TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKA TEADUSKOND
Arvutiteaduse instituut
Infotehnoloogia õppekava

Vjatšeslav Krõšin

Asukohapõhiste veebisündmuste algoritmid (Twitteri andmete näitel)

Magistritöö (30 EAP)

Juhendaja(d): MSc Elis Kõivumägi

Tartu 2014

Asukohapõhiste veebisündmuste algoritmid (Twitteri andmete näitel)

Lühikokkuvõte:

Käesoleva magistritöö raames on katsetatud erinevate algoritmide jõudlust Twitteri andmete põhjal. Töö eesmärgiks on testida algoritmide pädevust suurte andmete protsessimisel (kaardil kuvamisel) ja püüda andmeid kiiresti lõppkasutajani viia nii, et nad oleks sobivalt filtreeritud ja klasterdatud. Andmeid on võimalik kaardil näha nii agregeeritud kujul, kui ka agregeerimata. Veebirakendusega on võimalik otsida andmeid vabalt valitava raadiusega. Andmete klasterdamisel ja kauguste arvutamisel on katsetatud erinevaid algoritme ja võrreldud nende kiirusi. Andmebaasi päringute kiirendamiseks on katsetatud lisaks indekseerimisele ka riistvaralisi muudatusi. Andmeteks on Tartu Ülikooli poolt pakutud Twitteri andmed, kus on üle 24 miljoni kirjet.

Võtmesõnad:

algoritmid, valemid, veebirakendus, kaardistamine, Twitter, Groovy, Grails, Java, Postgres, PostGIS

Location-based algorithms for events on the Web (based on Twitter data)

Abstract:

With this master's thesis the performance of different algorithms was tested using data from Twitter. The goal of this study was to test the performance of the algorithms when processing large-scale data (when displaying a map) and to try to get the data to the final user as quickly as possible with it being appropriately filtered and clustered. The data can be seen on the map in an aggregated state but also in a disaggregated state. With this web application it is also possible to search for a certain location with freely chosen radius in which only the points in a fixed radius are mapped. In order to cluster the points and calculate the distances, different algorythms were used and their speed compared. To quicken the database queries, changing hardware was also tried. The data used is Twitter data offered by The University of Tartu where there are over 24 million entities.

Keywords:

algorithms, formulas, web application, mapping, Twitter**,** Groovy, Grails, Java, Postgres, PostGIS

Sisukord

Jooniste nimekiri

Algoritmide nimekiri

Tabelite nimekiri

Lühendid

API	Application Programming Interface
CSS	Cascading Style Sheets
GB	Gigabyte
GIN	Generalized Inverted Indexes
GiST	Generalized Search Tree
GIS	Geographic Information Systems
GROM	Grails Object Relational Mapping
HDD	Hard Disk Drive
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IOPS	Input/Output Operations Per Second
JSON	JavaScript Object Notation
MVC	Model View Controller
OS	Operating System
ORDBMS	Object-Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
SSHD	Solid State Hybrid Drive
SRID	Spatial Reference Identifier
WGS	World Geodetic System

1 Sissejuhatus

Iga aastaga tuleb sotsiaalvõrgustikel järjest rohkem kasutajaid, mille andmete haldamine ja otsimine toimub järjest mahukamate andmete seast. Töö raames proovitakse välja selgitada, kuidas on võimalik teostada mahukaid andmebaasi päringuid, ilma et see võtaks kaua aega. Antud töös räägitakse nii riistvaralistest muudatustest, kui ka algoritmides. Andmeteks võivad olla nii numbrilised, kui sõnalised andmed. Magistritöös on süvenetud geograafilistele andmetele, mis koosnevad pikkuskraadidest ja laiuskraadidest.

Twitter on üks suurematest sotsiaalvõrgustikkest, kus on võimalik kirjutada ja ka lugeda väiksemaid uudiseid inimeste eraelust ja ettevõttete saavutustest. Ühe tweedi pikkus peab olema maksimaalselt 140 tähemärki ja paljud tweetidest sisaldavad geograafilisi andmeid. Töö raames on testitud algoritmide pädevust suurte andmete protsessimisel ja püütud andmeid kiiresti lõppkasutajani viia nii, et nad oleks sobivalt filtreeritud ja klasterdatud. Andmeteks oli Tartu Ülikooli poolt korjatud tweedid, milleks oli üle 24 miljoni kirje. Töö raames on valmistatud rakendus, millega on testitud algoritme suuremahuliste andmete peal ja tulemusi kajastatakse Google Maps kaardil agregeerituna.

Täpsemaks selgituseks võib võtta näitliku olukorra, kus populaarses kohas on tehtud palju tweete ning tekib olukord, kus on mitu punkti teineteise peal ja sellepärast ei ole võimalik valida ja vajutada just nendele punktidele, mida inimesed sooviksid uurida. Kui nüüd vaadata kaarti, siis oleks hea, kui saaks neid kõiki punkte vaadata agregeeritud viisil. Käesolevas magistritöös on võimalik lugeda, kuidas on võimalik saavutada head tulemust, kus oleks andmebaasi päring efektiivne ja ka punktide klasterdamine piisavalt kiire mahukate andmete peal.

Antud magistritöö eesmärgiks katsetada erinevaid algoritme ja muid vahendeid seotud päringu optimiseerimisega, millega on võimalik otsida tweete ja kaardistada neid Google Maps abil. Otsimise kriteeriumiteks oleksid järgmised väljad:

- Algus kuupäev
- Lõpp kuupäev
- Otsitav sõna
- Klasterdamise algoritm
- Terve kaardi otsing või kindla punkti raadiuse otsing

Väljade täitmine ei ole kohustuslik. Lisaks otsitavatele parameetritele saab valida meetodi, millega on võimalik andmeid klasterdada. Otsingut võimalik teostada ka kindla punkti raadiuse järgi. Raadius on vabalt sisestav väli, mille mõõtmisühikuks on meetrid. Kui kaardil vajutada kindlale punktile, siis tekib ka raadiuse ring, mille järgi on võimalik näha otsitavat ala. Selleks, et saavutada selline funktsionaalsus, tuleb teha õiged valikud kasutajaliideses.

Käesolev magistritöö koosneb 6-st põhi osast. Kõige alguses on juttu tehnoloogiate kasutusest ja milleks nad on vajalikud. Kuna suurt rolli mängib ka riistvara, siis on võrreldud ka kõvakettaid erinevate kiirustega ja seda, kuidas see võib muuta tulemuste kiirusi. Järgmisena tuleb juttu sellest, kuidas oleks võimalik optimiseerida päringute kiirusi. Selles osas räägitakse nii indekseerimisest, kui ka erinevatest katsetest indekseerimistega. Kuna tegemist on Postgres andmebaasiga, siis rakendusse on võetud kasutusele plugin, millega oleks võimalik kiirendada tulemusi.

Kui päringute optimiseerimine lõppeb, tuleb suurem osa andmebaasist saadud tulemuste klasterdamisest, kus selgitatakse välja klasterdamise vajadust ja tehakse mitmeid klasterdamise erinevate algoritmidega.

Kuna tegemist on geograafiliste koordinaatidega, siis klasterdamise jaoks on vaja arvutada ka punktide omavahelisi kaugusi, aga kuna koordinaadid on geograafilised, siis ei piisa lihtsalt Euclidean valemi kasutamisest, sest koordinaatidel tuleb mängu ka maakera raadius. Punktide omavahelisi kaugusi on võimalik leida mitmel viisi, mõni algoritm ei pruugi olla 100%-se täpsusega, kuid sellegi poolest on võimalik, et kiiruse vahe on niivõrd suur, et täpsust võib ohvriks tuua. Punktide kauguste mõõtmise kiirusi mõõdeti erinevate andmemahtudega ja hiljem selgus, kas kiiruse vahe on niivõrd suur, et tuleks võtta kasutusele midagi muud, kui alguses oli plaanitud.

2 Töövahendid ja tehnoloogia

Magistritöö raames kasutati arvutit, mis kasutas SSHD kõvaketast, aga kuna kõvaketta kiirusest jäi puudu, sest IOPS oli liiga väike ja mõned andmete teisendused võisid kesta mitmeid päevi tavalise kõvakettaga. Kuna tekkis vajadus kiirema kõvaketta järgi, siis tuli kiirendamiseks rentida server, kus oli kasutuses SSD kõvaketas. SSD mõjutab ka päringu kiirust ning milline vahe oli tavalise kõvakettaga selgitatakse järgmistes peatükkides. SSD kõvakettaga serveri teenuspakkujaid on palju, kuid hea hinna ja kvaliteedi suhtes asetus hetkelisel seisul DigitalOcean [Dig14]**.** Kui serveri ruum oli saadaval, siis sinna tuli installeerida operatsioonisüsteem CentOS 6.5, koos muude vajalike vahenditega, näiteks Postgres.

Projekti raamistikuks oli valitud Grails raamistik, mis kasutab Groovy keelt. Groovy on Java põhinev keel, mis võib teha tavalise Java koodi umbes 50% lühemaks ja palju loetavamaks. Grails raamistik kasutab põhiliselt Groovy keelt, kuid võib kirjutada ka tavalises Java keeles. Raamistik on ehitatud Spring MVC ja Hibernate peale. Näitena saab tuua koodi, millega saab kätte andmeaasist objekti tema Id järgi, kuid siiski, kui teha Hibernate-ga andmebaasi päring, siis on selleks vaja kirjutada mitu rida koodi, kuid Grails-i GROM abil on võimalik teha keerulisi päringuid ühesuguste koodidega. [Gra14] Järgmisena on välja toodud näide, kuidas saab teha andmete päringut andmebaasi Hibernate abil:

```
Session session = HibernateUtil.getSessionFactory().openSession();
Object tweet =  (Tweet) session.get(Tweet.class, tweet_id);
```

Grails-il on olemas Domeen klassid, mis on seotud andmebaasiga. Selleks, et pärida domeen klassist andmeid, piisab sellest, kui kirjutada järgmine kood:

```
def tweet = Tweet.get(tweet_id)
```

GROM teeb taustal ise kõik vajalikud teisendused ja avab/sulgeb ka vajadusel Hibernate sessiooni. Eelmise koodi tulemuseks on objekt, mida saab manipuleerida samamoodi nagu tavalist java objekte. [Gra14]

Rakendusserveriks oli valitud Tomcat 7 ja andmebaasi vahendiks oli Postgres 9.3, millega koos kasutati ja võrreldi tavalisi SQL päringuid Postgres plugina päringutega, kus pluginaks oli PostGIS. Vaadete kirjutamiseks oli kasutatud järgmisi vahendeid:

