONB tüüpi väärtustena salvestamiseks mõeldud tabel.

```
CREATE INDEX studies_series_title_idx ON study_json USING
   GIN ((data->>'seriesTitle') gin_trgm_ops);
```

Joonis 34. Trigram indeks JSONB tüüpi veerus olevale json väljale.

Proovime andmete otsimist koos ja ilma trigram indeksita (vt. joonis 35).

```
SELECT * FROM concept WHERE concept_data->>'label' ILIKE ''%
   yahoo%;
```

Joonis 35. Kontsepti leidmiseks mõeldud päring.

Ilma indeksita kulus selle täitmiseks 210ms ja koos indeksiga 45ms. Päringu täitmisplaanid joonistel 36 ja 37 näitavad, et indeksi olemasolul seda ka kasutati.

```
Gather  (cost=1000.00..8489.65 rows=119 width=80) (actual
   time=22.180..84.245 rows=20 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    ->  Parallel Seq Scan on concept  (cost=0.00..7477.75
       rows=50 width=80) (actual time=29.816..76.104 rows=7
       loops=3)
        Filter: ((concept_data ->> 'label'::text) ~~* '%
           trader%'::text)
        Rows Removed by Filter: 124193
```

Joonis 36. Kontsepti päringu täitmisplaan ilma trigram indeksita.

```
Bitmap Heap Scan on concept  (cost=60.65..480.84 rows=119
   width=80) (actual time=0.513..0.633 rows=20 loops=1)
    Recheck Cond: ((concept_data ->> 'label'::text) ~~* '%
       trader%'::text)
    Rows Removed by Index Recheck: 5
    Heap Blocks: exact=25
    ->  Bitmap Index Scan on concept_label_idx  (cost
       =0.00..60.62 rows=119 width=0) (actual time
       =0.480..0.482 rows=25 loops=1)
        Index Cond: ((concept_data ->> 'label'::text) ~~* '%
           trader%'::text)
```

Joonis 37. Kontsepti päringu täitmisplaan koos trigram indeksiga.

Sellest saab järeldada, et kui JSONB tüüpi veerus luuakse indeks konkreetsetele json väljadele, siis kasutatakse seda ILIKE operaatorit kasutava päringu täitmiseks. Samas kui indeks luuakse json-väärtuses sisalduvatele listidele, siis indeksit ei kasutata. Indeksi kasutamine ei tule välja *jsonb_array_elements* funktsiooni välja kutsumisel. Samuti, kui tabelis olevate andmete hulk on väike, siis võib andmebaasisüsteem pidada mõistlikumaks tabeli skaneerimist.

Seega ei õnnestunud PostgreSQLis luua efektiivset otsingulahendust JSONB tüüpi veerus olevate andmete põhjal. Samas tasub märkida, et suure andmehulga salvestamine on JSONB-põhise lahenduste korral kiire.

6. Eksperimentide kirjeldus

Erinevate lahendusvariantide sobivuse võrdlemiseks teostatakse neli eksperimenti:

1. otsingute kiirus e jõudlus,
2. andmete salvestamise kiirus,
3. koormustestimine,
4. edasiarendamise lihtsus.

Eksperimendid tehti autori arvutis, mille riistvara näitajad on esitatud tabelis 3.

Tabel 3. Arvuti riistvara näitajad.

Protsessor	AMD Ryzen 7 PRO 4750U with Radeon Graphics, 1700 Mhz, 8 Core(s), 16 Logical Processor(s)
Muutmälu	40GB

Elasticsearch ja PostgreSQL eksemplarid käivitati Dockeri konteineris. Eksperimentides kasutati Elasticsearch versiooni 8.11 ja PostgreSQL versiooni 16. Eksperimentide tulemused võivad erineda sõltuvalt riistvarast, kus neid käivitatakse ja ei pruugi olla samasugused, kui eksperimente teistsuguses masinas jooksutada.

6.1 Testandmete hulk

Testide läbiviimiseks genereeritakse testandmed, mille hulk on võimalikult lähedane sellele, mis on oodatud toodangu keskkonnas. Testandmete kirjutamiseks kasutatakse lähenemist, mida kirjeldatakse alajaotises 5.1.

Tabel 4. Andmete arv testimiseks.

Andmetabel	**Ridade arv testimise tarbeks**
metadata_domain	3
sub_domain	9
series	27
study	135
other_material	405

Jätkub...

Jätkub...

Andmetabel	**Ridade arv testimise tarbeks**
data_file	540
keyword	1620
subject	1620
logical_record	5400
variable	124200
concept	372600
quality_indicator	388800
code_list	100000
category	300000

6.2 Otsingute kiirus

Otsingu kiirus on peamine näitaja mille järgi otsustatakse lahenduse sobivuse üle. Otsingu kiiruse testimiseks realiseeriti Spring vaherakenduses lõpp-punkt, mis vahendab päringut Elasticsearchi või PostgreSQLi andmebaasi, koostab JSON formaadis tulemuse ja tagastab päringu vastuse. Tulemused tagastatakse lehekülgede kaupa ja lisaks otsingutulemustele sisaldab päringu vastus infot selle kohta kui palju on tulemusi kokku, mis lehekülg on hetkel tagastatud ja kui palju lehekülgi on saadaval. Otsinguks kulunud ajaks loetakse aeg, mis kulub HTTP kliendis alates päringu tegemisest kuni vastuse saamiseni.

Päringute testimiseks valiti 13 sõnet, millel on erinev sisaldus testimise jaoks genereeritud andmehulgas. Iga sõnega tehti viis päringut (kokku 65 päringut) ja tulemustest on võetud geomeetriline keskmine. Kasutatakse geomeetrilist keskmist, sest nende mõõtmiste tulemused on üksteisest sõltuvad, kuna operatsiooni korduval täitmisel saab süsteem kasutada eelmise täitmiste tulemusi (koostatud täitmisplaanid, muutmällu loetud andmed) [31].

Tabel 5. Testimiseks valitud test-andmetes sisalduvad sõnad.

Sõna	**Vastete arv test-andmetes**
considering	135
soldier	79
blood	86
StudyTitleTest	27
rasp	61
ginseng	55

Jätkub...

Jätkub...

Sõna	**Vastete arv test-andmetes**
trader	48
abaft	135
atop	135
quirkily	135
diffract	58
bandwidth	135
ferociously	135

6.3 Andmete salvestamise kiirus

Andmete salvestamise kiiruse testimiseks loetakse vajalikku infot JSONi failist. Seejärel mõõdetakse kui palju läheb aega, et antud JSON dokumenti vaherakenduses töödelda ja andmehoidlasse salvestada. Andmete hulk salvestamisel on sama, mida kasutatakse otsingukiiruse mõõtmiseks. Eksperimendi käigus käivitatakse salvestamist kümme korda ning täitmise aegadest võetakse geomeetriline keskmine. Testimise jaoks genereeritud andmeid sisaldava JSON dokumendi suurus on 380 MB.

6.4 Koormustestimine

Koormustestimisel simuleeritakse olukorda, kus 100 kasutajat proovivad ühesekundiliste vahedega teha otsinguteenuses päringuid. Iga kasutaja teeb kokku 10 päringut. Koormustestimise jaoks kasutatakse Apache Jmeter tööriista.

Koormustestimise tulemuse näitajad on järgmised.

1. Läbilaskevõime (*throughput*) näitab kui suure koormusega saab server hakkama. Teiste sõnadega, kui mitu päringut suudab see teatud aja jooksul töödelda ja täita. Mida suurem läbilaskevõime, seda parem.
2. Hälve (*deviation*) näitab kõrvalekallet keskmisest päringu täitmise kiirusest. Mida väiksem on hälve, seda parem.
3. Keskmine näitab päringu täitmise keskmist kiirust.
4. Mediaan näitab päringu täitmise kiiruse mediaani.

6.5 Edasiarendamise lihtsus

Edasiarenduse lihtsuse hindamiseks saab loetleda tegevuste arvu (või hinnata tegevuste keerukust), mis ühe või teise lisafunktsionaalsuse lisamise korral peab Elasticsearch ja PostgreSQL realisatsioonides sooritama. Mida väiksem on vajalik tegevuste arv (või mida väiksem on nende keerukus), seda lihtsam on edasiarendus. Üks selline edasiarenduse võimalus on hägus otsing, mida proovitakse mõlema lahenduse puhul realiseerida ning hinnatakse selle keerukust.

7. Eksperimentide tulemused

Selles peatükis esitatakse eksperimentides saadud tulemused. Kood, mis loodi eksperimentide läbiviimiseks, on kättesaadav siit: `https://github.com/borzah/full-text-search-comparison`

7.1 Otsingute kiirus

Tabel 6. Otsingu keskmised kiirused.

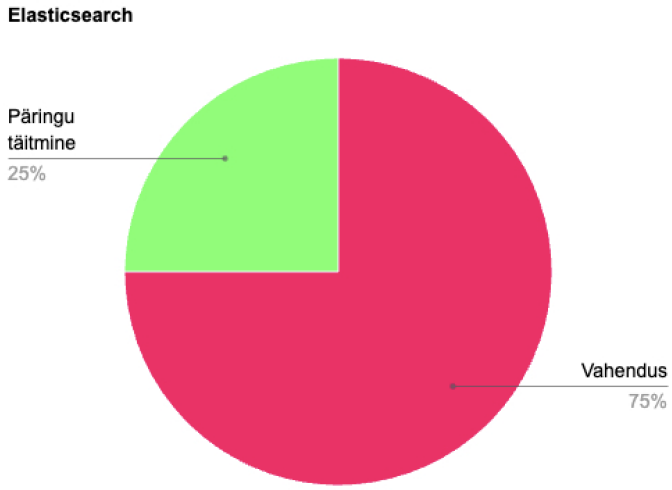
Realisatsioon	Elasticsearch	PostgreSQL+tsvector	PostgreSQL+jsonb
Aeg (ms)	144.43	46.85	84241.68

Asjaolu, et PostgreSQL lahendus otsingute puhul kiiremaks osutus, on üllatav (vt. Tabel 6). Töökiiruse korral saab vaadelda eraldi kahe komponendi töökiirust:

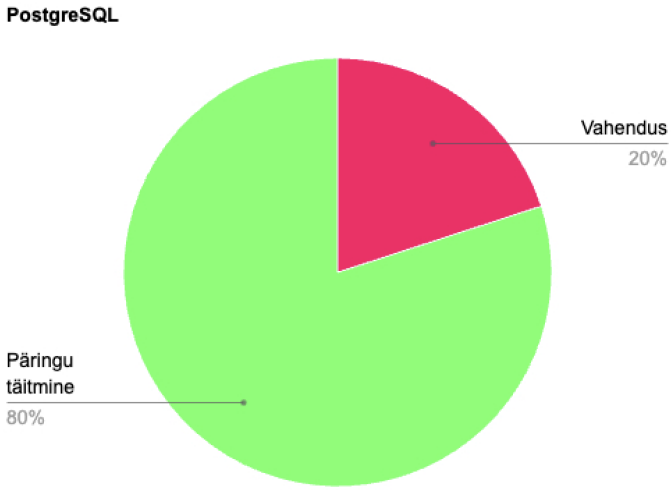
1. kui palju läheb aega vaherakendusel, et otsingumootorilt vastust küsida ja esitada
2. kui palju läheb aega otsingumootoril, et päringu tulemus valmis saada.

