TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Sarah Marion Mikk 153003 IAPM

GRAAFIDE ESITAMINE SQL-ANDMEBAASIDES

Magistritöö

Juhendaja: Erki Eessaar
Doktor

Tallinn 2017

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: Sarah Marion Mikk

07.05.2017

Annotatsioon

Magistritöö üheks eesmärgiks on süstematiseerida SQL-andmebaasides graafide esitamise disainilahendusi ning ühtlustada ja struktureerida nende esitust. Magistritöö teiseks eesmärgiks on katsetada konkreetse probleemvaldkonna (lennumarsruutide andmebaas) näitel mõnda nendest disainilahendustest ning uurida nende lahenduste mõningaid omadusi.

Töö esimeses pooles esitan ma SQL-andmebaasis graafide hoidmise disainide kataloogi. Ma ei too välja disaine kus graafid on esitatud XML või JSON dokumentidena, sest see kuuluks pigem töö alla, kuidas esitada graafe dokumendipõhise andmemudeli korral. Samuti ei esita ma disaine, mis võimaldavad säilitada informatsiooni servade ja sõlmede eksisteerimise kohta kindlal ajahetkel. Esitan disainid struktureeritult mustrite formaadis. Toon välja kokkuvõtliku võrdlustabeli selle kohta, milliste omadustega graafide esitamiseks mingi disain sobib.

Seejärel kavandan ning viin läbi eksperimendi PostgreSQL (9.5) ja Oracle (12c Enterprise Edition, Release 1) andmebaasisüsteemides, kasutades näitena lennuliinide andmebaasi. Graafiks on antud juhul suunatud kaalutud tsükliline multigraaf. Viin eksperimendi läbi kahe lähteülesande jaoks sobiva disainiga – *külgnevusnimistu* ja *seoste suunatud graaf*. Selleks projekteerin magistritöö teises osas eksperimendis kasutatavad andmebaasid. Iga eksperimendis käsitletava disaini puhul võrdlen päringute ja andmemuudatuse operatsioonide kiiruseid, andmemahte ning andmekäitluskeele koodi keerukust. Kiiruste võrdlemiseks viin andmebaasides läbi andmete lugemise, muutmise ja kustutamise operatsioonid. Eksperimendi tulemuste põhjal analüüsin disainilahenduste sobivust konkreetse ülesande lahendamiseks erinevate päringute, andmehulkade ning andmebaasisüsteemide korral.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 108 leheküljel, 8 peatükki, 45 joonist, 12 tabelit.

Abstract

Representing Graphs in SQL Databases

One goal of the Master's Thesis is to systematize different design solutions for representing graphs in SQL databases and make representations of these designs more uniform and structured. Another goal of the thesis is to experiment with some of those design solutions using airline routes database and explore some properties of those solutions. The thesis results will help database designers, application designers, and data architects to choose the design solution for representing graphs in SQL databases that is the most suitable for their needs. It will also help them to choose the data model for representing graph data. In order to choose between data models (for instance, SQL, graph, network, document) the chooser has to be well informed on the possibilities and limitations of using each data model in order to represent their data. The thesis would be an input to the process. As of the spring 2017, SQL database management systems (DBMSs) are still the most popular DBMSs. Thus, in case of choosing a DBMS for representing your data, a SQL DBMS is still a viable alternative. Using DBMSs with different data models for different data is good in terms of having the best representation for data but it also increases technical complexity of the system. Moreover, in case of NoSQL graph-based systems there is no central standardization process, the systems are not very mature yet, the systems do things differently and compete with each other, and thus there is a bigger possibility that a selected DBMS will be at some point discontinued that leads to the costly migration process.

I bring forth possible designs for representing graphs in SQL databases. The design catalogue does not include designs where graphs are represented as XML or JSON documents. In those designs graph nodes and edges are hidden in documents. Thus, these are more like designs for document data model and a SQL database would only be used as their storage environment. I also do not present designs that allow users to keep the history of graph structure. The designs are presented in a pattern format to give a better overview as well as to have a structured and more uniform representation. In addition,

there is a summary comparison table of designs about how these could be used for graphs with specific properties.

Next, I design and conduct an experiment in PostgreSQL (9.5) and Oracle (12c Enterprise Edition, Release 1) DBMSs by using an airplane routes database. The graph there is a weighted directed multigraph (multidigraph) that can have cycles. The experiment is carried out with two designs suitable for the problem domain – *Adjacency List* and *Connection Directed Graph*. I build the databases that I will use in the experiments. For each used design, I compare query and data modification performance, data size, and the complexity of code. To compare performance, I execute select, update, and delete operations in all the constructed databases. Finally, I analyze the design solutions' suitability for the given problem based on the results.

The results of the experiment show that data takes up less space in case of *Adjacency List* design than in case of *Connection Directed Graph* design. In addition, it is easier to write queries in case of *Adjacency List* design. There is not much difference between the complexity of queries written in Oracle and PostgreSQL. However, base tables (tables) and indexes take up less space in PostgreSQL. In Oracle, database queries and data modification operations are faster in case of *Adjacency List* than in case of *Connection Directed Graph* design. However, in PostgreSQL recursive queries to greater depth are faster in case of using *Connection Directed Graph* design. For most queries it takes more time to execute the query when there is more data in database. Only insertion of rows does not depend on data size.

There are multiple ways to continue the work done. One line of research could be to pick a different problem, which has a graph with different properties than the one in current experiment. In this case, one has to compare different designs suitable for picked graph and do experiments that are relevant in case of the chosen problem. Another way to expand the experiment is to compare graph representation using different graph representation designs in SQL database and graph data models.

The thesis is in Estonian and contains 108 pages of text, 8 chapters, 45 figures, 12 tables.

Lühendite ja mõistete sõnastik

ACID	Atomicity, Consistency, Isolation, Durability
CSR	Compressed Sparse Row
CTE	Common Table Expression
GOOD	Graph-Oriented Object Database
IDS	Integrated Data Store
JSON	JavaScript Object Notation
LDM	Logical Data Model
LUBM	Lehigh University Benchmark
RDF	Resource Description Framework
SLOC	Source Lines of Code
SQL	Structured Query Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

Sisukord

1 Sissejuhatus 12
2 Teoreetiline taust 15
2.1 Graafid 15
2.2 Graafide esitamine erinevate andmemudelite korral 16
2.2.1 Graafide esitamine ja otsimine SQL-andmebaasides 20
3 Varasemad uuringud 23
4 Erinevad disainilahendused graafide esitamiseks SQL-andmebaasides 27
4.1 Adjacency List 32
4.2 Node and Edge Undirected Graph 34
4.3 Simple Directed Graph 36
4.4 Split Node Nested Sets 38
4.5 Nested Intervals 41
4.6 Flat Table 45
4.7 Multiple Lineage Columns 48
4.8 Hardcoded Graph 50
4.9 Structured Directed Graph 52
4.10 Connection directed graph 55
4.11 Path directed graph 57
4.12 Child List Directed Graph 60
4.13 Triple-store RDF 62
4.14 Type-oriented RDF 64
4.15 Predicate-oriented RDF 66
4.16 Entity-oriented RDF 68
4.17 Ainult hierarhiate jaoks sobivad disainid 70
4.17.1 Closure Table 70
4.17.2 Materialized Path 70
4.18 Disainide kokkuvõte 72
5 Eksperiment 75
5.1 Eksperimendi eesmärk 75

5.2 Eksperimendi kirjeldus 76
5.2.1 Kasutatavad andmebaasisüsteemid 77
5.3 Andmebaasi projekteerimine 77
5.3.1 Kontseptuaalne andmemudel 78
5.3.2 Andmebaasi realiseerimine 79
5.3.3 Andmebaasi disaini mudel külgnevusnimistu korral 79
5.3.4 Andmebaasi disaini mudel seoste suunatud graafi korral 80
5.4 Testandmed 81
5.5 Eksperimendis testitavad operatsioonid 82
6 Eksperimendi tulemused 88
7 Tulemuste analüüs ja järeldused 93
7.1 Päringute ja andmemuudatuse operatsioonide kiirused 93
7.1.1 Päringute ja andmemuudatuse operatsioonide kiiruste võrdlus erinevates andmebaasisüsteemides sama disaini ja andmemahu korral 93
7.1.2 Päringute ja andmemuudatuse operatsioonide kiiruse võrdlus erinevate disainide korral sama andmebaasisüsteemi ja andmemahu korral 94
7.1.3 Päringute ja andmemuudatuse operatsioonide kiiruste võrdlus erinevate andmemahtude korral sama andmebaasisüsteemi ning disaini korral 94
7.1.4 Päringu S2 kiiruse sõltuvus lennuliinide arvust 96
7.1.5 Välisvõtme indeksite lisamise mõju päringute ja andmemuudatuste kiirusele 97
7.2 Päringute ja andmemuudatuse operatsioonide keerukuse võrdlus 100
7.3 Andmebaasi salvestamiseks kulunud ruum 101
7.4 Järeldused 101
8 Kokkuvõte 103
Kasutatud kirjandus 105
Lisa 1 – tabelite loomise laused Oracle andmebaasis 109
Lisa 2 – tabelite loomise laused PostgreSQL andmebaasis 111
Lisa 3 – andmete sisestamine seoste suunatud graafi disaini tabelitesse 114
Lisa 4 – välisvõtme kitsenduste lisamine Oracle andmebaasi 114
Lisa 5 – välisvõtme kitsenduste lisamine PostgreSQL andmebaasi 115
Lisa 6 – tabelite andmemahu leidmine Oracle andmebaasis..................................... 115
Lisa 7 – tabelite andmemahu leidmine PostgreSQL andmebaasis 116

Jooniste loetelu

Joonis 1. Linnadevaheliste vahemaade graaf 29
Joonis 2. Atsükliline linnadevaheliste vahemaade graaf 30
Joonis 3. Hierarhia kujule teisendatud atsükliline linnadevaheliste vahemaade graaf... 31
Joonis 4. Külgnevusnimistu disaini tabel City koos näiteväärtustega 33
Joonis 5. Külgnevusnimistu disaini tabel City_relation koos näiteväärtustega 33
Joonis 6. Sõlmede ja servadega suunamata graafi disaini tabel Distance koos näiteväärtustega 35
Joonis 7. Sõlmede ja servadega suunamata graafi disaini tabel City_connection koos näiteväärtustega 35
Joonis 8. Lihtsa suunatud graafi disaini tabel City_relation koos näiteväärtustega 36
Joonis 9. Linnadevaheliste vahemaade graafi nummerdamine poolitatud sõlmedega pesastatud hulkade disaini jaoks 39
Joonis 10. Poolitatud sõlmedega pesastatud hulkade disaini tabel Graph koos näiteväärtustega 39
Joonis 11. Poolitatud sõlmedega pesastatud hulkade disaini tabel City_nested_sets koos näiteväärtustega 40
Joonis 12. Pesastatud intervallide kodeerimine 43
Joonis 13. Pesastatud intervallide disaini tabel City_nested_intervals koos näiteväärtustega 43
Joonis 14. Lametabeli disain 46
Joonis 15. Lametabeli disaini tabel City_list koos näiteväärtustega 47
Joonis 16. Hulk pärilikkuse veerge disaini tabel City_relation koos näiteväärtustega... 49
Joonis 17. Tase-tabeli disaini tabel City_level_1 koos näiteväärtustega 50
Joonis 18. Tase-tabeli disaini tabel City_level_2 koos näiteväärtustega 51
Joonis 19. Tase-tabeli disaini tabel City_level_3 koos näiteväärtustega 51
Joonis 20. Tase-tabeli disaini tabel City_level_4 koos näiteväärtustega 51
Joonis 21. Struktureeritud suunatud graafi disaini tabel City koos näiteväärtustega 53
Joonis 22. Struktureeritud suunatud graafi disaini tabel Middle_city koos näiteväärtustega 53

Joonis 23. Struktureeritud suunatud graafi disaini tabel End_city koos näiteväärtustega 53
Joonis 24. Seoste suunatud graafi disaini tabel City_connection koos näiteväärtustega 56
Joonis 25. Teega suunatud graafi disaini tabel City_relation koos näiteväärtustega 58
Joonis 26. Teega suunatud graafi disaini tabel City_path koos näiteväärtustega........... 58
Joonis 27. Järglaste nimekirjaga suunatud graafi disaini tabel City koos näiteväärtustega 60
Joonis 28. Kolmikute hoidla disaini tabel Triple_store koos näiteväärtustega 63
Joonis 29. Tüübile orienteeritud RDFi disaini tabel City koos näiteväärtustega 64
Joonis 30. Tüübile orienteeritud RDFi disaini tabel Connection koos näiteväärtustega 65
Joonis 31. Predikaadi-põhise RDFi disaini tabel Name koos näiteväärtustega 66
Joonis 32. Predikaadi-põhise RDFi disaini tabel Source koos näiteväärtustega 67
Joonis 33. Predikaadi-põhise RDFi disaini tabel Sink koos näiteväärtustega................ 67
Joonis 34. Predikaadi-põhise RDFi disaini tabel Distance koos näiteväärtustega 67
Joonis 35. Olemile orienteeritud RDFi disaini tabel Entity koos näiteväärtustega 69
Joonis 36. Lennuliinide andmebaasi olemi-suhte diagramm. 78
Joonis 37. Külgnevusnimistu disaini andmebaasi diagramm....................................... 80
Joonis 38. Seoste suunatud graafi disaini andmebaasi diagramm 80
Joonis 39. S2C päringu täitmisplaan Oracle andmebaasisüsteemis (65 611 rida) 95
Joonis 40. S2C päringu täitmisplaan Oracle andmebaasisüsteemis (16 400 rida) 96
Joonis 41. Päringu S2 kiiruse sõltuvus CTEsse lisatavatest marsruutide arvust............ 97
Joonis 42. D1A päringu täitmisplaan PostgreSQL andmebaasisüsteemis ilma välisvõtme indeksiteta... 97
Joonis 43. D1A päringu täitmisplaan PostgreSQL andmebaasisüsteemis koos välisvõtme indeksitega 98
Joonis 44. S2A päringu täitmisplaan PostgreSQL andmebaasisüsteemis ilma välisvõtme indeksiteta... 99
Joonis 45. S2A päringu täitmisplaan PostgreSQL andmebaasisüsteemis koos välisvõtme indeksitega 100

Tabelite loetelu

Tabel 1. Graafide esitamise ja otsimise võimalused Oracle ja PostgreSQL andmebaasisüsteemides............ 21
Tabel 2. Graafide SQL-andmebaasides esitamiseks mõeldud disainide võrdlus........... 72
Tabel 3. Eksperimendi andmemahud............ 82
Tabel 4. Eksperimendis kasutatavad päringud ja andmemuudatuse operatsioonid........ 83
Tabel 5. Päringu S1 ja andmemuudatuse operatsioonide geomeetrilised keskmised kiirused (sekundites)............ 88
Tabel 6. Päringu S2 kiirused (sekundites)............ 89
Tabel 7. Päringu S2 kiirused Oracle andmebaasisüsteemis suurima andmemahu korral (sekundites)............ 90
Tabel 8. Päringu S2 marsruutide arv Oracle andmebaasisüsteemis suurima andmemahu korral............ 90
Tabel 9. Päringu S1 ja andmemuudatuse operatsioonide geomeetrilised keskmised kiirused (sekundites) kasutades välisvõtme indekseid............ 91
Tabel 10. Päringu S2 kiirused (sekundites) kasutades välisvõtme indekseid................ 91
Tabel 11. Andmebaasi maht erinevate disainide korral (MB)............ 92
Tabel 12. Koodiridade arv erinevate disainide korral............ 92

1 Sissejuhatus

Paljud andmed on esitatavad graafide kujul ja nende töötlemiseks kasutatakse graafitöötluse algoritme. Näiteks on graafidena esitatavad sotsiaalvõrgustikud nii et inimesed on graafi sõlmedeks ja nendevahelised suhteid kujutavad graafi servad. Logistikaülesanded on veel üheks suureks valdkonnaks, kus graafiteooria kasutamine on laialt levinud.

Terminiga andmemudel tähistatakse kahte erinevat mõistet. Andmemudel ühes tähenduses on abstraktne kirjeldus andmebaaside ülesehitamiseks mõeldud üldistest ehitusplokkidest, operatsioonide tüüpidest, kitsenduste tüüpidest ja andmetüüpidest. Selliste andmemudelite näideteks on relatsiooniline andmemudel, SQL andmemudel, võrkmudel ja graafi andmemudel. Andmemudel teises tähenduses on konkreetse andmebaasi kirjeldus (nt ettevõtte X andmebaasi kontseptuaalne andmemudel).

Graafide andmeid on võimalik hoida erinevatel andmemudelitel (esimeses tähenduses) põhinevates andmebaasisüsteemides. Kui hakatakse disainima graafide hoidmiseks sobilikku andmebaasi võib graafi andmemudelil põhinev andmebaasisüsteem tunduda parima lahendusena. Need süsteemid on spetsiaalselt loodud graafidena esitatavate andmete hoidmiseks ning nende süsteemide jaoks on olemas erinevaid graafipäringute jaoks sobivaid päringukeeli.

Samas on mitmeid põhjuseid, miks uurida, kuidas oleks võimalik hoida graafe SQL-andmebaasis (andmebaas, mis on loodud SQL andmemudelit toetavas andmebaasisüsteemis e SQL-andmebaasisüsteemis). SQL-andmebaasisüsteemid on töö kirjutamise ajal (2017. aasta kevad) populaarseimad andmebaasisüsteemid. 2017. aasta mai seisuga on andmebaasisüsteemide populaarsuse indeksi [1] esikümnes seitse SQL-andmebaasisüsteemi. Esimene graafimudelil põhinev andmebaasisüsteem (Neo4j) on aga alles 21. kohal. SQL-andmebaasisüsteemid on olnud kasutusel juba ligi 40 aastat. Selle aja jooksul on neid pidevalt optimeeritud ja täiendatud. Need süsteemid suudavad läbi viia tugevate garantiidega (ACID) tehingutöötlust ja pakuvad palju võimalusi andmete kaitsmiseks. Lisaks pakuvad mitmed SQL-andmebaasisüsteemid võimalust

hoida andmeid JSON või XML tüüpi veergudes. Graafiandmete hoidmist SQL-andmebaasis on kindlasti mõistlik kaaluda juhul, kui süsteemi ainsaks ülesandeks pole töö graafidega, vaid millel lisaks muudele andmetele oleks vaja hoida ka graafidena esitatavaid andmeid. Kui olemasolevale, SQL-andmebaasi kasutavale süsteemile, lisandub vajadus hallata ka graafidena esitatavaid andmeid, siis on samuti kaalumise koht, kas kasutada edasi SQL-andmebaasisüsteemi, minna üle graafimudelil põhineva andmebaasisüsteemi kasutamisele või võtta kasutusele mõlemad. Viimast lähenemist propageerivad näiteks Sadalage ja Fowler [2] ja Fowler [3], leides et moodsad tarkvarasüsteemid võivad kasutada korraga mitmeid andmebaasisüsteeme (*polyglot persistence*). Sellise lähenemise miinuseks on suurenev tehniline keerukus ning arendajatelt/administraatoritelt nõutava oskusteabe hulk. Mõnikord on selline tehniliselt keerulisem lahendus vajalik, kuid mitte alati. Lisaks sellele, et andmed on erinevate andmebaasisüsteemide hoole all võivad need olla ka erinevatel serveritel. CAP teoreemist tulenevalt on arendajatel vaja hakata valima terviklikkuse (erinevatel serveritel olevates andmetes pole vastuolusid) ja käideldavuse (süsteem on andmete lugemiseks ja muutmiseks alati kättesaadav) vahel ning süsteemi keerukus kasvab, samas kui süsteem ei suuda enam pakkuda nii tugevaid transaktsioonilisi garantiisid (ACID) kui üks SQL-andmebaasisüsteem ühel serveril.

Graafide hoidmiseks SQL-andmebaasides on olemas erinevaid võimalusi. Andmebaaside disainimisel on vaja mõelda, milline disainilahendus sobiks kõige paremini andmete struktuuri ning huvipakkuvate päringute jaoks. Samas pole ma leidnud mitte ühtegi allikat, kus oleks ülevaatlikult esitatud kõik viisid, kuidas disainida SQL-andmebaaside jaoks graafide esitamiseks mõeldud baastabeleid (nimega tabeleid, mis pole loodud teiste tabelite põhjal) e tabeleid.

Magistritöö *üheks eesmärgiks* on mustrite formaati kasutades süstematiseerida graafide SQL-andmebaasides esitamise disainilahendusi. Magistritöö *teiseks eesmärgiks* on võrrelda lennumarsruutide andmebaasi näitel lähteülesande esitamiseks sobivaid disainilahendusi kahes populaarses ja autorile tuttavas andmebaasisüsteemis – PostgreSQL ja Oracle. Magistritöö aitab andmebaasi disaineritel, rakenduste disaineritel ja andmebaasi arhitektidel valida rakenduse vajadustele vastavat graafide esitamise disaini SQL-andmebaasis. Samuti võiks see aidata neil valida, millist andmemudelit graafide esitamiseks kasutada. Magistritöö on loogiliseks jätkuks Katerina Krönströmi tehtud magistritööle hierarhiate esitamise kohta SQL-andmebaasides [4].

Kõigepealt annan ülevaate teoreetilisest taustast. Toon välja töö lugemiseks vajalikud graafidega seotud mõisted. Käsitlen graafide esitamist erinevate andmemudelite korral. Seejärel annan ülevaate antud valdkonnas varem tehtud töödest. Järgnevalt toon mustrite formaadis välja ja võrdlen erinevaid SQL-andmebaasis graafide hoidmise võimalusi.

Edasi kirjeldan graafiandmetega tehtavat eksperimenti. Toon välja andmebaasi struktuurid ja andmetega tehtavad operatsioonid. Lõpuks esitan mõõtmiste tulemused ning analüüsin eksperimendi tulemusi.

2 Teoreetiline taust

Kirjeldan selles peatükis graafidega seotud mõisteid, mida on vaja teada magistritööd lugedes. Esitan ka lühiülevaade graafide esitamiseks sobivatest andmemudelitest.

2.1 Graafid

Jaotises toodud mõistete kirjeldused põhinevad raamatul [5].

Graaf G koosneb sõlmede (ehk tippude) hulgast V ja servade (ehk seoste) hulgast E. Iga serv seob omavahel kaht sõlme u ja v, moodustades nii sõlmepaari (u,v). Serva, kus u ja v on sama sõlm, nimetatakse silmuseks.

Suunamata graafi korral pole servadel suunda. See tähendab, et servad (u,v) ja (v,u) on samaväärsed. Seega servaks on sõlmede hulk {u,v}.

Suunatud graafi korral on iga serv ühesuunaline. Servale seatakse vastavusse järjestatud paar (u,v), kus u on serva lähtesõlm ning v sihtsõlm. Suunatud graafi serva nimetatakse kaareks e nooleks e suunatud servaks.

Ahelaks (ehk teeks) sõlmest u sõlme v nimetatakse sellist servade järjendit, kus iga eelneva serva lõppsõlm on järgneva serva algussõlmeks ning kus esimese serva algussõlmeks on u ning viimase serva lõppsõlmeks on v.

Tsükkel on kinnine ahel, kus alustades liikumist sõlmest u, on võimalik samasse sõlme u tagasi jõuda. Tsükliline graaf sisaldab vähemalt üht tsüklit. Tsükliteta (ehk atsükliline) graaf ei sisalda ühtegi tsüklit.

Sidus graaf on graaf, kus asendades kõik suunatud seosed suunamata seostega, leidub iga sõlmepaari {u,v} korral ahel sõlmest u sõlme v.

Puu on graafi alamliik, mille korral on graaf sidus ja ilma tsükliteta. Puud esitaval graafil on üks juursõlm, millest leidub üks tee igasse teise sõlme.

Hierarhia on suunatud servadega puu, millel on lisaomadused pärimine ja alluvus. [6, p. 4]

Kaalutud servadega graafi korral on servadega seotud arvud, mis näitavad serva kaalu.

Multigraafi korral võib kahe sõlme vahel olla mitu serva. [7]

Graaf $G_1 = (V_1, E_1)$ on graafi $G_2 = (V_2, E_2)$ alamgraafiks juhul, kui $V_1 \subseteq V_2$ (V_1 on V_2 alamhulk) ja $E_1 \subseteq E_2$ (E_1 on E_2 alamhulk).

Sõlm v on sõlme u naabriks juhul, kui leidub serv sõlmest u sõlme v. [8]

2.2 Graafide esitamine erinevate andmemudelite korral

Antud peatükis toon välja lühiülevaate erinevatest graafide esitamist võimaldavatest andmemudelitest. Pikemalt on antud teemal kirjutatud artiklis [9].