- Twitter API
- Javascripti
- jQuery
- jQueryUI
- HTML
- CSS

PostGIS on on vaba ja avatud lähtekoodiga raamatukogu, mis on mõeldud postgreSQL relatsioonilise andmebaasi haldus süsteemi jaoks. PostGIS-il on olemas üle 300 funktsiooni ja andmetüüpi. [Pos14]

Selleks, et PostGIS saaks kasutada oma funktsioone ja andmetüüpe, peab andmebaas võimaldama ruumilisust. Ruumilisust võimaldav andmebaas on mõeldud spetsiaalset tüüpi andmete jaoks, milleks on geomeetrilised andmed. Sellise andmebaasiga saab salvestada tavalistesse tabelitesse geomeetrilisi andmeid, mis on enamasti geograafilise päritoluga. Kuna tavaline Postgres andmebaas ei toeta ruumilisust, tuleb tõmmata pugin nimega PostGIS. Peale pugina tõmbamist ja paigaldamist on võimalik kasutusele võtta tavalisse Postgres andmebaasi ruumilisi andmeid. Selleks, et andmebaas toetaks PostGIS funktsioone ja andmetüüpe, peab käivitama tavalise Postgres andmebaasis järgmise SQL koodi:

```
CREATE EXTENSION postgis;
```

Peale SQL koodi käivitamist hakkab andmebaas pakkuma spetsiaalseid funktsioone ja indekseerimise meetodit päringute ja andmete manipuleerimiseks. Selle andmebaasiga on võimalik nii andmeid salvestada kui ka analüüsida. PostGIS-il on olemas andmetüüpe, millega võimalik teha 2D kaardistamist. Kolm põhi andmetüüpi on näiteks punkt, joon, hulknurk. Uurimuses on vaja ainult punkti ja hulknurka, millega võimalik saavutada palju, mida tavaliste vahenditega oleks palju keerukam teha. [OS11]

3 Andmed

Tartu Ülikool oli andnud kasutuseks nende poolt kogutud Twitteri andmeid. Andmete faili suuruseks oli 5,62 GB ja andmete formaat oli JSON. Kokku oli andmeid 24347024 ja andmete rida nägi välja järgmiselt:


```
{
  "id":"3e1474f4b009e4576ed92adced4625fad74930ee",
  "time":"2012-10-21T21:21:50.000Z",
  "author":"cdec17f532cb4fa8184c09cc49d2830aa95e29e0",
  "body":"f8238760c504ca9fea2fe6708dedcad9d4bbdd9c",
  "loc":{
    "type":"Point","coordinates":[51.56628129,-1.74492274]
  }
}
```


Kasutajatunnus ning tweedi sisu oli turvalisuse kaalutlustel hashitud. Tavaliselt Twitteri andmed tulevad REST teenusest ja nendel andmetel on palju rohkem lisa välju. Kuna Twitteril on piirang, et päring saab tagastada maksimaalselt 1500 vastust, siis otsustati, et ei hakata implementeerima Twitteri live versiooni. Live versiooniga ei saa teha oma päringuid andmebaasi jaoks ja ei saa kasutada mingeid lisavahendeid, näiteks PostGIS.

Enne seda, kui saab hakata andmebaasi päringuid tegema, tuleb kõik andmed teisendada just sellisesse formaati, nagu rakenduse jaoks oleks tarvis. Andmete teisendamisel pole mingeid piiranguid. Postgres kahjuks ei toeta JSON kujul andmefaile.

4 Andmebaasi kõvaketta valik

Kuna rakenduse päringu kiirus ei sõltu ainult klasterdamise algoritmidest, vaid ka andmete otsimise ja tagastamise kiirusest. Selleks, et kiirendada andmebaasi tööd, oli tehtud katsetatud nii SSD, kui ka SSHD kõvaketta peal. Nende kahel kõvaketta peamine vahe on nende kiirus. Kui rääkida kõvaketta andmetest, siis SSHD andmed on järgmised:

- SSHD: Seagate ST1000LM014-1EJ164 (88.4MB/s; 1500.8 IOPS)

Kõvaketta kiiruse andmed on saadud *SiSoftware* kodulehelt, kus on palju andmeid riistvaraliste võrdlustega. [Sis14]

Kuna SSD kõvaketta teenuse pakkujaga näol oli tegemist jagatud serveritega, siis kõvaketta andmeid ei olnud võimalik saada. Selleks, et saada SSD kiirust tehti kõvaketta test ja tulemus oli järgmine:

- SSD: 247 MB/s 10889 IOPS

Selgus, et nende kõvaketaste lugemise ja kirjutamise kiiruse vahe on umbes 7.3 kordne.

Lisaks võib välja tuua, et kui oleks kasutatud ka tavalist HDD-d, siis enamusel IOPS on 150-200 ja sellega oleks mahukamad päringud raskesti teostatavad.

5 SQL päringute kiirused ja nende optimeerimise meetodid

Tavaliselt kõige suurem viga, mida tehakse on see, et ei kasutata indekseerimist. Kõige populaarsemaks indekseerimise meetodiks on B-Tree indekseeriminem, mis töötab peaaegu identselt ka paljudes teistes SQL andmebaasides.

Indeks on eraldi objekt andmebaasis ja selle tegemine ei muuda tabeli struktuuri, vaid loob lihtsalt uue andmete struktuuri, mis viitab kindlale tabelile. Põhi ideeks on andmete representeerimine sorteeritud järjekorras. Indekseerimisega on andmete kättesaadavus väga kiire. [Win12]

Järgmisena vaatame, miks B-Tree indekseerimine on nii populaarne. B-tree indekseerimise lähemal uurimisel, võib välja tuua kolme tasemelist andmepuud, kus iga puu leht saab hoida nelja kirjet. Kolme tasemelisel puul saab olla kokku 64 kirjet, nagu on näha tabelis 1. Kui puu sügavus kasvab ainult ühe võrra, siis nüüd puu võib hoida 256 kirjet. Iga puu taseme lisamisel kirjete koguarv neljakordistub. See tähendab, et kui läbida ainult 9 taset, siis juba on võimalik läbi otsida üle miljoni kirjet. [Win12]

Tabel 1. B-Tree tasemed

Puu sügavus	Maksimaalne kirjete arv
3	64
4	256
5	1 024
6	4 096
7	16 384
8	65 536
9	262 144

Järgmisena tehti katseid erinevate indekseerimistega. Siin võeti kasutusele ka PostGIS plugin, kuid algul prooviti B-Tree indekseerimist, mida kasutab Postgres vaikimisi. Järgmisena tehti mõned katsed päringutega erinevatel andmemahtudel. Selleks võeti järgmised tulemuste piirid: 100, 500, 1000, 5000, 10000, 25000 ja 50000. Kõik katsed hakkavad toimusid nii SSD kõvakettaga, millel on keskmiselt 10000 IOPS ja SSHD kõvakettaga, mille keskmine IOPS on 1500 ringis.

Kuna on vaja teha mitu päringut, et saada üks keskmine tulemus, siis on oluline teha eeltööd, enne kõikide päringute järjest jooksma panekut. Probleemiks on see, et Postgres enamasti hakkab kasutama vahemälu sama päringu jaoks ja sellega ei ole võimalik enam õiget tulemust saada. Lisaks sellele on olemas ka operatsioonisüsteemi vahemälu, mis tuleb ka kasutusse. Selleks, et teha ühte päringut mitu korda, tuleb enne igat päringut käivitada koodi, mis on **joonisel 1:**

```
[root@twitterApp1 ~]# sync; /etc/init.d/postgresql-9.3 stop; echo 1 > /proc/sys/
vm/drop_caches; /etc/init.d/postgresql-9.3 start
Stopping postgresql-9.3 service:                           [  OK  ]
Starting postgresql-9.3 service:                           [  OK  ]
```

Joonis 1 Linuxi ja Postgresi vahemälu tühjendamine

Sellel joonisel olev käsurea kood teeb kolme asja järjest:

1. Lõpetab Postgres teenuse tööd
2. Puhastab operatsioonisüsteemi vahemälu
3. Käivitab Postgres teenust

Esimese katsena on tehtud päring ilma indekseerimiseta ja siin selgitatakse välja, kui kiiresti päring tagastab vastuse. Päringuks on järgmine SQL kood:

```
SELECT * FROM Tweet t
WHERE longitude BETWEEN 57.058772 AND 63.51437
AND latitude BETWEEN 15.534538 AND 33.74987
AND time BETWEEN fromDate AND toDate
AND body ILIKE '%search%';
LIMIT 1000;
```

Koordinaatide järgi on asukohaks valitud suurem ala, mille sisse mahub Eesti, Soome, Rootsi ja osa Venemaast. Test alal on kokku üle 100 000 tweedi. Päringu vastuseks saadi järgmine tulemus:

```
Seq Scan on tweet t  (cost=0.00..2250938.44 rows=4312 width=202) (actual ti-
me=24841.976..24861.691 rows=1000 loops=1)
            ........
          Rows Removed by Filter: 19925"
        Total runtime: 8072.858 ms"
```

Tulemusest on näha, et päring kasutab *sequential scan*-i, mis tähendab terve tabeli järjestikust otsimist. Selleks, et saada 1000 õiget punkti, pidi päring läbima 19925 tabeli rida, millele kulus tervelt 8072.858 ms.

Järgmiseks on kasutusele võetud B-Tree indekseerimine, mida rakendatatkse ka Postgres puhul ja millega peaks tulemus tunduvalt paranema. Tekitati indekseerimist järgmise SQL käsuga:

```
CREATE INDEX idx_lon_lan_bTree
ON Tweet
USING btree(longitude, latitude);
```

Indekseerimine on tehtud kahel tulbal korraga, kuna rakenduses ei saa olla olukorda, kus B-tree indekseerimise puhul otsitakse ainult ühe parameetri - pikkuskraadi või laiuskraadi järgi.