Elasticsearchi puhul tagastatakse päringu vastuses ka tulemuse koostamiseks kulunud aeg millisekundites. PostgreSQLi puhul saab kasutada täitmisplaani vaatamiseks mõeldud EXPLAIN lauses ANALYZE klauslit, mis annab infot, kui palju läks aega SQL lause planeerimiseks ja täitmiseks.

Eksperimenti tulemusi vaadates on näha, et Elasticsearchi puhul on otsingumootori poolt päringu täitmise kiirus umbes 25 protsenti koguajast (vt. Joonis 38). PostgreSQL puhul on otsingu ja kokkulugemise päringute täitmine umbes 80 protsenti koguajast (vt. Joonis 39). Järelikult Elasticsearchi puhul kulub enamik ajast vaherakenduse poolt lause vahendamisele.



Joonis 38. Elasticsearh realisatsiooni päringu täitmiseks kuluva aja jaotus.



Joonis 39. PostgreSQL realisatsiooni päringu täitmiseks kuluva aja jaotus.

Tasub välja tuua, et kui vaadelda päringu täitmise kiirusi, siis varieeruvad PostgreSQLi tulemused rohkem kui Elasticsearchi omad. PostgreSQLi tulemuste hulka kuuluvad sõltuvalt kasutatud otsingusõnest 3ms, 5ms, 7ms, 37-100ms ja 300-400ms . Elasticsearchi puhul jääb tulemus alati 18-60ms vahemikku. Tabelis 7 on näidatud andmebaasisüsteemi/otsingumootori poolt päringu täitmise kiiruste geomeetrilised keskmised.

Tabel 7. Päringu täitmise kiiruste geomeetrilised keskmised.

Realisatsioon	Elasticsearch	PostgreSQL+tsvector
Aeg (ms)	35.66	24

Tulemus näitab, et PostgreSQLi lahenduse puhul on võimalik antud otsingut teostada kiiremini. Põhjuseks on päringu suurem täitmise kiirus andmebaasisüsteemi poolt. Tõenäoliselt mängib rolli ka andmevahetuse viis, mille abil vaherakendusega suheldakse ning sõnumivahetuses osalevate infokildude suurus. Elasticsearchi puhul on andmeva-

hetuse viis REST API, mis kasutab HTTP-d ning PostgreSQL kasutab TCP/IP põhist PostgreSQL Frontend/Backend protokolli. Tõenäoliselt kulutab Elasticsearch palju aega, et suuremahulisi JSONit sisaldavaid HTTP sõnumeid vahetada, samal ajal kui PostgreSQL saab enda protokolli kasutades selle ülesandega kiiremini hakkama.

Otsingu kiiruse mõõtmiste tulemusena on näha, et antud PostgreSQL JSONB-põhine lahendus ei sobi suure hulga salvestatud andmete korral, kuna otsingu kiirus ei ole sellisel juhul vastuvõetav.

Tekstivektoritel põhineva PostgreSQL disaini ja Elasticsearch disaini otsingukiiruste põhjalikuma võrdluse graafikud on näidatud lisades 4, 5 ja 6.

7.2 Salvestamise kiirus

Tabelist 8 on näha, et salvestamine võttis kõige rohkem aega PostgreSQL+tsvector realisatsiooni korral, kuna lisaks suure hulga andmete tabelitesse lisamisele on vaja ka programselt koostada sõned, mis lähevad otsingutabelisse lisamisel *to_tsvector* funktsiooni argumendiks ning sooritada suuremahulised INSERT laused. On näha, et võrreldes teiste realisatsioonidega on JSONB tüüpi veeruga variandi korral salvestamine kõige kiirem.

Tabel 8. Salvestamise keskmised kiirused.

Realisatsioon	Elasticsearch	PostgreSQL+tsvector	PostgreSQL+jsonb
Aeg (ms)	34582.03	65080.7	14580.3

Tekstivektoritel põhineva PostgreSQL disaini ja Elasticsearch disaini otsingukiiruste põhjalikuma võrdluse graafikud on näidatud lisades 7, 8 ja 9.

7.3 Koormustestimine

Elasticseach realisatsioon (vt tabel 9).

Tabel 9. Elasticsearch realisatsiooni koormustestimise tulemused.

Läbilaskevõime (req/min)	Hälve	Keskmine	Mediaan
550.126	156	153	133

PostgreSQL + tsvector realisatsioon (vt tabel 10).

Tabel 10. PostgreSQL + tsvector realisatsiooni koormustestimise tulemused.

Läbilaskevõime (req/min)	Hälve	Keskmine	Mediaan
605.388	2	12	11

PostgreSQL + JSONB realisatsioon (vt tabel 11).

Tabel 11. PostgreSQL + JSONB realisatsiooni koormustestimise tulemused.

Läbilaskevõime (req/min)	Hälve	Keskmine	Mediaan
51.62	35606	37474	30015

Koormustestimisel keskenduti kahele näitajale: läbilaskevõime ja hälve (vt tabel 12).

Tabel 12. Koormustestimise koondtulemused.

Realisatsioon	Elasticsearch	PostgreSQL+tsvector	PostgreSQL+jsonb
Läbilaskevõime (req/min)	550.126	605.388	51.62
Hälve	156	2	35606

Nii Elasticseach kui ka PostgreSQL+tsvector realisatsioonid taluvad koormust hästi. JSONB variant ei saa koormusega hakkama, sest ühe päringu täitmine võtab palju aega.

7.4 Edasiarendamise lihtsus

Juhul kui muutuvad otsitavad andmeväljad, siis on Elasticsearhi puhul vaja täiendada indeksi loomise lauset (et uued vajalikud andmeväljad saaks indekseeritud) ning täiendada päringut. PostgreSQL variandi puhul on vaja muuta vaherakenduse koodi, mis andmeid eeltöötleb ja otsingu tabelisse sisestab. Kuna PostgreSQL otsingus rakendatakse otsingutingimus ühele tekstivektori veerule, milles olevasse väärtusesse on kõik huvipakkuvad sõned kokku ühendatud, siis otsingut muutma ei pea. Mõlemal juhul on need lihtsad täiendused.

Filtrite lisamine on kergem Elasticsearchi puhul, sest nõuab ainult väikesemahulist päringu täiendamist. PostgreSQLi puhul ei ole keeruline lisada filtreid, mis kontrollivad veerus oleva väärtuse võrdumist etteantud väärtusega. Samas näiteks alamelementide ajalise kaetuse filtri puhul on see keeruline. Uuringu alamelemendis - andmefailis - on olemas väljad alguskuupäev ja lõpp-kuupäev. Võib tekkida vajadus otsida uuringuid, mille alamelementide seas olevad andmefailid on kindlast ajavahemikus. Sellisel juhul peab kokku ühendama otsingu jaoks eeltöödeldud tabeli ja andmefailide andmeid sisaldavat tabeli, mis võib mõjutada jõudlust. Seda oli näha baastabelites sisalduvate tekstivektorite variandis, kus pidi päringus suurt hulka tabeleid kokku ühendama ning see mõjutas päringu täitmise

kiirust. Seega oleks ajalise kaetuse filtrite lisamine PostgreSQLi keerulisem probleem ja võib nõuda andmete eeltöötlemise täiendamist või tuleb otsingu tarbeks loodud tabelite struktuuri muuta. Elasticsearchi puhul on see üks lisaklausel päringus.

7.4.1 Hägus otsing Elasticsearchis ja PostgreSQLis

Üks tõenäoline täiendus, mis võidakse süsteemis tulevikus teha, on hägusa otsingu võimaluse lisamine. Näiteks kui tahetakse, et otsingusõne "datab" tooks tulemused, kus on olemas sõna "database". Või kui on soov, et vigadega kirjutatud sõna "bangwidht" annaks tulemusi kohtades, kus sisaldub sõna "bandwidth".

Pooliku sõneosa järgi otsimiseks saab Elasticsearchis teha *wildcard* päringut (vt. Joonis 40). Lisaks sellele pakutakse erinevaid indeksi seadistusi, näiteks n-grammi kasutamine analüsaatoris [32].

```
GET /_search
{
    "query": {
        "wildcard": {
            "fieldName": {
                "value": "datab*"
            }
        }
    }
}
```

Joonis 40. Sõneosa järgi otsing Elasticsearchis.

Siinkohal saab * märkide alusel näidata, kas otsime vastet sõna alguses (datab*), keskel (*datab*) või lõpus (*datab), mis on sarnane "%" märgiga PostgreSQLi LIKE ja ILIKE klauslite abil otsingu puhul.

Selleks, et arvestada otsingu sõna kirjavigu või ebatäpset esitamist on Elasticsearchi puhul olemas mitmeid võimalusi [33]. Kõige lihtsam nendest, mille abil saab olemasolevat uuringute otsingut laiendada, on päringu parameeter *fuzziness*, mille väärtuse muutmine võimaldab realiseerida hägusat otsingut (vt. joonis 41). Selle parameetri võimalikud väärtused on: 0, 1, 2 ja AUTO.

- 0 tähendab, et hägusust ei ole, otsitakse ainult täisvasteid.
- 1 tähendab, et vasteid leitakse, kui otsingusõne ja vaste vahel on ühe tähemärgiline erinevus.
- 2 tähendab, et vasteid leitakse, kui otsingusõne ja vaste vahel on kahe tähemärgiline erinevus.
- AUTO puhul määratakse hägusus vastavalt otsingusõne pikkusele. Kui see on 2-5 tähemärki, siis määratakse hägususeks 1. Pikemate otsingusõnede puhul määratakse hägususeks 2.