Andmemudeli all peetakse antud juhul silmas abstraktset kirjeldust, mis määrab ära andmebaasis andmete esitamiseks kasutatavate andmestruktuuride tüübid, sellist tüüpi andmestruktuuride põhjal tehtavate operatsioonide tüübid, sellist tüüpi andmestruktuuride korral kasutatavad kitsenduste tüübid ning võimalik, et ka andmestruktuuride koostamisel kasutatavad andmetüübid. Iga selline andmemudel võib olla aluseks ühele või mitmele konkreetsele andmebaasikeelele, mida võimaldavad kasutada konkreetsed andmebaasisüsteemid. Andmemudeli näiteks on relatsiooniline andmemudel, kus andmestruktuuri tüübiks on relatsiooniline muutuja, üheks operatsiooni tüübiks on ühendamine, üheks kitsenduse tüübiks on kandidaatvõti ning nõutud andmetüübi näiteks on relatisoonilised tüübid või tõeväärtustüüp. Relatsioonilise mudeli alusel loodud konkreetse andmebaasikeele näide on SQL. Andmebaasisüsteeme, kus saab kasutada SQL keelt nimetan SQL-andmebaasisüsteemideks.

1960ndate alguses lõi Charles W. Bachman võrkmudelil põhinevate andmebaaside haldussüsteemi IDS (*Integrated Data Store*). Selle abil loodud andmebaas meenutab graafi, kus igale kirjele vastab sõlm graafis ning kirjete vahelisi seoseid kujutavad graafi servad. Samas pole selle süsteemi aluseks oleva andmemudeli näol tegemist graafi andmemudeliga. Kleppmann toob oma raamatus [10] ära neli põhilist tunnust, mille poolest võrkandmemudel erineb graafi andmemudelist. Esiteks on võrkandmemudel piiravam – andmebaasil on skeem, mis kirjeldab ära, kuidas on andmed omavahel seotud.

Teiseks pole soovitud kirjeni (sõlmeni) jõudmiseks võimalik kasutada selle unikaalset identifikaatorit, vaid tuleb läbida üks võimalikest teekondadest (juurdepääsutee) andmete otsijale etteantud sõlmest kuni otsitava sõlmeni. Kolmandaks on võrkandmemudeli korral kirje (sõlme) järglased sorteeritud. Viimaseks oluliseks erinevuseks on andmete otsimise võimalused. Enamikes graafiandmebaasisüsteemides on võimalik kasutada kõrgtasemelist deklaratiivset päringukeelt. Võrkandmemudelit realiseerivate andmebaasisüsteemide korral olid päringud imperatiivsed (kirjeldasid otsingu algoritmi) ja seega oli päringute kirjutamine keerukam ning skeemist sõltuv. Võrkandmemudelil põhinevaid andmebaase kasutati laialdaselt 1970ndatel ja 1980ndatel aastatel. [11, 10, p. 60]

Graafe on võimalik hoida ka dokumendimudelil põhinevates andmebaasides, kus andmeid hoitakse JSON või XML dokumentidena. Sellisel juhul salvestatakse iga sõlm eraldi dokumendina. Kui andmebaasisüsteem ei võimalda ühendamisoperatsioone, siis salvestatakse iga sõlme *s* korral selle naabrid *s* listi tüüpi atribuudi väärtusena. Naabrite listi hoidmine sõlme dokumendis pole aga efektiivne. Näiteks pole võimalik lugeda dokumenti ilma selle naabrite informatsioonita, mistõttu on paljude seostega sõlmi juba kulukas vaadatagi. Kui dokumendimudelil põhinevas andmebaasisüsteemis on võimalik kasutada ühendamisoperatsioone, võiks hoida informatsiooni seoste kohta eraldi dokumendis. Dokumenti saab vaadata kui suunatud ja järjestatud puud, kus sõlmedel on märgendid. Nii skeem kui andmed esitatakse samas dokumendis. Sõlmed, kust on võimalik edasi liikuda, defineerivad struktuuri ning lehed kujutavad endast andmeid. XML pakub viitamisvõimalust elementide vahel, mis võimaldab hoida suvalisi graafe. Viidataval elemendil on sel juhul atribuut identifikaatori jaoks ning viitavas elemendis on viite atribuut, mille väärtuseks on viidatava elemendi identifikaator. [9, 12] Ka JSONi korral on võimalikud dokumentide vahelised viited [13, 14]. Kui andmebaasisüsteem ei toeta nende alusel dokumentide ühendamist, siis peab selle ühendamise läbi viima dokumente lugev rakendus.

1980ndate alguses tutvustasid Hull ja Yap formaaditud mudelit, mis üritas üldistada relatsioonilist ja hierarhilist mudelit. Andmebaasi ehitusplokiks on formaadid e märgistega puud. Lehed vastavad relatsioonilise mudeli atribuutidele ning sisemised sõlmed kujutavad endas andmete vahelisi seoseid. Formaadi eksemplariks on formaadi alusel puustruktuuri paigutatud andmed. [15]

1980ndatel tuldi välja ideega loogilisest andmemudelist (*Logical Data Model*, LDM), mis kombineeriks omavahel hierarhilise andmemudeli, võrkandmemudeli ning relatsioonilise mudeli parimaid külgi. Andmebaasi skeem oleks modelleeritav suunatud multigraafina, kus lehed esitaksid atribuute ning sisemised sõlmed atribuutide vahelisi seoseid. Sarnaselt relatsioonilisele mudeli relatsioonarvutusele ja relatsioonialgebrale nähti ette loogiline (mitte-protseduurne) ja algebraline (protseduuriline) keel. [16] Nende keelte abil saaks lisaks päringutele defineerida vaateid ja jõustada kitsendusi.

1980ndatel sai alguse ka objektorienteeritud andmemudel, eesmärgiga võimaldada objektorienteeritud tarkvara töö käigus loodavate objektide võimalikult valutut säilitamist. Andmeid esitatakse selle mudeli alusel objektide kollektsioonidena, mis on organiseeritud klassidesse. Sarnaselt objektorienteeritud keeltele võib objekt sisaldada keeruka sisemise struktuuriga andmevälju, sealhulgas viiteid teistele objektidele. See võimaldab realiseerida üks-mitmele ja mitu-mitmele seoseid ning seega esitada ka graafistruktuuriga andmeid. Objektidega võivad olla seotud meetodid. [9] 2017. aasta kevade seisuga on objektorienteeritud andmemudelil põhinevad andmebaasisüsteemid jäänud nishitooteks ja 2017. aasta mai seisuga on populaarseim neist (Db4o) andmebaasisüsteemide populaarsuse indeksis 105. kohal. [1]

Graafile suunatud objekti (*Graph-Oriented Object Database*, GOOD) andmemudeli korral kujutatakse skeemi ja selle eksemplare suunatud graafidena. Erinevalt objektoritenteeritud andmebaasidest, ei esitata andmeid mitte viitade kaudu seotud objektidena objektorienteeritud programmeerimiskeele mõttes, vaid sõlmede ja servadega, st kõrgemal abstraktsioonitasemel. Olemeid (reaalse maailma füüsilisi või abstraktseid asju) kujutatakse graafi sõlmedena. Servad kujutavad olemite vahelisi seoseid ning olemite omadusi. Andmetöötlus on samuti graafipõhine. Andmete muutmiseks on neli põhilist graafioperatsiooni – sõlmede ja servade lisamis- ning kustutamisoperatsioonid. Lisaks on olemas võimalus duplikaatide eemaldamiseks. On võimalik kasutada ka erinevat tüüpi andmete jaoks oluliste operatsioonide läbiviimiseks mõeldud operaatoreid nagu näiteks arvutada päevade arvu kahe kuupäeva vahel. [17] Sellised operaatorid võimaldavad töötada servade või sõlmedega seotud andmetega.

Graafi andmemudelite (RDF, omaduste graaf) väljamõtlejad üritasid parandada ülejäänud andmemudelite piiratud võimalusi graafide andmete esitamiseks. Graafi andmemudelite korral esitatakse andmed ja/või skeem graafidena või andmestruktuuridena, mis

üldistavad graafi mõistet. Andmetöötluses kasutatakse graafile orienteeritud operatsioone realiseerivaid operaatoreid nagu näiteks naabrite leidmine, teekonna leidmine või alamgraafi leidmine. Graafi mudelil põhinevad andmebaasisüsteemid olid populaarsed juba 1990ndate esimeses pooles. Kuna paljud andmed on oma olemuselt graafistruktuuriga, on nende populaarsus viimasel ajal jälle tasapisi kasvama hakanud. Populaarseim graafi andmemudelit pakkuv andmebaasisüsteem – Neo4j – on 2017. aasta mai seisuga kõikide andmebaasisüsteemide seas populaarsuselt 21. kohal. [9, 1]

Erinevalt SQL-andmebaasisüsteemidest pole graafipõhistel andmebaasisüsteemidel standardset päringukeelt ega üht formaalset andmemudelit. Enamik süsteeme pakuvad rakendusliideseid (APIsid) populaarsete programmeerimiskeelte jaoks ning mõningatel süsteemidel on oma päringukeel. Mitmed andmebaasisüsteemid kasutavad SPARQL päringukeelt, mis on RDF (*Resource Description Framework*) graafimudeli standardne keel. RDF graafimudeli korral esitatakse andmed kolmikute (subjekt, predikaat, objekt) kujul. [18, 19]

Teiseks graafi andmemudeliks on omaduste graafi (*Property Graph*) mudel, milles saavad nii sõlmed kui servad olla seotud võti-väärtus paaridena esitatud omadustega. Tuntud konkreetseks andmebaasikeeleks sellisele andmemudelile on Cypher, mis nagu RDF päringukeel SPARQL, põhineb graafi otsimisel mustri alusel (*pattern matching*). Graafi otsimisel mustri alusel defineeritakse muster ja otsitakse kõiki sellele vastavaid graafe või selle alamgraafe [20]. Cypher aga ei paku mõningaid vajalikke graafi päringute funktsionaalsusi nagu regulaarse tee päringud (*regular path queries*). Regulaarse tee päring leiab teega ühendatud sõlmed, kus teel olevate servadega seotud andmed rahuldavad mingit regulaaravaldist. Seega on loodud uus päringukeel PGQL, mis kombineerib mustri alusel graafide otsimise SQLi süntaksi ja funktsionaalsusega ning toetab regulaarse tee päringuid ja graafi konstrueerimist. PGQL järgib tugevalt SQLi süntaktilist struktuuri ja pakub SQLi-laadset funktsionaalsust. Lisaks SQLi operatsioonidele pakub PGQL operaatoreid graafide mustrite sobitamiseks: sõlmede, servade ning teede sobitamine. [19]

Erinevat liiki andmete hoidmiseks on kõige sobivamad erinevatel andmemudelitel põhinevad andmebaasid. Mis siis, kui selliste andmetega peab töötama üks ja sama rakendus? Üks võimalus on panna rakendus suhtlema erinevate andmebaasisüsteemidega. See muudab rakenduse loomise ja andmete haldamise ning

administreerimise keerukamaks. Veel üks võimalus on kasutada mitme-mudelilisi andmebaasisüsteeme. Sellised andmebaasisüsteemid toetavad mitut andmemudelit, mille alusel loodud andmestruktuure on võimalik läbi ühe liidese kasutada. Sellised andmebaasisüsteemid on näiteks ArangoDB [21] ja OrientDB [22] Mõlemad toetavad graafi- ja dokumendimudelit ning võti-väärtus paare. [23] Probleemiks on, et sellised süsteemid pole saavutanud turul kauem olnud süsteemidega võrreldavat küpsust ning populaarsust ja võrreldes küpsete süsteemidega on suurem ka nende turult kadumise risk. ArangoDB ja OrientDB tulid turule vastavalt 2012. ja 2010. aastal ning 2017. aasta mai seisuga on need populaarsuselt 80. ja 46. kohal [1]. Kuna erinevalt SQL andmemudelist ei ole graafi- ja dokumendipõhiseid andmemudeleid tsentraalselt standardiseeritud, siis iga süsteem teeb asju väga omamoodi (SQL andmebaasisüsteemid teevad ka, aga vähem) ja see tõstab andmebaasisüsteemi väljavahetamise kulusid. Seega on üheks kaalumist väärt lähenemiseks ikkagi iga rakenduse jaoks ühel andmemudelil põhineva andmebaasi kasutamine. Selleks on vaja aga teadmisi ning oskuseid, kuidas erineva struktuuriga andmeid sellisesse andmebaasi kõige paremini "ära mahutada".

2.2.1 Graafide esitamine ja otsimine SQL-andmebaasides

1969. aastal kirjeldas Codd esmakordselt relatsioonilist mudelit [24]. Relatsioonilises andmemudelis on andmed esitatud relatsiooniliste muutujate väärtustena (relatsioonidena). Igal relatsioonil on päis ja kehand. Relatsiooni kehand koosneb hulgast korteežidest, millel on relatsiooni päisega määratud struktuur. Päise moodustavate atribuutide hulga ja kehandi moodustava korteežide hulga korral pole hulga elementide järjekorral tähendust. Enamik relatsioonilisi andmebaasisüsteeme pakuvad kasutajatele andmebaasikeelt SQL, mis põhineb, kuid ei järgi täielikult relatsioonilist mudelit. SQL aluseks on mudel, kus relatsiooni kujutatakse tabelina ja korteež kujutab üht andmerida tabelis. Erinevalt relatsioonilisest mudelist on SQLis read ja veerud järjestatud ning mõnikord on vaja seda järjekorda teada. Samuti kasutatakse SQLis puuduvate andmete esitamiseks NULL markereid, mida relatsiooniline mudel ette ei näe. [25, 26]

Relatsioonilises/SQL-andmebaasis võib andmeid hoida ka RDF kujul, st disainida relatsioonilised muutujad/tabelid, mille väärtusena hoida RDF kolmikuid. Artiklis [27] on võrreldud nelja sellist moodust andmete salvestusruumi kulu ja päringute täitmise kiiruste vaatepunktist. RDF jaoks on olemas päringukeel SPARQL. Paljud RDF hoidlad (nt Jena [28], 3Store [29]) kasutavad andmete hoidmiseks

SQL-andmebaase. On välja toodud mooduseid, kuidas SPARQL päringuid SQL päringuteks teisendada [27, 30]. Lisaks on olemas tööriistu, mis võimaldavad kasutada SPARQL päringuid SQL-andmebaasides: Ontop [31], D2RQ [32].

Oracle ja PostgreSQL on SQL-andmebaasisüsteemid, st nendes kasutatavaks andmebaasikeeleks on SQL ning andmeid organiseeritakse nendes SQL mudeli alusel. Tegemist on andmebaasisüsteemidega, mida kasutan käesoleva töö eksperimentaalses osas. Seega on vajalik mainida võimalusi, mida need süsteemid pakuvad graafe esitavate andmetega töötamiseks. Tabelis (Tabel 1) on toodud ära Oracles ja PostgreSQL'is olevad võimalused graafe esitavate andmete kasutamiseks. Mõlemad süsteemid toetavad SQL standardis ette nähtud WITH klauslit ja võimaldavad selle abil kasutada CTEsid (*Common Table Expression*) [33]. CTEst võib mõelda kui abipäringust, mis on defineeritud SELECT, INSERT, UPDATE või DELETE lause osana. CTEd ei salvestata eraldiseisva andmebaasiobjektina nagu baastabelit või hetktõmmist. Selle abil leitud tulemus kestab ainult päringu aja. CTE võib päringu jooksul iseennast mitmeid kordi välja kutsuda ning sellele võib sama lause piires mitu korda viidata. [34] Rekursiivsed päringud on päringud, mis viitavad rekursiivselt CTEle [35].

Tabel 1. Graafide esitamise ja otsimise võimalused Oracle ja PostgreSQL andmebaasisüsteemides

	Oracle	**PostgreSQL**
Rekursiivsed päringud	Võimaldab kasutada CTEsid (*Common Table Expression*). Kasutab selleks WITH klauslit. Tsüklitega toimetulekuks on seal klausel CYCLE. [36]	Võimaldab kasutada CTEsid (*Common Table Expression*). Kasutab selleks WITH klauslit. Tsüklite tuvastamiseks kasutatakse nimekirja, mis sisaldab teed antud sõlmeni. [37]
Päringud hierarhiate/ graafide põhjal enne WITH	Võimaldab teha hierarhilisi päringuid CONNECT BY operaatori abil. Versioonis 10g lisandus võtmesõna NOCYCLE, mis võimaldab avastada tsükleid,	Lisamoodul *tablefunc* võimaldab teha hierarhilisi päringuid *connectby* funktsiooni abil. Paistab, et antud funktsioon

	Oracle	PostgreSQL
klausli toe lisamist	tehes seega võimalikuks päringud tsükliliste graafidega. [38]	ei võimalda teha päringuid tsükliliste graafidega. [39]

Nii PostgreSQLis kui Oracle viimastes versioonides on võimalik hoida XML ja JSON formaadis andmeid [40, 41, 42, 43]. Tõepoolest, graafe saab esitada ka XML või JSON dokumentidena. Kuid kuna käesolevas töös selliseid disaine ei käsitleta (vt jaotis 4), siis ka siinkohal nendest võimalustest pikemalt ei kirjutata.

3 Varasemad uuringud

Selles peatükis annan ülevaate varem graafide SQL-andmebaasides esitamise kohta tehtud töödest. Toon välja teadustööd ja raamatud, kus on esitatud rohkem kui üks graafide SQL-andmebaasis esitamise disain. Annan ülevaate teadusartiklitest, kus on võrreldud SQL-andmebaasis graafide põhjal tehtud päringute kiiruseid graafi andmemudelitel põhinevate lahenduste kiirustega. Nende tööde otsimiseks kasutasin Google Scholarit ning järgmiseid võtmesõnu: *relational*, *database*, *benchmark*, *graph*, *model*, *comparison*, *SQL*.

Krönströmi magistritöös [4] vaadeldakse andebaasisüsteeme PostgreSQL 9.3.0 ja Oracle 12c Enterprise Edition Release 1. Töös tuuakse välja mustrid graafi alamliigi – hierarhia – esitamiseks nimetatud andmebaasisüsteemide abil loodud SQL-andmebaasides. Iga muster kirjeldab ühte võimalikku andmebaasi disaini. Kokku esitatakse kümme mustrit. Eksperiment viiakse läbi kolme disaini alusel: külgnevusnimistu, materialiseeritud tee ja pesastatud hulgad. Eksperimendi käigus võrreldakse päringute kiirust ja andmekäitluskeele koodi keerukust erinevate andmemahtude ja päringute puhul. Magistritöös järeldatakse, et enamasti on lihtsaimad päringud materialiseeritud tee korral. Samas võtavad selle disaini tabelid enim salvestusruumi. Muutmisoperatsioonide puhul on kiireimaid operatsioone võimaldavaks lahenduseks külgnevusnimistu. Keeruka koondpäringu täitmisel on töökiirus parim pesastatud hulkade disaini puhul. Samas on muutmisoperatsioonid selle disaini puhul kõige aeglasemad.

Celko raamatu [44] 37. peatükk on pühendatud graafide esitamisele SQL-andmebaasis. Välja tuuakse kaks disaini – külgnevusnimistu ja poolitatud sõlmedega pesastatud hulgad. Celko kirjeldab tabelite struktuuri ja mõningaid päringuid. Samuti toob ta välja algoritmi külgnevusnimistu disaini teisendamiseks pesastatud hulkade disainiks atsüklilise graafi puhul.

Blaha raamatu [45] kolmas ja neljas peatükk kirjeldavad graafide SQL-andmebaasides esitamiseks mõeldud disaine. Esimeses neist esitatakse suunatud graafide (neli disaini ja kahe disaini jaoks variatsioonid, mis võimaldavad hoida infot graafi elementide – sõlmed ja servad – kehtivusperioodide kohta) ja teises suunamata graafide (kaks disaini, millest ühe jaoks ka elementide kehtivusperioodidega variatsioon) disainid. Kehtivusperioodidega disainid võimaldavad säilitada infot graafi struktuuri muutumise

ajaloo kohta. Iga disaini kohta tuuakse välja kontseptuaalne andmemudel, kitsendused ja paar lihtsamat päringut. Lisaks illustreeritakse iga disaini korral, kuidas üks näitegraaf tabelite kujul välja näeb. Iga disaini jaoks on olemas ka UMLi klassidiagrammi abil esitatud näiteülesanne, mille esitamiseks antud disain sobib.

Artiklis [46] võrreldakse sotsiaalvõrgustike tüüpiliste päringute kiiruseid erinevates andmebaasisüsteemides: omaduste graafi mudelil põhinevad (Dex 4.7 ja Neo4j 1.8.2 Community), RDF mudelil põhinev (RDF-3X) ja SQL mudelil põhinevad (Virtuoso 7.0 ja PostgreSQL 9.1). Proovitakse luua võimalikult päriseluline andmebaasi struktuur ja testandmed. Andmebaasi struktuuri kohaselt on isikud seotud teiste isikutega suunamata servaga „sõber“ ning veebilehtedega suunatud serva „meeldib“ abil. Seega on antud skeemi eksemplariks atribuutidega mitmeosaline graaf, millel on suunatud ja suunamata servad ning kahte tüüpi sõlmed (isik, veebileht) ja servad (sõber, meeldib). SQL-andmebaasides luuakse iga sõlme- ja servatüübi kohta eraldi tabel. Mõlemad tekkinud graafid esitatakse külgnevusnimistu disainiga tabelites. RDF-baasis on sõlmede jaoks genereeritud URId ja atribuutide ning servade kohta genereeritud RDF-kolmikud. Iga süsteemi jaoks realiseeritakse katse kasutades Java't, kuna eesmärgiks on võrrelda sotsiaalvõrgustike rakenduste kiirust erinevate andmebaaside korral, mitte üksnes päringute kiirust. Seega luuakse lisaks andmebaasile ka neid kasutavad rakendused ja mõõdetakse mitte päringu kiirust andmebaasis, vaid aega, mis kulub rakendusest päringu tegemiseks. Seega kuulub mõõdetud aja sisse ka aeg, mis kulus rakendusel andmebaasiga ühendumiseks, päringu saatmiseks ning vastuse tagasi saamiseks. Katsed tehakse viie erineva andmehulga juures tuhandest kümne miljoni sõlmeni. Parimad tulemused saadi graafiandmebaasides. Vaadeldud SQL-andmebaasisüsteemidest oli enamikul juhtudel kiirem Virtuoso.

Artiklis [47] vaadeldakse kaht andmebaasisüsteemi: MySQL Community Server 5.1.42 (SQL) ja Neo4j version 1.0-b11 (graafipõhine). Kasutatav andmebaas on mõeldud andmete päritoluga seotud info hoidmiseks ning seega on tegemist suunatud atsüklilise graafiga. Töös viiakse läbi valdkonna ülesandeid lahendavaid päringuid. Lisaks päringute kiirusele võrreldakse ka andmebaasisüsteemide subjektiivsemaid tunnuseid nagu programmeerimise lihtsus ja andmete turvalisuse tagamise võimalused. Mõlema süsteemi abil luuakse 12 andmebaasi. Iga andmebaas hoiab endas ühte suunatud graafi. Andmebaasis esitatav informatsioon on minimaalne – iga sõlmega on seotud üks andmeväli. Genereeritud andmebaasid on nelja erineva suurusega (1000 – 100 000 sõlme)

ja iga suuruse kohta kolme erineva andmevälja tüübiga (täisarv, 8KB sõne, 32KB sõne). Päringud jagunevad kahte liiki: struktuuri- ja andmepäringud, mida mõlemat on kolm tükki. Struktuuripäringud on seotud graafi struktuuriga. Näiteks loetakse üle kõik eraldiseisvad sõlmed (sõlmed, millel puuduvad sissetulevad ja väljaminevad servad). Andmepäringu korral küsitakse andmeid kaarte või sõlmedega seotud andmeväljadest. Näiteks täisarvu tüübiga andmevälja korral loetakse üle sõlmed, kus andmevälja väärtus on võrdne etteantud väärtusega. Struktuuripäringute puhul oli Neo4j selgelt kiirem. Arvutüüpi andmevälja jaoks mõeldud andmepäringud töötasid kiiremini MySQL's. Sõnede korral oli MySQL veidi kiirem juhul, kui sõned koosnesid ainult tähtedest. Kui aga sõne sisaldas ka tühikuid, oli MySQL suuremate andmemahtude korral tunduvalt aeglasem kui Neo4j.

Artikkel [48] kirjeldab Dijkstra lühima tee algoritmi realisatsiooni kolmes erinevas süsteemis: Green-Marl, Neo4j 1.8.2 (graafipõhine andmebaasisüsteem) ja Oracle Database 12c (SQL-andmebaasisüsteem). Green-Marl on süsteem, kus kasutaja saab graafialgoritme kirjeldada valdkonnapõhises keeles ning mis tõlgib selle kirjelduse üldotstarbelisse programmeerimiskeelde nagu C++. Süsteemis on graafi sõlmed ja servad salvestatud CSR'na (*Compressed Sparse Row*). Igal platvormil realiseeritakse vastavale süsteemile kõige sobilikum algoritmi versioon. Süsteemide võrdlemiseks kasutatakse kahte sotsiaalvõrgustiku graafi: LIVEJ ja TWITTER. Uuringu tulemusena selgus, et sotsiaalvõrgustike süsteemi puhul oli Dijkstra lühima tee algoritmi optimeeritud implementatsioon SQL-andmebaasis ligilähedane või isegi kiirem kui parim implementatsioon Neo4j's. Parim algoritmi implementatsioon graafianalüüsiks arvuti muutmälus töötab tunduvalt paremini kui andmebaasipõhised lahendused.