Katsetati ka sama SQL päringuga, mis oli kasutuses siis, kui ei olnud indekseerimist. Seekord saadi järgmise tulemuse:

```
Index Scan using idx_lon_lan_btree on tweet t  (cost=0.44..15098.50 rows=3356
width=202) (actual time=1.257..32.756 rows=1000 loops=1)"
"Total runtime: 6470.284 ms"
```

Selles tulemuses on näha, et päring on kiirem umbes 753 korda, kui päring ilma indekseerimiseta. Kui kasutada indekseerimist, tuleb alati jälgida seda, kas indekseerimine edendab rakendust nii, et selle tagajärjed ei segaks saadud kasu. Indekseerimisega on võimalik otsinguid küll kiirendada, aga suureks miinuseks on see, et sellesse tabelisse andmete lisamine ja muutmine muutub aeglaseks. Kui oleks tegemist üksikute muutmistega ja lisamistega, siis indekseerimine tasub ära, aga kui tunni jooksul peaks tuhandeid uusi andmeid tulema indekseeritud tabelisse, siis see teeks rakenduse hoopis aeglasemaks, kui kiiremaks. Kuna lõputöö rakenduses on vaja ainult otsimist, siis antud töö raames oli indekseerimine kasulik

Enne, kui saab proovida järgmisi päringuid, tuleb välja uurida, kuidas tekitada geomeetrilisi andmeid ja kuidas saab neid kasutada. Kuna päring teostab otsingu ristküliku

kujulisest kastist, siis algul oleks vaja tekitada geomeetrilisi punkte. Kui rakendus hakkab teostama päringuid raadiusega, siis tuleb luua ja kasutada geograafilisi punkte. Hetkel on vaja ainult geomeetrilisi punkte, kuna nende kasutamisel on üks eelis geograafiliste punktide üle ja see on kiirus.

Selleks, et saaks tekitada geomeetrili punkte on vaja tweetide pikuskraade ja laiuskraade. Et seda saavutada, tuleb algul tekitada tulp, mille andmetüübiks oleks geomeetriline punkt. Selleks on vaja käivitada järgmine SQL kood:

```
ALTER TABLE tweet
ADD COLUMN geom geometry(Point,4326);
```

Selle koodiga lisatakse tulp nimega *geom*, mille andmetüübiks on geomeetriline punkt. PostGIS funktsioon *geometry* lisab geomeetrilisele punktile juurde ka märke, et tegemist on punktiga, mis kasutab ruumilisuse viidete (4326) süsteemi, mida tuntakse ka *WGS-84* nime all. See tähendab, et tegemist punktiga, millel on pikkuskraad ja laiuskraad. [OS11]

Ruumilisuse viidete süsteem tähistab koordinaat süsteeme, mida kasutatakse geomeetriliste punktidega. Nad tähistavad, kuidas geograafilised andmed on representeeritud tasapinnalisel kaardil ja mis mõõtmis ühikud on kasutuses. Selleks võivad olla näiteks kraadid, meetrid jms. [OS11]

Järgmise sammuga oleks vaja täita tühja tulpa, mille nimeks sai pandud *geom*. Et seda saavutada, tuleb rakendada järgmine SQL kood, mis uuendab tulpa ja tekitab sinna geomeetrilisi punkte koos pikkuskraadiga ja laiuskraadiga:

```
UPDATE tweet
SET geom = ST_SetSRID(ST_MakePoint(longitude, latitude), 4326);
```

Kui vaadata andmebaasist, mis väärtused on uues tulbas, siis on näha, et seal on tekkinud binaarne kirje, mida pole võimalik lugeda. Näitena on välja toodud üks vabalt valitud väärtus:

```
0101000020E6100000000000008032381A40000000209AEB0940
```

Peale seda, kui kõik geomeetrilised punktid on tekitatud, tuleb uut tulpa indekseerida, et kiirendada andmebaasi otsingut. Selleks, et indekseerida geomeetrilisi punkte, tuleb kasutada PostGIS indekseerimist. Tekitame järgmisena indeksi, mille nimeks paneme näiteks *idx_tweet_geom*:

```
CREATE INDEX idx_tweet_geom
ON tweet
USING GIST(geom);
```

Kasutades SSD kõvaketast, siis 24347024 väärtuse indekseerimine võtab aega umbes 13-14 minutit.

Eelnevalt toodud koodi näide, mis sisaldas *ST_SetSRID*, tähendab, et ta lisab ruumilisuse viite identifitseerijat punktile ja see on numbriline väärtus, mis tähistab primaarset võtit *meta* tabelis nimega *numega spatial_ref_sys*. PostGIS kasutab seda tabelit selleks, et kategoriseerida kõik ruumilisuse viidete süsteemid, mis on andmebaasile saadaval. [OS11]

Kui tahetakse mõõta kahe linna vehelist kaugust PostGIS abil ja tekitada tavalised punktid funktsiooniga *ST_MakePoint(x, y)* ning mõõta nende vahelisi kaugusi funktsiooniga *ST_Distance*, siis vastus saadakse tavalise Pythagorase teoreemi valemiga, mis tagastab vastuse kraadides. Lisaks sellele poleks siin arvestatud ka maakera raadiust, isegi kui oleks õiged laiuskraadid ja pikkuskraadid, mis ei mõjutaks tulemust. Seega peab alati kasutama SRID süsteemi, et PostGIS arvutaks õigeid vahemaid. [OS11]

Peale indekseerimist võime teha päringu ja kontrollida, kas Postgres hakkab kasutama indekseerituid tulemusi või teeb ikka järjestikust otsimist. Selleks, et seda kontrollida, võib käivitada järgmist SQL koodi:

```
SELECT t FROM Tweet t
WHERE ST_Within(geom, ST_GeomFromText ('POLYGON((coords[0] coords[1], coords[0]
coords[3], coords[2] coords[3], coords[2] coords[1], coords[0] coords[1]))',
4326))
AND time BETWEEN fromDate AND toDate
AND body ILIKE '%search%'
LIMIT 1000;
```

Kui käivitada eelnev SQL kood, siis tulemuseks võib näha järgmist koodi rida ja veenduda, et andmebaas kasutab indekseeritud geomeetrilisi andmeid:

```
Index Scan using idx_tweet_geom on tweet  (cost=0.42..640823.69 rows=50586
width=186) (actual time=2.486..3253.870 rows=1000 loops=1)
...
Total runtime: 3258.335 ms
```

Kuna vaja oli 4 nurka, et tekitada ristküliku kujuline otsinguala, kuid rakendus tagastab ainult 2 nurka, milleks on alumine vasak nurk ja ülemine parem nurk, siis sellest oli või-

malik tuletada ka ülejäänud 2 nurka. Javascript tagastab kaardi kaks nurka järgmisel kujul: *lat_lo,lng_lo,lat_hi,lng_hi*. Sellest võime tuletada järgmised nurgad:

- SW – lat_lo, lng_lo
- NW – lat_hi, lng_lo
- SE – lat_lo, lng_hi
- NE – lat_hi, lng_hi

Eelmises päringus on kasutatud funktsiooni *POLYGO)*, millega saab tegelikult teha piiranguid rohkem, kui nelja nurgaga. Kui teha nelinurkne piirang *POLYGO)* funktsiooni abil, siis on vaja kirja panna viis nurka, sest viimane nurk peab alati olema ka esimene nurk. Kuna funktsioonid vajavad punkte ehk *Point* andmetüüpe, siis *ST_GeomFromText* võimaldab teha tekstist erinevaid geomeetrilisi kujundeid ilma, et peaks looma eraldi punkte.

5.1 Indekseerimise võrdlused

Selles osas on võetud kokku kõik indekseerimise katsetused erinevate otsingu tulemuste piirangutega. Testimised on tehtud nii SSDH, kui ka SSD peal, et saada ülevaate, kas SSD on ikka niipalju parem, kui algul oli arvatud. Võrdlemises on kasutatud tavalist B-Tree indekseerimis, kui ka GIST indekseerimist. Seda indekseerimist oli kasutatud nii geomeetriliste, kui ka geograafiliste andmete puhul.

Et olla kindel, kas õige indekseerimine oli kasutuses, tuli leida võimalus, kuidas saaks ajutiselt välja lülitada teised indekseerimise tüübid. Selleks kasutati järgmised SQL koodid:

```
update pg_index set indisvalid = false where indexrelid =
'idx_lon_lan_btree'::regclass;
update pg_index set indisvalid = true where indexrelid =
'idx_tweet_geom'::regclass;
update pg_index set indisvalid = false where indexrelid =
'idx_tweet_geography'::regclass;
```

Esimeseks tabeliks on valitud SSHD kõvaketta peal tehtud katsetused, mille IOPS oli keskmiselt 1500 ringis.

8462.Tabel 2. SSHD indekseerimise kiirusetestid

Suurus	**Indekseerimiseta**	**B-Tree**	**GIST (geometry)**	**GIST (geography)**
100	783.937 ms	304.115 ms	49.637 ms	59.520 ms
500	1370.495 ms	511.985 ms	216.930 ms	250.169 ms
1 000	2155.812 ms	727.549 ms	587.775 ms	448.944 ms
5 000	8335.830 ms	2101.367 ms	1819.119 ms	19004.064 ms
10 000	25589.560 ms	49388.649 ms	3283.864 ms	86211.129 ms
25 000	43467.317 ms	131900.683 ms	8462.630 ms	183194.403 ms
50 000	49984.596 ms	210024.429 ms	16952.429 ms	374056.180 ms

Teises tabelis on katsetused tehtud SSD kõvaketta peal tehtud katsetused. SSD keskmine IOPS on umbes 10 00

Tabel 3. SSD indekseerimise kiirusetestid

Suurus	**Indekseerimiseta**	**B-Tree**	**GIST (geom.)**	**GIST (geog.)**
100	122.890 ms	207.023 ms	29.759 ms	29.115 ms
500	348.561 ms	255.207 ms	121.549 ms	290.648 ms
1 000	609.211 ms	345.285 ms	211.365 ms	420.381 ms
5 000	2175.926 ms	1313.384 ms	1189.661 ms	2445.261 ms
10 000	4233.545 ms	2519.840 ms	1290.656 ms	3839.276 ms
25 000	16480.287 ms	5745.899 ms	3714.838 ms	4747.867 ms
50 000	25057.343 ms	7827.054 ms	5763.177 ms	8748.053 ms

Katsetamise tulemustes on näha, et kõvaketta valikust võib päringu kiirus erineda mitmeid kordi. Hetkeline kiiruse vahe oli tulemuste järgi keskmiselt kahekordne. Võrdluste järgi saab järeldada, et kui rakenduse jaoks on oluline päringu kiirus, siis tuleks alati kaalutleda ka kõvaketta valikud, sest sellest võib tulla mitmekordne kiiruse vahe.