Elasticsearchi puhul on hägusa otsingu kiirused samad, mis tavalise otsingu puhul: 100-300ms.

```
GET /studies/_search
{
    "query": {
        "multi_match": {
            "query": "bis",
            "fuzziness": "AUTO"
        }
    }
}
```

Joonis 41. Hägus otsing Elasticsearchis.

Seega Elasticsearchi puhul on hägusa otsingu lisamine suhteliselt lihtne ja nõuab ainult ühte sammu - otsingupäringu täiendamist hägususe klausliga.

PostgreSQLi puhul ei oleks pooliku sõneosa järgi otsingu puhul abi tesktivektoritest, kuna need on loodud sõnade, mitte sõnaosade vastete leidmiseks. Siinkohal oleks abiks trigram indeks ja üks võimalus sellise realisatsiooni loomiseks on kasutada eeldtöödeldud andmetega tabelites tesktivektori tüüpi veeru asemel TEXT tüüpi veergu, kus oleks vajalike sõnede kokkuliitmise tulemus. Sellele veerule saaks luua trigram indeksi (vt. joonis 42).

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;

CREATE TABLE study_search_document_store
(
    study_id UUID PRIMARY KEY,
    series_title TEXT,
    universe_label TEXT NOT NULL,
    title TEXT,
    purpose TEXT,
    reference_area TEXT,
    time_coverage TEXT,
    study_search_document TEXT,
    CONSTRAINT study_search_document_store_study_fk FOREIGN KEY (study_id) REFERENCES
        study (study_id)
);

CREATE INDEX idx_study_search_document_store_study_search_document ON
    study_search_document_store USING gin((study_search_document) gin_trgm_ops);

CREATE TABLE variable_search_document_store
(
    variable_id UUID PRIMARY KEY,
    study_id UUID NOT NULL,
    logical_record_id UUID NOT NULL,
    name TEXT,
    label TEXT,
    representation_type TEXT,
    variable_search_document TEXT,
    CONSTRAINT variable_search_document_study_fk FOREIGN KEY (study_id) REFERENCES study
         (study_id)
);

CREATE INDEX idx_variable_search_document_store_study_id ON
    variable_search_document_store (study_id);
CREATE INDEX idx_variable_search_document_store_variable_search_document ON
    variable_search_document_store USING gin((variable_search_document) gin_trgm_ops);
```

Joonis 42. Tabelite loomine PostgreSQLis hägusa otsingu jaoks.

Otsing oleks sellisel juhul sarnane tekstivektori variandiga (vt. joonis 43) kasutades ILIKE predikaati.

```
WITH variables_match AS (
    SELECT variable_id AS variable_id,
           study_id AS study_id,
           logical_record_id AS logical_record_id,
           label  AS variable_label,
           name AS variable_name,
           representation_type AS representation_type
    FROM variable_search_document_store
    WHERE variable_search_document ILIKE :search
),
     variables_selected AS (
         SELECT variable_id,
                study_id,
                logical_record_id,
                variable_label,
                variable_name,
                representation_type
         FROM variables_match
     )
SELECT  study_id as study_id,
        series_title AS series_title,
        universe_label AS universe_label,
        title AS title,
        purpose AS purpose,
        reference_area AS reference_area,
        time_coverage AS time_coverage,
        json_agg(jsonb_build_object(
            'variable', variable_id,
            'logicalRecordId', logical_record_id,
            'variableLabel', variable_label,
            'variableName', variable_name,
            'representationType', representation_type)) FILTER ( WHERE variable_id IS
                NOT NULL ) AS variables
FROM study_search_document_store LEFT JOIN variables_selected USING (study_id)
WHERE study_search_document ILIKE :search
GROUP BY study_id, series_title, universe_label, title, purpose, reference_area,
    time_coverage
OFFSET 0
FETCH NEXT 10 ROWS ONLY;
```

Joonis 43. Hägusa otsingu võimalus PostgreSQLis.

Antud päringuga leitakse otsingu tulemused kiirusega 300ms-1.5s ja siin ei ole veel lahendatud relevantsuse ning otsingusõne esiletõstmise funktsionaalsust. Seega erinevalt Elasticsearchist ei ole PostgreSQLi puhul sõneosa järgi otsingu lisamine ühe sammu küsimus.

Arvestamaks kirjavigu otsingusõnes või selle ebatäpsust pakub PostgreSQL erinevaid lähenemisi [34]. Sõnade sarnasusi on võimalik leida Soundex, Metaphone ja Levenshteini algoritmide alusel. Funktsioonid nende algoritmide järgi sarnasuste leidmiseks on PostgreSQLis olemas - selleks tuleb andmebaasi lisada CREATE EXTENSION lausega laiendus *fuzzystrmatch*. Antud PostgreSQL lahenduse puhul tähendaks sellise otsingu võimaluse lisamine ulatusliku lisatööd või ümbertegemist, sest hetkel on iga otsitava elemendi kohta (uuring ja muutuja) salvestatud üks suur tekstivektor, mis sellisena ei võimalda hägusa otsingu algoritme ära kasutada.

Üks võimalik lahendusviis oleks järgmine.

1. Lisaks sellele, et iga elemendi juurde salvestatakse selle ning selle alamelementide sõnedest moodustatud tekstivektor, salvestatakse iga elemendi kohta ka nimekiri unikaalsetest sõnadest kas eraldi tablisse, või näiteks samasse tabelisse JSONB veergu json listina.
2. Otsingu korral leitakse esiteks unikaalsete sõnade seast otsingusõnele kõige lähedasem sõna Levenshtein algoritmi abil (vt. Joonis 44).
3. Leitud sõnega sooritatakse varem kirjeldatud otsing kasutades tekstivektorit.

```
SELECT * FROM unique_lexeme WHERE levenshtein_less_equal(unique
   word, 'otsing', 2) < 2;
```

Joonis 44. Vigadega kirjutatud otsingusõnele kõige lähedasema vaste leidmine tekstivektori päringus kasutamiseks.

See lahenduskäik ei ole realiseerimise kaudu läbiproovitud, kuid see kirjeldus näitab võimalikke samme, mida saaks hägusa otsingu lisamise eesmärgi täitmise suunas ette võtta. Samuti on näha, et sammude arv, mida tuleb selle jaoks teha, on suurem kui Elasticsearchi lahenduse korral.

Kokkuvõttes näeme, et hägusa või sõne järgi otsingu lisamiseks olemasolevatesse lahendustesse vajab PostgreSQL disain rohkem samme kui Elasticsearchi oma. Sellega loeme viimast edasiarenduse mõttes lihtsamaks.

7.5 Andmemahud

Kuigi andmemahte alternatiivide vahel valiku tegemisel kriteeriumina ei arvestata, siis huvi pärast uuriti ka seda, kui palju salvestusruumi võtab kummaski tööriistas otsingu tarbeks andmete hoiustamine. Elasticsearchi puhul on taolist statistikat tagastav päring näidatud joonisel 45 ja PostgreSQLi puhul joonisel 46.

```
GET _cat/indices?v
```

Joonis 45. Elasticsearchi päring, mis tagastab statistikat salvestatud indeksite kohta.