Artiklis [49] võrreldakse ühe graafitöötluse põhiprobleemi – graafide mustrite alusel otsimise – lahenduste kiirusi erinevates andmebaasisüsteemides. Artiklis on võetud tuntud etalon LUBM ja konstrueeritud sellest sobiva struktuuriga andmebaasid erinevat tüüpi andmebaasisüsteemidele. LUBM on ülikooli valdkonda modelleeriv RDF süsteemide võrdlemiseks mõeldud etalon. See hõlmab andmeid ülikoolide, osakondade, teadusgruppide, õppejõudude ja üliõpilaste kohta. Katsed viiakse läbi kolmes erinevat tüüpi andmebaasisüsteemis: RDF süsteemides Virtuoso (Virtuoso pakub erinevaid andmemudeleid, sealhulgas RDF ja SQL andmemudelit) 6.1.8 ja 7.1.0 ja TripleRush, SQL-andmebaasisüsteemis Virtuoso 7.1.0 ning graafipõhistes andmebaasisüsteemides Neo4j 2.0.1 ja Sparksee 5.0.0. Katse käigus leiti, et graafide mustrite alusel otsimise

ülesannetes pole graafipõhised süsteemid teistest paremad – LUBM-8000 puhul täideti märgatavat osa päringutest üle ettenähtud 10 minuti. Väikese andmehulga juures olid TripleRush'i kiirused konkurentsivõimelised, aga suure mälukasutuse tõttu ei skaleerunud see suurtele andmemahtudele. Autorid leiavad, et graafide mustrite alusel otsimise ülesannete ja üleüldiselt graafide analüüsimiseks on parimad hübriidsüsteemid, mis laiendavad SQL süsteemi graafi-spetsiifiliste operatsioonidega ja keelega.

Artiklis [50] vaadeldakse veerupõhise andmete salvestamisega SQL-andmebaasisüsteemi Vertica. Võrreldakse graafi-analüütika operatsioonide kiirusi, mälukasutust ning kettale kirjutatud ning kettalt loetud baitide arvu Verticas ja graafitöötlusele pühendatud programmides Giraph ja GraphLab. Lisaks analüüsitakse Vertica päringute erinevaid realisatsioone. Näiteks võrreldakse omavahel jagatud mäluga (kogu graaf laetakse alguses ja salvestatakse jagatud mällu) kasutaja-defineeritud tabelifunktsiooni, kettapõhise (iga iteratsiooni tulemused salvestatakse kettale) kasutaja-defineeritud tabelifunktsiooni ja SQL-lausega tehtud päringute kiiruseid. Tabelifunktsiooni väärtuseks on tabel. Uuringus leiti, et SQL-andmebaasisüsteemid (eriti veerupõhise andmete salvestamisega) võimaldavad efektiivselt kombineerida graafide analüüsi relatsioonialgebra operatsioonidega nagu projektsioon ja ühendamine. Graafitöötlusele pühendatud programmi Giraph ja Vertica kiiruste võrdluses andsid päringud, mis sisaldasid nii graafi analüüsi kui ka relatsioonialgebra operatsioone, tunduvalt paremaid tulemusi Verticas.

4 Erinevad disainilahendused graafide esitamiseks SQL-andmebaasides

Graafide SQL-andmebaasides esitamist puudutavates töödes (vt peatükk 3) on järgmiseid puudujääke, mida soovin käesoleva tööga parandada.

- Ei ole allikaid, kus oleks kokku koondatud ja süstematiseeritud kõikvõimalikud graafide SQL-andmebaasides esitamise disainid. Üle ühe graafide esitamiseks mõeldud disaini on välja toodud Blaha [45] ja Celko [44] raamatutes.
- Disaine ei esitata mustri formaadi järgi struktureerituna. Struktuur aitab kaasa nii disaini paremale mõistmisele, kui võimaldab ka paremini erinevaid disaine võrrelda. Vaid Blaha raamatus [45] väljatoodud disainide esitusel on kindel struktuur (UML mall, IDEF1X mall, SQL päringute näited, tabelid näiteandmetega ja näide kasutusest).
- Ei ole ühte kohta kokku koondatud ülevaadet, millised disainid sobivad milliste omadustega graafide esitamiseks. Blaha [45] välja toodud disainidest on olemas koondtabel, kus on toodud iga disaini kokkuvõte koos sellega, millal antud disaini kasutada. Disainide kirjeldustes on enamasti öeldud, kas nad on suunatud graafide jaoks sobivad ning kui nendega pole võimalik esitada tsükleid sisaldavaid graafe.

Nagu eelnevalt kirjutatud, siis on võimalik esitada graafe ka dokumendimudeli põhjal. Kuna SQL-andmebaasides on võimalik salvestada XML ja JSON dokumente, siis on põhimõtteliselt ka SQL-andmebaaside korral võimalik graafe sellest lähtuvalt esitada. Selline lahendus kombineerib kokku SQL aluseks oleva ja dokumendimudeli ning kaldub pigem dokumendimudeli poole, sest sõlmed ja servad on maetud dokumendi sisse. SQL-andmebaasisüsteem pakub lihtsalt platvormi selliste disainide realiseerimiseks. Seetõttu jäävad sellised disainid käesolevast tööst välja. Minu hinnangul kuuluksid need töösse, mis uurib graafide esitamist dokumendimudeli järgi.

Krönströmi [4] uuris võimalusi hierarhiate esitamiseks SQL-andmebaasides kujul, kus hierarhias osalevate olemitüüpide atribuudid on esitatud eraldiseisvate tabelite või veergudena (st hierarhiate andmeid ei esitata JSON, XML või muus hierarhilises formaadis dokumentidena). Erandiks on materialiseeritud tee disain, mis kodeerib iga elemendi juurest mineva tee juursõlmeni ühe stringi abil. Graafide esitamise

disainilahenduste leidmise üheks meetodiks on hinnata iga Krönströmi töös välja toodud disainilahendust selle sobivuse seisukohalt graafide üldjuhu esitamiseks. Lubamaks rohkem graafe kui ainult puud / hierarhiad, on vaja tabelite disaini muuta.

Lisaks Krönströmi töös välja toodud hierarhiate esitamise viisidele, esitan mõned graafide SQL-andmebaasi disainid, mida hierarhiate töös pole käsitletud. Kasutasin nende leidmiseks Google Scholaris järgmiseid otsisõnu *relational*, *database*, *graph*, *dag*, *model*. Lisaks otsisin artikleid, mis viitavad juba leitud artiklitele ja vaatasin leitud artiklite allikaid.

Väljatoodud disaine omavahel kombineerides on võimalik saada uusi lahendusi, mida antud töös välja ei tooda. Lisaks on osade disainide puhul kirjanduses olemas ka variandid, mis võimaldavad hoida graafi ajalugu. Nende korral on võimalik hoida informatsiooni servade ja sõlmede eksisteerimise kohta kindlal ajahetkel. Neid variatsioone ei tooda samuti mustrite formaadis välja.

Iga leitud ja töösse valitud graafide esitamise disaini kohta toon välja järgnevad alampunktid.

Nimi eesti keeles: Eestikeelne nimi, mis on enamasti tuletatud Krönströmi töös olevast disaini eestikeelsest nimest.

Nimed inglise keeles: Ingliskeelsed nimed, mis on enamasti võetud lähteallikast või tuletatud Krönströmi töös olevatest disainide ingliskeelsetest nimedest.

Hierarhia esitamise disaini nimi inglise keeles: Juhul kui disaini leidmise aluseks oli hierarhiate esitamise disain, siis nimi, mida Krönström oma töös alajaotuste pealkirjadena kasutas.

Viited artiklitele: Viited kõigile leitud materjalidele, kus kirjeldatakse sellise disaini kasutamist graafide esitamiseks.

Graafide esitamine: Selgitan põhimõtet ja toon välja piirangud graafidele, mida selle disaini abil saab esitada.

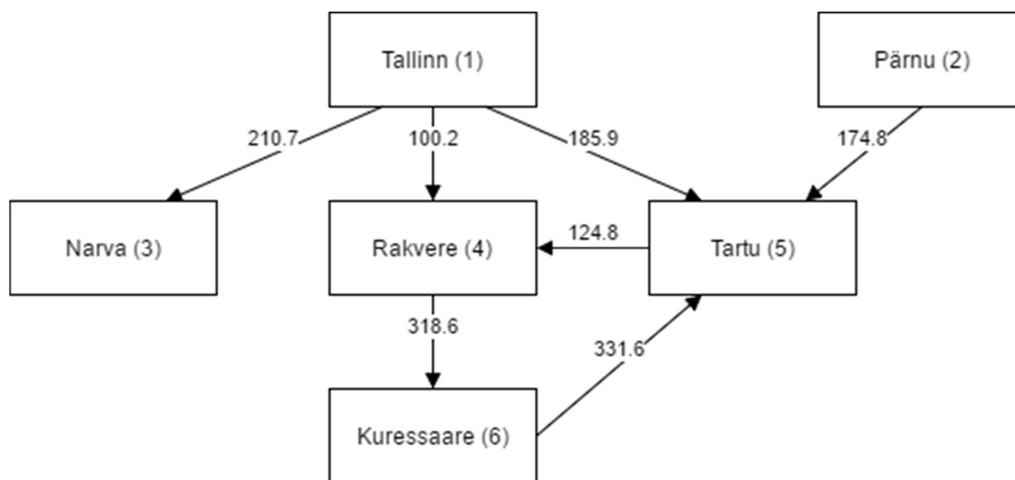
Eelised: Sellise esitusviisi eelised kirjanduse põhjal.

Puudused: Sellise esitusviisi puudused kirjanduse põhjal.

Näide: Iga disaini puhul toon näite, kuidas esitada selle alusel loodud tabelites ühte ja sama graafi. Kui mõnda disaini pole tsüklilise graafi esitamiseks võimalik rakendada, esitan näite ilma tsükliteta (atsüklilise) graafi jaoks. Tabeleid, mis korduvad erinevate mustrite näidetes (nt *City*) ei korrata, vaid neile viidatakse. Kõikide näidetes esitatud tabelite näol on tegemist baastabelitega – nimega tabelid, mis pole defineeritud teiste tabelite põhjal.

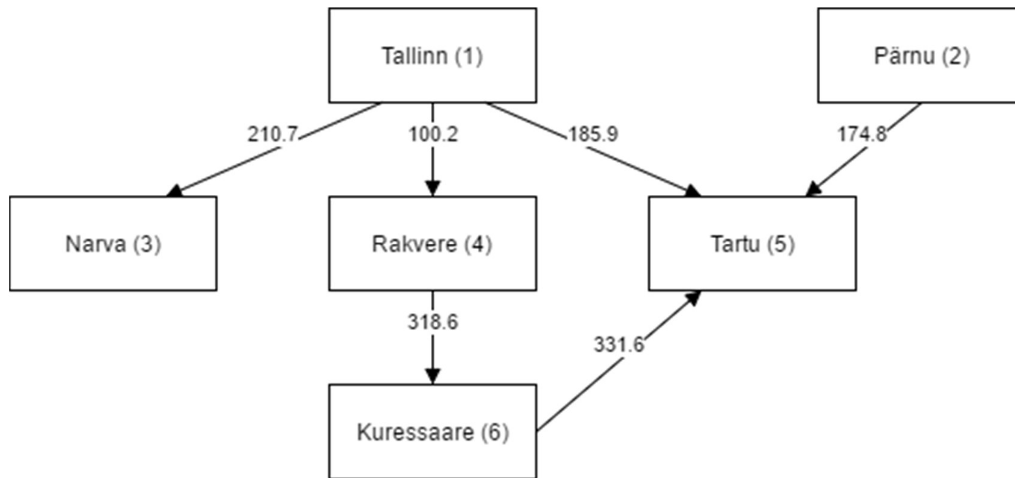
Iga tabeli päis esitab üldistatud väite (predikaadi) reaalse maailma olemite kohta. Iga näite tabeli juures toon välja selle predikaat – sõnaline kirjeldus inimkasutajale, mis peaks aitama tabelis olevate andmete tähendust paremini mõista. Selle predikaadi parameetritele vastavad veergude nimed ning loetavuse huvides esitan need predikaadis suurtähtedega.

Erinevate disainilahenduste korral kasutatav graaf on toodud Joonis 1. Tegemist on tsüklit sisaldava suunatud graafiga. Graafi sõlmedeks on mõned minu poolt vabalt valitud Eesti linnad ja kaartel on toodud vahemaa kilomeetrites, mis autoga sõitmisel ühest linna teise sõitmiseks Google Maps andmetel [51] läbida tuleb. Nooleotsa pool on see linn, kuhu poole sõidetakse. Iga linna juures on sulgudes toodud selle näites kasutatav numbriline identifikaator (primaarvõtme väärtus tabelis *City*). Kasutan neid identifikaatoreid näite tabelites.



Joonis 1. Linnadevaheliste vahemaade graaf

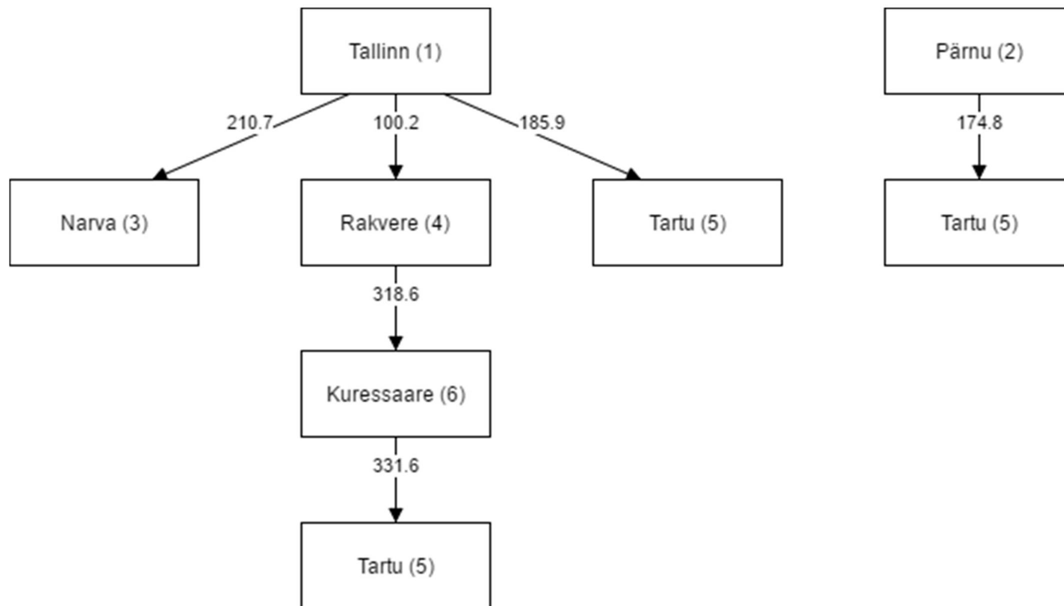
Osade disainide puhul on võimalik kujutada ainult atsüklilisi graafe. Nende puhul on kasutatud antud graafi lihtsustatumat varianti, kus puudub kaar Tartu ja Rakvere vahel (Joonis 2). Sulgudes on linna numbriline identifikaator.



Joonis 2. Atsükliline linnadevaheliste vahemaade graaf

Celko [44] toob poolitatud sõlmedega pesastatud hulkade disaini kirjelduse juures välja meetodi atsükliliste graafide teisendamiseks hierarhia kujule, kasutades selleks sõlmede poolitamist. Kuna antud meetodit kasutatakse atsükliliste graafide korral ka teiste hierarhiate jaoks kirjeldatud mustrite juures, toon siinkohal ära antud meetodi sõnalise kirjelduse konkreetse graafi näitel.

Sõlmede poolitamist alustatakse sõlmedest, millest enam edasi liikuda ei anna (antud juhul linnad Tartu ja Narva) ja poolitamisega liigutakse graafi sõlmede suunas, millel puuduvad sissetulevad kaared (antud juhul kõigepealt Tartust Pärnusse, Kuressaarde ning Tallinnasse ning Narvast Tallinnasse). Kui jõutakse sõlmeni, millesse sissetulevate kaarte hulk on rohkem kui üks, siis asendatakse see vastava arvu sõlme koopiatega ning iga sissetulev kaar seotakse ühega nendest. Näiteks Tartusse on kolm sissetulevat kaart ning Joonis 3 on Tartust kolm koopiat, millest igaühel on üks sissetulev kaar. Atsüklilise linnade graafi (Joonis 2) teisendamise teel tekkinud hierarhiad on näha Joonis 3. Sulgudes on linna numbriline identifikaator. [44, p. 693]



Joonis 3. Hierarhia kujule teisendatud atsükliline linnadevaheliste vahemaade graaf

Kommentaar suunamata graafide kohta: Eelnevalt toodud ja läbivalt kasutatud näide on suunatud graafide kohta. Kui mul on midagi lisaks öelda samades andmestruktuurides suunamata graafide hoidmise kohta, siis kirjutan selle kommentaari siia.

Strateegia: Toon välja disaini tabelitega seotud deklaratiivsed kitsendused, sh välis- ja primaarvõtmed.

4.1 Adjacency List

Nimi eesti keeles: Külgnevusnimistu

Nimed inglise keeles: Node and Edge Directed Graph

Hierarhia esitamise disaini nimi inglise keeles: Adjacency List, Degenerate Node and Edge

Viited artiklitele: [44, 45, 52]

Graafide esitamine: Celko väitel [44, p. 682] on külgnevusnimistu levinuim viis graafide esitamiseks SQL-andmebaasisüsteemides. Selle disaini korral on kaks põhitabelit: üks sõlmede ja teine nendevaheliste servade jaoks. [44, p. 682, 45, pp. 45-47]

Eelised:

- Võimaldab hoida infot nii sõlmede kui servade kohta, st nende kohta saab salvestada atribuutide väärtuseid. [45]
- Võimaldab samade sõlmede vahel mitut samasuunalist serva. [45]
- Lisamis-, muutmis- ja kustutamisoperatsioonid on lihtsad ja kiired. [52]
- Lihtne leida otseseid järglasi. [52]

Puudused:

- Kui süsteem ei võimalda kasutada rekursiooni, siis on keerukas teha päringuid selle kohta, millistesse sõlmedesse antud sõlmest saab. Kui rekursiooni ei saa kasutada, siis on vaja teha palju ühendamisoperatsioone ning on vaja teada sügavust, milleni otsinguga minna. [52]

Näide: Luuakse tabelid *City* (Joonis 4) ja *City_relation* (Joonis 5).

Tabeli *City* predikaat: Eksisteerib suuremat sorti Eesti linn numbrilise identifikaatoriga CITY_ID ja nimega CITY.

CITY_ID	CITY
1	Tallinn
2	Pärnu
3	Narva
4	Rakvere
5	Tartu
6	Kuressaare

Joonis 4. Külgnevusnimistu disaini tabel City koos näiteväärtustega

Tabeli *City_relation* predikaat: On olemas numbrilise identifikaatoriga CITY_RELATION_ID seos, et Eesti linnast numbrilise identifikaatoriga FROM_ID saab liikuda otse teise Eesti linna numbrilise identifikaatoriga TO_ID ning selleks tuleb läbida DISTANCE kilomeetri pikkune vahemaa.

CITY_RELATION_ID	FROM_ID	TO_ID	DISTANCE
1	1	3	210.7
2	1	4	100.2
3	1	5	185.9
4	2	5	174.8
5	4	6	318.6
6	5	4	124.8
7	6	5	331.6

Joonis 5. Külgnevusnimistu disaini tabel City_relation koos näiteväärtustega

Kommentaar suunamata graafide kohta: Suunamata graafi disain on toodud eraldi disainina *Sõlmede ja servadega suunamata graaf* (jaotis 4.2).

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

Tabeli *City* primaarvõtmeks on *(city_id)*, *City_relation* primaarvõtmeks *(city_relation_id)*.

Tabelile *City_relation* lisatakse veergudele *from_id* ja *to_id* välisvõtme kitsendused, mis viitavad tabeli *City* primaavõtmele.

Tabelile *City_relation* lisatakse unikaalsuse kitsendus (*from_id*, *to_id*, *distance*).

4.2 Node and Edge Undirected Graph

Nimi eesti keeles: Sõlmede ja servadega suunamata graaf

Nimed inglise keeles: -

Hierarhia esitamise disaini nimi inglise keeles: Adjacency List, Degenerate Node and Edge

Viited artiklitele: [45]

Graafide esitamine: Tegemist on külgnevusnimistu disaini variatsiooniga suunamata graafi jaoks. On kolm tabelit: sõlmede, servade ja nendevaheliste seoste tabel. Seoste tabelis on iga serva kohta kaks rida. [45, pp. 64-66]

Eelised:

- Võimaldab esitada suunamata graafi. [45]
- Lihtne disain.

Puudused [45]**:**

- Pole võimalik lisada serva sõlmest endasse.

Näide: Luuakse tabelid *City* (Joonis 4), *Distance* (Joonis 6) ja *City_connection* (Joonis 7).

Tabeli *Distance* predikaat: Leidub vahemaa numbrilise identifikaatoriga DISTANCE_ID ja pikkusega DISTANCE kilomeetrit.

DISTANCE_ID	DISTANCE
1	210.7
2	100.2
3	185.9
4	174.8
5	318.6
6	124.8

DISTANCE_ID	DISTANCE
7	331.6

Joonis 6. Sõlmede ja servadega suunamata graafi disaini tabel Distance koos näiteväärtustega

Tabeli *City_connection* predikaat: Eesti linnal numbrilise identifikaatoriga CITY_ID leidub ühendus, mille pikkuse numbriline identifikaator on DISTANCE_ID.

CITY_ID	DISTANCE_ID
1	1
3	1
1	2
4	2
1	3
5	3
2	4
5	4
4	5
6	5
5	6
4	6
6	7
5	7

Joonis 7. Sõlmede ja servadega suunamata graafi disaini tabel City_connection koos näiteväärtustega

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

Tabeli *City* primaarvõtmeks on *(city_id)*, *Distance* primaarvõtmeks *(distance_id)* ning *City_connection* primaarvõtmeks *(city_id, distance_id)*.

Tabelile *City_connection* lisatakse veergudele *city_id* ja *distance_id* välisvõtme kitsendused, mis viitavad vastavalt tabeli *City* ja *Distance* primaavõtmetele.

4.3 Simple Directed Graph

Nimi eesti keeles: Lihtne suunatud graaf

Nimed inglise keeles: -

Hierarhia esitamise disaini nimi inglise keeles: Simple Tree

Viited artiklitele: [45]

Graafide esitamine: Lihtsa suunatud graafi disaini põhitabeliks on sõlmede tabel. Graafil võib olla mitu juurt ja igal sõlmel mitu vanemat. Kuna nii tekib sõlmede vahel mitu-mitmele seos, on vaja lisada vahetabel seoste hoidmiseks sõlmede vahel. [45, pp. 37-39]

Eelised [45]**:**

- Lihtsa struktuuriga.

Puudused [45]**:**

- Ei võimalda lisada servadega seotud informatsiooni.
- Iga sõlmepaari vahel võib olla maksimaalselt üks samasuunaline serv.

Näide: Luuakse tabelid *City* (Joonis 4) ja *City_relation* (Joonis 8).

Tabeli *City_relation* predikaat: Eesti linnast numbrilise identifikaatoriga FROM_ID saab liikuda otse teise Eesti linna numbrilise identifikaatoriga TO_ID.

FROM_ID	TO_ID
1	3
1	4
1	5
2	5
4	6
5	4
6	5

Joonis 8. Lihtsa suunatud graafi disaini tabel City_relation koos näiteväärtustega

Kommentaar suunamata graafide kohta: Antud disaini jaoks pole suunamata graafi jaoks mõeldud vastet. Suunatud graafi disainiga sarnase disaini kasutamine oleks sümmeetria antimuster. Antimuster kirjeldab halba lahendust probleemile. Kui iga serva kohta oleks seoste tabelis üks rida, siis pole selge, milline veergudest peaks olema esimene ja milline teine (antud juhul *from_id* ja *to_id*). Kuna lisataks aga iga serva kohta mitu rida, siis kahekordistuks tabeli ridade arv ning muutmise korral oleks vaja muuta kaks korda rohkem ridu. [45, pp. 64; 97-99]

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

City tabeli primaarvõtmeks on *(city_id)*, *City_relation* tabeli primaarvõtmeks on *(from_id*, *to_id)*.