6 Kaardil punktide klasterdamine

Google Maps kaardil punktide klasterdamisega on võimalik lahendada kahte probleemi. Esiteks, kui tuleks näidata tuhandeid punkte kaardil, siis nende joonistamine oleks aega nõudev protsess, mis võtaks väga palju arvuti ressurssi, mille tagajärjeks on aeglane arvuti ja ebameeldiv kogemus. Aegluse põhjustab see, et kaardile peab joonistama JavaScripti abiga ja selleks, et tekitada kaardile üks punkt, tuleb algul teha üks Marker objekt, ja lisada see kaardile. Tuhandete objektide kaardile joonistamine võtaks kindlasti palju arvuti ressursse, mille tagajärjel võib veebilehitseja kokku joosta.

Teiseks probleemiks on kasutusmugavus. Kui kaardil on väga palju marker-eid ja nad kõik on üksteise peal, siis selle tagajärjeks on halb kasutatavus, kuna on võimatu vajuta-da kindlale punktile.Mõlemaid probleeme saaks väga edukalt lahendada mõne klasterdamise algoritmiga, mis võtaks kokku kõik punktid kindla raadiusega ja tagastaks punktid agregeeritud viisil, et oleks palju arusaadavam ja mugavam kasutada. Nii saab kergelt identifitseerida näiteks populaarsemaid kohti kaardil.

6.1 Kahe punkti vaheline kaugus

Selleks, et saaks klasterdada punkte, peab algul arvutama punktide vahelisi kaugusi. Sel-le teostamiseks on neli võimalust. Edasi on välja toodud kõik võimalused ja seletused nende sobivuse kohta. Lõpus tehakse ka kaugus algoritmide kiiruse võrdlusi, mille järgi on valitud algoritm rakenduse jaoks.

Euclidean kaugus

Euclidean valem sobib siis, kui pole vaja arvutada kaugusi meetritesse Kuna see valem ei arvesta maakera kumerust, siis seda rakenduses kasutada ei saa. Euclidean kaugust kahe punkti vahel arvutatakse sirgel joonel, kuid rakenduse punktid sisaldavad geograafilisi koordinaate. Euclidean kaugust arvutatakse järgmise valemiga [Sol10]:

$$d(A,B) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

Kui seda valemit oleks võinud kasutada rakenduses, siis selle kood näeks välja järgmisena:

```
float calcDistance(Point p1, Point p2) {
     Math.sqrt( Math.pow(p2.latitude - p1.latitude, 2) +
     Math.pow(p2.longitude - p1.longitude, 2)).round(6)
}
```

Haversine kauguse valem

Kõige täpsemaks variandiks oleks Haversine valem, mis võtab arvesse ka maakera raadiuse. Kuna andmebaasis on tegemist punktidega, millel on pikkuskraadid ja laiuskraadid, siis see valem oleks üks kandidaatidest, mida oleks võimalik võtta kasutusele. Haversine kauguse valem näeb välja järgmisena:

$$d = 2r\sin^{-1}\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\psi_2 - \psi_1}{2}\right)}\right)$$

Kus ψ on pikkuskraad, φ on laiuskraad ja r on maakera raadius. Valemi järgi on võimalik tuletada järgmine pseudo-kood:

```
dlon = lon2 - lon1
dlat = lat2 - lat1
a = (sin(dlat/2))^2 + cos(lat1) * cos(lat2) * (sin(dlon/2))^2
c = 2 * atan2( sqrt(a), sqrt(1-a) )
d = R * c (R on maakera raadius)
```

Kui see pseudo-kood teisendada Java/Groovy koodiks, siis see väljenduks järgmiselt:

```
float haversineDistance(float lat1,float lat2,float lon1,float lon2) {
   float dlat = Math.toRadians(lat2 - lat1)
   float dlon = Math.toRadians(lon2 - lon1)
   float lat1R = Math.toRadians(lat1)
   float lat2R = Math.toRadians(lat2)
   float a = (Math.sin(dlat / 2) * Math.sin(dlat / 2)) +
          (Math.sin(dlon /2) * Math.sin(dlon / 2) *
           Math.cos(lat1R) * Math.cos(lat2R))
   float c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a))
   float d = 6371 * c // 6371 == maakera raadius meetrites
   return d
}
```

Kauguse mõõtmine maapinnal koosinuse abil

Järgmine valem sarnaneb harvesine valemiga, kuid kui vaadata seda valemit, siis tundub et harvesine valem on rohkem ressurssi nõudlikum, kuid täpsem.

Valem:

$$d = \text{acos}(\sin(\varphi_2)\ \sin(\varphi_1) + \ \cos(\varphi_2)\ \cos(\varphi_1)\ \cos(\psi_1 - \ \psi_2))R$$

Seda valemit on võimalik kirjutada koodis järgmisel:

```
float sphericalLawOfCosinesDistance(float lat1,float lat2,float lon1,float lon2)
{
  return Math.acos(Math.sin(Math.toRadians(lat2)) *
   Math.sin(Math.toRadians(lat1)) + Math.cos(Math.toRadians(lat2)) *
   Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lon2 - lon1))) *6371
}
```

Õigepikkuselise projektsioonil ligikaudse kauguse arvutamine

Kuna on tegemist väga suure andmemahuga, siis Haversine valem võib-olla üsna ressurssi nõudev, kui tuleks arvutada kaugusi tuhandete punktide jaoks. Väiksemate kauguste jaoks, näiteks linnad, võib kasutada pütagorose teoreemi õigepikkuselise projektsioonide peal. See on projektsioon, mis säilitab moonutusteta teatud punktide vahelised jooned [SS12]:

```
float equirectangularApproximation(float lat1,float lat2,float lon1,float lon2)
{
   float dlat = Math.toRadians(lat2 - lat1)
   float dlon = Math.toRadians(lon2 - lon1)
   float x = (dlon) * Math.cos(Math.toRadians((lat2 + lat1)/2))
   float d = Math.sqrt(x*x + dlat*dlat) * 6371
   return d
}
```

Google Maps API-ga kauguse arvutamine

Viimaseks võimaluseks, kuidas võiks arvutada kahe punkti omavahelist kaugust on *Google Maps API v3* kasutamine, kuna rakendus on tehtud kasutades *Google Maps-i*. Selleks, et saaks kasutada kauguse mõõtmis funktsiooni, tuleb algul täiendada *JavaScrip-ti* raamatukogu, selleks on vaja allikale lõppu lisada *&libraries=geometry*. Järgmisena on välja toodud väike koodi jupp, kuidas saab kasutada *Google Maps* meetodit, et mõõta kahe punkti vahelist kaugust. [EG06]

```
<script type="text/javascript"
src=http://maps.google.com/maps/api/js?sensor=false&libraries=geometry />

var p1 = new google.maps.LatLng(45.463688, 9.188145);
var p2 = new google.maps.LatLng(46.043832, 9.759362);

//kauguse arvutamine kilomeetriteks
function calcDistance(p1, p2){
  var distance = google.maps.geometry.spherical.computeDistanceBetween(p1, p2)
  return (distance / 1000).toFixed(2);
}
calcDistance(p1, p2);
```

See meetod on hea ainult siis, kui oleks valikus väga vähe punkte. Magistritöö raames tehtud rakendusele selline variant ei sobi, kuna selle meetodiga tuleks serverile teha päring ja server peaks tagastama kõik punktid kliendi veebilehitsejasse ja peale seda kõik oleneks kasutaja arvutist, kui kiiresti ta arvutaks kõikide nende punktide vahelisi kaugusi. Rakenduses püüti teha kõik kasutajasõbralikumaks, kuna rakenduse töötamise kiirus oleneks lõppkasutaja arvutist.

Kaugusvalemite kiiruste võrdlused

Kuna see, et lõppkasutaja arvuti mõjutaks tulemust ei olnud soovitud, siis ei mõõdetud *Google Maps* meetodit. Samuti ei saa kasutusele võtta ka lihtsa *Euclidean* valemi, kuna tegemist on geograafiliste koordinaatidega, mitte tavaliste punktidega. Seega jääb kolm valemit, mida saab võrrelda, nendeks on:

- Haversine
- Spherical Law Of Cosines
- Õigepikkuselise projektsioonil ligikaudne arvestus

Test oli teostatud kahe punktiga 100, 1 000, 10 000, 100 000 ja 1 000 000 korda. Seda testi tegi rakendus 5 korda järjest ja keskmised tulemused on järgmised:

Tabel 4. Kauguste valemite kiirusetestid

Mitu korda	**Haversine**	**Spherical Law Of Cosines**	**Equirectangular Approximation**
100x	17 ms	4.2 ms	4.2 ms
1 000x	174.8 ms	43.8 ms	39 ms
10 000x	879.6 ms	469 ms	436.8 ms
100 000x	4387.2 ms	3760.2 ms	3574.8 ms
1 000 000x	41306.6 ms	38160 ms	36803.4 ms

Peale katse tegemist, selgus, et valemite kiiruse vahe tegelikult ei olegi nii suur, kui algul oli arvatud. Tulemuse põhjal võeti rakenduse jaoks kasutusele *Harvesine* kauguse valem, kuna see on välja toodud kolme valemi seast kõige täpsem ja kiiruse vahe pole nii suur, et see kaaluks üle täpsuse.