```
SELECT pg_size_pretty(sum(pg_relation_size(quote_ident(
   schemaname) || '.' || quote_ident(tablename)))::bigint) FROM
   pg_tables
WHERE schemaname = 'schemaName'
```

Joonis 46. PostgreSQL päring, mis tagastab kui palju salvestusruumi kasutab konkreetne andmebaasi skeem.

Tabel 13. Salvestusruumi kasutus andmete hoiustamiseks.

Realisatsioon	Elasticsearch	PostgreSQL+tsvector	PostgreSQL+jsonb
Mälu (MB)	201	238	0.024

Tasub märkida, et JSONB lahendus suudab suhteliselt vähese salvestusruumi kasutusega terve metaandmete JSON dokumendis sisalduvad struktuuri ühte tabeli ritta salvestada.

7.6 Võrdlus olemasolevate tulemustega

Tudor Golubenco artiklis *Full-text search engine with PostgreSQL (part 2): Postgres vs Elasticsearch* [7] võrreldakse muuhulgas Elasticsearchi ja PostgreSQLi täistekstotsingu päringute jõudlust. Testimine käib antud tööga võrreldes lihtsama struktuuriga (üks baastabel üheksa andmeväljaga) kuid suurema hulga andmete peal (2.3 miljonit elementi). Seal esitatud tulemustes on Elasticsearch otsingute tegemisel kiirem. Kohati on PostgreSQL kriitiliselt aeglasem (näiteks Elasticsearchi kõige halvem päringu täitmisaeg 67ms *vs* PostgreSQL kõige parem päringu täitmisaeg 8267ms).

Töös *Comparing Data Store Performance for Full-Text Search: to SQL or to NoSQL?* [35], võrreldakse PostgreSQL, MongoDB, Solr ja Elasticsearch erinevate otsingu ja andmete sisestamise operatsioonide täitmise kiiruseid. Samuti nagu käesolevas töös on seal mõõdetud ja võrreldud täistkeskotsingu ja andmete salvestamise kiiruseid. Erinevalt käesolevast tööst otsiti täistekstotsingu korral vasteid fraaside järgi, mitte üksikute sõnade järgi. Andmemaht, mille põhjal uurimistöö teostati, oli 0.5-3 miljonit kirjet. Kasutati kahte

andmestikku. Otsingu korral olid Elasticsearchi tulemused 5.1 sekundit kuni 18.6 sekundit ning PostgreSQL tulemused 155 sekundit kuni 2766 sekundit. Andmete salvestamise korral olid PostgreSQLi tulemused 70 sekundit kuni 1000 sekundit, Elasticsearchi tulemused 500 sekundit kuni 4000 sekundit.

Töös *Comparing Databases for Inserting and Querying Jsons for Big Data* [36] on võrreldud JSON data sisestamist ja otsimist erinevate otsingu- ja andmebaasisüsteemide puhul suurandmete (*Big Data*) võtmes. Muuhulgas võrreldakse Elasticsearchi ja PostgreSQLi. Testid on tehtud andmestiku peal, mis sisaldab 839122 JSON dokumenti ning mille suurus on 5 GB. Tasub märkida, et töös on PostgreSQLi puhul kasutatud andmetüüpi JSON, mitte JSONB, mida on kasutatud käesoleva töö puhul. Andmete otsimisele keskmiselt kulunud aeg oli PostgreSQLi puhul 20 sekundit ning Elasticsearchi puhul 0.35 sekundit. Andmete salvestamisel kulus PostgreSQLil keskmiselt 8000 sekundit ja Elasticsearchil 18000 sekundit.

Nagu näha, siis võrreldavates töödes näitas Elasticsearch nii andmete otsimisel kui salvestamisel paremaid tulemusi. Käesolevas töös ei ole tegemist nii suurte andmehulkadega nagu nendes katsetustes kasutati. Peamine otsingu väljakutse on suure arvu mitmetasemeliste alamelementide seast vastete otsimine ja osade vastet saanud alamelementide tulemuses näitamine. Sellega sai parema tulemuse PostgreSQLi kasutades.

8. Parima alternatiivi valimine

Nagu on eelnevast näha, siis PostgreSQLis saab täistekstotsingu realiseerimiseks kasutada erinevaid disainilahendusi. Eksperimentide tulemused näitasid, et osad nendest on nii halva päringute täitmise töökiirusega, et neid ei tasu parima alternatiivi valikusse kaasata. Parim alternatiiv valitakse Elasticseach ja PostgreSQL + otsingu tarbeks eeltöödeldud andmeid sisaldavad *tsvector* tüüpi veergudega baastabelid realisatsioonide seast. JSONB-põhine variant jääb valikust välja kuna selle puhul ei ole otsingute kiirused vastuvõetavad.

Parima lahenduse valimiseks kasutatakse analüütiliste hierarhiate meetodit e Saaty meetodit. Meetodi poolt nõutud arvutuste läbiviimiseks kasutatakse vahendit *123ahp* [37].

8.1 Kriteeriumite valik

Kriteeriumid on järgmised.

1. Otsingute kiirus
   - Mugavaks otsinguks ja positiivse kasutajakogemuse pakkumiseks on vaja, et otsing töötaks võimalikult kiiresti.
2. Realiseerimise lihtsus
   - Selleks, et arendusega vähese ajakuluga hakkama saada, on oluline valida lahendusvariant, mille ülesehitamine ei võta palju aega.
3. Edasiarenduse lihtsus
   - Otsingu aluseks olevate andmete struktuur ei muutu tõenäoliselt ajas palju. Küll aga on võimalik, et tulevikus nõuded otsingu funktsionaalsusele täienevad ning tuleb lisada uusi filtreid või suurendada/vähendada otsingu aluseks olevate andmeväljade hulka. Näiteks on võimalik, et lisatakse hägusa otsingu tegemise võimalus. Tarkvarakomponendi edasiarendus ja laiendamine peaks olema mugav ning ei tohiks nõuda ulatuslikku ümbertegemist.
4. Koormustaluvus