Tabelile *City_relation* lisatakse veergudele *from_id* ja *to_id* välisvõtme kitsendused, mis viitavad tabeli *City* primaarvõtmele.

4.4 Split Node Nested Sets

Nimi eesti keeles: Poolitatud sõlmedega pesastatud hulgad

Nimed inglise keeles: -

Hierarhia esitamise disaini nimi inglise keeles: Nested Sets

Viited artiklitele: [44, 52]

Graafide esitamine: Hierarhiat saab esitada joonisel hulkadena, kus järglased on kujutatud vanema alamhulgana. Igale sõlme kirjeldamiseks kasutatakse kahte täisarvu: vasak ja parem. Arvude määramine algab juursõlmest ja puule tehakse ring peale. Sõlmeni jõudes pannakse arv sellele sõlme poolele, mida parasjagu külastatakse ja suurendatakse loendurit. Lõpuks saab iga sõlm endale kaks arvu – ühe parema ja teise vasaku külje jaoks. [44, pp. 673-674]

Sõlmi poolitades on võimalik kasutada atsükliliste suunatud graafide esitamiseks ka pesastatud hulkade mudelit. Sõlmede poolitamise teel teisendatakse atsükliline graaf puu kujule nagu näha jooniselt (Joonis 3). [44, p. 693]

Eelised:

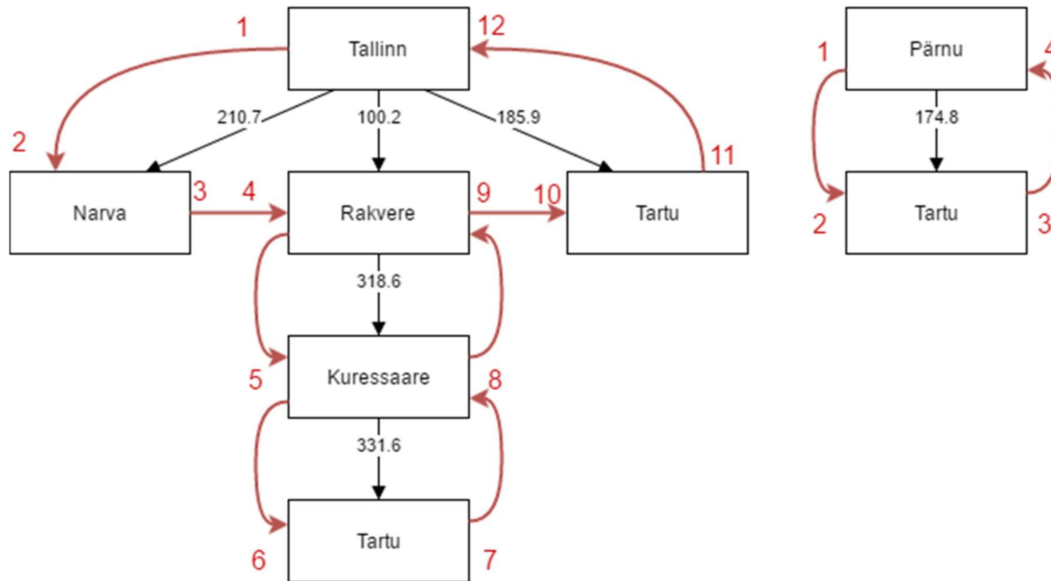
- Sõlme kõikide tasemete järglased on kiiresti leitavad. Sõlm [lft1, rgt1] on sõlme [lft2, rgt2] järeltulijaks juhul kui lft1 > lft2 ja rgt1 < rgt2. [44]
- Sõlme kõikide tasemete eellased on kergesti leitavad. Sõlm [lft2, rgt2] on sõlme [lft1, rgt1] eellaseks juhul kui lft2 < lft1 < rgt2. [52]
- Võimaldab teha päringuid ilma rekursiooni kasutamata. [44]

Puudused:

- Ei võimalda esitada tsüklilisi graafe [44].
- Võtab lisaruumi, kuna sõlmede poolitamise teel tekivad duplikaadid. [44]
- Servade lisamine ja kustutamine on ajakulukas, kuna on vaja lft ja rgt väärtused ümber arvutada. [44]

- Otseste eellaste ja järeltulijate leidmine on keerukas. See tähendab, et pole lihtne aru saada, milline leitud eellastest/järeltulijatest on sõlme otsene ning milline kaugem eellane/järeltulija. [52]

Näide: Joonis 9 näidatakse, kuidas leitakse lft ja rgt väärtused. Sõlme vasakul poolel olev arv näitab lft ja paremal poolel olev arv rgt väärtust. Iga graafi korral alustatakse nummerdamist 1-st.



Joonis 9. Linnadevaheliste vahemaade graafi nummerdamine poolitatud sõlmedega pesastatud hulkade disaini jaoks

Luuakse tabelid *City* (Joonis 4) ja *Graph* (Joonis 10).

Tabeli *Graph* predikaat: Leidub Eesti linnade vahelisi teid iseloomustav graaf numbrilise identifikaatoriga GRAPH_ID.

GRAPH_ID
1
2

Joonis 10. Poolitatud sõlmedega pesastatud hulkade disaini tabel Graph koos näiteväärtustega

Luuaks tabel *City_nested*_sets (Joonis 11). Veerg *Distance* näitab vahemaad vanemast vastaval real olevasse sõlme. Kuna antud näite puhul tekib hierarhiaks teisendamisel kaks puud, on tabelisse lisatud ka veerg *graph_id*, mis viitab tabelile *Graph*.

Tabeli *City_nested_sets* predikaat: Tehes Eesti linnade vaheliste teede graafile numbrilise identifikaatoriga GRAPH_ID vastupäeva ringi peale ja sõlmeni identifikaatoriga CITY_ID jõudes, pannakse järgmine arv LFT välja, kui jõuti sõlme vasakule poole ning RGT välja, kui jõuti paremale poolele. DISTANCE näitab vahemaad kilomeetrites, mis tuleb läbida otsesest eellasest olevast linnast numbrilise identifikaatoriga CITY_ID linna.

CITY_ID	**DISTANCE**	**LFT**	**RGT**	**GRAPH_ID**
1	0	1	12	1
2	0	1	4	2
3	210.7	2	3	1
4	100.2	4	9	1
5	331.6	6	7	1
6	318.6	5	8	1
5	185.9	10	11	1
5	174.8	2	3	2

Joonis 11. Poolitatud sõlmedega pesastatud hulkade disaini tabel City_nested_sets koos näiteväärtustega

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

City tabeli primaarvõtmeks on (*city_id)*. *City_nested_sets* tabeli primaarvõtmeks on (*lft*, *graph_id)*. *City_nested_sets* tabeli alternatiivvõtmeks on (*rgt*, *graph_id*).

City_nested_sets tabelil on kaks välisvõtit: *city_id* viitab tabeli *City* primaarvõtmele ja *graph_id* tabeli *Graph* primaarvõtmele.

City_nested_sets tabelile on lisatud CHECK kitsendused *lft* >= 1 ja *rgt* > *lft*.

Tabelile *City_nested_sets* on lisatud unikaalsuse kitsendus (*rgt*, *graph_id*).

4.5 Nested Intervals

Nimi eesti keeles: Pesastatud intervallid

Nimed inglise keeles: Matrix Encoding

Hierarhia esitamise disaini nimi inglise keeles: Nested Intervals

Viited artiklitele: [53, 54]

Graafide esitamine: Pesastatud intervallid on pesastatud hulkade üldistus. Nagu pesastatud hulkadegi puhul kodeeritakse iga sõlm arvupaariga. Kodeerimine on aga vabam ja kasutada võib kõiki reaalarve. See võimaldab lisada ning eemaldada sõlmi ilma, et peaks kogu graafi ümber kodeerima. Seda põhjusel, et kahe reaalarvu vahele on alati võimalik lisada uus reaalarv. Ainult täisarvude kasutamise korral võib aga tekkida olukord, kus oleks vaja uut täisarvu kahe järjestikuse täisarvu vahele lisada ning kuna see pole võimalik, tuleks graaf ümber kodeerida. Järglase intervall sisaldub alati tema vanema intervallis. Kui on teada järglase kodeering, siis vanema kodeering on võimalik selle põhjal arvutada. Seega on võimalik saada kätte kogu tee mingist sõlmest juursõlmeni ainult arvutamise teel. [53]

Kodeerimiseks on mitmeid erinevaid võimalusi. Üheks selliseks on Farey järjend. Farey järjend F_n, on järjestatud hulk nulli ja ühe vahel asuvatest taandamata murdudest a/b, kus a on mittenegatiivne b-st väiksem täisarv, b ja n on positiivsed täisarvud ning b on ühe ja n-i vahel. [55]

Mul ei õnnestunud leida ühtegi artiklit, kus oleks kirjeldatud pesastatud intervallide kasutamist graafide puhul. Kuna tegemist on pesastatud hulkade laiendusega, siis peaks teoreetiliselt olema võimalik sarnaselt sellele atsüklilisi suunatud graafe esitada.

Eelised [53]**:**

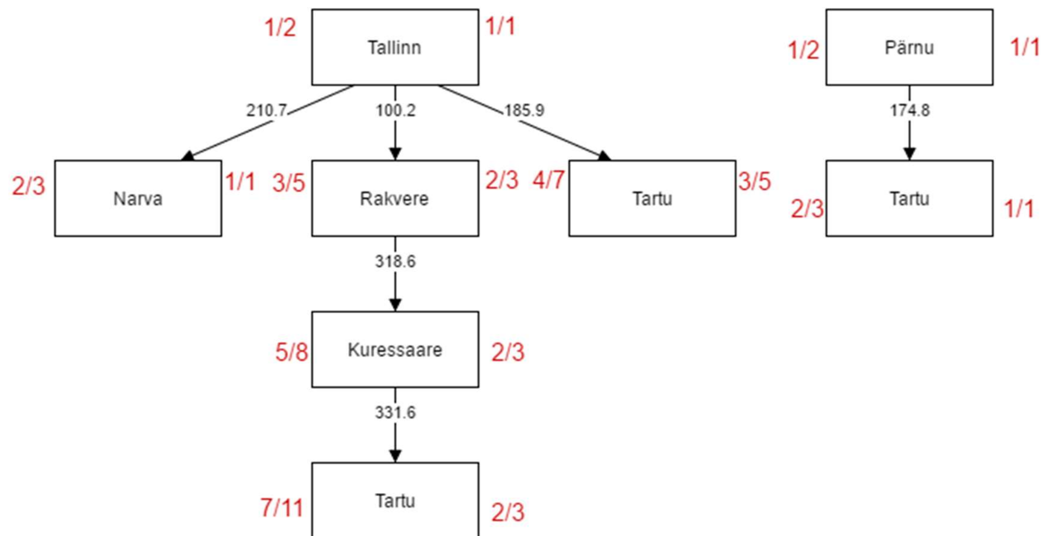
- Lihtne leida järglasi.
- Lihtne leida vanemaid, kuna sõlmed kodeeritakse algoritmi põhjal. Seega on võimalik välja arvutada vanema kodeering ning kogu tee sõlme kaugeima eellaseni.

- Servi lisades või eemaldades pole vaja kogu graafi ümber kodeerida.

Puudused:

- Ei võimalda esitada tsüklilisi graafe.
- Duplikaadid sõlmede poolitamisest.

Näide: Näide (Joonis 12) on realiseeritud atsüklilise graafi jaoks. Antud näites on intervalli kodeerimiseks kasutatud Farey järjendit. Vasakpoolne arv (*lft*) ei kuulu intervalli sisse, parempoolne (*rgt*) kuulub. Oletame, et mida vasakul pool sõlm joonisel on, seda vanem (ehk varem lisatud) järglane see on. Seega näiteks sõlm „Narva“ on vanem kui „Rakvere“. Sõlme sisestamisel otsitakse kõigepealt üles sama vanema kõige uuem otsene järglane. Kõige uuemaks otseseks järglaseks on see sõlm, mille *lft* väärtus on kõikide otseste järglaste seast väikseim. Ütleme, et selle leitud sõlme *lft* väärtuseks on a/b ning vanema *lft* väärtuseks on c/d. Sõlme enda *lft* väärtuseks saab (a+c)/(b+d). Näiteks hakkame sisestama sõlme „Rakvere“ ning graafis on eelnevalt olemas vaid sõlmed „Tallinn“ ja „Narva“. Sõlme vanemaks on „Tallinn“, mille *lft* väärtus on 1/1. Sõlme „Tallinn“ nooreimaks järglaseks antud ajahetkel on „Narva“, mille *lft* väärtuseks on 2/3. Seega saab sõlme *lft* väärtuseks (2+1)/(3+2)=2/5. Kui sisestatakse esimest järglast, siis kasutatakse viimati lisatud sõlme *lft* väärtuse asemel vanema *rgt* väärtust. Näiteks sõlme „Narva“ sisestades on olemas ainult tema vanem „Tallinn“. Seega saab sõlme *lft* väärtuseks (1+1)/(1+2)=2/3. Sõlme *rgt* väärtust on võimalik leida kasutades vaid sõlme enda *lft* väärtust. Kõikide d’de korral ühest *lft* nimetajani, üritatakse leida mod(*lft* lugeja * d + 1, *lft* nimetaja), mis annaks tulemuseks nulli. Esimese sellise leitud d korral saab *rgt* lugejaks (*lft* lugeja * d + 1 / *lft* nimetaja) ning nimetajaks d. Näiteks leiame sõlme „Rakvere“ *rgt* väärtuse. Esimeseks sobivaks d väärtuseks on 3, mille korral saame mod(3*3+1,5)=mod(10,5)=0. *Rgt* nimetajaks saab seega (3*3+1)/5=2 ning lugejaks 3. Ehk sõlme „Rakvere“ *rgt* väärtuseks saab 2/3. [54]



Joonis 12. Pesastatud intervallide kodeerimine

Luuakse tabelid *City* (Joonis 4), *Graph* (Joonis 10) ja *City_nested_intervals* (Joonis 13).

Tabeli *City_nested_intervals* predikaat: GRAPH_ID on Eesti linnade vaheliste teede graafi numbriline identifikaator. DISTANCE näitab vahemaad kilomeetrites, mis tuleb läbida otsesest eellasest olevast linnast numbrilise identifikaatoriga CITY_ID linna. Järglase intervall sisaldub alati käesoleva sõlme intervallis (LFT, RGT].

CITY_ID	**DISTANCE**	**LFT**	**RGT**	**GRAPH_ID**
1	0	1/2	1/1	1
2	0	1/2	1/1	2
3	210.7	2/3	1/1	1
4	100.2	3/5	2/3	1
5	331.6	7/11	2/3	1
6	318.6	5/8	2/3	1
5	185.9	4/7	3/5	1
5	174.8	2/3	1/1	2

Joonis 13. Pesastatud intervallide disaini tabel City_nested_intervals koos näiteväärtustega

Strateegia: Strateegia on samasugune, kui *poolitatud sõlmedega pesastatud hulkade* disaini juures (vt jaotis 4.4).

Veerud *lft* ja *rgt* on PostgreSQL's tüüpi DECIMAL(p,s) ja Oracle's tüüpi NUMBER(p,s). Ka teistes andmebaasisüsteemides tuleks eelistada kümnendmurde ujukomatüüpi

(FLOAT) arvudele. Seda põhjusel, et ujuvkomakohtadega arve omavahel võrreldes võivad tulla ebatäpsed tulemused. [56]

4.6 Flat Table

Nimi eesti keeles: Lametabel

Nimed inglise keeles: -

Hierarhia esitamise disaini nimi inglise keeles: Flat Table

Viited artiklitele: [57]

Graafide esitamine: Hierarhiat on võimalik kujutada listina, kus taane kujutab vanema-järglase suhet. Antud disain on sellest lähtuvalt kujundatud. Andmed esitatakse ühes tabelis, kus iga sõlme jaoks on eraldi rida. Igal real on sõlme järjekorranumber nimekirjas, taande tase ja viide vanemale. [57]

Antud disaini vaadates tundub, et tsüklilist graafi sellega realiseerida ei saa. Samas nagu me eelnevalt nägime, on võimalik atsüklilist graafi teisendada hierarhiaks, mis võimaldab ka selle disaini realiseerimist atsüklilise suunatud graafi jaoks.

Eelised:

- Lihtne leida otseseid vanemaid ja järglasi.

Puudused:

- Ei võimalda esitada tsüklilisi graafe. [57]
- Sõlmede lisamine ja kustutamine on ajakulukas. Iga operatsiooni korral tuleb muuta listis sõlmest allpool olevate (suurema järjekorranumbriga) sõlmede järjekorranumbrit. [57]
- Kui leidub palju mitme vanemaga sõlmi, siis tekib palju duplikaate. Mida kõrgemal hierarhia tasemel selline sõlm asub, seda rohkem duplikaate tekib.

Näide: Antud näite saab kirjutada lahti järgneva listina, kus esimesel kohal on linna nimi ja teisel kohal teekond vanemast antud sõlme.

Tallinn 0

Narva 210.7

Rakvere 100.2

Kuressaare 318.6

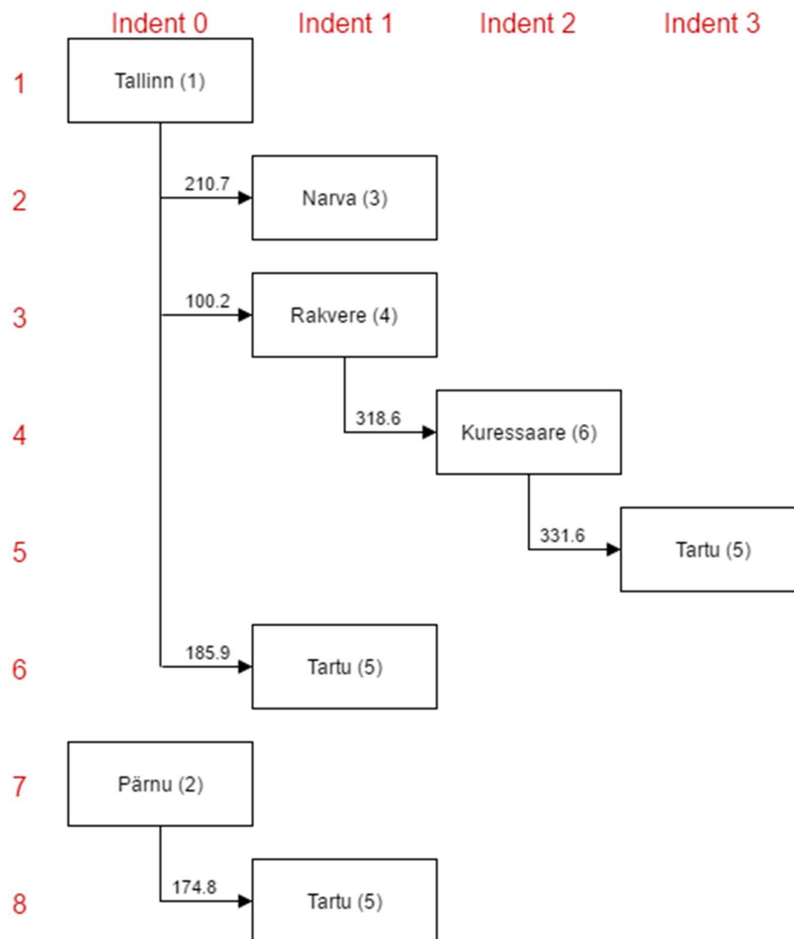
Tartu 185.9

Tartu 331.6

Pärnu 0

Tartu 174.8

Joonisel (Joonis 14) on näha hierarhiaks teisendatud atsüklilise graafi joonis paigutatuna selliselt, et oleks aru saada, kuidas lametabeli disaini tabel on tekkinud. Sulgudes olev number näitab linna numbrilist identifikaatorit. Joonise ülaosas olevad arvud näitavad taanet (*City_list* veerg *indent*). Vasakul olevad arvud näitavad linna järjekorranumbrit nimekirjas (*City_list* veerg *order*).



Joonis 14. Lametabeli disain

Luuakse tabelid *City* (Joonis 4) ja *City_list* (Joonis 15).

Tabeli *City_list* predikaat: Leidub listi element numbrilise identifikaatoriga CITY_LIST_ID, et Eesti linna numbrilise identifikaatoriga CITY_ID on võimalik saada otse teisest Eesti linnast PARENT_ID, läbides vahemaa DISTANCE kilomeetrit. Linnad on võimalik panna järjekorda selliselt, et linna järjekorranumber nimekirjas on ORDER. Linnal on INDENT eellast.

CITY_LIST_ID	PARENT_ID	CITY_ID	DISTANCE	ORDER	INDENT
1		1	0	1	0
2		2	0	7	0
3	1	3	210.7	2	1
4	1	4	100.2	3	1
5	1	5	331.6	6	1
6	4	6	318.6	4	2
7	6	5	185.9	5	3
8	2	5	174.8	8	1

Joonis 15. Lametabeli disaini tabel City_list koos näiteväärtustega

Strateegia: Kõik veerud peale parent_id on kohustuslikud (NOT NULL).

Tabeli *City* primaarvõtmeks on (*city_id)*, *City_list* primaarvõtmeks on (*city_list_id)*. Tabeli *City_list* veerg *city_id* on välisvõti, mis viitab tabeli *City* primaarvõtmele. Veerg *parent_id* on välisvõti, mis viitab tabeli *City_list* primaarvõtmele.

City_list tabelis on CHECK kitsendus *city_id* != *parent_id*.

Tabelis *City_list* on unikaalsuse kitsendus *(order).* Antud näites on *order* väärtus nummerdatud järjest üle kõikide graafide. Teiseks võimaluseks oleks lisada veerg *graph_id* ning iga graafi kohal nummerdust uuesti ühest alustada. Sel juhul oleks unikaalsuse kitsenduseks *(graph_id, order)*.

4.7 Multiple Lineage Columns

Nimi eesti keeles: Hulk pärilikkuse veerge

Nimed inglise keeles: David Chandler Model

Hierarhia esitamise disaini nimi inglise keeles: Multiple Lineage Columns

Viited artiklitele: [58]

Graafide esitamine: Antud disain kasutab ühte tabelit. Tabelis on iga hierarhia taseme jaoks eraldi veerg. Lisaks on veerg puu taseme jaoks, mis näitab ära, mitmenda hierarhia taseme elemendiga on tegemist. Samas pole see veerg hädavajalik, kuna taseme saab leida ka hierarhia tasemete veergude järgi. Puu identifikaator võimaldab salvestada mitme puu informatsiooni samas SQL tabelis. [58]

Antud disainilahendusega pole võimalik tsüklitega graafi esitada. Sarnaselt ka mitmele eelpool toodud disainile, tuleb atsükliline graaf kõigepealt hierarhiaks teisendada, et seda oleks võimalik antud disainiga esitada.

Eelised:

- Kiire leida vanemaid ja järeltulijaid. [58]
- Kiire lisada, kustutada ja muuta lehti. [58]

Puudused:

- Ei võimalda esitada tsüklilisi graafe. [58]
- Sisemiste sõlmede lisamine, kustutamine ja muutmine on ajakulukas, kuna on vaja muuta ka alamsõlmi. [4]
- Uue taseme lisandumisel on vaja muuta tabeli struktuuri (lisada uus veerg). [4]
- Piiratud tasemete arv, kuna veergude arv tabelis on piiratud. [4]

Näide: Luuakse tabelid *City* (Joonis 4), *Graph* (Joonis 10) ja *City_relation* (Joonis 16).

Tabeli *City_relation* predikaat: Eesti linnade vahelisi teid näitavas graafis numbrilise identifikaatoriga GRAPH_ID leidub seos numbrilise identifikaatoriga CITY_RELATION_ID, et tasemel LEVEL olevasse Eesti linna numbrilise identifikaatoriga CITY_ID saab sellele eelneva taseme linnast otse, läbides vahemaa DISTANCE kilomeetrit. L1_ID näitab ära kõige kaugema ning L[LEVEL-1]_ID linnale kõige otsese vanema numbrilise identifikaatori.

CITY_RELATION_ID	GRAPH_ID	LEVEL	CITY_ID	DISTANCE	L1_ID	L2_ID	L3_ID
10	1	1	1	0			
11	2	1	2	0			
12	1	2	3	210.7	1		
13	1	2	4	100.2	1		
14	1	2	5	185.9	1		
15	1	3	6	318.6	1	4	
16	1	4	5	331.6	1	4	6
17	2	2	5	174.8	2		

Joonis 16. Hulk pärilikkuse veerge disaini tabel City_relation koos näiteväärtustega

Strateegia: Kõik veerud peale *l1_id*, *l2_id* ja *l3_id* on kohustuslikud (NOT NULL).

Tabeli *City* primaarvõtmeks on (*city_id)*, tabelil *City_relation* on primaarvõtmeks (*city_relation_id).*

Veerg *city_id* on välisvõti, mis viitab tabeli *City* primaarvõtmele ning *graph_id* on välisvõti, mis viitab tabeli *Graph* primaarvõtmele.