6.2 Geograafiline keskpunkt

Klasterdamise jaoks on vaja leida ühest klastrist kõikidest punktidest moodustuva keskmise, selle jaoks on olemas kaks võimalust. Esimeseks variandiks oleks kasutada *Euclidean* kauguse valemit, kuid see ei arvesta, et maakera ei asu sirgel joonel ja mõõdab otse punktist A punkti B. Teiseks võimaluseks, kuidas arvutada kahe punkti keskmist on järgmine kood:

```
float dLon = Math.toRadians(grid2.centroid.longitude - grid1.centroid.longitude)
float lat1 = Math.toRadians(grid1.centroid.latitude)
float lat2 = Math.toRadians(grid2.centroid.latitude)
float lon1 = Math.toRadians(grid1.centroid.longitude)

float Bx = Mh.cos(lat2) * Math.cos(dLon)
float By = Math.cos(lat2) * Math.sin(dLon)
float lat3 = Math.atan2(Math.sin(lat1) + Math.sin(lat2),
Math.sqrt((Math.cos(lat1) + Bx) * (Math.cos(lat1) + Bx) + By * By))
loat lon3 = lon1 + Math.atan2(By, Math.cos(lat1) + Bx)

float midLon = Math.toDegrees(lon3)
float midLat = Math.toDegrees(lat3)
```

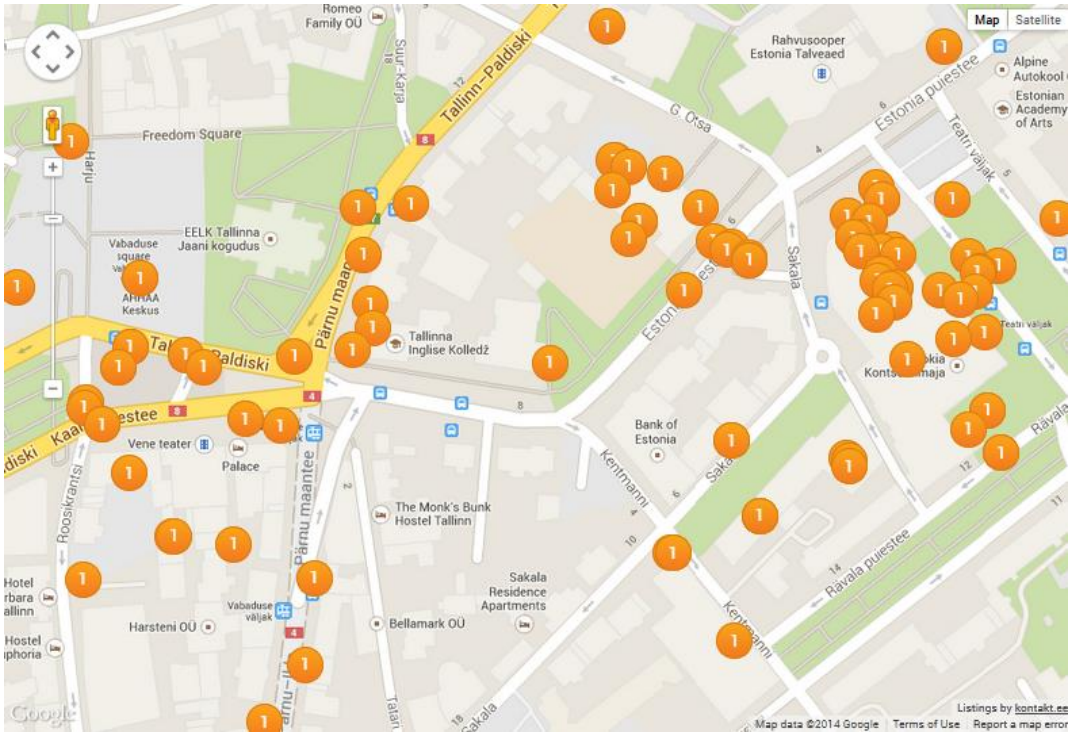
6.3 Klasterdamise algoritmi valimine

Rakendusel on olemas suur andmebaas, milles on punkte 2-dimensioonilises koordinaadi süsteemis, mida oleks vaja klasterdada, et saada loetavama tulemus. Klasterdada tuleb neid arvestades nende kaugust teineteisest. Kauguse arvutamisest, oli juttu eelmises punktis. Kui on teada, kuidas arvutada punktide omavahelisi kaugusi ja keskpunkte, siis edasi uuriti algoritme, millega saab teostada punktide klasterdamist. Selles osas uuriti ja implementeeriti klasterdamise algoritme ning võrreldi nende kiirusi.

Joonisel 2 on näha, et palju tweete üksteisele nii lähedal, et nad puudutavad teineteist või on isegi üksteise peal. Sellest tekib küsimus, et kuidas saaks seda probleemi lahendada?

Selles peatükis on välja toodud kolme klasterdamise algoritmi proovi tulemused. Iga algoritmi juurde on toodud välja nii nende miinused kui ka plussid.. Nendeks algoritmideks on:

- K-keskmiste klasterdamine
- Hierarhiline klasterdamine
- Naiivne võrgustiku põhine klasterdamine

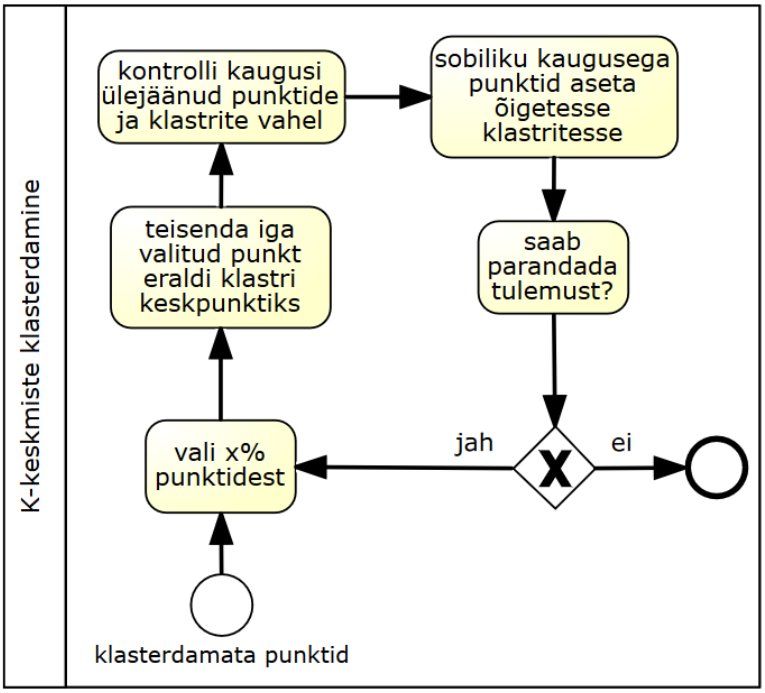


Joonis 2 Klasterdamata punktid

Kuna paljud punktid on üksteise peal, siis lähestikku olevatest punktidest tuleb teha üks agregeeritud punkt. Sellisel viisil võib tekkida kaardile üsna palju uusi agregeerituid punkte ning on vaja laiendada punktide otsitavust ning need punktid, mis on üksteise peal tuleb kokku panna nendega, mis on kindaks määratud kaugusel.

K-keskmiste klasterdamine

Alustuseks on võetud algoritm, mis on üks populaarsemaid klasterdamise algoritme just selle lihtsuse pärast. Järgmisena on välja toodud algoritmi sammud, mille järgi on rakendatud kood. Neid samme on võimalik näha joonisel 3.



Joonis 3 K-keskmiste klasterdamine

Selleks, et saada toimivad K-keskmise arvutamise algoritmi, tuleb jälgida järgmisi samme:

1. Peab valima x protsenti punktidest.
2. Asetada iga punkt kõige lähimale kesk punktile.
3. Peale selle, kui kõik punktid on asetatud lähimale keskpunktile, tuleb liigutada iga klastri keskpunkti kõikide punkti koordinaatide keskmise juurde.
4. Korda punktist 1, kuni kesk punktid lõpetavad liikumise.

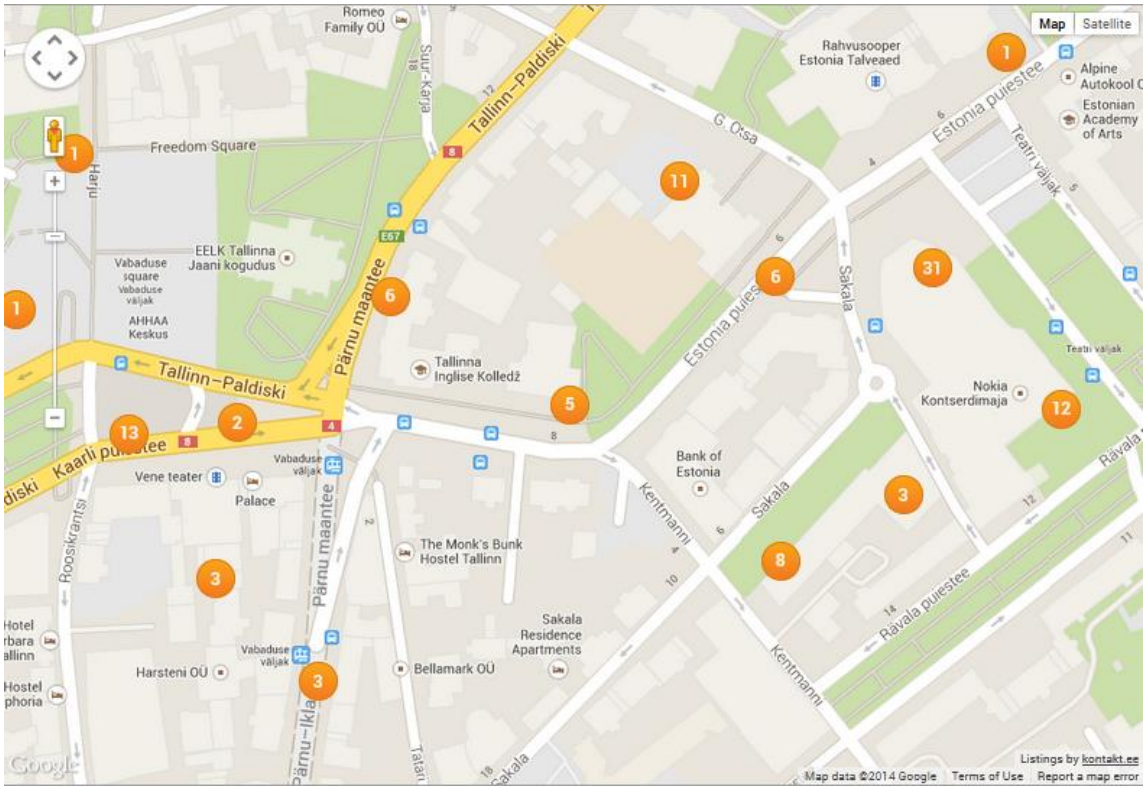
Teine eelis, mis on K-keskmiste klasterdamisel, peale selle, et seda on kerge implementeerida, on selle kiirus. Kõige suuremaks miinuseks on see, et enne klasterdamist tuleb valida k keskmine punkti, kuid selleks võib kasutada vabalt valitud punkte olemasolevatest punktidest või tekitada uusi klastrite keskpunkte. Arvestati, et kui tekitada vaba valikuga klastrite keskpunkte, siis esineb klastreid, kus ei ole läheduses mitte ühtegi punkti. Lisaks, kui valida vaba valikuga punkte ja teha nendest keskmised punktid, siis võib tekkida ka väga lähestikku klastreid, nagu on näha joonisel 4, mida aga sooviti vältida. [Wu12]

Joonis 4 K-keskmiste klasterdamise probleem

Iga katsetus valida k-punkt, kas punktide seast või tekitada klastreid juhuslikku kohta, tekitas enamasti just eelnevalt mainitud probleeme.