   - Otsingu kasutajate hulk ei ole suur, kokku mõnikümmend inimest. Kuid süsteem peaks olema valmis ka suurematekskoormusteks.
5. Andmete salvestamise kiirus
   - Andmete uuendamine toimub üks kord päevas ja see toimub öisel ajal, kui süsteemi aktiivse kasutuse tõenäosus on väike. Vaatamata sellele on salvestamise kiirus üks näitaja ja seega ka valikukriteerium.

Kriteeriumid, mille kasutamist analüüsi käigus kaaluti, kuid mis jäid alternatiivide vahel valiku tegemisest lõpuks välja on järgmised.

1. Tarkvara hind
   - Antud kontekstis pole valikkuse sattunud ükski tasulist litsentsi nõudev tööriist. Kõik alternatiivid on tasuta lahendused.
2. Tarkvara lähtekoodi avalikkus
   - Antud juhul on mõlemad süsteemid (nii Elasticsearch kui PostgreSQL) avatud lähtekoodiga.
3. Andmemahud.
   - Süsteemis salvestatavate metaandmete hulk on väike.

8.2 Kriteeriumite suhteline olulisus

Kriteeriumite suhtelist olulisust kirjeldab tabel 14.

Tabel 14. Kriteeriumite suhteline olulisus.

	Otsingute kiirus	Realiseerimise lihtsus	Edasiarenduse lihtsus	Koormustaluvus	Andmete salvestamise kiirus
Otsingute kiirus	1	5	5	7	9
Realiseerimise lihtsus	1/5	1	2	3	3
Edasiarenduse lihtsus	1/5	1/2	1	3	3
Koormustaluvus	1/7	1/3	1/3	1	3
Andmete salvestamise kiirus	1/9	1/3	1/3	1/3	1

Tabel 15. Kriteeriumite kaalud.

Kriteeriumid	**Kaalud**
Otsingute kiirus	0.5757
Realiseerimise lihtsus	0.1732
Edasiarenduse lihtsus	0.1316
Koormustaluvus	0.0747
Andmete salvestamise kiirus	0.0448

Antud kontekstis on kõige olulisemad kriteeriumid otsingu kiirus ning seejärel realiseerimise ja edasiarenduse lihtsus. Koormustaluvus on tähtsuselt teistest allpool - seda põhjusel, et antud süsteemi ei ole oodata suur hulk aktiivseid kasutajaid. Andmete salvestamise kiirus ei ole käesolevas olukorras kuigi oluline, kuna andmete uuendamine toimub harva.

8.3 Alternatiivide omavaheline võrdlus

Selles jaotises võrreldakse omavahel alternatiive valitud kriteeriumite suhtes.

8.3.1 Otsingute kiirus

Nii Elasticsearch kui ka PostgreSQL realisatsioonide korral on otsingute täitmised kiired. Keskmised näitajad olid vastavalt 147.63 ms ja 79ms. PostgreSQL lahenduse keskmine kiirus on Elasticsearchi omast 1.83 korda väiksem, mis annab suure paremuse. Võrdlust kirjeldab tabel 16.

Tabel 16. Alternatiivide võrdlus otsingu kiiruse suhtes.

	Elasticsearch	PostgreSQL
Elasticsearch	1	1/5
PostgreSQL	5	1

8.3.2 Realiseerimise lihtsus

Elasticsearchil põhinev realisatsioon on lihtsam, kuna Elasticsearchi ja vastava Java teegi õpetusi järgides kujunes koheselt välja töötav ja vastuvõetav lahendus. PostgreSQLi puhul oli vaja proovida ja katsetada erinevaid lähenemisi, enne, kui vastuvõetava lahenduseni jõuti. See annab Elasticsearch variandile suure paremuse PostgreSQLi. Võrdlust kirjeldab tabel 17.

Tabel 17. Alternatiivide võrdlus realiseerimise lihtsuse suhtes.

	Elasticsearch	PostgreSQL
Elasticsearch	1	5
PostgreSQL	1/5	1

8.3.3 Edasiarenduse lihtsus

Tulevikus võivad muutuda andmeväljad, mille järgi otsingut teostada ja lisanduda piiranguid e filtreid. Elasticsearchi puhul on see tehtav lihtsa indeksi loomise lause täiendamisega või filtrite puhul lisaklausliga otsingupäringu JSON esituses. Käesoleva PostgreSQL variandi puhul tuleks muuta vaherakenduse loogikat, mis otsinguks vajalike sõnesid üheks

tekstivektoriks kokku paneb. Filtrite puhul tuleks täiendada otsingutabelit lisaveergudega ja vajadusel ühendada päringus juurde tabeleid (näiteks kui tekib vajadus otsida uuringute all olevate olemite ajavahemike järgi). Autori arvates on Elasticsearch variandi edasiarendus lihtsam ja see annab antud kriteeriumi võrdluses suure paremuse. Võrdlust kirjeldab tabel 18.

Tabel 18. Alternatiivide võrdlus edasiarenduse lihtsuse suhtes.

	Elasticsearch	PostgreSQL
Elasticsearch	1	5
PostgreSQL	1/5	1

8.3.4 Koormustaluvus

Koormustaluvuse näitajad on mõlema lahenduse puhul lähedased. Elasticseachi hälve on PostgreSQLi omast märgatavalt suurem. Autori arvates annab see PostgreSQL lahendusele nõrga paremuse Elasticsearch ees. Võrdlust kirjeldab tabel 19.

Tabel 19. Alternatiivide võrdlus koormustaluvuse suhtes.

	Elasticsearch	PostgreSQL
Elasticsearch	1	1/2
PostgreSQL	2	1

8.3.5 Andmete salvestamise kiirus

Andmete salvestamise puhul on Elasticseach variandi tulemus 1.87 korda kiirem kui PostgreSQLi puhul. See annab suure paremuse. Võrdlust kirjeldab tabel 20.

Tabel 20. Alternatiivide võrdlus andmete salvestamise kiiruse suhtes.

	Elasticsearch	PostgreSQL
Elasticsearch	1	5
PostgreSQL	1/5	1

8.4 Tulemuste analüüs

Pärast alternatiivide omavahelist võrdlust tekib igale alternatiivile arvuline näitaja. Mida suurem on näitaja, seda suurem on sobivus.

Tabel 21. Alternatiivide kaalud.

	Otsingute kiirus	Realiseerimise lihtsus	Edasiarenduse lihtsus	Koormustaluvus	Andmete salvestamise kiirus