Tabelile *City_relation* lisatakse unikaasuse kitsendus *(graph_id, level, city_id, distance, l1_id, l2_id, l3_id*).

Tsüklite vältimiseks on lisatud on CHECK kitsendus, et veergude *l1_id*, *l2_id*, *l3_id* ja *city_id* väärtused oleksid erinevad.

4.8 Hardcoded Graph

Nimi eesti keeles: Tase-tabel

Nimed inglise keeles: -

Hierarhia esitamise disaini nimi inglise keeles: Hardcoded Tree

Viited artiklitele: [45]

Graafide esitamine: Tase-tabeli disaini puhul esitatakse iga hierarhia tase eraldi tabelis. Igal tasemetabelil on vanema välisvõti eelmise taseme tabelisse. [45, pp. 12-13]

Antud disain pole graafide esitamiseks eriti sobilik. Nagu mitmed teised disainidki võimaldab see ainult hierarhiaks teisendatud atsükliliste graafide esitamist.

Eelised [45]**:**

- Kergesti arusaadav.

Puudused [45]**:**

- Ei võimalda esitada tsüklilisi graafe.
- Andmete struktuuri muutmisel on vaja teha muudatusi andmebaasi struktuuris. Näiteks uue taseme lisandumisel tuleb lisandunud taseme jaoks luua uus tabel.
- Järglaste ja eellaste leidmiseks on vaja läbi viia palju ühendamisoperatsioone.

Näide: Luuakse tabelid *City* (Joonis 4), *City_level_1* (Joonis 17), *City_level_2* (Joonis 18), *City_level_3* (Joonis 19) ja *City_level_4* (Joonis 20).

Tabeli *City_level_1* predikaat: Leidub teekondade graafi mõttes esimese taseme Eesti linn, taseme numbrilise identifikaatoriga LEVEL_1_ID ja numbrilise identifikaatoriga CITY_ID, kuhu ei saa ühestki teisest linnast.

LEVEL_1_ID	CITY_ID
1	1
2	2

Joonis 17. Tase-tabeli disaini tabel City_level_1 koos näiteväärtustega

Tabeli *City_level_2* predikaat: Leidub teekondade graafi mõttes teise taseme Eesti linn, taseme numbrilise identifikaatoriga LEVEL_2_ID ja numbrilise identifikaatoriga CITY_ID, kuhu saab liikuda otse esimese taseme Eesti linnast taseme numbrilise identifikaatoriga LEVEL_1_ID läbides DISTANCE kilomeetri pikkuse vahemaa.

LEVEL_2_ID	CITY_ID	DISTANCE	LEVEL_1_ID
1	3	210.7	1
2	4	100.2	1
3	5	185.9	1
4	5	174.8	2

Joonis 18. Tase-tabeli disaini tabel City_level_2 koos näiteväärtustega

Tabeli *City_level_3* predikaat: Leidub teekondade graafi mõttes kolmanda taseme Eesti linn, taseme numbrilise identifikaatoriga LEVEL_3_ID ja numbrilise identifikaatoriga CITY_ID, kuhu saab liikuda otse teise taseme Eesti linnast taseme numbrilise identifikaatoriga LEVEL_2_ID, läbides DISTANCE kilomeetri pikkuse vahemaa.

LEVEL_3_ID	CITY_ID	DISTANCE	LEVEL_2_ID
1	6	318.6	2

Joonis 19. Tase-tabeli disaini tabel City_level_3 koos näiteväärtustega

Tabeli *City_level_4* predikaat: Leidub teekondade graafi mõttes neljanda taseme Eesti linn, taseme numbrilise identifikaatoriga LEVEL_4_ID ja numbrilise identifikaatoriga CITY_ID, kuhu saab liikuda otse kolmanda taseme Eesti linnast taseme numbrilise identifikaatoriga LEVEL_3_ID, läbides DISTANCE kilomeetri pikkuse vahemaa.

LEVEL_4_ID	CITY_ID	DISTANCE	LEVEL_3_ID
1	5	331.6	1

Joonis 20. Tase-tabeli disaini tabel City_level_4 koos näiteväärtustega

Strateegia: Kõikide loodud tabelite primaarvõtmeks on (*level_n_id)*, kus n on vastava taseme number.

Tabeli *City_level_n* (kus n>1) veerg level_(*n-1)*_id on välisvõti, mis viitab tabeli *City_level_(n-1)* primaarvõtmele.

4.9 Structured Directed Graph

Nimi eesti keeles: Struktureeritud suunatud graaf

Nimed inglise keeles: -

Hierarhia esitamise disaini nimi inglise keeles: Structured Tree

Viited artiklitele: [45]

Graafide esitamine: Struktureeritud graafi disaini puhul eraldatakse sisemisi sõlmi (haru e *branch*) sõlmedest, kust enam edasi liikuda pole võimalik (leht e *leaf*). On neli tabelit: sisemiste sõlmede, lehtede ja üldine sõlmede tabel ning servade tabel. Üldises sõlmede tabelis on mõlemal sõlmetüübil (haru ja leht) olevatele atribuutidele vastavad veerud. Lehtede ja sisemiste sõlmede tabelite primaarvõtmeks on üldtabeli kandidaatvõti. Antud disaini võiks kasutada juhul, kui sisemistel sõlmedel ja lehtedel on erinevad atribuudid, seosed või semantika. [45, pp. 40-43]

Eelised [45]**:**

- Lihtne teada saada, kas antud sõlmest saab kuhugi edasi liikuda.
- Sisemistel sõlmedel ja lehtedel võivad olla erinevad atribuudid.

Puudused [45]**:**

- Võimalik ainult üks serv kahe sõlme vahel.

Näide: Luuakse tabel *City*.

Tabeli *City* predikaat: Eksisteerib Eesti linn numbrilise identifikaatoriga CITY_ID ja nimega CITY, mille kauguste graafi sõlme tüübiks on NODE_TYPE.

CITY_ID	CITY	NODE_TYPE
1	Tallinn	*branch*
2	Pärnu	*branch*
3	Narva	*leaf*
4	Rakvere	*branch*

CITY_ID	**CITY**	**NODE_TYPE**
5	Tartu	*leaf*
6	Kuressaare	*branch*

Joonis 21. Struktureeritud suunatud graafi disaini tabel City koos näiteväärtustega

Luuakse tabel *Middle_city*, kus hoitakse sisemiste sõlmede identifikaatoreid (sõlme tüüp on *branch*).

Tabeli *Middle_city* predikaat: Eksisteerib Eesti linn numbrilise identifikaatoriga CITY_ID, kust on kauguste graafi mõttes võimalik saada teistesse linnadesse.

CITY_ID
1
2
4
6

Joonis 22. Struktureeritud suunatud graafi disaini tabel Middle_city koos näiteväärtustega

Luuakse tabel *End_city*, kus hoitakse lehtede identifikaatoreid (sõlme tüüp *leaf*).

Tabeli *End_city* predikaat: Eksisteerib Eesti linn numbrilise identifikaatoriga CITY_ID, kust pole kauguste graafi mõttes võimalik teistesse linnadesse saada.

CITY_ID
3
5

Joonis 23. Struktureeritud suunatud graafi disaini tabel End_city koos näiteväärtustega

Luuakse tabel *City_relation*, kus on kirjeldatud sõlmede vahelised seosed (Joonis 8).

Kommentaar suunamata graafide kohta: Antud disain ei sobi suunamata graafi esitamiseks, kuna oleks sümmeetria antimuster. [45, p. 64]

Strateegia: Tabelite *City*, *Middle_city* ja *End_city* primaarvõtmeks on *(city_id)*. *City_relation* tabeli primaarvõtmeks on *(from_id, to_id)*.

Tabelite *Middle_city* ja *End_city* veerg *city_id* on välisvõti, mis viitab tabeli *City* primaarvõtmele. *City_relation* tabelis olevad veerud *from_id* ja *to_id* on välisvõtmed, mis viitavad tabeli *City* primaarvõtmele.

4.10 Connection directed graph

Nimi eesti keeles: Seoste suunatud graaf

Viited artiklitele: [45]

Graafide esitamine: Antud disainis on kolm tabelit: sõlmede, servade (ühendus lähte ja sihi vahel) ja ühenduste (seos sõlme ja serva vahel) tabel. Disaini kasutatakse juhul, kui on vaja hoida infot ühenduse kohta. Iga ühendus on kas lähe (*source*) või siht (*sink*). [45, pp. 48-51]

Eelised [45]**:**

- Võimaldab hoida infot ühenduse kohta.

Puudused:

- Keerukamad päringud.
- Keeruline tagada, et iga servaga oleks seotud kaks sõlme, millest üks oleks lähte ning teine siht tüüpi.

Näide: Luuakse tabelid *City* (Joonis 4), *Distance* (Joonis 6) ja *City_connection* (Joonis 24).

Tabeli *City_connection* predikaat: Eesti linnal numbrilise identifikaatoriga CITY_ID ja serva kontekstis omatava tüübiga SOURCE_OR_SINK leidub ühendus CITY_CONNECTION_ID, mille pikkuse numbriline identifikaator on DISTANCE_ID. SOURCE_OR_SINK „source“ korral on linn ühendusteel lähteks, „sink“ korral sihiks.

CITY_CONNECTION_ID	CITY_ID	DISTANCE_ID	SOURCE_OR_SINK
1	1	1	*source*
2	3	1	*sink*
3	1	2	*source*
4	4	2	*sink*
5	1	3	*source*
6	5	3	*sink*
7	2	4	*source*

CITY_CONNECTION_ID	CITY_ID	DISTANCE_ID	SOURCE_OR_SINK
8	5	4	*sink*
9	4	5	*source*
10	6	5	*sink*
11	5	6	*source*
12	4	6	*sink*
13	6	7	*source*
14	5	7	*sink*

Joonis 24. Seoste suunatud graafi disaini tabel City_connection koos näiteväärtustega

Kommentaar suunamata graafide kohta: Kuna seosed on mõlemasuunalised, siis seoste tabelis pole vaja veergu (antud näite korral *source_or_sink*), mis näitaks ära, kas tegemist on lähte või suudmega. [45, pp. 66-69]

Strateegia: Tabelite *City*, *Distance* ja *City_connection* primaarvõtmeteks on vastavalt (*city_id)*, (*distance_id)* ja (*city_connection_id)*.

Tabelis *City_connection* on kaks välisvõtit. Veerg *city_id* viitab tabelis *City* primaarvõtmele ning *direction_id* tabeli *Direction* primaarvõtmele.

Iga *Distance* reaga peab olema seotud kaks *City_connection* rida.

4.11 Path directed graph

Nimi eesti keeles: Teega suunatud graaf

Viited artiklitele: [59]

Graafide esitamine: Antud disain on mõeldud atsükliliste graafide esitamiseks. Disain koosneb kolmest tabelist: sõlmede, servade ja teede omast. Sõlmede tabel sisaldab infot iga sõlme kohta. Servade tabelis on otsesed seosed sõlmede vahel. Teede tabel näitab ära kõik sõlmede vahelised seosed. See näitab ära, kas suvalise kahe sõlme vahel leidub tee. [59]

Eelised [59]**:**

- Teede tabel võimaldab kiirelt teada saada, kas sõlmede vahel leidub tee.

Puudused [59]**:**

- Vajalik hoolitseda selle eest, et terviklikkus kolme tabeli vahel oleks tagatud. Seega on sõlmede vahel olevate servade lisamise ja kustutamise operatsioonid keerukad.

Näide: Kuna antud disain on mõeldud atsükliliste graafide esitamiseks, kasutatakse atsüklilise graafi joonist (Joonis 2).

Luuakse tabelid *City* (Joonis 4), *City_relation* (Joonis 25) ja *City_path* (Joonis 26).

Tabeli *City_relation* predikaat: Eesti linnast numbrilise identifikaatoriga FROM_ID on võimalik kauguste graafi mõttes saada otse teise Eesti linna numbrilise identifikaatoriga TO_ID läbides DISTANCE kilomeetri pikkuse vahemaa.

FROM_ID	TO_ID	DISTANCE
1	3	210.7
1	4	100.2
1	5	185.9
2	5	174.8

FROM_ID	TO_ID	DISTANCE
4	6	318.6
6	5	331.6

Joonis 25. Teega suunatud graafi disaini tabel City_relation koos näiteväärtustega

Tabeli *City_path* predikaat: Eesti linnast numbrilise identifikaatoriga FROM_ID on kauguste graafi mõttes võimalik kas otse või läbides teisi linnu saada teise Eesti linna numbrilise identifikaatoriga TO_ID.

FROM_ID	TO_ID
1	3
1	4
1	5
1	6
2	5
4	5
4	6
6	5

Joonis 26. Teega suunatud graafi disaini tabel City_path koos näiteväärtustega

Kommentaar suunamata graafide kohta: Kasutades suunamata graafi jaoks sama disaini kui suunatud graafil, oleks see sümmeetria antimuster (vt jaotis 4.3).

Disaini suunamata graafi jaoks sobivaks teisendamine võiks teoreetiliselt käia sarnaselt *sõlmede ja servadega suunamata graafi* disainile (vt jaotis 4.2). Sõlmede tabelis oleks sõlmede ning servade tabelis servade info. Lisaks oleks tabel teede ning kaks tabelit seoste jaoks: üks otseste ja teine kõikide seoste jaoks. Teede tabelis hoitaks teede identifikaatoreid. Lisaks võib seal ka hoida nt tee pikkust. Sümmeetria antimustri vältimiseks oleks iga seose kohta kaks rida. Otseste seoste tabelis oleksid veergudeks sõlme ja serva identifikaator. Kaudsete seoste tabelis oleks aga üheks veeruks sõlme identifikaator ning teiseks tee identifikaator. Selline disain oleks aga raskesti arusaadav ning hallatav.

Strateegia: Tabeli *City* primaarvõtmeks on (*city_id)*. Tabelite *City_relation* ja *City_path* primaarvõtmeteks on (*from_id*, *to_id)*.

Veerud *from_id* ja *to_id* on välisvõtme veerud, mis viitavad tabeli *City* primaarvõtmele.

City_path tabelis on CHECK kitsendus *from_id* != *to_id*.

4.12 Child List Directed Graph

Nimi eesti keeles: Järglaste nimekirjaga suunatud graaf

Viited artiklitele: [60]

Graafide esitamine: Antud disaini puhul on kasutusel ainult üks tabel. Seal on salvestatud igale reale info ühe sõlme kohta. Rea ühes veerus on nimekiri linnadest, kuhu antud sõlmest on suunatud serv. [60]

Eelised:

- Võimalik on ühe lausega sisestada ja leida nii sõlm kui ka kõik servad, milles sõlm on lähteks.

Puudused:

- Disain ei võimalda salvestada lisainformatsiooni servade kohta (antud juhul kaugus).

- Sõlme kustutamisel tuleb ise selle eest hoolitseda, et kustutada teiste sõlmede juurest viited sellele sõlmele.

Näide: Luuakse tabel *City* (Joonis 27).

Tabeli *City* predikaat: Eesti linnast numbrilise identifikaatoriga CITY_ID ja nimega CITY on kauguste graafi mõttes võimalik saada otse teistesse Eesti linnadesse numbriliste identifikaatoritega TO_IDS.

CITY_ID	CITY	TO_IDS
1	Tallinn	3 4 5
2	Pärnu	5
3	Narva	
4	Rakvere	6
5	Tartu	4
6	Kuressaare	5

Joonis 27. Järglaste nimekirjaga suunatud graafi disaini tabel City koos näiteväärtustega

Strateegia: Tabeli *City* primaarvõtmeks on (*city_id)*.

Veerg *to_ids* on Oracle andmebaasisüsteemis tüübiga NESTED TABLE. Selle tüübi abil saab realiseerida mitmeväärtuselisi atribuute.

4.13 Triple-store RDF

Nimi eesti keeles: Kolmikute hoidla

Viited artiklitele: [27]

Graafide esitamine: On üks kolme veeruga (subjekti, predikaadi ja objekti jaoks) tabel. Iga RDF kolmik on üks rida sellest tabelist. [27]

Eelised [27]**:**

- Tuleb toime dünaamilise skeemiga, kuna kolmikuid saab sisestada ilma eelneva teadmiseta, millised predikaadid või subjekti tüübid on kasutusel.

Puudused [27]**:**

- Efektiivne andmete otsimine nõuab spetsiaalseid tehnikaid.

Näide: Luuakse tabel *Triple_store* (Joonis 28).

Tabeli *Triple_store* predikaat: Subjektil SUBJECT (v tähendab sõlme (*vertex*) ja e serva (*edge*)) leidub omadus PREDICATE väärtusega OBJECT.

SUBJECT	PREDICATE	OBJECT
v1	*name*	Tallinn
v2	*name*	Pärnu
v3	*name*	Narva
v4	*name*	Rakvere
v5	*name*	Tartu
v6	*name*	Kuressaare
e1	*source*	v1
e1	*sink*	v3
e1	*distance*	210.7
e2	*source*	v1
e2	*sink*	v4
e2	*distance*	100.2
e3	*source*	v1

SUBJECT	PREDICATE	OBJECT
e3	*sink*	v5
e3	*distance*	185.9
e4	*source*	v2
e4	*sink*	v5
e4	*distance*	174.8
e5	*source*	v4
e5	*sink*	v6
e5	*distance*	318.6
e6	*source*	v5
e6	*sink*	v4
e6	*distance*	124.8
e7	*source*	v6
e7	*sink*	v5
e7	*distance*	331.6

Joonis 28. Kolmikute hoidla disaini tabel Triple_store koos näiteväärtustega

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

Tabeli primaarvõti on *(subject, predicate, object)*.

4.14 Type-oriented RDF

Nimi eesti keeles: Tüübile orienteeritud RDF

Viited artiklitele: [27]

Graafide esitamine: Iga olemitüübi kohta (nt isik, kaubaartikkel) luuakse oma kolmikute tabel.

Eelised [27]**:**

- Efektiivsemad päringud kui üldise kolmikute hoidla puhul.

Puudused [27]**:**

- Vaja muuta skeemi, kui lisandub uut tüüpi subjekt (lisada uus tabel).
- Tabelite arv võib väga suureks muutuda, kui on palju erinevaid olemitüüpe.

Näide: Luuakse tabelid *City* (Joonis 29) ja *Connection* (Joonis 30).

Tabeli *City* predikaat: Linnal identifikaatoriga SUBJECT leidub omadus PREDICATE väärtusega OBJECT.

SUBJECT	PREDICATE	OBJECT
v1	*name*	Tallinn
v2	*name*	Pärnu
v3	*name*	Narva
v4	*name*	Rakvere
v5	*name*	Tartu
v6	*name*	Kuressaare

Joonis 29. Tüübile orienteeritud RDFi disaini tabel City koos näiteväärtustega

Tabeli *Connection* predikaat: Seosel identifikaatoriga SUBJECT leidub omadus PREDICATE väärtusega OBJECT.

SUBJECT	PREDICATE	OBJECT
e1	*source*	v1
e1	*sink*	v3

SUBJECT	PREDICATE	OBJECT
e1	*distance*	210.7
e2	*source*	v1
e2	*sink*	v4
e2	*distance*	100.2
e3	*source*	v1
e3	*sink*	v5
e3	*distance*	185.9
e4	*source*	v2
e4	*sink*	v5
e4	*distance*	174.8
e5	*source*	v4
e5	*sink*	v6
e5	*distance*	318.6
e6	*source*	v5
e6	*sink*	v4
e6	*distance*	124.8
e7	*source*	v6
e7	*sink*	v5
e7	*distance*	331.6

Joonis 30. Tüübile orienteeritud RDFi disaini tabel Connection koos näiteväärtustega

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

Tabelite primaarvõti: *(subject, predicate, object)*.

4.15 Predicate-oriented RDF

Nimi eesti keeles: Predikaadi-põhine RDF

Viited artiklitele: [27]

Graafide esitamine: Iga predikaadi kohta (nt elab, töötab) luuakse eraldi tabel. Tabelites on kaks veergu – üks subjekti, teine objekti jaoks. [27]

Eelised [27]**:**

- Efektiivsemad päringud kui kolmikute hoidla puhul.

Puudused [27]**:**

- Vaja muuta skeemi, kui lisandub uut tüüpi predikaat (lisada uus tabel).
- Kui on palju erinevaid predikaate, siis tabelite arv võib väga suureks muutuda.

Näide: Luuakse tabelid *Name* (Joonis 31), *Source* (Joonis 32), *Sink* (Joonis 33) ja *Distance* (Joonis 34).

Tabeli *Name* predikaat: Subjektil SUBJECT leidub nimi OBJECT.

SUBJECT	OBJECT
v1	Tallinn
v2	Pärnu
v3	Narva
v4	Rakvere
v5	Tartu
v6	Kuressaare

Joonis 31. Predikaadi-põhise RDFi disaini tabel Name koos näiteväärtustega

Tabel *Source* predikaat: Subjektil SUBJECT leidub lähe OBJECT.

SUBJECT	OBJECT
e1	v1
e2	v1
e3	v1

SUBJECT	OBJECT
e4	v2
e5	v4
e6	v5
e7	v6

Joonis 32. Predikaadi-põhise RDFi disaini tabel Source koos näiteväärtustega

Tabeli *Sink* predikaat: Subjektil SUBJECT leidub suue OBJECT.

SUBJECT	OBJECT
e1	v3
e2	v4
e3	v5
e4	v5
e5	v6
e6	v4
e7	v5

Joonis 33. Predikaadi-põhise RDFi disaini tabel Sink koos näiteväärtustega

Tabeli *Distance* predikaat: Subjektil SUBJECT leidub vahemaa pikkusega OBJECT kilomeetrit.

SUBJECT	OBJECT
e1	210.7
e2	100.2
e3	185.9
e4	174.8
e5	318.6
e6	124.8
e7	331.6

Joonis 34. Predikaadi-põhise RDFi disaini tabel Distance koos näiteväärtustega

Strateegia: Kõik veerud on kohustuslikud (NOT NULL).

Tabelitele lisatakse primaarvõti: *(subject, object)*.

Keerukas kontrollida, et igale lähtele vastaks suue.

4.16 Entity-oriented RDF

Nimi eesti keeles: Olemile orienteeritud RDF

Viited artiklitele: [27]

Graafide esitamine: Sama olemiga seotud andmeid hoitakse üksteise lähedal, võimalusel ühes reas. Tabel koosneb olemi veerust ja k-st predikaadi ja väärtuse veergude paarist. Igale predikaadile määratakse dünaamiliselt üks predikaadi veerg, kuhu sellele vastavaid väiteid registreeritakse. Ühele veerule võib olla määratud mitmeid predikaate. Ühel olemil võib olla ühe predikaadi kohta mitmeid väärusi. Näiteks võib ühel inimesel olla mitmeid kontakttelefone. Selliste väärtuste jaoks kasutatakse lisatabelit, kus on kaks veergu: hulga identifikaator ja väärtus. Põhitabelis on sel juhul viide hulga identifikaatorile. [27]

Eelised [27]**:**

- Uue RDF andmetüübi (predikaadi, subjekti tüübi) lisandumisel pole vaja muuta skeemi.
- Kõik olemi väärtused on salvestatud üksteisele lähedal.
- Predikaat x on iga olemi korral samas veerus.

Puudused:

- Keerukam realiseerida kui teisi RDF disaine.
- Tabel võib mahukaks minna, kuna kõik andmed samas tabelis.

Näide: Luuakse tabel *Entity* (Joonis 35).

Tabeli *Entity* predikaat: Olemil SUBJECT leiduvad omadused PRED1, PRED2 ja PRED3 vastavate väärtustega OBJ1, OBJ2 ja OBJ3.

SUBJECT	PRED1	OBJ1	PRED2	OBJ2	PRED3	OBJ3
v1	*name*	Tallinn				
v2	*name*	Pärnu				

SUBJECT	PRED1	OBJ1	PRED2	OBJ2	PRED3	OBJ3
v3	*name*	Narva				
v4	*name*	Rakvere				
v5	*name*	Tartu				
v6	*name*	Kuressaare				
e1	*source*	v1	*sink*	v3	*distance*	210.7
e2	*source*	v1	*sink*	v4	*distance*	100.2
e3	*source*	v1	*sink*	v5	*distance*	185.9
e4	*source*	v2	*sink*	v5	*distance*	174.8
e5	*source*	v4	*sink*	v6	*distance*	318.6
e6	*source*	v5	*sink*	v4	*distance*	124.8
e7	*source*	v6	*sink*	v5	*distance*	331.6

Joonis 35. Olemile orienteeritud RDFi disaini tabel Entity koos näiteväärtustega

Strateegia: Veerg subject on kohustuslik (NOT NULL).