Tundub, et k-keskmiste algoritm ei garanteeri globaalselt optimaalse klasterdamise toimumist. Veelgi enam, see ei garanteeri lokaalselt optimaalse klasterdamise toimumist. See tähendab seda, et k-klasterdamine ei sobi antud tööks.

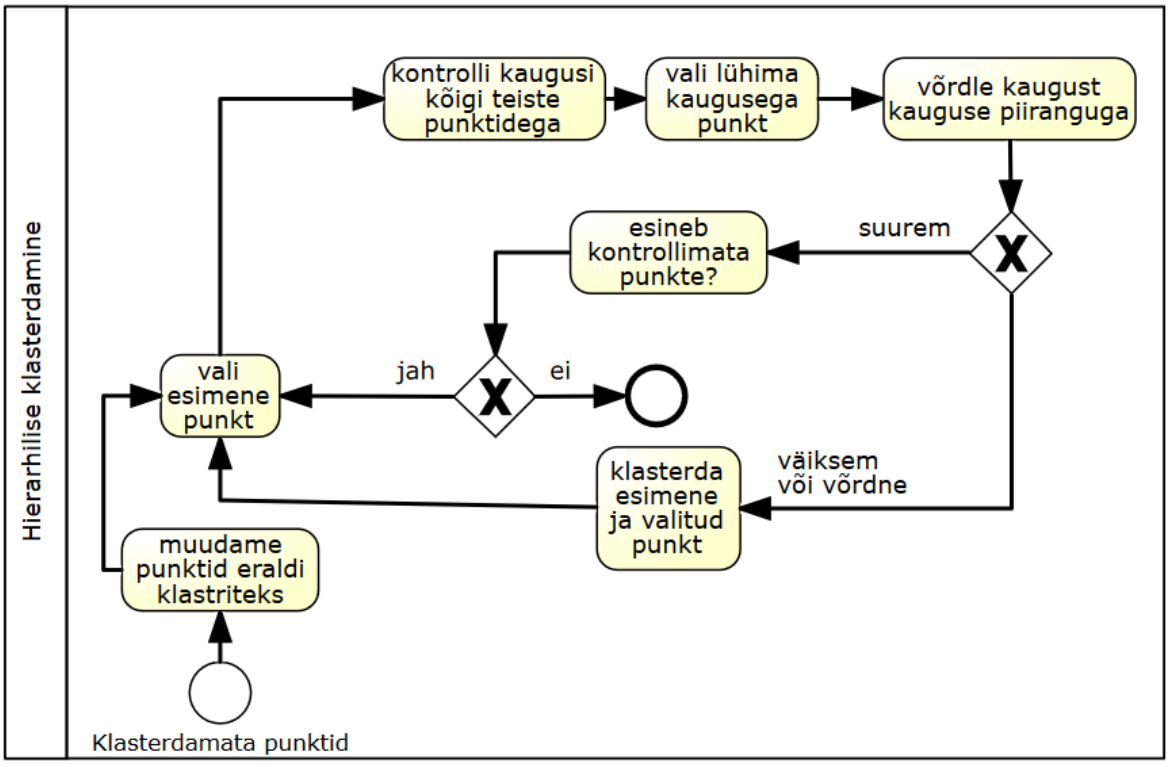
Joonisel 5 on näha tulemust, kuidas kaart näeb välja peale klasterdamist, kus on punktid piisavalt agregeeritud, et ei tekiks sellist segadust, nagu oli joonisel 2.



Joonis 5 Klasterdatud punktid

Hierarhiline klasterdamine

Järgmisena prooviti Hierarhilist klasterdamist. Hierarhiline klasterdamine toimub sarnaselt k-keskmiste klasterdamisega. K-keskmiste klasterdamine tekitab algul klastrid, kas vabalt valitud punktidest või juhuslikult tekitatud uutest punktidest. Peale seda tuleb lisada tekkinud klastritesse punktid, kuid hierarhiline klasterdamine alustab juba klastritest, see tähendab, et algul peab kõik punktid muutma eraldi asetsevateks klastriteks. Hierarhilise klaserdamisel on selline eripära, et algul tuleb läbida kõik punktid ja alles peale seda tuleb ühendada kahe kõige väiksema vahemaaga klastreid. Hierarhiline klasterdamine toimib järgmiselt:



Joonis 6 Hierarhiline klasterdamine

1. Tekita igast punktist eraldi klaster.
2. Arvuta vahemaa kõigi klastrite vahel
3. Leia kaks teineteisele kõige lähemal olevat klastrit ja kombineeri nad omavahel üheks klastriks.
4. Tuleb korrata punktist kaks, kuni tulemuseks on nii palju klastreid, kui on vaja või nad on teineteisest piisaval kaugusel, et ei saaks enam uusi klastreid luua.

Hierarhilise klasterdamisega on võimalik saada globaalselt optimaalne tulemus, kuid optimaalsusel on hind: hierarhiline klasterdamine võib joosta $O(n^3)$ ajaga kõige halvemal juhul. Hierarhilist klasterdamist võib oluliselt kiirendada, kui muuta andmete struktuuri, näiteks salvestada kõik kaugused tabelisse. Hierarhilise klasterdamisega võimalik saada häid tulemusi, mitte nagu k-keskmiste puhul. Kokkuvõttes on hierarhiline klasterdamine tunduvalt aeglasem, kui antud uurimuses vaja. [XW09]

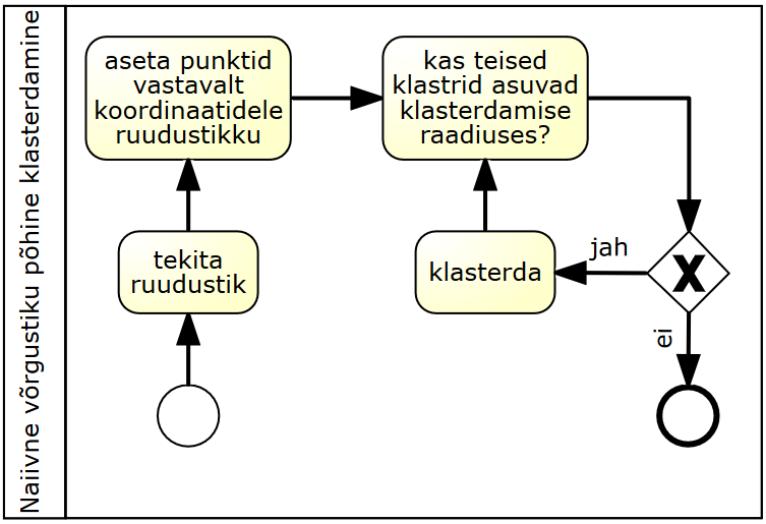
Seda algoritmi oleks võimalik kasutada juhul, kui n oleks väike arv, kuid meil on vaja lähenemist, mida saaks kasutada ka väga suure n puhul (kümneid tuhandeid punkte ja mõist-

liku ajaga HTTP päringu jaoks.. Hierarhiline klasterdamine pole ikka see, mida saaks selles rakenduses kasutada.

Naiivne võrgustiku põhine klasterdamine

Google Maps kaardi jaoks pole vaja optimaalset või matemaatiliselt täpset klastrist, näteks sellist, kus klastri keskmine punkt oleks optimiseeritud läbides kõiki variante, mida klaster võimaldab. See algoritm, mida antud töös vaja, pidi oskama kahte asja:

1. Punktid, mida me peame näitama kaardil, ei tohiks teineteisega kokku puutuda või teine teise peal olema.
2. Kõiki punkte ei tohi näidata korraga, vaid tuleb agregeerida neid üheks klastriks, kus oleks ainult üldine info, näiteks punktide arv.



Joonis 7 Naiivne võrgustiku põhine klasterdamine

Selleks, et lahendada neid punkte, meil oli vaja ruudustikku, nagu on näidatud joonisel 8. Iga ruut pidi olema vähemalt nii suur, et punkt mahuks sisse. Järgmisena tuli meil võtta kõik punktid, mis on ühe ruudu sees ja teha nendest üks klaster. Peale seda saime vaadata kõrval olevaid ruudustikke ja kui seal sees oli piisava kaugusega klaster/punkt, siis ühedasime kokku mõlema ruudustiku punktid ühte suuremasse klastrisse ja leidsime nende keskmise.

Joonis 8 Võrgustik Google Maps-il

Et see algoritm toimiks efektiivselt, tuleb seda implementeerida moel, mis on näidatud joonisel 7.

Seda algoritmi on võimalik teha veelgi efektiivsemaks, kui muuta punktide paigutamist õigetesse ruudu koordinaatidesse. Seda võimalik järgmiselt: kui ruudustik on näiteks 20x25 ja tuleb läbi käia n punkti, siis tuleb tekitada üks tühi nimekiri, kuhu hiljem tuleb asetada kõik ruudud, milles on vähemalt 1 punkt. Kui järgmine punkt hakkab otsima rudustikest oma asukohta, siis esimesena tuleb läbida täidetud ruutude nimekirja, kuna on väga suur tõenäosus, et see ruut on juba kasutuses ja sellega võib saavutada suurema ajavõidu, kui otsides iga punktiga kõikidest ruutudest õiget asukohta.

Klasterdamise algoritmide võrdlused

Kui kõik algoritmid olid implementeeritud, siis oli võimalik teha ka klasterdamise kiiruse teste. Kuna järgmised testid ei olene kõvaketta kiirusest, vaid protsessori kiirusest, siis ei ole ka välja toodud kahe kõvaketta võrdlusi.

Naiivne võrgupõhine klasterdamine pole küll kõige täpsem algoritm, kuna sellega asetatakse punktid teatud ruudustikku põhinedes koordinaatidele, kuid rakenduses oli eesmär-

giks kiirus, mitte täpsus. Algoritm on piisavalt täpne, et silmaga vaadates ei näe erinevust selle algoritmi ja matemaatiliselt korrektse valemi puhul. Huvitav on ka see, et kui päringu vastuseks oli umbes 100 tulemust, siis K-keskmine algoritm oli teistest mitu korda kiirem.