Elasticsearch	0.0960	0.1443	0.1097	0.0249	0.0373
PostgreSQL	0.4797	0.0289	0.0219	0.0498	0.0075

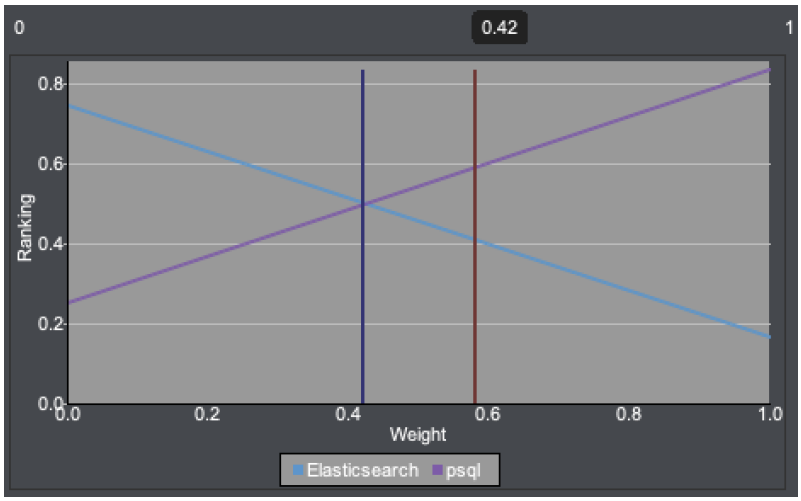
Tabel 22. Alternatiivide kaalud kokku.

	Kaalud
Elasticsearch	0.4122
PostgreSQL	0.5878

Valitud kriteeriumite ja nende kaalude järgi osutus antud olukorras parimaks alternatiiviks PostgreSQLil põhinev lahendus. Tasub pöörata tähelepanu sellele, et Elasticsearchil põhineva lahenduse halvemat otsingu kiirust põhjustab vaherakenduse poolt otsingumootori päringu vastuse töötlemine. Kui antud vaherakenduse komponenti õnnestuks optimeerida, siis muutuksid otsingu kiiruse ja koormustaluvuse kriteeriumite kaalud Elasticsearchi kasuks.

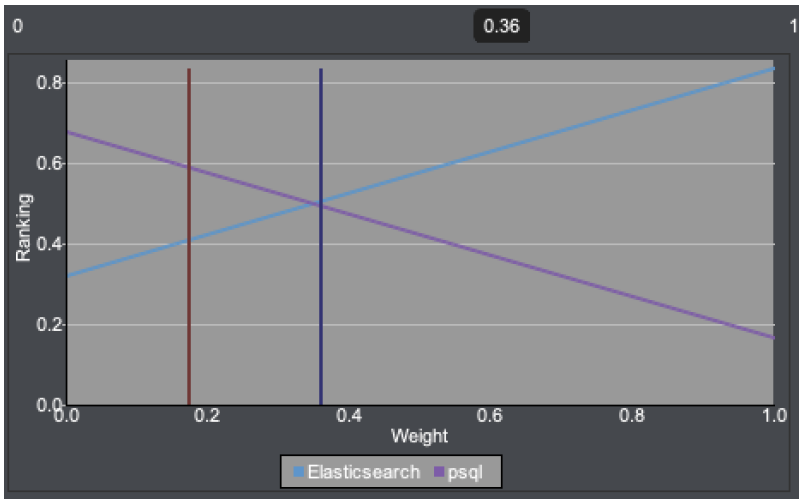
8.5 Tundlikkuse analüüs

Uurimaks, kuivõrd muutub parimaks valitud alternatiiv kriteeriumite kaalude muutumisel, teostame tundlikkuse analüüsi. Tundlikkuse analüüsi läbiviimiseks kasutatakse vahendit Priority Estimation Tool (AHP) [38].



Joonis 47. Otsingu kiiruse kriteeriumi tundlikkuse analüüs.

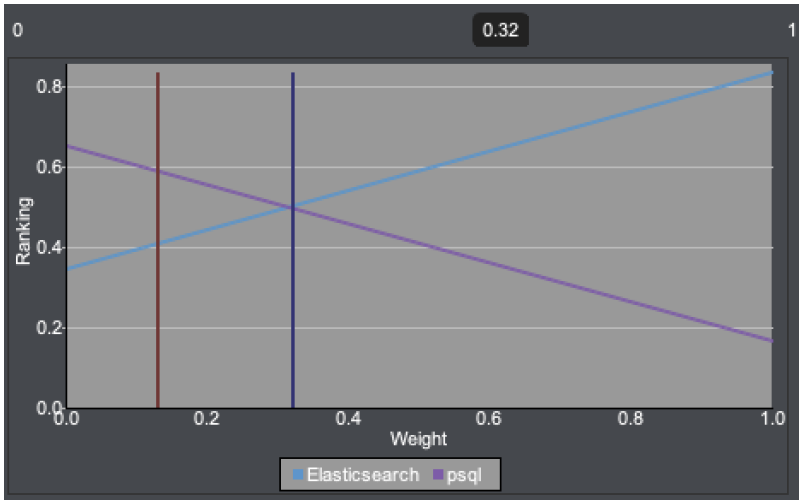
Joonisel 47 on otsingu kiiruse kriteeriumi tundlikkuse analüüsi tulemus. Sellest järeldub, et kui kriteeriumi kaalu vähendada 0.58 pealt 0.42 peale, siis oleks parimaks alternatiiviks valitud Elasticsearch. Kaalude suhe teistesse kaaludesse enne muudatust on 0.58/(1-0.58)=1.38 ja peale muudatust 0.42/(1-0.42)=0.72. Seega otsingu kiiruse kriteeriumi tähtsust teiste suhtes tuleks vähendada keskmiselt 1.38/0.72=1.91 korda. Otsingukiiruse tähtsuse tõttu ei ole tõenäoline, et selle kriteeriumi kaalu langus juhtuda saaks.



Joonis 48. Realiseerimise lihtsuse kriteeriumi tundlikkuse analüüs.

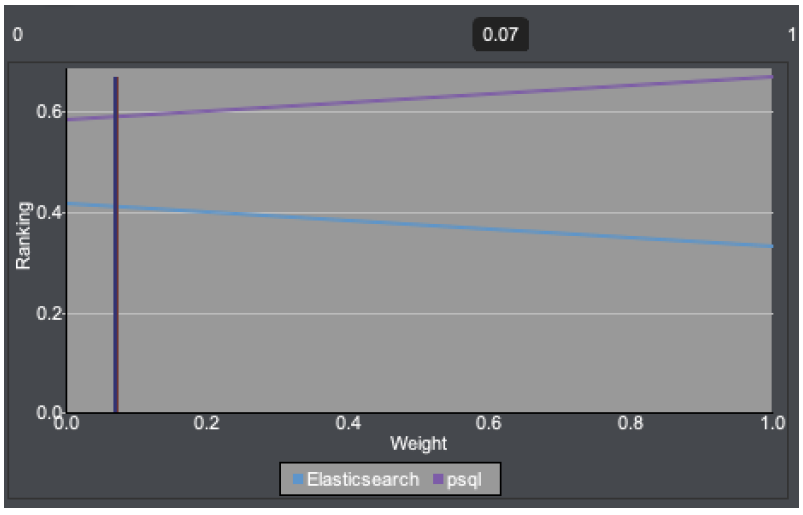
Joonisel 48 on realiseerimise lihtsuse kriteeriumi tundlikkuse analüüsi graafik. Graafikust järeldub, et kui kriteeriumi kaalu suurendada 0.17 pealt 0.36 peale, siis oleks parimaks

alternatiiviks valitud Elasticsearch. Kaalude suhe teistesse kaaludesse enne muudatust on 0.17/(1-0.17)=0.20 ja peale muudatust 0.36/(1-0.36)=0.56. Seega otsingu kiiruse kriteeriumi tähtsust teiste suhtes tuleks suurendada keskmiselt 0.56/0.20=2.8 korda. Autori arvates ei ole selle kriteeriumi puhul selline olulisuse suurendamine mõistlik.



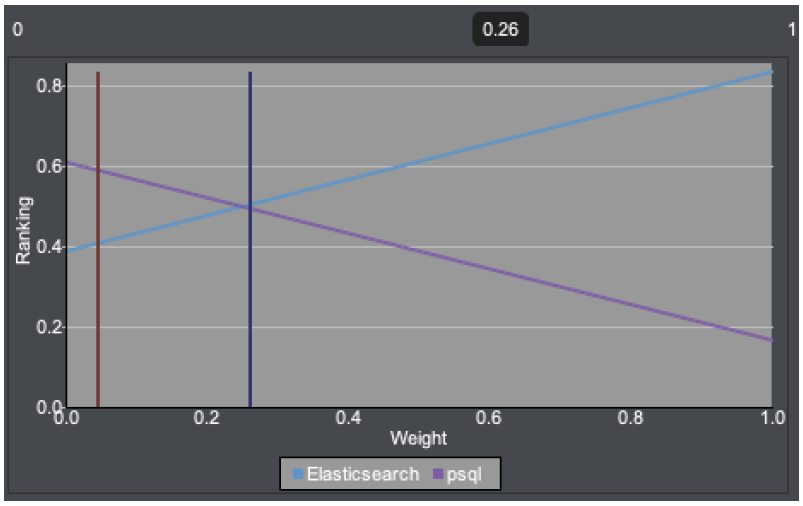
Joonis 49. Edasiarenduse lihtsuse kriteeriumi tundlikkuse analüüs.

Joonisel 49 on edasiarenduse lihtsuse kriteeriumi tundlikkuse analüüsi tulemus. Sellest järeldub, et kui kriteeriumi kaalu suurendada 0.13 pealt 0.32 peale, siis oleks parimaks alternatiiviks valitud Elasticsearch. Kaalude suhe teistesse kaaludesse enne muudatust on 0.13/(1-0.13)=0.15 ja peale muudatust 0.32/(1-0.32)=0.47. Seega otsingu kiiruse kriteeriumi tähtsust teiste suhtes tuleks suurendada keskmiselt 0.47/0.15=3.13 korda. Autori arvates ei ole selline kriteeriumi osatähtsuse suurendamine tõenäoline.



Joonis 50. Koormustaluvuse kriteeriumi tundlikkuse analüüs.

Joonisel 50 on koormustaluvuse kriteeriumi tundlikkuse analüüsi tulemus. Sellest järeldub, et kriteeriumi kaalu muutmine ei mõjuta lõpp-tulemust.



Joonis 51. Andmete salvestamise kiiruse kriteeriumi tundlikkuse analüüs.

Joonisel 51 on andmete salvestamise kiiruse kriteeriumi tundlikkuse analüüsi tulemus. Sellest järeldub, et kui kriteeriumi kaalu suurendada 0.04 pealt 0.26 peale, siis oleks parimaks alternatiiviks valitud Elasticsearch. Kaalude suhe teistesse kaaludesse enne muudatust on 0.04/(1-0.04)=0.04 ja peale muudatust 0.26/(1-0.26)=0.35. Seega otsingu kiiruse kriteeriumi tähtsust teiste suhtes tuleks suurendada keskmiselt 0.35/0.04=8.75 korda. See on ekstreemne kaalu suurendamine. Antud kriteeriumi puhul ei ole selline suurendamine võimalik.

Tundlikkuse analüüs näitas, et kriteeriumite kaalude väike muutumine ei muuda parimaks valitud alternatiivi.

8.6 Soovitused olemasoleva lahenduse parandamiseks

Hüpotees, miks Elasticsearchi puhul päringu vahendus ja töötlemine päringu täitmisega võrreldes nii palju aega võtab (vt. Joonis 38) seisnes selles, et Java rakenduses on Elasticsearchi poolt soovitatud *ElasticsearchClient* klassi asemel kasutatud Spring Data Elasticsearchi teegi klassi *ElasticsearchOperations*. Lähimal uurimisel selgus, et tegelikult kasutatakse *ElasticsearchClient* klassi ka olemasoleva lahenduse puhul - *ElasticsearchOperations* kasutab seda klassi üks tase allpool ning selle abil tehakse kõik REST päringud vastu Elasticsearchi. Seega antud töö tulemusel ei leitud võimalusi, mis võiks Elasticsearch lahendust antud juhul efektiivsemaks teha.

9. Kokkuvõte