4.17 Ainult hierarhiate jaoks sobivad disainid

Eelnevas peatükis nägime, et enamikke hierarhiate jaoks mõeldud disaine on võimalik kasutada ka graafide esitamiseks. Isegi kui mõningate hierarhiate disainide kohta ei leidunud kirjandust graafide üldjuhu esitamiseks, siis oli neid võimalik ise tuletada kasutades Celko poolt välja toodud meetodit hierarhia teisendamiseks. Samas leidus ka mõningaid disaine, mida on võimalik kasutada ainult hierarhiate esitamiseks.

Antud jaotises on välja toodud Krönströmi töös esitatud hierarhiate disainid, mida pole võimalik graafide üldjuhu jaoks kasutada.

4.17.1 Closure Table

Nimi eesti keeles: Sulunditabel

Nimed inglise keeles: Bridge Table

Andmete esitamine: Sulunditabeli puhul luuakse lisatabel, kus hoitakse kõiki seoseid sõlmede vahel ning tuuakse välja, kui pikk on tee sõlmede vahel. [61]

Antud disaini kasutamise kohta üldisemate graafide korral ma kirjandust ei leidnud. Hierarhia jaoks kirjeldatud kujul pole seda võimalik graafide üldjuhule rakendada, kuna ühest sõlmest teise võib leiduda mitu erineva pikkusega teed. Üheks võimaluseks antud disaini atsüklilisele graafile kohandamiseks oleks jätta ära sügavuse hoidmise veerg, mis juhul saadaks sarnane disain kui jaotises 4.11 kirjeldatud *teega suunatud graafi* disain.

4.17.2 Materialized Path

Nimi eesti keeles: Materialiseeritud tee

Nimed inglise keeles: Path Enumaration, Lineage Column

Andmete esitamine: Antud mudelis salvestatakse iga sõlme juures teekond juursõlmest temani. [44, p. 670]

Materialiseeritud tee ei võimalda olukordi, kus sõlmel on mitu eellast, kuna tee veerus saab esitada ainult üht teekonda sõlmeni. Seega ei võimalda see disain esitada isegi atsüklilist graafi. [62]

Samas leian, et teoreetiliselt on võimalusi, kuidas ka materialiseeritud tee disaini korral graafi andmeid esitada. Üheks selliseks võiks olla *hulk pärilikkuse veerge* (vt jaotis 4.7) disainile sarnane disain, kus eellaste viiteid sisaldavad veerud oleks kõik ühte veergu teena kokku pandud. See võimaldaks hoida ainult hierarhiaks teisendatud atsüklilisi graafe. Teiseks võimaluseks oleks kasutada teede hoidmise veerul andmetüüpi, mis võimaldab esitada info kõikide teede kohta. Näiteks Oracle puhul saaks kasutada NESTED TABLE andmetüüpi, mille väärtuseks oleks teede hulk. Kõiki võimalikke teid sõlmeni hoitaks ühes sellist tüüpi veerus. Sel juhul poleks vaja atsüklilist graafi eelnevalt hierarhiaks teisendada.

4.18 Disainide kokkuvõte

Esitan antud jaotises ülevaate vaadeldud disainide sobivusest erinevate omadustega graafide esitamiseks. Nagu eelnevalt mainitud, on atsüklilist graafi võimalik teisendada hierarhiaks, mis võimaldab nende realiseerimiseks kasutada samu disaine nagu hierarhiate esitamiseks.

Kõikides välja toodud disainides, peale järglaste nimekirjaga suunatud graafi, on võimalik salvestada ka servadega seotud informatsiooni.

Tabel 2 näitab, milliseid disaine on võimalik kasutada milliste omadustega graafide esitamiseks. Nende disainide hulgast valin välja ka need, mida järgnevates peatükkides realiseerima ja võrdlema hakkan.

Esitan tabeli lahtrites väiteid disainide sobivuse kohta. Kui see väide põhineb mõnel kirjalikul allikal, siis on selle juures esitatud viited allikatele, kust see väide pärineb. Kui viidet juures ei ole, siis on tegemist minu väitega, mis põhineb disaini uurimisel saadud ettekujutusel. Ma ei ole leidnud ühtegi allikat, kus erinevate SQL-andmebaaside graafi disainide kohta oleks selline info kokkuvõtlikult välja toodud.

Tabel 2. Graafide SQL-andmebaasides esitamiseks mõeldud disainide võrdlus

Disain	Võimalik esitada tsüklilist graafi	Võimalik esitada suunatud graafi	Võimalik esitada suunamata graafi	Võimalik esitada multigraafi	Võimalik esitada kaalutud graafi
Adjacency List	**Jah** [44, 45]	**Jah** [44, 45]	Ei [45] (sellele vastab suunamata graafi disain *Node and Edge Undirected Graph*)	**Jah** [45]	**Jah** [44, 45]
Node and Edge Undirected Graph	**Jah** [45] (va serv iseendasse)	Ei [45] (sellele vastab suunatud graafi disain *Adjacency List*)	**Jah** [45]	**Jah** [45]	**Jah** [45]

Disain	**Võimalik esitada tsüklilist graafi**	**Võimalik esitada suunatud graafi**	**Võimalik esitada suunamata graafi**	**Võimalik esitada multigraafi**	**Võimalik esitada kaalutud graafi**
Simple Directed Graph	**Jah** [45] *	**Jah** [45]	Ei [45]	Ei [45]	Ei [45]
Split Node Nested Sets	Ei	**Jah** [44]	Ei	**Jah**	**Jah**
Nested Intervals	Ei	**Jah**	Ei	**Jah**	**Jah**
Flat Table	Ei	**Jah**	Ei	**Jah**	**Jah**
Multiple Lineage Columns	Ei	**Jah**	Ei	**Jah**	**Jah**
Hardcoded Graph	Ei	**Jah**	Ei	**Jah**	**Jah**
Structured Directed Graph	**Jah** [45] *	**Jah** [45]	Ei [45]	Ei [45]	Ei [45]
Connection Directed Graph	**Jah** [45]	**Jah** [45]	**Jah** [45]	**Jah** [45]	**Jah** [45]
Path Directed Graph	Ei	**Jah** [45]	**Jah**	**Jah**	**Jah**
Child List Directed Graph	**Jah**	**Jah** [45]	**Jah**	Ei	Ei
Triple-store RDF	**Jah** [63]	**Jah** [27]	**Jah**	**Jah** [27]	**Jah** [64]
Type-oriented RDF	**Jah** [63]	**Jah** [27]	**Jah**	**Jah** [27]	**Jah** [64]
Predicate-oriented RDF	**Jah** [63]	**Jah** [27]	**Jah**	**Jah** [27]	**Jah** [64]

Disain	Võimalik esitada tsüklilist graafi	Võimalik esitada suunatud graafi	Võimalik esitada suunamata graafi	Võimalik esitada multigraafi	Võimalik esitada kaalutud graafi
Entity-oriented RDF	**Jah** [63]	**Jah** [27]	**Jah**	**Jah** [27]	**Jah** [64]

* - allikas väidab, et kuna põhineb puu disainil, siis tsüklite lubamine ei paista sobilik.

Milline on minu subjektiivne hinnang nendele disainidele? Kui on vaja hoida lihtsalt suunatud tsüklilisi graafe, kus servade kohta ei soovita informatsiooni hoida ning sõlmepaari vahel võib olla ainult üks seos, siis tundus kõige loomulikuma lahendusena *Simple Directed Graph*. Suunamata graafide hoidmiseks paistis loomulikuim disain *Node and Edge Undirected Graph.* Muudel juhtudel paistis mulle kõige loomulikuma valikuna disain *Adjacency List*. Kõige keerukama lahendusena paistis minu jaoks *Nested Intervals* disain.

Mingi ülesande lahendamiseks sobiva graafi disaini valimisel lähtuksin ma järgmistest kriteeriumitest (tähtsuse järjekorras).

- Disain peab võimaldama esitada lähteülesandele vastavat graafi.
- Enim kasutatavad päringud ja andmemuudatuse operatsioonid peaksid olema kiired.
- Päringute kirjutamine võiks olla võimalikult lihtne. Päringud peaksid olema kergesti arusaadavad.
- Tabelite ja indeksite andmemaht võiks olla võimalikult väike.
- Andmete struktuur oleks lihtne (oleks võimalik tabelitele peale vaadates aru saada sõlmede vahelistest seostest).

5 Eksperiment

Peatükis kirjeldatakse läbiviidavat eksperimenti, et selgitada tehtavat praktilist katsetamise tööd ning võimaldada seda soovi korral korrata.

5.1 Eksperimendi eesmärk

Varasemates graafide SQL-andmebaasides esitamist puudutavates töödes (vt peatükk 3) on järgnevad puudujäägid.

- Ei ole võrreldud omavahel erinevate disainide alusel loodud andmebaasides toimuvate andmekäitluskeele lausete täitmise kiiruseid ja lausete koodi keerukust.
- Oracle andmebaasisüsteemi on vaadeldud ainult Dijkstra lühima tee algoritmi kiiruste võrdlemisel.

Eksperimendis võrreldakse disainilahendusi lennumarsruutide andmebaasi näitel. Võtan andmed *OpenFlights* andmebaasist [65] ja ülesandest lähtuvalt valin ka võrreldavad disainid. Millised on lennumarsruutide graafi omadused?

- Graafi sõlmedeks on lennujaamad ning nende servadeks on nendevahelised marsruudid.
- Lennuliinil on kindel lähte- ja sihtkoht, seega on tegemist suunatud graafiga.
- Ühest sihtkohast alustades on võimalik sinna tagasi jõuda, seega graafis võib esineda tsükleid.
- Lennuliiniga on seotud informatsioon vahepeatuste arvu kohta – seega on tegemist kaalutud graafiga.
- Ühel lennuliinil võivad pakkuda lendu erinevad lennufirmad, seega on tegemist multigraafiga e graaf peab võimaldama sama sõlmepaari vahel mitut seost.

Vaadates disainide kokkuvõtte tabelisse (Tabel 2), leidub ainult kaks disaini, mis rahuldaks kõiki tingimusi – *külgnevusnimistu (Adjacency List)* (jaotis 4.1) ja *seoste suunatud graaf (Connection Directed Graph)* (jaotis 4.10). Võrdlusest jätan välja RDF disainid, kuna tegemist pole puhaste SQL-andmebaaside jaoks mõeldud disainidega.

5.2 Eksperimendi kirjeldus

Eksperimendis luuakse kontseptuaalse andmemudeli (Joonis 36) alusel kaks erinevat andmebaasi – üks Oracle ja teine PostgreSQL andmebaasisüsteemis. PostgreSQL andmebaasisüsteemis luuakse iga disaini kohta samas andmebaasis eraldi skeemid nimedega vastavalt *kylgnevus* ja *seos*. Oracle andmebaasisüsteemis pannakse disainilahendused ühte skeemi (C##TUD10). Oracle andmebaasi tabelite nimedele pannakse eesliited vastavalt disaini nimele: KYLGNEVUS ja SEOS.

Disaine võrreldakse järgmiste kriteeriumite alusel.

- Päringute kiirus
- Ridade lisamise kiirus
- Ridade kustutamise kiirus
- Päringute ja operatsioonide keerukus (koodiridade arvu metoodikal)
- Tabelite ja nendele loodud indeksite andmemaht

Sama operatsiooni erinevatel käivitamistel võib tekkida juhuslikke täitmiskiiruse erinevusi näiteks kontekstkommutatsiooni, pakettide kaotsimineku, prügikoristuse, ploki eksliku kettalt lugemise, mehhaanilise vibratsiooni jms tõttu [10, p. 14]. Seetõttu käivitatakse kõiki päringuid ja andmemuudatuse operatsioone viis korda. Saadud tulemuste põhjal arvutatakse geomeetriline keksmine. Aritmeetilise keskmise asemel kasutan geomeetrilist keskmist, sest operatsiooni korduva täitmise tulemusel läheb süsteem „soojaks“ (st valmistab ette ja jätab meelde täitmisplaanid ning loeb mällu andmed, mida pole järgnevatel täitmistel enam vaja teha). [66] Järelikult on need täitmised üksteisest sõltuvad.

Päringute ja andmemuudatuse operatsioonide keerukuse hindamiseks kasutan koodiridade arvu metoodikat (*Source Lines of Code*, SLOC). Koodiridade arvu metoodika on meetrika, mida kasutatakse tarkvara mahu ja keerukuse hindamiseks. Leian füüsilise koodiridade arvu, mis tähendab, et koodiridade hulka ei loeta tühje ridu ja kommentaare. [67] Selle kasutamiseks määran koodi vormistamise reeglid.

- SELECT, INSERT, DELETE, WHERE, JOIN, UNION, ORDER BY ja GROUP BY klauslid algavad uuelt realt.
- Iga WHERE alamtingimus algab uuelt realt.

- SELECT klauslis on iga veeru nimi uuel real.
- FROM klauslis on iga tabeli nimi uuel real.

PostgreSQL's saab tabeli andmemahu teadasaamiseks baitides kasutada funktsiooni `pg_total_relation_size(regclass)`. Funktsioon arvutab tabeli ning tabeliga seotud indeksite andmemahu. [68] Näiteks skeemis *kylgnevus* oleva tabeli *lennujaam* mahu saamiseks käivitatakse lause `SELECT pg_total_relation_size('"kylgnevus". "lennujaam"')`. Kogu skeemis olevate tabelite mahu leidmiseks megabaitides kasutasin lisas 7 toodud päringut, mis on kirjutatud [69] põhjal.

Oracle andmebaasisüsteemis on võimalik tabeli andmemahtu koos indeksite mahuga leida päringuga, mis on toodud lisas 6. Päring on võetud allikast [70] ning seda on muudetud vastavalt vajadusele.

5.2.1 Kasutatavad andmebaasisüsteemid

Kasutan andmebaasisüsteeme, mis kuuluvad 2017. aasta mai seisuga nelja kõige populaarsema andmebaasisüsteemi hulka [1]. Valisin need andmebaasisüsteemid, sest olen nendega tuttav ning mul on nendele juurdepääs. Eksperimentide läbiviimiseks kasutan kaht andmebaasisüsteemi: PostgreSQL 9.5 ja Oracle 12c Enterprise Edition Release 1. Oracle andmebaasisüsteemis kasutan päringute käivitamiseks ning nende täitmiseks kulunud aja mõõtmiseks Oracle SQL Developer'i versiooni 4.1.5.21. PostgreSQL jaoks kasutan phpPgAdmin versiooni 5.1. Täitmisplaanide graafilisel kujul nägemiseks PostgreSQL andmebaasisüsteemis kasutan pgAdmin versiooni 1.22.1.

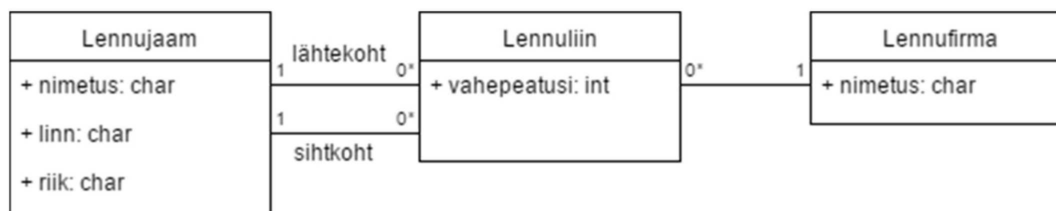
Eksperimendi käigus kasutan TTÜ serverit apex.ttu.ee, kus on võimalik mõlemat andmebaasisüsteemi kasutada. Serveri tehnilised andmed: 40 GB RAM, 15 CPU-d, operatsioonisüsteem CentOS 6.4.

5.3 Andmebaasi projekteerimine

Toon jaotises välja projekteeritava andmebaasi kontseptuaalse andmemudeli ning andmebaasi diagrammid eksperimendis kasutatavate disainide kohta.

5.3.1 Kontseptuaalne andmemudel

Joonis 36 esitab lähteülesande olemi-suhte diagrammi, mis on osaks kontseptuaalsest andmemudelist.



Joonis 36. Lennuliinide andmebaasi olemi-suhte diagramm.

Antud mudelis on esitatud kolm olemitüüpi: *Lennujaam*, *Lennuliin* ja *Lennufirma*. Igal lennuliinil on kindel seda teenindav lennufirma. Iga lennuliin saab alguse ühest lennujaamast ning lõpeb teises lennujaamas. Lisainfoks lennuliinide kohta on vahepeatuste arv, mis peab olema vahemikus null kuni üheksa. Tallinn-Frankfurt ja Frankfurt-Tallinn oleksid süsteemis registreeritud kui kaks eraldi lennuliini.

Marsruudi all mõistan teekonda lennujaamast a lennujaama b. See võib koosneda mitmest lennuliinist. Süsteem on paindlik. Näiteks marsruut Tallinnast läbi Frankfurti Barcelonasse võib koosneda kahest lennuliinist: Tallinn-Frankfurt ja Frankfurt-Barcelona, millest esimene on näiteks seotud lennufirmaga Lufthansa ja teine lennufirmaga Iberia. Mõlemal lennuliinil ei ole vahepeatusi, st vahepeatuste arv on null. Samas, kui neid lennuliine teenindab sama lennufirma, siis võimaldaks andmebaas selle marsruudi registreerida ka ühe lennuliinina – Tallinn-Barcelona, millel on registreeritud vahepeatuste arvuks üks. Teisalt, ka sellisel juhul on võimalik registreerida kaks lennuliini, millel vahepeatuste arv on null. Lähteandmetes on üldjuhul kasutatud lähenemist, kus lennuliinid salvestatakse eraldi ning vahepeatuste arvuks on null.

Igal lennujaamal on unikaalne rahvusvaheline ICAO (*International Civil Aviation Organization*) kood [71]. Lennujaamad võivad olla väljapool linnu, seega atribuut *linn* ei ole kohustuslik. Lennujaama riigina registreeritakse riigi nimi.

Linn ja *Riik* on klassifikaatorid, mida võiks vastavalt modelleerida ja realiseerida. Sellised täiendused oleksid reaalse marsruutide haldamise süsteemi puhul olulised. Kuid käesoleva töö jaoks, mis vajab näidet andmekäitlusoperatsioonidega eksperimenteerimiseks, ei oma see tulemusele mõju.

5.3.2 Andmebaasi realiseerimine

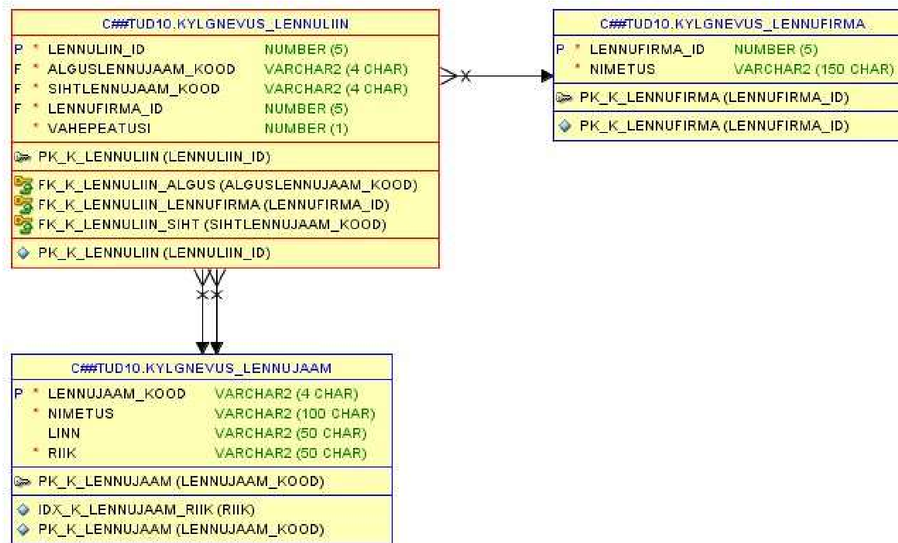
Realiseerin kontseptuaalse andmemudeli alusel baastabelite (nimega tabelid, mis pole defineeritud teiste tabelite põhjal) e tabelite loomiseks CREATE TABLE laused ning käivitan need andmebaasis.

Oracle ja PostgreSQL loovad primaarvõtmetele ning unikaalsuse kitsenduse alusel automaatselt indeksid. Lisaks loon ise indeksi tabeli *Lennujaam* veerule *riik*, et kiirendada riigi alusel otsingut. Välisvõtmetele need andmebaasisüsteemid automaatselt indekseid ei loo. Et leida, kuidas välisvõtmetele lisatavad indeksid mõjutavad päringute kiiruseid, jätan need esialgu lisamata ning teen töökiiruse katsetused. Hiljem lisan ka välisvõtme indeksid ja viin töökiiruse katsetused uuesti läbi koos lisatud indeksitega.

Kõigile välisvõtmetele (välja arvatud klassifikaatori *Seose_suund_kood* viitele *seoste suunatud graafi* disaini tabelis *Lenuliin*) määran ON DELETE CASCADE kompenseeriva tegevuse. Kuna eksperimenti on plaanis läbi viia erinevate andmehulkadega, siis lihtsustab see andmete kustutamist. Alustan eksperimenti kõige suurema andmehulgaga (kõige rohkem ridu tabelites) ja seejärel kustutan ridu tabelitest *Lennujaam* ning *Lennufirma*.

5.3.3 Andmebaasi disaini mudel külgnevusnimistu korral

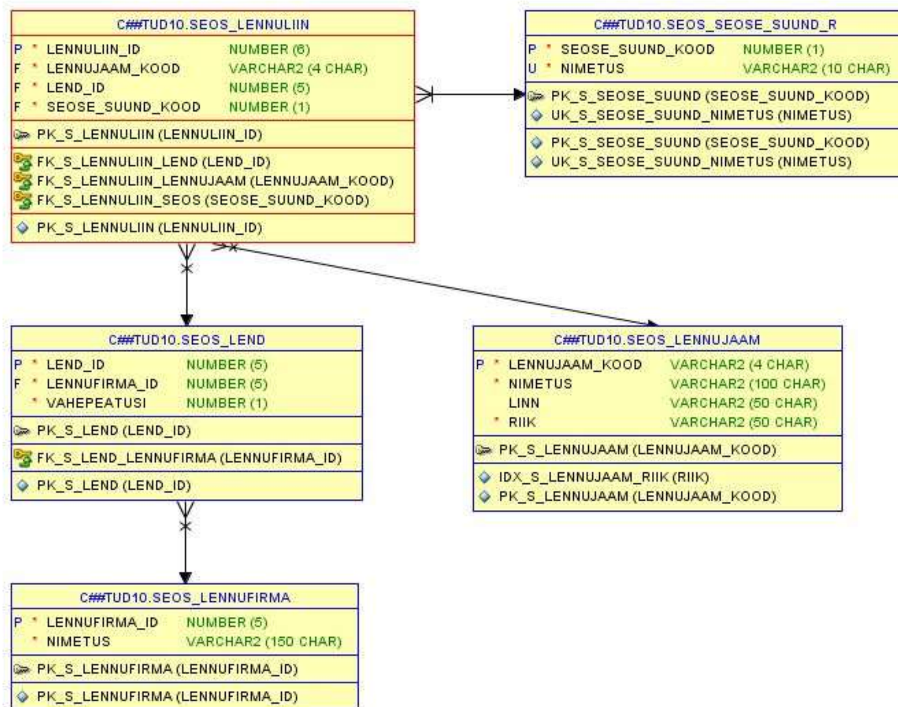
Joonis 37 on Oracle SQL Developeri abil loodud külgnevusnimistu disaini andmebaasi diagramm. Tarkvara genereeris selle vastavalt andmebaasis loodud tabelitele. PostgreSQL andmebaasis kasutasin tüüpide NUMBER(n) ja VARCHAR2(n) asemel vastavalt tüüpe INTEGER ja VARCHAR(n).



Joonis 37. Külgnevusnimistu disaini andmebaasi diagramm

5.3.4 Andmebaasi disaini mudel seoste suunatud graafi korral

Joonis 38 on Oracle SQL Developeri abil loodud seoste suunatud graafi disaini andmebaasi diagramm. Tarkvara genereeris selle vastavalt andmebaasis loodud tabelitele.



Joonis 38. Seoste suunatud graafi disaini andmebaasi diagramm

Tabelis *Lend* hoitakse infot lennu üldandmete kohta (vahepeatuste arv, lennufirma).

5.4 Testandmed

Täidan eksperimendi jaoks loodud andmebaasid testandmetega. Sain testandmed OpenFlights andmebaasist [65]. Seal esitatud andmetes on eksperimendi jaoks võetud kasutusele alamhulk andmetest, mis vastab ülesande kontseptuaalsele andmemudelile.