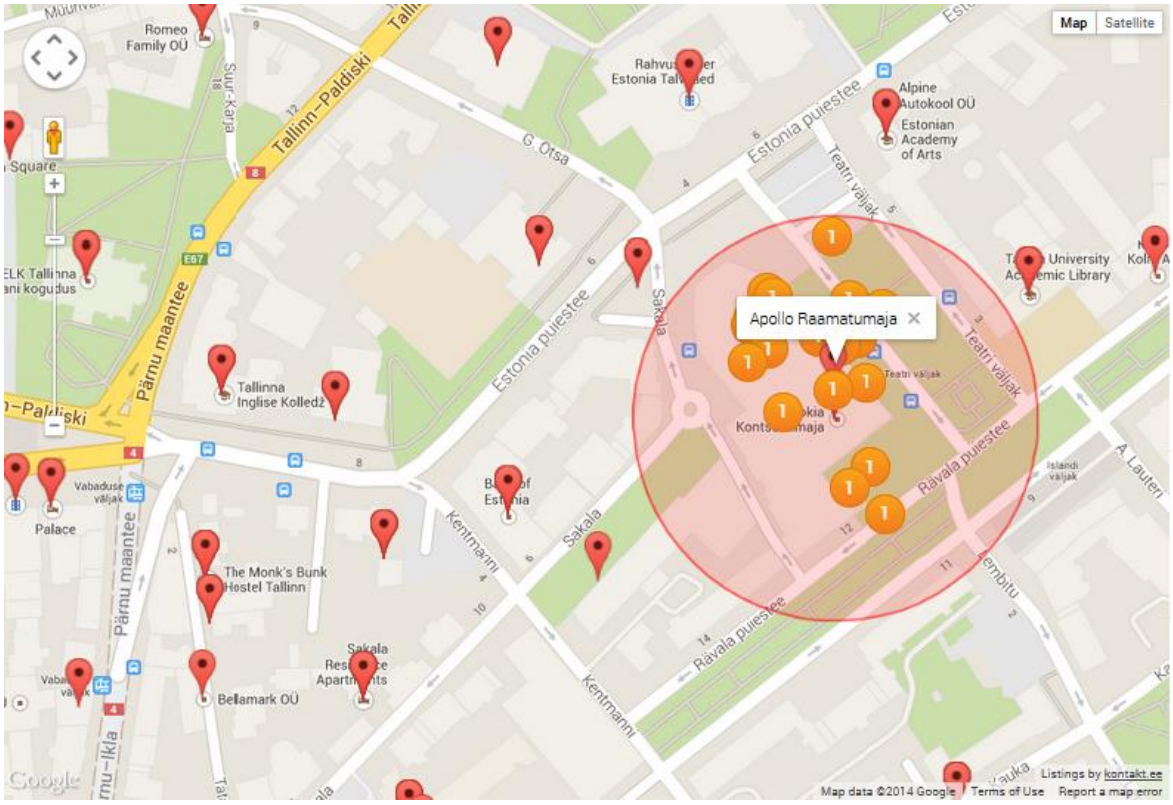
Nende tulemuste põhjal, hakkab rakendus kasutama vaikimisi naiivset võrgustiku põhist algoritmi. Nagu on näha tabelis 4, siis mõni algoritm võttis nii kaua aega, et polnud võimalik tabelit täita mõistlikkuse piirides.

Tabel 5. Klasterdamise algoritmide kiirusetestid

Suurus	Naiivne võrgustiku põhine	K-keskmiste	Hierarhiline
100	**108 ms**	**32 ms**	**270 ms**
500	**223 ms**	**553 ms**	**31097 ms**
1 000	**389 ms**	**1909 ms**	**575763 ms**
5 000	**1262 ms**	**29803 ms**	
10 000	**2455 ms**	**136444 ms**	
25 000	**8689 ms**		

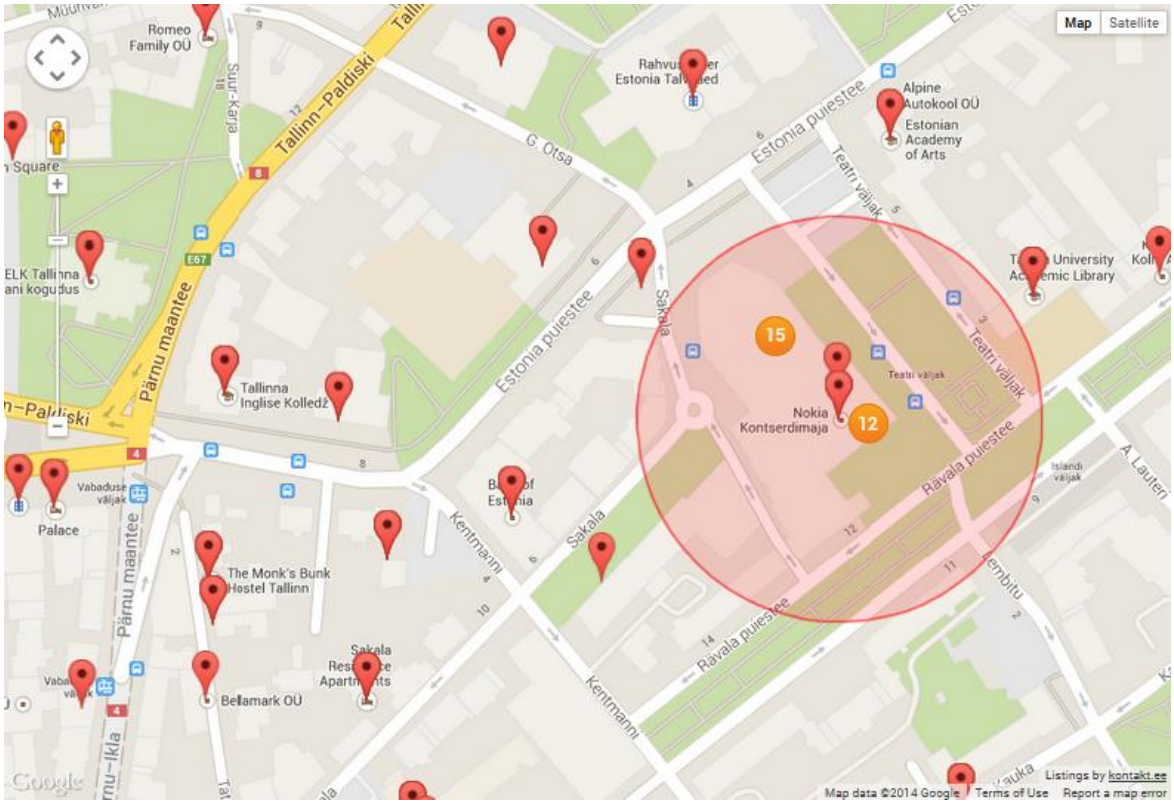
7 Populaarsete asukohtade tweetide otsimine

Selles osas püüti leida punktid, mis asuvad vabalt valitud raadiuse kaugusel kindlast asukohast. Selleks tuleb rakenduses vajutada ühele populaarsemale kohale (punane marker). Joonisel number 9 on näide, kuidas rakendus leidis punktid kindla raadiuse raamides. Kaardile võib tekitada ka rohkem otsimisi, kuid lihtsuse mõttes on välja toodud ainult üks.



Joonis 9 Klasterdamata populaarne asukoht

Siin tekib jälle sama probleem, mis varem. Leitud punktidele tuleb rakendada ka klasterdamise algoritm. Algoritmi valik oli siin sama, mis eelmises osas, ehk naiivne võrgupõhine algoritm. Kui lülitda sisse klasterdamine, hakkab sama päring näitama järgmist pilti klasterdatud punktidega:



Joonis 10 Klasterdatud populaarne asukoht

Tekib küsimus, kuidas see sai teostatud, kuna siin ei saa enam võtta kahe punkti koordinaadid ja edastada nad andmebaasile, et tekitada päring leitud nelja nurgaga. Selle teostamiseks võeti kasutusele PostGIS funktsionaalsus ja loodud rakendusele vajaliku struktuuriga punktid. Kuna viiendas peatükis räägiti geomeetrilistest punktidest, siis see oli hetk, kus tuli tekitada ja võtta kasutusele geograafilisi punkte.

PostGIS geograafilised vs. geomeetrilised mõõtmed

Geograafilisi andmeid peab alati salvestama WGS 84 andmetüübis, mis tähendab pikkuskraadi ja laiuskraadi. PostGIS puhul kõik geograafilised mõõtmed väljenduvad meetrites. Kui kõik andmed oleks teises mõõtmisühikus ja meil oleks vaja geograafilist andmetüüpi andmeid, siis see tähendaks, et kõiki neid andmeid peaks algul salvestama geomeetrilisteks andmeteks, siis teisendada korrektseteks mõõtühikuteks ja seejärel saab muuta juba geograafilisteks andmeteks. [CMKP14]

Geomeetriliste andmete eelis ongi see, et neid saab salvestada ka muudes mõõtmisühikutes, lisaks sellele nendega teostatud päringud on kiiremad, kui geograafiliste andmetega. See selgitati viiendas osas, kus mõõdeti indekseerimisete kiirusi.

Kui salvestada geomeetriline punkt lihtsalt pikkuskraadi ja laiuskraadiga, siis mõõtmisühikuks oleks kraadid. Geomeetriliste andmetüüpide eeliseks on see, et nad toetavad palju rohkem PostGIS funktsionaalsust, kuid geograafiliste andmetel on olemas ainult üksikud funktsioonid.

Andmete teisendamine geograafilisteks andmeteks

Kõigepealt on vaja lisada andmebaasi tabelile juurde uus veerg, mis hakkab hoidma geograafilisi andmeid. Järgmise SQL koodiga tekitame uue geograafilise veeru:

```
ALTER TABLE tweet
ADD COLUMN geography geography(Point,4326);
```

Kuna geograafilised andmed peavad olema WGS 84 andmetüüpi, siis tuleb märkida juurde ka 4326 andmetüüp, mis tähendab, et tegemist on andmetega, millel on olemas pikuskraadid ja laiuskraadid. Vaja oli tekitada geograafilisi andmeid ja seda sai teha järgmise SQL koodiga:

```
update tweet
set geography = ST_GeogFromText('POINT(' || longitude || ' ' || latitude ||
')');
```

Siin oli kasutuses *ST_GeogFromText*, mis võimaldab teha tavalisest teksti reast vajalik andmetüüpi, ilma et peaks ise tekitama punkti ja siis selle teisendama geograafiliseks punktiks. Geograafiliste andmetüüpide teisendamine võttis aega umbes 8 tundi SSD kõvakettaga, mis omakorda lisas andmemahtu 7.8GB. See on rohkem kui topelt varasemast andmebaasi mahust.