Käesolev töö tõukus autori töökohal tehtud valikust realiseerida täistekstotsingut võimaldav süsteem kasutades Elasticsearchi. See süsteem on töö tegemise ajaks realiseeritud ja kasutusel. Autorit jäi painama küsimus, kas oleks võinud kasutada hoopis PostgreSQLi, mis on süsteemi teistes osades kasutusel. Käesoleva bakalaureusetöö eesmärk oli võrrelda Elasticsearch ja PostgreSQL sobivust täistekstotsingu probleemi lahendamiseks konkreetses süsteemis. Konkreetne võrdlus põhines Elasticsearchi versioonil 8.11 ja PostgreSQLi versioonil 16. PostgreSQLis on erinevaid täistekstotsingu realiseerimise võimalusi. Töös tutvustati neid ja tehti nendega katsetusi. Lõpuks valiti üks, mida võrreldi Elasticsearchi-põhise lahendusega kasutades analüütiliste hierarhiate meetodit e Saaty meetodit, mis on mõeldud alternatiivide seast valiku tegemisel subjektiivsuse vähendamiseks. Valitud lahendus põhineb *tsvector* tüüpi veergude kasutamisel spetsiaalselt otsingu jaoks välja töötatud struktuuriga baastabelites. Valitud lahenduste hulka ei kuulunud päringute tegemine PostgreSQL andmebaasis JSONB tüüpi veerus salvestatud andmete põhjal, sest see oli eelnevate katsetuste kohaselt liiga aeglane. Katsetustes kasutatud kood on kättesaadav: `https://github.com/borzah/full-text-search-comparison`.

Töös tehtud mõõtmiste tulemusi võrreldi teiste sarnaste uuringute tulemustega. Võrreldavates töödes näitas Elasticsearch andmete otsimise kiiruse osas paremaid tulemusi kui PostgreSQL. Samas selle töö aluseks olevas süsteemis ei ole tegemist nii suurte andmehulkadega nagu kasutati võrreldavates töödes.

Uuring näitas, et antud olukorras oleks tegelikult parim alternatiiv PostgreSQLi-põhine lahendus. Töö tulemusena ei võeta täistekstotsingu tegemiseks konkreetses süsteemis kasutusele PostgreSQLi, sest loodud tarkvaras ei saa hetkel teha sellist ulatuslikku muudatust. Samas on töös analüüsitud erinevaid võimalusi täistekstotsingu realiseerimiseks, millega saadud teadmisi saab kasutada muude probleemide lahendamisel. Kindel on see, et täistekstotsingu tööriista peaks valima lähtudes konkreetse probleemi spetsiifikast, mitte tehes kiirustades järeldusi üldlevinud arvamuste alusel.

Kasutatud kirjandus

[1] *Elasticsearch*, [Accessed: 18-12-2023]. [Online]. Available: https://www.elastic.co/elasticsearch.

[2] *Postgresql*, [Accessed: 18-12-2023]. [Online]. Available: https://www.postgresql.org/.

[3] *Db-engines ranking*, [Accessed: 18-12-2023]. [Online]. Available: https://db-engines.com/en/ranking.

[4] M. Chhabra, *Elasticsearch postgresql comparison: 6 critical differences*, [Accessed: 18-12-2023], 2021. [Online]. Available: https://hevodata.com/learn/elasticsearch-postgresql/.

[5] *When does it become necessary to use elastic search*, [Accessed: 18-12-2023]. [Online]. Available: https://www.quora.com/When-does-it-become-necessary-to-use-Elastic-Search-currently-using-PostgreSQL.

[6] *Postgresql full text search documentation*, [Accessed: 18-12-2023]. [Online]. Available: https://www.postgresql.org/docs/current/textsearch.html.

[7] T. Golubenco, *Full-text search engine with postgresql (part 2): Postgres vs elasticsearch*, [Accessed: 18-12-2023], 2023. [Online]. Available: https://xata.io/blog/postgres-full-text-search-postgres-vs-elasticsearch.

[8] *Db-engines ranking of search engines*, [Accessed: 18-12-2023]. [Online]. Available: https://db-engines.com/en/ranking/search+engine.

[9] *Spring data elasticsearch*, [Accessed: 18-12-2023]. [Online]. Available: https://spring.io/projects/spring-data-elasticsearch.

[10] A. Alamir, *Needle in a haystack: A nifty large-scale text search algorithm tutorial*, [Accessed: 08-01-2024]. [Online]. Available: https://www.toptal.com/algorithms/needle-in-a-haystack-a-nifty-large-scale-text-search-algorithm.

[11] M. Davtyan, *Making text search learn from feedback*, [Accessed: 08-01-2024], 2018. [Online]. Available: https://medium.com/filament-ai/making-text-search-learn-from-feedback-4fe210fd87b0.

[12] E. Nam, *Understanding the levenshtein distance equation for beginners*, [Accessed: 18-12-2023], 2019. [Online]. Available: https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0.

[13] L. Otwell, *The soundex algorithm*, [Accessed: 18-12-2023], 2021. [Online]. Available: https://medium.com/@lukehenryotwell/the-soundex-algorithm-d39c2f8d8756.

[14] P. Factor, *String comparisons in sql: The metaphone algorithm*, [Accessed: 18-12-2023], 2017. [Online]. Available: https://www.red-gate.com/simple-talk/blogs/string-comparisons-sql-metaphone-algorithm/.

[15] A. Padmanabhan, *N-gram model*, [Accessed: 18-12-2023]. [Online]. Available: https://devopedia.org/n-gram-model#:~:text=It's%20a%20probabilistic%20model%20that's,and%20then%20estimating%20the%20probabilities..

[16] A. Oussous, *A comparative study of different search and indexing tools for big data*, Jordanian Journal of Computers and Information Technology 8.1. [Accessed: 09-01-2024], 2022. [Online]. Available: https://www.researchgate.net/publication/358386535_A_COMPARATIVE_STUDY_OF_DIFFERENT_SEARCH_AND_INDEXING_TOOLS_FOR_BIG_DATA.