Kõigepealt impordin andmed Oracle andmebaasi külgnevusnimistu disaini tabelitesse. Andmete Oracle andmebaasi importimiseks kasutan Oracle SQL Developeris olevat Data Import Wizardit. Lähteandmetes on lennujaamadele lisatud süsteemi spetsiifiline identifikaator, mida kasutatakse lennujaama sidumiseks lennuliiniga. Samas on igal lennujaamal olemas ka ICAO kood, mis on unikaalne rahvusvaheline identifikaator. Seega kasutan eksperimendi andmemudelis lennujaama identifikaatorina ICAO koodi. Et andmeid omavahel siduda, lisan tabelitesse *Lennujaam* ja *Lennuliin* andmete importimise ajaks veeru *lennujaam_id*, kuhu impordin lähteandmetes kasutatava lennujaama identifikaatori. Pärast seose loomist läbi ICAO koodi eemaldan need veerud. Lähteandmetes leidub lennuliine, mis on seotud lennujaamadega, mis lennujaamade failis puuduvad. Ilmselt on tegemist lennujaamadega, mis on süsteemist kustutatud, kuid lennuliinides on viited lennujaamale alles jäänud. Importimisel kustutan sellised lennuliinid, mille lähte- või sihtkohale puudub vaste lennujaamade tabelis. Samuti eemaldan andmed, mis ei lähe kokku kontseptuaalse andmemudeliga – lennuliinid, millel puudub lähte- või sihtlennujaam. Lisaks kustutan ühe lennuliini, kus lähte- ja sihtkohaks on sama lennujaam. Nii jääb esialgsest 67 663st lennuliinist alles 65 612 lennuliini.

Külgnevusnimistu tabelitest seoste suunatud graafi tabelitesse andmete ülekandmise laused esitab Lisa 3.

Täidan PostgreSQL tabelid Oracle andmebaasis olevate andmete alusel. Eesmärk on, et andmebaasisüsteemide võrdlemise huvides tehtaks eksperiment nendega samal riistvaral ja samade andmete põhjal. Tingimustele vastavate andmete hulgad mõjutavad andmebaasisüsteemide tehtavaid valikuid ja võrreldavuse huvides peavad need olema ühesugused. Ekspordin andmete lisamise SQL INSERT laused Oracle andmebaasist kasutades Oracle SQL Developeris olevat Export Wizardit. Seejärel viin need laused vastavusse PostgreSQL tabelite struktuuriga – asendan skeemi ja tabelite nimed. Selleks asendan sõned ’C##TUD10.KYLGNEVUS_’ ja ’C##TUD10.SEOS_’ vastavalt

sõnedega ’KYLGNEVUS.’ ja ’SEOS.’. Seejärel käivitan andmete sisestamise laused PostgreSQL andmebaasisüsteemis.

Hindamaks andmemahtude mõju päringute ja andmemuudatuse operatsioonide kiirustele, viin eksperimendi läbi kolme erineva andmehulgaga (Tabel 3). Teen seda, et tuleks välja disainide omaduste sõltuvus andmemahu muutumisest.

Tabel 3. Eksperimendi andmemahud

	Lennujaamu	**Lennufirmasid**	**Lennuliine**
Andmemaht 1	7184	6162	65 611
Andmemaht 2	4600	3450	16 400
Andmemaht 3	3100	2100	4100

Alustan eksperimenti kõige suuremast andmemahust, milleks on valitud OpenFlight andmebaasist saadav maksimaalne võimalik testandmete hulk. Väiksemate mahtude saamiseks kustutan teatud hulga lennujaamu ja lennufirmasid. ON DELETE CASCADE tõttu kustutatakse selle tulemusel ka lennuliinid, millega seotud lennujaam või lennufirma kustutati.

Värskendan iga andmemahu korral enne eksperimendi algust statistikat, et andmebaasisüsteemil oleks ülevaade andmete hulgast ja jaotusest. See võimaldab andmebaasisüsteemil valida päringu jaoks sobivaima täitmisplaani. Oracle’s käivitan statistika uuendamiseks iga tabeli kohta lause `EXEC dbms_stats.gather_table_stats( 'C##TUD10', [tabeli nimi], cascade=>TRUE)`, PostgreSQL’s lause `ANALYZE`, mis värskendab terve andmebaasi statistikat.

5.5 Eksperimendis testitavad operatsioonid

Viin eksperimendi käigus läbi järgnevad operatsioonid.

- Andmete lugemine (S1, S2)
- Andmete lisamine (I1)

- Andmete kustutamine (D1)

Päringus S1 leitakse riik, kust on võimalik lennata otselennuga ja ilma vahepeatusteta kõige rohkematesse erinevatesse lennujaamadesse. Päringu tulemusena tuuakse välja ka sihtlennujaamade arv. Päringu alamosaks on sõlme naabrite leidmine, mis on üks graafitöötluses levinud ülesannetest.

Päringuga S2 antakse ette lähtelennujaam ning leitakse kõik lennujaamad, kuhu antud riigist maksimaalselt n ümberistumisega on võimalik saada. Et võrrelda, kuidas sõltub päring otsingu sügavusest, viiakse päringut läbi kolme erineva ümberistumiste arvuga n vahemikus 0-2. Mida tähendab ümberistumiste arv?

- Lend 1: A=>B, vahepeatuste arv = 1
- Lend 2: B=>C, vahepeatuste arv = 1
- A-st C-sse saab antud juhul kolme ümberistumisega.

Rekursiivse päringu aluseks on võetud Celko raamatus toodud päring [44, p. 690]. Päringuga alamosana leitakse kõikvõimalikud teed ühest sõlmest teistesse sõlmedesse. Erinevad teede leidmise ülesanded on levinud graafiülesanded. Ka lennujaamade süsteemi puhul on oluline teede leidmine erinevate lennujaamade vahel. Kuna SQL standardi viimased versioonid [33] kirjeldavad WITH klauslit, siis kasutatakse seda nii PostgreSQL kui Oracle andmebaasis ülesande lahendamiseks ja eelistatakse seda nende andmebaasisüsteemide pakutavatele vanematele mittestandardiseeritud võimalustele.

Operatsiooniga I1 lisatakse uus lennujaam ning sellega seotud lennuliin.

Operatsiooniga D1 kustutatakse kõik päringuga S1 leitud riigis asuvad lennujaamad.

Tabel 4. Eksperimendis kasutatavad päringud ja andmemuudatuse operatsioonid

	Oracle	**PostgreSQL**

S1 A	`SELECT riik, sihtkohti FROM (`
	`  SELECT j.riik,`
	`         COUNT(DISTINCT l.sihtlennujaam_kood) AS sihtkohti,`
	`         rank() OVER (ORDER BY COUNT(DISTINCT`
		`SELECT riik, sihtkohti FROM (`
		`  SELECT j.riik,`
		`         COUNT(DISTINCT l.sihtlennujaam_kood) AS sihtkohti,`
		`         rank() OVER (ORDER BY COUNT(DISTINCT`

<table>
<tr><th></th><th>Oracle</th><th>PostgreSQL</th></tr>
<tr><td></td><td>l.sihtlennujaam_kood) DESC) AS rank
  FROM kylgnevus_lennuliin l
  INNER JOIN kylgnevus_lennujaam j ON l.alguslennujaam_kood = j.lennujaam_kood
  WHERE l.vahepeatusi = 0
  GROUP BY j.riik) foo
WHERE rank=1;</td><td>l.sihtlennujaam_kood) DESC) AS rank
  FROM kylgnevus.lennuliin l
  INNER JOIN kylgnevus.lennujaam j ON l.alguslennujaam_kood = j.lennujaam_kood
  WHERE l.vahepeatusi = 0
  GROUP BY j.riik) foo
WHERE rank=1;</td></tr>
<tr><td>S1 C</td><td>SELECT riik, sihtkohti FROM (
  SELECT lj.riik,
         COUNT(DISTINCT s_ll.lennujaam_kood) AS sihtkohti,
         rank() OVER (ORDER BY COUNT(DISTINCT s_ll.lennujaam_kood) DESC) AS rank
  FROM seos_lennuliin ll
  INNER JOIN seos_lend l ON l.lend_id = ll.lend_id
  INNER JOIN seos_lennujaam lj ON ll.lennujaam_kood = lj.lennujaam_kood
  INNER JOIN seos_lennuliin s_ll ON s_ll.lend_id = ll.lend_id AND s_ll.seose_suund_kood = 2
  WHERE l.vahepeatusi = 0 AND ll.seose_suund_kood = 1
  GROUP BY lj.riik) foo
WHERE rank=1;</td><td>SELECT riik, sihtkohti FROM (
  SELECT lj.riik,
         COUNT(DISTINCT s_ll.lennujaam_kood) AS sihtkohti,
         rank() OVER (ORDER BY COUNT(DISTINCT s_ll.lennujaam_kood) DESC) AS rank
  FROM seos.lennuliin ll
  INNER JOIN seos.lend l ON l.lend_id = ll.lend_id
  INNER JOIN seos.lennujaam lj ON ll.lennujaam_kood = lj.lennujaam_kood
  INNER JOIN seos.lennuliin s_ll ON s_ll.lend_id = ll.lend_id AND s_ll.seose_suund_kood = 2
  WHERE l.vahepeatusi = 0 AND ll.seose_suund_kood = 1
  GROUP BY lj.riik) foo
WHERE rank=1;</td></tr>
<tr><td>S2 A</td><td>WITH marsruut (sihtlennujaam_kood, vahepeatusi, teekond)
  AS (SELECT DISTINCT sihtlennujaam_kood, vahepeatusi, CAST(alguslennujaam_kood || ',' || sihtlennujaam_kood AS VARCHAR(255))
    FROM kylgnevus_lennuliin
    WHERE alguslennujaam_kood = :algusLennujaamaKood AND</td><td>WITH recursive marsruut (sihtlennujaam_kood, vahepeatusi, teekond)
  AS (SELECT DISTINCT sihtlennujaam_kood, vahepeatusi::INTEGER,
alguslennujaam_kood || ',' || sihtlennujaam_kood
    FROM kylgnevus.lennuliin
    WHERE alguslennujaam_kood = :algusLennujaamaKood AND vahepeatusi <= :maksimaalneVahepeatusteArv</td></tr>
</table>

	Oracle	PostgreSQL
	vahepeatusi <= :maksimaalneVahepeatusteArv UNION ALL SELECT uus_lend.sihtlennujaam_kood, eelmine_lend.vahepeatusi + uus_lend.vahepeatusi + 1, eelmine_lend.teekond || ',' || uus_lend.sihtlennujaam_kood FROM marsruut eelmine_lend, kylgnevus_lennuliin uus_lend WHERE eelmine_lend.sihtlennujaam_kood = uus_lend.alguslennujaam_kood AND eelmine_lend.teekond NOT LIKE '%' || uus_lend.sihtlennujaam_kood || '%' AND eelmine_lend.vahepeatusi + uus_lend.vahepeatusi + 1 <= :maksimaalneVahepeatusteArv) SELECT DISTINCT sihtlennujaam_kood FROM marsruut;	UNION ALL SELECT DISTINCT uus_lend.sihtlennujaam_kood, eelmine_lend.vahepeatusi + uus_lend.vahepeatusi + 1, eelmine_lend.teekond || ',' || uus_lend.sihtlennujaam_kood FROM marsruut eelmine_lend, kylgnevus.lennuliin uus_lend WHERE eelmine_lend.sihtlennujaam_kood = uus_lend.alguslennujaam_kood AND eelmine_lend.teekond NOT LIKE '%' || uus_lend.sihtlennujaam_kood || '%' AND eelmine_lend.vahepeatusi + uus_lend.vahepeatusi + 1 <= :maksimaalneVahepeatusteArv) SELECT DISTINCT sihtlennujaam_kood FROM marsruut;
S2 C	WITH marsruut (sihtlennujaam_kood, vahepeatusi, teekond) AS (SELECT DISTINCT s_ll.lennujaam_kood, l.vahepeatusi, CAST(ll.lennujaam_kood || ',' || s_ll.lennujaam_kood AS VARCHAR(255)) FROM seos_lennuliin ll INNER JOIN seos_lend l ON l.lend_id = ll.lend_id INNER JOIN seos_lennuliin s_ll ON s_ll.lend_id = l.lend_id AND s_ll.seose_suund_kood = 2	WITH recursive marsruut (sihtlennujaam_kood, vahepeatusi, teekond) AS (SELECT DISTINCT s_ll.lennujaam_kood, l.vahepeatusi::INTEGER, ll.lennujaam_kood || ',' || s_ll.lennujaam_kood FROM seos.lennuliin ll INNER JOIN seos.lend l ON l.lend_id = ll.lend_id INNER JOIN seos.lennuliin s_ll ON s_ll.lend_id = l.lend_id AND s_ll.seose_suund_kood = 2 WHERE ll.lennujaam_kood = :algusLennujaamaKood AND

<table>
<tr><th></th><th>Oracle</th><th>PostgreSQL</th></tr>
<tr><td></td><td>WHERE ll.lennujaam_kood = :algusLennujaamaKood AND ll.seose_suund_kood = 1 AND l.vahepeatusi <= :maksimaalneVahepeatusteArv
UNION ALL
SELECT saabumine.lennujaam_kood,

eelmine_lend.vahepeatusi + l.vahepeatusi + 1,
eelmine_lend.teekond || ',' || saabumine.lennujaam_kood
FROM marsruut eelmine_lend
INNER JOIN seos_lennuliin valjumine ON eelmine_lend.sihtlennujaam_kood = valjumine.lennujaam_kood AND valjumine.seose_suund_kood = 1
INNER JOIN seos_lend l ON l.lend_id = valjumine.lend_id
INNER JOIN seos_lennuliin saabumine ON saabumine.lend_id = l.lend_id AND saabumine.seose_suund_kood = 2
WHERE eelmine_lend.teekond NOT LIKE '%' || saabumine.lennujaam_kood || '%'
AND eelmine_lend.vahepeatusi + l.vahepeatusi + 1 <= :maksimaalneVahepeatusteArv
)
SELECT DISTINCT sihtlennujaam_kood FROM marsruut;</td><td>ll.seose_suund_kood = 1 AND l.vahepeatusi <= :maksimaalneVahepeatusteArv
UNION ALL
SELECT DISTINCT saabumine.lennujaam_kood,

eelmine_lend.vahepeatusi + l.vahepeatusi + 1,
eelmine_lend.teekond || ',' || saabumine.lennujaam_kood
FROM marsruut eelmine_lend
INNER JOIN seos.lennuliin valjumine ON eelmine_lend.sihtlennujaam_kood = valjumine.lennujaam_kood AND valjumine.seose_suund_kood = 1
INNER JOIN seos.lend l ON l.lend_id = valjumine.lend_id
INNER JOIN seos.lennuliin saabumine ON saabumine.lend_id = l.lend_id AND saabumine.seose_suund_kood = 2
WHERE eelmine_lend.teekond NOT LIKE '%' || saabumine.lennujaam_kood || '%'
AND eelmine_lend.vahepeatusi + l.vahepeatusi + 1 <= :maksimaalneVahepeatusteArv
)
SELECT DISTINCT sihtlennujaam_kood FROM marsruut;</td></tr>
<tr><td>I1 A</td><td>BEGIN
INSERT INTO kylgnevus_lennujaam (lennujaam_kood, nimetus, riik)
VALUES ('TEST', 'UUS LENNUJAAM', 'Estonia');</td><td>DO $$
BEGIN
INSERT INTO kylgnevus.lennujaam (lennujaam_kood, nimetus, riik)
VALUES ('TEST', 'UUS LENNUJAAM', 'Estonia');</td></tr>
</table>

	Oracle	PostgreSQL
	INSERT INTO kylgnevus_lennuliin (alguslennujaam_kood, sihtlennujaam_kood, lennufirma_id, vahepeatusi) VALUES ('TEST', 'KATL', 6, 0); END;	INSERT INTO kylgnevus.lennuliin (alguslennujaam_kood, sihtlennujaam_kood, lennufirma_id, vahepeatusi) VALUES ('TEST', 'KATL', 6, 0); END; $$ LANGUAGE plpgsql;

I1 C	BEGIN
	INSERT INTO seos_lennujaam (lennujaam_kood, nimetus, riik)
	VALUES ('TEST', 'UUS LENNUJAAM', 'Estonia');
	
	INSERT INTO seos_lend (lend_id, lennufirma_id, vahepeatusi)
	VALUES (90000, 6, 0);
	
	INSERT INTO seos_lennuliin (lennujaam_kood, lend_id, seose_suund_kood)
	VALUES ('TEST', 90000, 1);
	INSERT INTO seos_lennuliin (lennujaam_kood, lend_id, seose_suund_kood)
	VALUES ('KATL', 90000, 2);
	END;
		DO $$
		BEGIN
		INSERT INTO seos.lennujaam (lennujaam_kood, nimetus, riik)
		VALUES ('TEST', 'UUS LENNUJAAM', 'Estonia');
		
		INSERT INTO seos.lend (lend_id, lennufirma_id, vahepeatusi)
		VALUES (90000, 6, 0);
		
		INSERT INTO seos.lennuliin (lennujaam_kood, lend_id, seose_suund_kood)
		VALUES ('TEST', 90000, 1);
		INSERT INTO seos.lennuliin (lennujaam_kood, lend_id, seose_suund_kood)
		VALUES ('KATL', 90000, 2);
		END;
		$$ LANGUAGE plpgsql;

D1 A	DELETE FROM kylgnevus_lennujaam WHERE riik IN (:riigiNimed);	DELETE FROM kylgnevus.lennujaam WHERE riik IN (:riigiNimed);
D1 C	DELETE FROM seos_lennujaam WHERE riik IN (:riigiNimed);	DELETE FROM seos.lennujaam WHERE riik IN (:riigiNimed);

6 Eksperimendi tulemused

Toon jaotises välja päringute ja andmemuudatuse operatsioonide kiiruste, andmemahtude ning lausete keerukuse mõõtmise tulemused.

Tabel 5, Tabel 6 ja Tabel 7 esitavad päringute ja andmemuudatuse operatsioonide kiiruste mõõtmistulemuste geomeetrilise keskmise Oracle ja PostgreSQL andmebaasisüsteemides. Mõõtmistulemused on sekundites.

Tabel 5 esitab esimese päringu (S1) ja andmemuudatuse operatsioonide (I1, D1) täitmise kiirused. Tabeli iga veerg tähistab operatsiooni kombinatsiooni disainiga (A – külgnevusnimistu; C – seoste suunatud graaf). Tabeli read viitavad andmebaasisüsteemile ja ridade arvule tabelis *Lennuliin*.

Tabel 6 esitab teise päringu (S2) täitmise kiirused. Tabeli iga veerg esitab disaini kombinatsiooni maksimaalse vahepeatuste arvuga. Näitekse veerus A1 on esitatud maksimaalselt ühe vahepeatusega sihtlennujaamade leidmise kiirused külgnevusnimistu disaini korral. Lähtelennujaamana kasutan lennujaama, mille ICAO koodiks on „KATL“. Tegemist on lennujaamaga, millest saab ilma vahepeatusteta alguse enim lennuliine. PostgreSQL võimaldab kasutada rekursiivses päringus ka korduste eemaldamise DISTINCT operatsiooni, Oracle seda ei võimalda. Päringute lausetes (Tabel 4) on see rasvaselt ja teises toonis välja toodud. PostgreSQL puhul on välja toodud päringute ajad nii DISTINCT operatsiooni kasutades kui ka ilma selleta. DISTINCT võimaldab vähendada leitud masruutide CTEsse lisatavaid ridu.

Tabel 5 ja Tabel 6 tähistatakse **rasvaselt** tulemused, mis on sama andmebaasisüsteemi, andmemahu ja lähteülesande korral parim (väiksem). See annab ülevaate, millist disaini millise andmebaasisüsteemi korral eelistada. <u>Allajoonitud</u> on tulemused, mis on sama disaini, andmemahu ja lähteülesande korral PostgreSQL ja Oracle võrdluses parimad (väiksemad). See aitab võrrelda andmebaasisüsteeme omavahel.

Tabel 5. Päringu S1 ja andmemuudatuse operatsioonide geomeetrilised keskmised kiirused (sekundites)

	S1A	**S1C**	**I1A**	**I1C**	**D1A**	**D1C**
4100 Oracle	**0,031**	0,068	0,033	**0,024**	1,741	**1,606**
4100 PostgreSQL	**<u>0,029</u>**	<u>0,042</u>	**<u>0,01</u>**	<u>0,013</u>	<u>1,009</u>	**<u>0,974</u>**

	S1A	S1C	I1A	I1C	D1A	D1C
16 400 Oracle	**0,08**	0,132	**0,043**	0,045	**2,758**	2,845
16 400 PostgreSQL	**0,087**	0,149	**0,012**	0,013	**4,493**	4,506
65 611 Oracle	**0,116**	0,291	**0,016**	0,018	**8,257**	9,760
65 611 PostgreSQL	**0,601**	0,699	**0,013**	**0,013**	**28,766**	32,607

Tabel 6. Päringu S2 kiirused (sekundites)

	A0	C0	A1	C1	A2	C2
4100 Oracle	**0,047**	0,082	1,102	**0,148**	**1,352**	1,496
4100 PostgreSQL	**0,018**	0,043	**0,08**	0,081	1,153	**0,834**
4100 PostgreSQL DISTINCT	**0,02**	0,046	**0,064**	0,075	0,666	**0,405**
16 400 Oracle	**0,098**	0,175	**0,368**	0,439	**14,126**	14,669
16 400 PostgreSQL	**0,038**	0,133	**0,386**	0,428	17,857	**11,04**
16 400 PostgreSQL DISTINCT	**0,043**	0,137	0,399	**0,294**	6,323	**4,229**
65 611 Oracle	**0,129**	0,242	**1,344**	1,889	**208,023**	230,584
65 611 PostgreSQL	**0,133**	0,620	3,040	**2,278**	*	253,301
65 611 PostgreSQL DISTINCT	**0,144**	0,539	1,502	**1,371**	65,455	**60,309**

* Ei õnnestunud saada serverilt 10 minuti jooksul vastust

S2 päringus lisatakse marsruutide CTEsse võimalikud marsruudid sihtlennujaamast maksimaalse sügavusega n. Kõik *Lennuliin* tabelis olevad lennuliinid lisatakse ühekordselt. See tähendab, et kui ühel lennuliinil pakuvad lende kaks erinevat lennufirmat, siis CTEsse lisatakse ainult üks rida. Oracle's lisatakse mitmest lennuliinist koosnevad marsruudid CTEsse kõikide võimalike kombinatsioonide korral, kuna Oracle ei võimalda kasutada rekursioonis DISTINCT operatsiooni. Seega näiteks kui on lennuliinid a=>b ja b=>c ning lennuliinil a=>b pakuvad lendu kaks lennufirmat, siis lisatakse lend a=>c marsruutide CTEsse kaks korda.

Leidmaks, kuidas mõjutab võimalike marsruutide arv päringu kiirust, tegin S2 päringut erinevate alguslennujaamadega, millest oleks võimalik saada erinevasse arvu

sihtkohtadesse. Tabel 7 esitab päringu S2 kiirused külgnevusnimistu disaini korral Oracle andmebaasisüsteemis. Katse viidi läbi suurima andmemahu korral. Tabel 8 esitab samade päringute korral rekursiooni käigus leitud marsruutide arvu. Lennujaamad on valitud selliselt, et ilma vahepeatusteta oleks lennujaamast saadavate sihtkohtade arv enam-vähem ühtlaselt jaotunud.

Tabel 7. Päringu S2 kiirused Oracle andmebaasisüsteemis suurima andmemahu korral (sekundites)

Lähtelennujaama kood	**0**	**1**	**2**
VYKG	0,034	0,061	0,834
EGNX	0,058	0,552	60,036
ULLI	0,051	0,991	126,513
EGKK	0,058	1,337	206,036
KATL	0,129	1,344	208,023

Tabel 8. Päringu S2 marsruutide arv Oracle andmebaasisüsteemis suurima andmemahu korral

Lähtelennujaama kood	**0**	**1**	**2**
VYKG	1	46	8844
EGNX	56	5425	865 272
ULLI	108	10 945	1 755 757
EGKK	164	16 483	2 717 875
KATL	217	18 566	3 121 448

Päringutes tehakse palju ühendamisoperatsioone. Seega on eeldus, et välisvõtmetele indeksite lisamine kiirendab päringute tätimist. Leidmaks, kas ja kuidas välisvõtme indeksid päringuid mõjutavad, lisasin peale eelnevalt kirjeldatud katsetusi andmebaasidesse välisvõtme veergudele indeksid. Pärast indeksite lisamist värskendasin ka statistikat. Seejärel käivitasin kõik Tabel 5 ja Tabel 6 koostamiseks kasutatud päringud uuesti. Tulemused on esitatud tabelites Tabel 9 ja Tabel 10. Rohelisega on esitatud tulemused, mis on paremad kui ilma indeksita juhul. Punasega esitatud tulemused on kehvemad kui ilma indeksita. **Rasvaselt** toodud tulemuste juures kasutati ilma indeksita juhust erinevat täitmisplaani.