Selleks, et kiirendada päringu tulemusi, peab geograafilisi andmeid indekseerima. See ei erine palju sellest, kuidas indekseerimine tehti geomeetriliste punktidega. Geograafiliste punktide indekseerimiseks saab teha järgmise SQL koodiga:

```
CREATE INDEX idx_tweet_geography
  ON tweet
  USING GIST(geography);
```

Geograafiliste andmete päring

Eelnevalt kirjeldatud seisuga olid andmebaasis ka geograafilised andmed, millega on võimalik katsetada päringuid. Üheks eeliseks, mis on geograafiliste andmetel üle geomeetriliste andmete on see, et nende kirjetega on võimalik leida punkti vahelisi kaugusi, kus PostGIS pugin juba ise arvutab, arvestades maakera raadiust. Samas see arvutamine võtab rohkem ressursse ja päringud on aeglasemad. Geograafiliste andmetega on võimalik leida raadiuses asuvaid punkte, ilma et peaks ise arvutama kõikide punktide olemasolevaid kaugusi. Selleks, et leida ühe punkti raadiuses olevaid punkte, tuleb teostada järgmine SQL päring:

```
SELECT t FROM Tweet t
WHERE ST_DWithin(geography, ST_GeogFromText('SRID=4326;POINT( ${coords[0]}
${coords[1]})'),${params.radius}, false)
```

Hetkeline kood on illustreeriv, kuna õige SQL päring on natuke keerulisem ning raskesti loetav. Selles koodis on näha, et me loome tavalisest tekstist geograafilise punkti, mille raadiuses on võimalik leida andmebaasist kõiki teisi olemasolevaid punkte. Kui see päring on tagastanud vajalikud andmed, siis tuleb neid andmed klasterdada, kuid sellest on räägitud põhjalikumalt kuuendas peatükis.

8 Kokkuvõte

Kuna sotsiaalvõrgustike andmemaht suureneb igapäevaselt, võib tekkida probleeme nii päringutega, kui ka klasterdamisega. Töö oli kitsendatud Twitteri geograafiliste andmete peale.

Magistritöö raames on valminud veebirakendus, millega testiti algoritmide pädevust suuremahuliste andmete protsessimisel. Selleks, et kaardistada suuremahulisi Twitteri geograafilisi andmeid sai katsetatud kolme klasterdamise algoritmi.

Enne seda, kui saab hakata tegelema klasterdamisega, tuleb otsida võimalust, kuidas arvutada geograafiliste koordinaatide vahemaid. Selleks oli valikuks mitu algoritmi. Kuna nende algoritmide kiiruse vahe oli nii väike, siis veebirakenduse jaoks sai valitud kõige täpsem algoritm, milleks oli Haversine kauguste valem.

Klasterdamise algoritmideks sai valitud K-keskmiste, hierarhiline ja naiivne võrgustikupõhine algoritm. Kõige aeglasema tulemuse andis hierarhiline klasterdamine. See algoritm on matemaatiliselt kõige täpsem nendest kolmest, kuid täpsusega on ka suur ressurssi kulu.

Järgmisena oli katsetatud K-keskmise klasterdamist. Selle algoritmi kiirus oli tunduvalt parem, kui hierarhilisel algoritmil, kuid sellega enamus ajast ei saanud sarnaseid tulemusi. See oli tingitud sellest, et K-keskmise klasterdamine vajas algus klastreid, kas vabalt valitud koordinaadid või tuli muuta mingi arv punktidest klastriteks ja moodustada nende järgi uusi klastreid. Seda tuli teha nii kaua, kuni olid täidetud teatud tingimused, näiteks klastritel pidi omavahel olema teatud, et saaks enam uusi moodustada klastreid.

Viimaseks oli proovitud naiivset võrgustikupõhilist klasterdamist. See oli kolmest algoritmist kõige kiirem ja andis ka stabiilse tulemuse. Selle kiirus seisnes sellest, et algul tuli tekitada kindel ruudustik, kuhu sai asetatud andmed vastavalt koordinaatidele. Kui ruudustik oli täidetud, tuli klasterdada täidetud ruudustiku punktid ja võrrelda klastrite kaugusi omavahel, selleks, et saaks ühendada klastreid omavahel tingitud kaugusest. See algoritm oli sellepärast nii kiire, et polnud vaja arvutada andmete vahelisi kaugusid, nii nagu tuli teha kahe eelmise algoritmiga. Selle algoritmiga pidi ainult tekitama ruudustiku, asetama andmed koordinaatide järgi õigetesse kohtadesse ja nendest klastrid tegema.

Selleks, et saada mahukast andmebaasist kiireid päringu tulemusi, oli vaja andmebaasi indekseerida. Rakenduse andmebaasi jaoks võeti kasutusele plugin, mis oli tehtud geomeetriliste ja geograafiliste andmete töötlemiseks. Andmebaas kasutab tavaliselt B-Tree indekserimist, kuid plugin võimaldas kasutada GIST indekseerimist, mis on mõeldud spetsiaalselt geograafiliste ja geomeetriliste andmete indekseerimiseks. Plugina abil kiiruse võit oli märgatav just suuremahuliste päringutega. Väiksemate päringutega oli kiirus väga sarnane.

Suure kiiruse võidu andis ka kiire SSD kõvaketas. Kuna andmebaasi päringute kiirusel on oluline kõvaketta IOPS, siis SSD kõvaketas on enamasti mitu korda kiirem, kui tavaline või hübriid kõvaketas..

9 Edasine töö

Andmebaasi päringud on võimalik veel kiiremaks teha. Selleks võib kasutusele võtta Postgresi plugin nimega PgPool-II. Selle plugina põhi eesmärgiks oleks paralleelne paring. Et seda saaks kasutada, peab olema vähemalt 3 serverit kasutuses. Üks server peaks tegema päringu vastu teisi servereid ja kasutades *dblink*-i on võimalik päringute vastused kokku panna, et saaks tagastada ühe suure vastuse. Sellisel viisil on võimalik jagada päring mitmeks jupiks, et saada parem töö jaotus. See töötaks hästi mahukamate päringutega, kuid väiksematega võib ta hoopis aeglustada vastuse saamist. [Pgp14]

10 Kasutatud kirjandus

[EG06] Schuyler Erle, Rich Gibson "Google Maps Hacks", 2006, 82-88

[SS12] Ranel Suurna, Eveli Sisas "Kartograafia alused", 2012, 25-28

[OS11] Regina O. Obe, Leo S. Shu "PostGIS in Action", 2011, 1-275

[Pos14] "PostGIS 2.1.4dev Manual". URL: http://postgis.net/stuff/postgis-2.1.pdf

[CMKP14] Paolo Corti, Stephen Vincent Mather, Thomas J Kraft, Bborie Park "PostGIS Cookbook", 2014

[Win12] Markus Winand "SQL Performance Explained", 2012, 10-22

[Pgp14] Pgpool-II. URL: http://www.pgpool.net/mediawiki/index.php/Main_Page

[XW09] Rui Xu, Don Wunsch " Clustering ", 2009, 31-62

[Wu12] Junjie Wu "Advances in K-means Clustering: A Data Mining Thinking", 2012, 1-12

[Dig14] DigitalOcean. URL: https://www.digitalocean.com/

[Gra14] Grails. URL: https://grails.org/

[Sis14] SiSoftware URL: http://www.sisoftware.eu/rank2011d/top_device_all.php

[Sol10] Mark Solomonovich "Euclidean Geometry: A First Course", 2010

Title in English

Masters’s thesis (30 ECP)

Vjatšeslav Krõšin

Summary

Because the amount of data in social networking is growing daily, problems might occur with both inquiries and clustering. The study was limited to Twitter's geographical data.

With this master's thesis a web application was made, with which the performance of algorithms was tested when processing large-scale data. In order to map large-scale Twitter geographical data three clustering algorithms were used.

Before clustering, a way to calculate the distances between geographical coordinates needs to be found. For this there are several algorithms to choose from. Because the difference in the speed of these algorithms was so minor, the most accurate algorithm - Harvestine distance formula, was chosen.

For clustering K-means, hierarchical ja naive grid based algorithm were chosen. The slowest result was given by hierarchical clastering. Mathematically, this algorithm was the most accurate, but a greater accuracy also means a bigger resource consumption.

Next k-means clastering was used. The speed of this algorithm was noticeably greater than that of the hierarchical algorithm, but it did not get similar results most of the time. This was due to fact that k-means clustering needed some base clusters, either with freely chosen coordinates or a number of the points needed to be changed into clusters and then turned into new clusters. This had to be done as long as certain conditions were met for example there had to be a certain distance between the clusters so that new clusters could not be made anymore.

Last the naive grid based clastering was tried. This was the fastest of the three algorithms and also gave the most stable result. At first a grid was made where the data was laid according to the coordinates, which was where it's speed came from. When the grid was filled the points on the grid needed to be clustered and the distances between the clusters measured in order to combine the clusters in conditioned distances This algorithm was faster because there was no need to calculate the distances between data as had to be done

with the previous two algorithms. With this algorithm only a grid had to be made and data inserted according to the coordinates into their correct positions and turned into clusters.

In order to get quick inquiries from a large-scale database the data needed to be indexed. For the applications database a plugin was used which was made for processing geometric and geographic data. The database usually uses B-Tree indexing, but the plugin allowed to use GIST indexing, which is specifically meant for indexing geographical and geometrical data. With the help of this plugin the gain in speed was noticeable particularly with large-scale inquiries. With smaller inquiries the speed was very similar.

A big gain in speed was also due to the SSD hard drive. Because the IOPS of the hard drive is important for the speed of the inquiries, the SSD hard drive is mostly many times faster than a normal or hybrid hard drive

Lisad

I. Terminid

Algoritm sammsammuline tegevusjuhis	Plugin rakenduse lisa funktsionaalsuste kogum
Klasterdamine objektide agreerimine	Indekseerimine Andmete kaardistamine algoritmi järgi
Live versioon reaalaja andmete töötlemine	Andmebaasi paring otsingu teostamine

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina **Vjatšeslav Krõšin** (sünnikuupäev: 25.04.1988)
(*autori nimi*)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose **Asukohapõhiste veebisündmuste algoritmid (Twitteri andmete näitel)**,
(*lõputöö pealkiri*)

mille juhendaja on Elis Kõivumägi,
(*juhendaja nimi*)

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **26.05.2014**