[17] *Apache lucene*, [Accessed: 18-12-2023]. [Online]. Available: https://lucene.apache.org/.

[18] *Apache software foundation*, [Accessed: 18-12-2023]. [Online]. Available: https://www.apache.org/.

[19] *Apache solr*, [Accessed: 18-12-2023]. [Online]. Available: https://solr.apache.org/.

[20] *Uploading data with solr cell using apache tika*, [Accessed: 18-12-2023]. [Online]. Available: https://solr.apache.org/guide/7_5/uploading-data-with-solr-cell-using-apache-tika.html.

[21] *Spatial search*, [Accessed: 18-12-2023]. [Online]. Available: https://solr.apache.org/guide/6_6/spatial-search.html.

[22] *Graylog*, [Accessed: 18-12-2023]. [Online]. Available: https://graylog.org/.

[23] *Who uses elasticsearch?* [Accessed: 18-12-2023]. [Online]. Available: ttps://stackshare.io/elasticsearch.

[24] *Who uses postgresql?* [Accessed: 18-12-2023]. [Online]. Available: https://stackshare.io/postgresql.

[25] *What is the analytic hierarchy process (ahp)?* [Accessed: 18-12-2023]. [Online]. Available: https://www.passagetechnology.com/what-is-the-analytic-hierarchy-process.

[26] *Create index api*, [Accessed: 18-12-2023]. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-create-index.html.

[27] *Search api*, [Accessed: 18-12-2023]. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-search.html.

[28] L. Fittl, *Understanding postgres gin indexes: The good and the bad*, [Accessed: 09-01-2024], 2021. [Online]. Available: https://pganalyze.com/blog/gin-index.

[29] *Json types*, [Accessed: 18-12-2023]. [Online]. Available: https://www.postgresql.org/docs/current/datatype-json.html.

[30] *Using jsonb in postgresql: How to effectively store index json data in postgresql*, [Accessed: 18-12-2023], 2020. [Online]. Available: https://scalegrid.io/blog/using-jsonb-in-postgresql-how-to-effectively-store-index-json-data-in-postgresql.

[31] C. Gallant, *Arithmetic mean vs. geometric mean: What's the difference?* [Accessed: 08-01-2024]. [Online]. Available: https://www.investopedia.com/ask/answers/06/geometricmean.asp.

[32] *Elasticsearch wildcard queries*, [Accessed: 08-01-2024]. [Online]. Available: https://opster.com/guides/elasticsearch/search-apis/elasticsearch-wildcard-queries/.

[33] *Elasticsearch fuzzy match: Advanced techniques and best practices*, [Accessed: 07-01-2024]. [Online]. Available: https://opster.com/guides/elasticsearch/search-apis/elasticsearch-fuzzy-match-techniques/.

[34] *F.17. fuzzystrmatch — determine string similarities and distance*, [Accessed: 08-01-2024]. [Online]. Available: https://postgresql.org/docs/current/fuzzystrmatch.html.

[35] G.Fotopoulos, P.Koloveas, P.Raftopoulou, and C.Tryfonopoulos, *Comparing data store performance for full-text search: To sql or to nosql?* [Accessed: 09-01-2024]. [Online]. Available: https://users.uop.gr/~trifon/papers/pdf/data23-FKRT.pdf.

[36] P.Ribarski, B.Ilijoski, and B.Tojtovska, *Comparing databases for inserting and querying jsons for big data*, [Accessed: 09-01-2024], 2019. [Online]. Available: https://proceedings.ictinnovations.org/attachment/paper/518/comparing-databases-for-inserting-and-querying-jsons-for-big-data.pdf.

[37] *123ahp*, [Accessed: 18-12-2023]. [Online]. Available: http://www.123ahp.com.

[38] *Priority estimation tool (ahp)*, [Accessed: 18-12-2023]. [Online]. Available: https://sourceforge.net/projects/priority/.

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

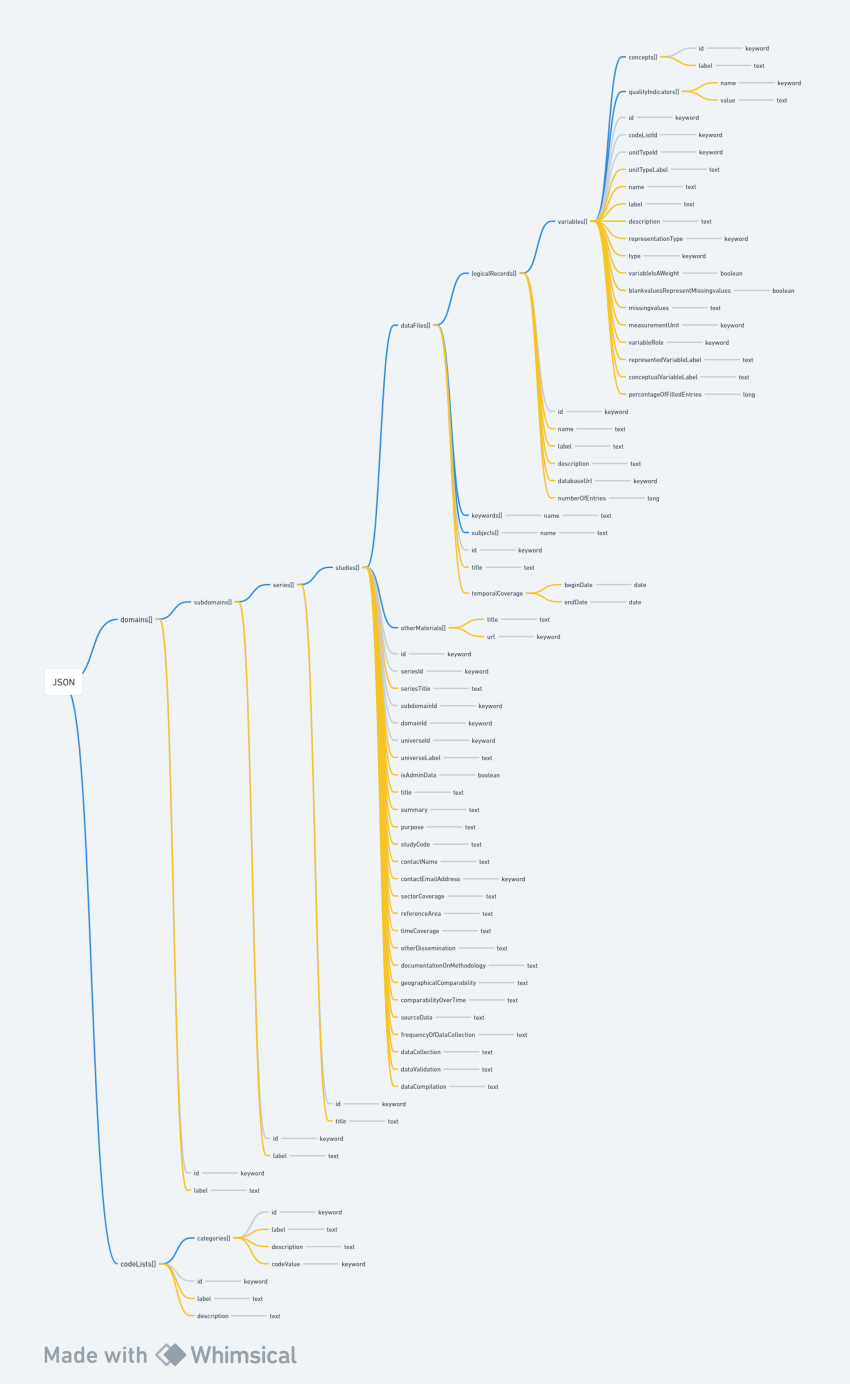
Mina, Boriss Zahharov

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Elasticsearchi ja PostgreSQLi sobivuse võrdlemine täistekstotsingu realiseerimiseks konkreetse süsteemi näitel", mille juhendaja on Erki Eessaar
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

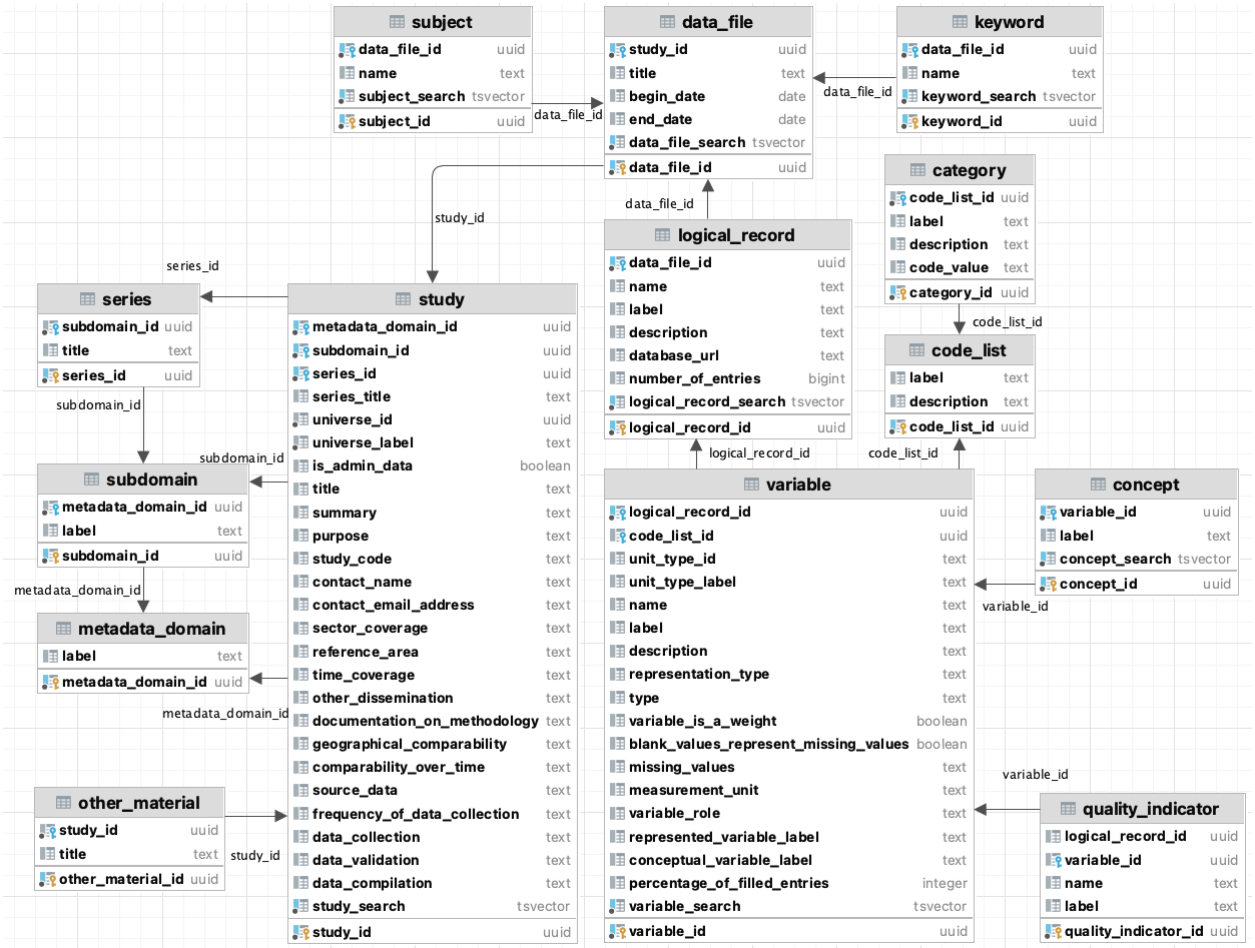
10.01.2024

[1] Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

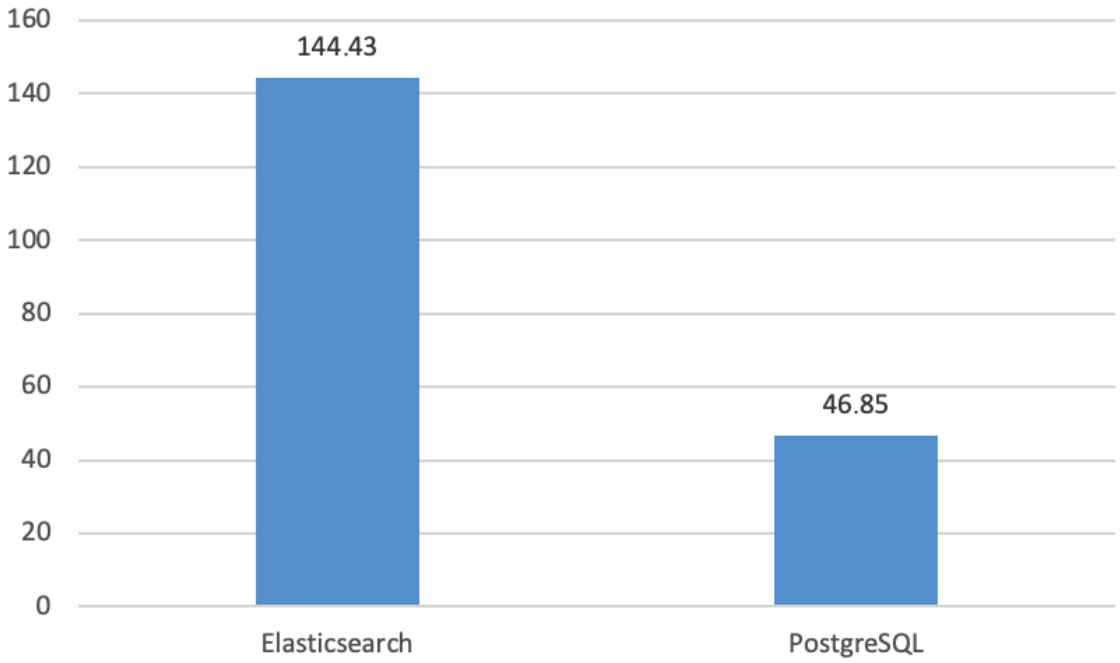
Lisa 2 - Metaandmete JSON struktuur



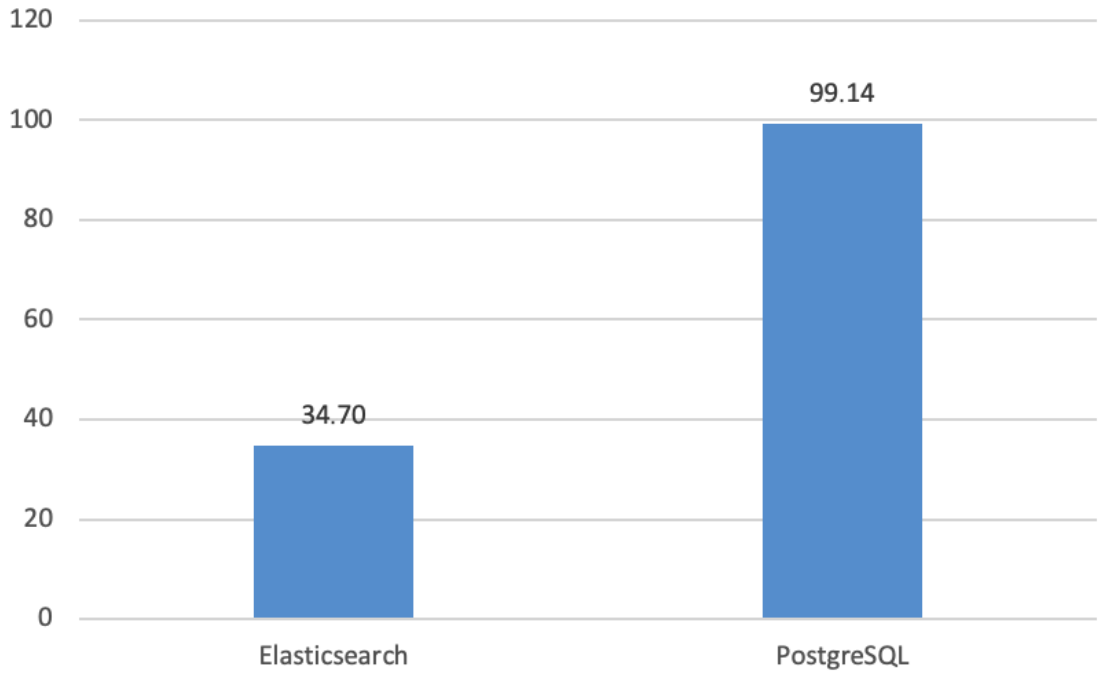
Lisa 3 - Baastabel iga olemitüübi kohta - tabelite struktuur



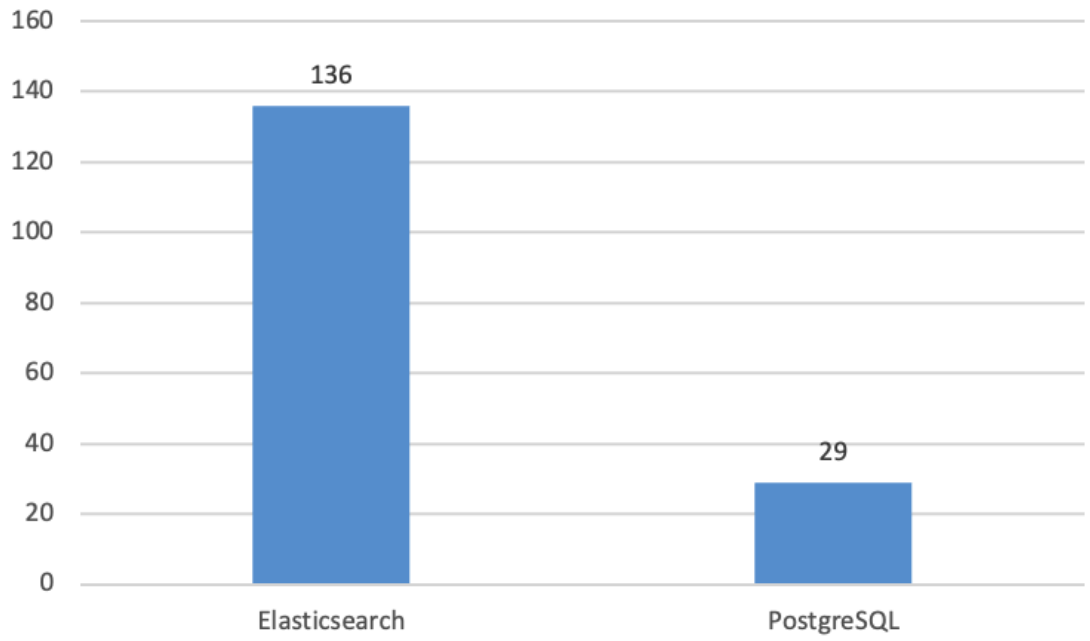
Lisa 4 - Elasticsearch ja PostgreSQL otsingule kulunud aja geomeetriline keskmine (millisekundites)



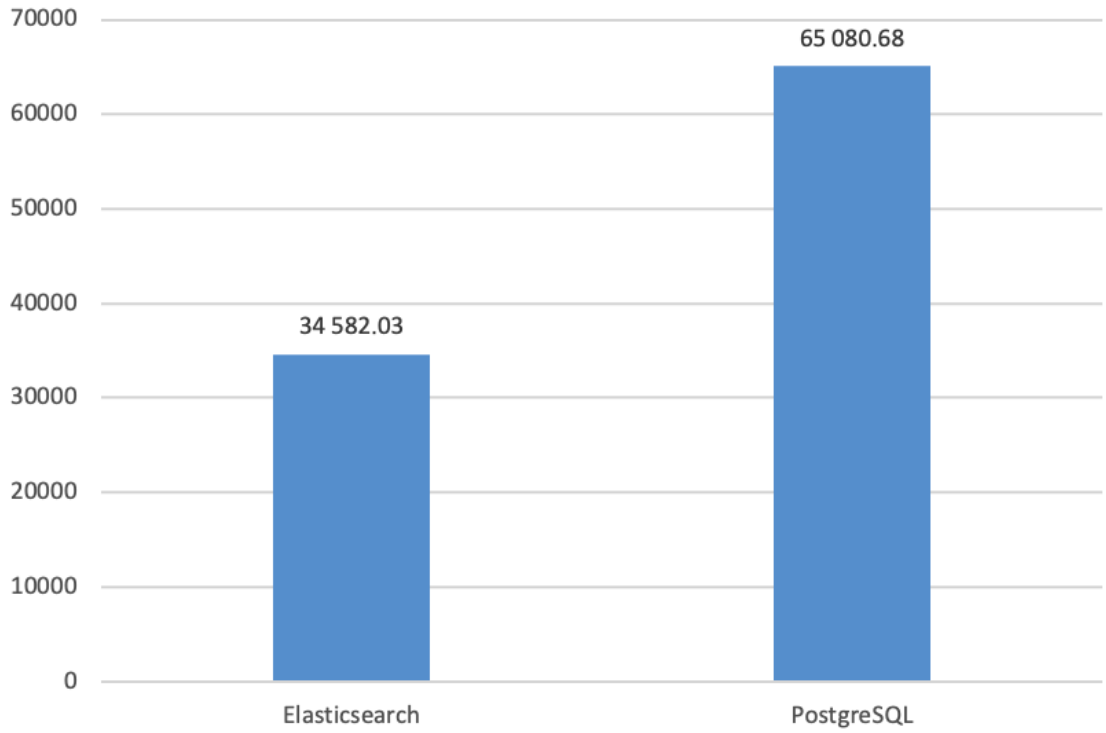
Lisa 5 - Elasticsearch ja PostgreSQL otsingule kulunud aja hälve (millisekundites)



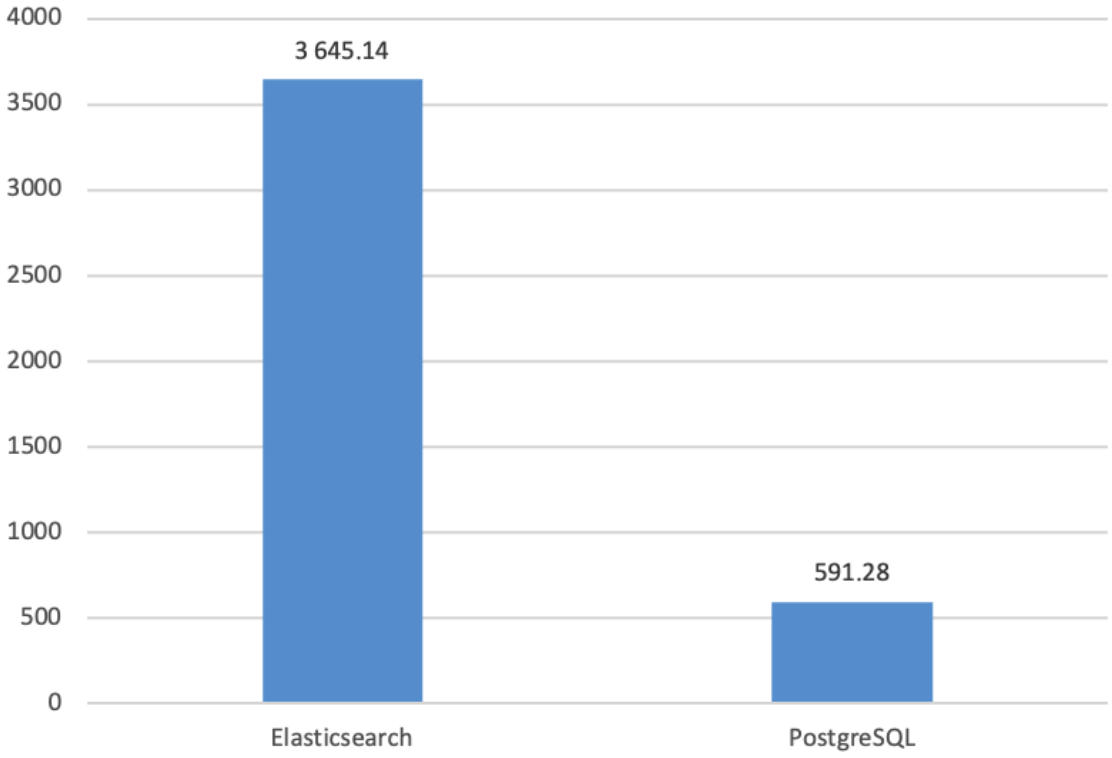
Lisa 6 - Elasticsearch ja PostgreSQL otsingule kulunud aja mediaan (millisekundites)



Lisa 7 - Elasticsearch ja PostgreSQL salvestamisele kulunud aja geomeetriline keskmine (millisekundites)



Lisa 8 - Elasticsearch ja PostgreSQL salvestamisele kulunud aja hälve (millisekundites)



Lisa 9 - Elasticsearch ja PostgreSQL salvestamisele kulunud aja mediaan (millisekundites)