Tabel 9. Päringu S1 ja andmemuudatuse operatsioonide geomeetrilised keskmised kiirused (sekundites) kasutades välisvõtme indekseid

	S1A	S1C	I1A	I1C	D1A	D1C
4100 Oracle	0,036	**0,071**	0,017	0,021	0,24	0,3
4100 PostgreSQL	0,025	0,038	0,012	0,015	**0,196**	**0,117**
16 400 Oracle	0,041	**0,106**	0,022	0,015	0,537	0,756
16 400 PostgreSQL	0,064	0,143	0,008	0,011	**0,102**	**0,062**
65 611 Oracle	0,136	0,28	0,011	0,015	0,241	0,205
65 611 PostgreSQL	0,627	2,896	0,017	0,022	**1,27**	**2,891**

Tabel 10. Päringu S2 kiirused (sekundites) kasutades välisvõtme indekseid

	A0	C0	A1	C1	A2	C2
4100 Oracle	0,051	**0,099**	0,11	**0,166**	1,01	**1,258**
4100 PostgreSQL	**0,019**	**0,048**	**0,091**	**0,082**	**1,169**	**0,791**
4100 PostgreSQL DISTINCT	**0,018**	**0,038**	**0,05**	**0,075**	**0,614**	**0,482**
16 400 Oracle	0,044	**0,142**	0,305	**0,446**	10,563	**13,776**
16 400 PostgreSQL	**0,029**	**0,121**	**0,417**	**0,263**	**18,142**	**10,66**
16 400 PostgreSQL DISTINCT	**0,028**	**0,078**	**0,296**	**0,18**	**6,282**	**3,905**
65 611 Oracle	0,081	**0,249**	1,573	**1,875**	180,116	**223,379**
65 611 PostgreSQL	**0,05**	**0,664**	**3,438**	**2,309**	*	**243,925**
65 611 PostgreSQL DISTINCT	**0,062**	**0,94**	**1,718**	**1,151**	**86,812**	**56,839**

* Ei õnnestunud saada serverilt 10 minuti jooksul vastust

Tabel 11 esitab andmebaasi mahu Oracle ja PostgreSQL andmebaasisüsteemis erinevate andmehulkade ja disainide korral. Mahud on esitatud megabaitides. Mahud on toodud koos välisvõtme indeksitega.

Tabel 11. Andmebaasi maht erinevate disainide korral (MB)

	Külgnevusnimistu	**Seoste suunatud graaf**
4100 Oracle	1,5	2,51
4100 PostgreSQL	1,25	1,23
16 400 Oracle	3,26	6,95
16 400 PostgreSQL	3,17	6,86
65 611 Oracle	11,44	27,69
65 611 PostgreSQL	10,34	24,92

Tabel 12 esitab lahenduste koodiridade arvu Oracle ja PostgreSQL andmebaasisüsteemides erinevate disainide korral.

Tabel 12. Koodiridade arv erinevate disainide korral

	S1A	**S1C**	**S2A**	**S2C**	**I1A**	**I1C**	**D1A**	**D1C**
Oracle	12	16	18	25	4	6	2	2
PostgreSQL	12	16	18	25	6	8	2	2

7 Tulemuste analüüs ja järeldused

Selles peatükis analüüsin eksperimendi käigus saadud tulemusi ning teen tulemuste põhjal järeldusi.

7.1 Päringute ja andmemuudatuse operatsioonide kiirused

Selles jaotises analüüsin eksperimendi tulemuste põhjal päringute ja andmemuudatuse operatsioonide kiiruste erinevusi.

7.1.1 Päringute ja andmemuudatuse operatsioonide kiiruste võrdlus erinevates andmebaasisüsteemides sama disaini ja andmemahu korral

Esimese päringu (S1) korral on väikeste andmehulkade korral kiiruste erinevus väike. Suurima andmehulga korral on Oracle's tehtav päring *külgnevusnimistu* disainiga umbes viis korda kiirem ning *seoste suunatud graafi* disainiga peaaegu kaks ja pool korda kiirem kui PostgreSQL's tehtud päringud.

Teise päringu (S2) puhul on enamasti kiireimaks PostgreSQL andmebaasisüsteem. Vaid suurtel andmehulkadel väikesele sügavusele otsides on Oracle andmebaasisüsteem mõnevõrra kiirem. PostgreSQL toetab rekursioonis korduste eemaldamiseks DISTINCT operaatorit, mis võimaldab teha efektiivsemaid päringuid. Tänu sellele lisatakse rekursiooni tehes marsruuti vähem ridu. Mida suurem on maksimaalne lubatud vahepeatuste arv, seda rohkem on kasu DISTINCT abil korduste eemaldamisest. Ilma vahepeatusteta marsruudi juures DISTINCT operatsioon pigem aeglustab päringut. Seda põhjusel, et vahepeatusteta marsruudi korral lisatakse ainult otsesed lennuliinid, mis on toodud *Lennuliin* tabelis ning korduste eemaldamine ei anna peale lisatöö midagi.

Lisamisoperatsiooni (I1) korral pole Oracle ja PostgreSQL vahel märgatavaid erinevusi.

Kustutamisoperatsiooni (D1) korral on väikeste andmemahtude juures PostgreSQL veidi kiirem. Andmehulga kasvades langeb PostgreSQL kiirus märgatavalt kiiremini. Suurima andmemahu korral on Oracle juba üle kolme koma kolme korra kiirem.

7.1.2 Päringute ja andmemuudatuse operatsioonide kiiruse võrdlus erinevate disainide korral sama andmebaasisüsteemi ja andmemahu korral

Esimene päring (S1) on kõikide andmehulkade korral veidi kiirem *külgnevusnimistu* disaini puhul. Samas ei ületa vahe ühelgi juhul 200ms.

Oracle's on teine päring (S2) alati kiirem külgnevusnimistu disaini korral. PostgreSQL puhul on ilma vahepeatusteta marsruutide korral kiirem *külgnevusnimistu* disain. Kui marsruut võib sisaldada mitmeid vahepeatusi, on kiireimaks DISTINCT operatsiooni sisaldav päring *seoste suunatud graafi* disaini tabelite põhjal.

Andmete sisestamise operatsiooni (I1) kiirused on kõikide andmehulkade korral mõlema disaini puhul sarnased.

Kustutamisoperatsiooni (D1) puhul on märgata, et mida suurem on andmehulk, seda kasulikum on kasutada *külgnevusnimistu* disaini.

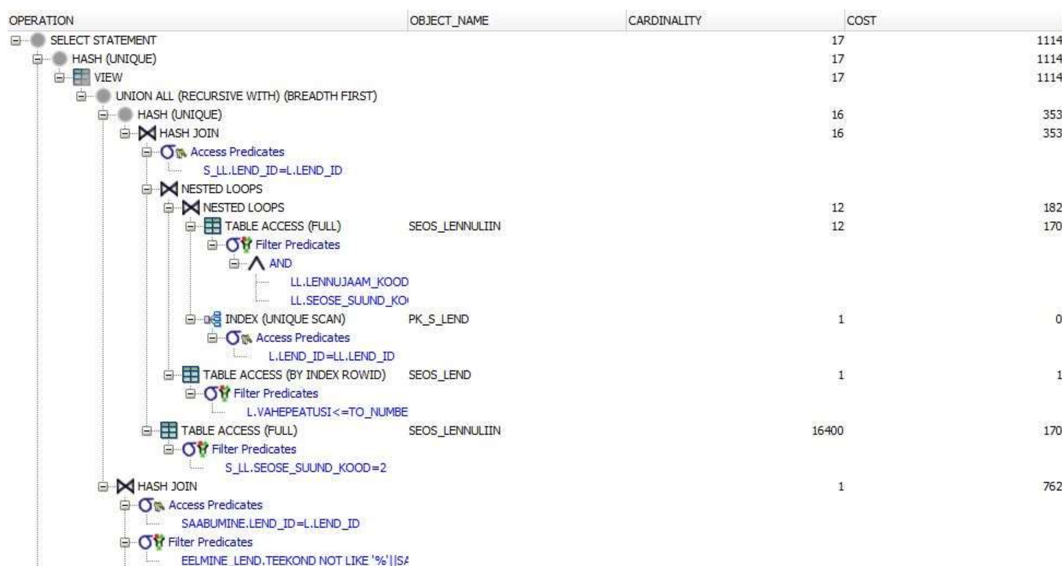
7.1.3 Päringute ja andmemuudatuse operatsioonide kiiruste võrdlus erinevate andmemahtude korral sama andmebaasisüsteemi ning disaini korral

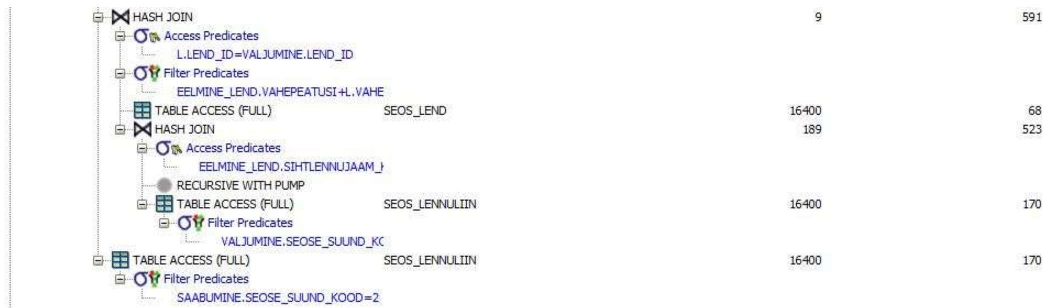
Oracle andmebaasisüsteemis on päringute S1A, S1C ja S2A täitmisplaanid samad olenemata andmehulgast. S2C korral erineb aga keskmise andmehulga korral täitmisplaan ülejäänud täitmisplaanidest.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		816	1233
HASH (UNIQUE)		816	1233
VIEW		816	1232
UNION ALL (RECURSIVE WITH) (BREADTH FIRST)			
HASH (UNIQUE)		775	411
HASH JOIN		775	411
Access Predicates			
S_LL.LEND_ID=L.LEND_ID			
HASH JOIN		775	240
Access Predicates			
L.LEND_ID=LL.LEND_ID			
TABLE ACCESS (FULL)	SEOS_LENNULIIN	775	171
Filter Predicates			
AND			
LL.LENNUJAAM_KOOD='KATL'			
LL.SEOSE_SUUND_KOOD=1			
TABLE ACCESS (FULL)	SEOS_LEND	65611	68
Filter Predicates			
L.VAHEPEATUSI<=2			
TABLE ACCESS (FULL)	SEOS_LENNULIIN	65611	171
Filter Predicates			
S_LL.SEOSE_SUUND_KOOD=2			
HASH JOIN		41	822
Access Predicates			
SAABUMINE.LEND_ID=L.LEND_ID			
Filter Predicates			
EELMINE_LEND.TEEKOND NOT LIKE '%'||SAABUMINE.LENNUJAAM_KOOD||'%'			
TABLE ACCESS (FULL)	SEOS_LENNULIIN	65611	171
Filter Predicates			
SAABUMINE.SEOSE_SUUND_KOOD=2			

Joonis 39. S2C päringu täitmisplaan Oracle andmebaasisüsteemis (65 611 rida)

Joonis 39 on toodud päringu S2C täitmisplaan Oracle andmebaasisüsteemis suurima andmemahu korral. Joonis 40 on toodud sama päring keskmise andmemahu juures. Suurima andmehulga korral kasutatab süsteem *Lennuliin* tabeli ühendamiseks sisemiselt HASH JOIN operatsiooni. See tähendab, et väiksema tabeli kohta luuakse räsitabel. Seejärel suuremat tabelit läbi käies ühendatakse väiksem tabel leitud räsi põhjal. Keskmise andmemahu korral toimub tabeli *Lennuliin* ühendamine sisemiselt NESTED LOOPS operatsiooni abil. Seda operatsiooni kasutab süsteem juhul, kui teise tabeli rea saamiseks on efektiivne võimalus. Antud juhul on selleks võimaluseks tabelil *Lennuliin* olev primaarvõtme indeks.





Joonis 40. S2C päringu täitmisplaan Oracle andmebaasisüsteemis (16 400 rida)

PostgreSQL puhul ei sõltu *külgnevusnimistu* disaini täitmisplaan andmehulgast. *Seoste suunatud graafi* disaini puhul erineb S1C päringu korral suurima andmehulgaga päringu täitmisplaan teistest. S2C täitmisplaan on kõikide andmehulkade korral erinev.

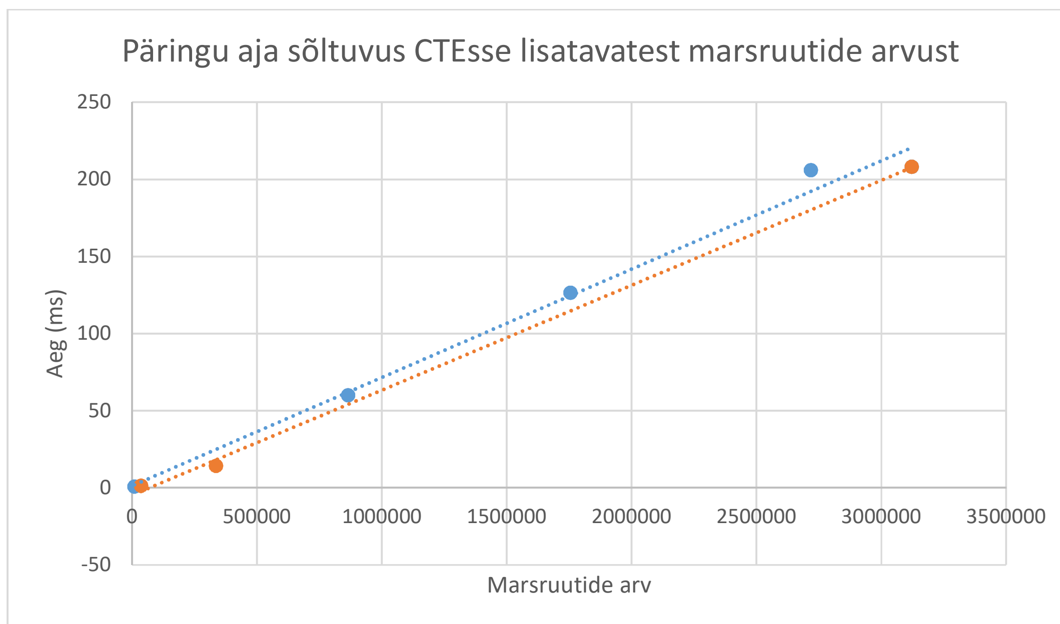
Andmehulkade suurenemine mõjutab enim S2 päringut mitmete vahepeatuste lubamise korral. Samuti avaldab andmehulkade suurenemine märgatavat negatiivset mõju andmete kustutamise operatsioonile (D1) juhul, kui pole lisatud välisvõtme indekseid.

7.1.4 Päringu S2 kiiruse sõltuvus lennuliinide arvust

Joonis 41 kirjeldab päringu S2 sõltuvust leitud marsruutide arvust Oracle andmebaasisüsteemis *külgnevusnimistu* disaini korral. Joonisel on toodud maksimaalselt kahe vahepeatusega päringu tulemused. Sinisega on toodud erinevate lennujaamadega tehtud päringute kiirused suurimal andmehulgal (Tabel 7 ja Tabel 8 veerg „2“). Punasega on toodud lennujaamaga „KATL“ tehtud päringute kiirused erinevate andmehulkade juures (Tabel 6 veerg „A2“ Oracle andmebaasisüsteemi korral).

Jooniselt on näha, et sõltuvus leitud marsruutide arvu ja päringu kiiruse vahel on lineaarne. Selles veendumiseks leidsin Pearsoni korrelatsiooni koefitsiendi kasutades kalkulaatorit [72]. Kasutades ainult maksimaalselt kahe vahepeatuste arvuga saadud mõõtmiste tulemusi, sain korrelatsiooni koefitsiendiks 0.9946. Kui leida koefitsient kasutades kõiki Tabel 7 ja Tabel 8 olevaid väärtusi, tuli koefitsiendiks 0.9978. See kinnitab eeldust, et CTEsse lisatavate marsruutide arvu ja päringu tegemiseks kuluva aja vahel on tugeva positiivne korrelatsioon.

Andmebaasis olev andmemaht (*lennuliinide* arv) aga joonise järgi kiirusele nii suurt mõju ei avalda kui CTEsse lisatavad marsruudid.

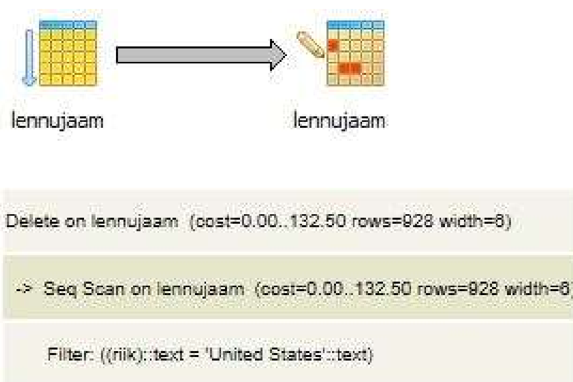


Joonis 41. Päringu S2 kiiruse sõltuvus CTEsse lisatavatest marsruutide arvust

7.1.5 Välisvõtme indeksite lisamise mõju päringute ja andmemuudatuste kiirusele

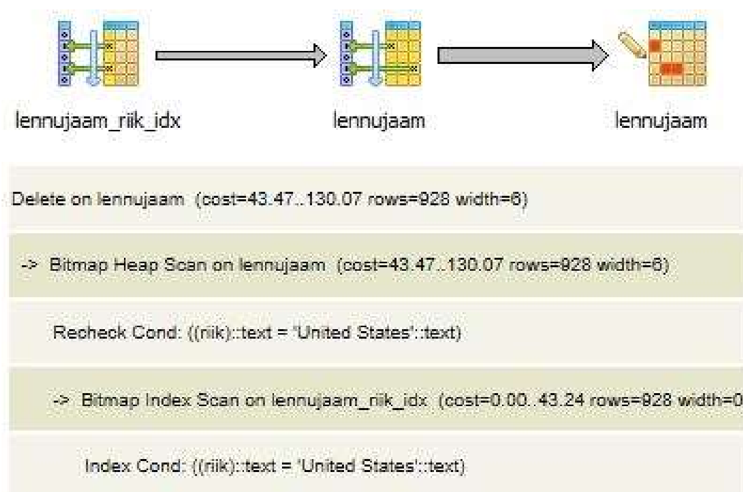
Päringu S1 korral ei võtnud PostgreSQL ühtegi loodud välisvõtme indeksitest kasutusele. Oracle kasutas S1C päringu tegemiseks seose suuna koodile lisatud indeksit. Suuremate andmemahtude korral muutus päring tänu selle veidi kiiremaks.

Kõige märgatavamat kasutegurit indeksite kasutusele võtust on näha kustutamisoperatsiooni juures (D1). Suurima andmemahu korral paranesid PostgreSQL'i tulemused *külgnevusnimistu* disaini korral üle 22 korra ning *seoste suunatud graafi* disaini korral üle 11 korra.



Joonis 42. D1A päringu täitmisplaan PostgreSQL andmebaasisüsteemis ilma välisvõtme indeksiteta

Joonis 42 on toodud päringu D1A täitmisplaan PostgreSQL andmebaasis keskmisel andmehulgal ilma välisvõtme indeksiteta. PostgreSQL otsustab sel juhul lugeda kõik tabeli kasutuses olevad plokid.



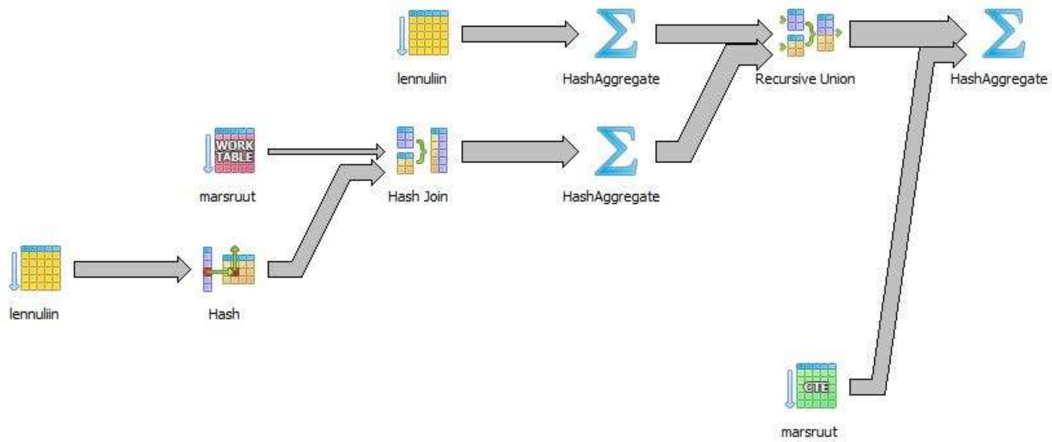
Joonis 43. D1A päringu täitmisplaan PostgreSQL andmebaasisüsteemis koos välisvõtme indeksitega

Joonis 43 on toodud päringu D1A täitmisplaan PostgreSQL andmebaasis keskmisel andmehulgal, kui on lisatud indeksid välisvõtmetele. *Bitmap Index Scan* osa loeb kõik indeksis olevad tingimusele vastavad viited ühekorraga ning sorteerib need mälus oleva „bitikaardi“ andmestruktuuri alusel. *Bitmap Heap Scan* loeb seejärel read selles järjekorras nagu need on kettale kirjutatud. [73]

S2 päringule lisatud indeksid erilist mõju ei avaldanud. Oracle võttis loodud indeksitest mõne kasutusele vaid *seoste suunatud graafi* disaini korral. Kõige märgatavam erinevus on näha PostgreSQL suurima andmemahu korral. Mitmete vahepeatuste korral *külgnevusnimistu* disaini korral muutus päring märgatavalt aeglasemaks. *Seoste suunatud graafi* disaini korral päringu kiirus aga kasvas.

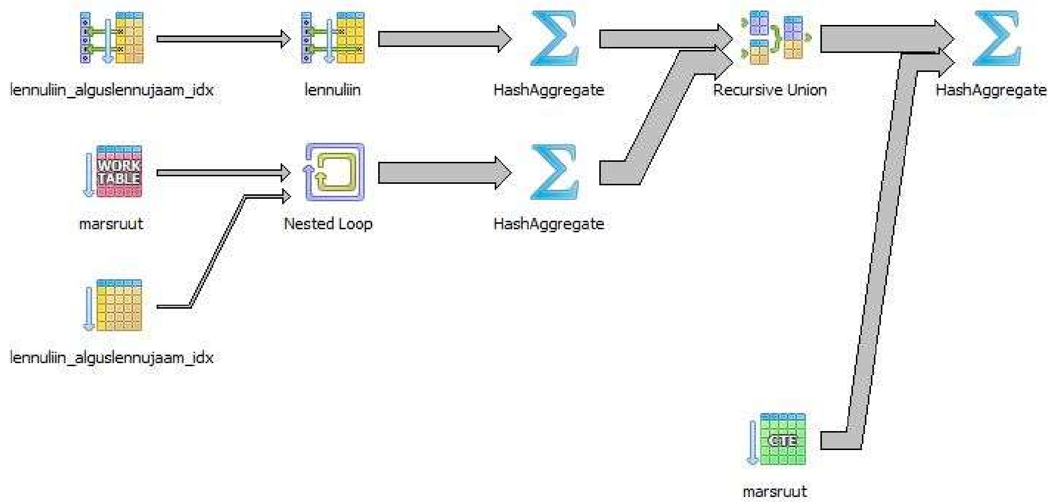
Joonis 44 on toodud päringu S2A täitmisplaan PostgreSQL andmebaasis suurima andmehulga korral ilma välisvõtme indeksiteta. Joonis 45 on toodud sama päring kasutades välisvõtme indekseid. Ilma indeksiteta juhul kasutab süsteem tabeli *Lennuliin* andmete saamiseks *sequencial scan*'i. See tähendab, et käiakse läbi kogu tabel nii nagu see kettale salvestatud on. Indeksitega juhul kasutab süsteem aga *Bitmap Heap Scan*'i. Ilma indeksiteta juht kasutab tabeli *Lennuliin* sidumiseks CTEga *Hash Join*'i. See tähendab, et tabeli *Lennuliin* kohta luuakse räsitabel. CTE seotakse seejärel tabeli *Lennuliin* ridadega kasutades loodud räsitabelit. Välisvõtme indeksitega andmebaasis

tehtud päringu korral kasutab süsteem nimetatud tabelite sidumiseks *Nested Loop* operatsiooni. Tabeli *Lennuliin* väärtus ühendatakse CTE väärtusega kasutades alguslennujaama koodi veerule loodud välisvõtme indeksit.




```
HashAggregate  (cost=52837.46..52839.46 rows=200 width=20)
 Group Key: marsruut.sihtlennujaam_kood
 CTE marsruut
  ->  Recursive Union  (cost=1418.03..50919.56 rows=85240 width=43)
        ->  HashAggregate  (cost=1418.03..1419.78 rows=100 width=11)
              Group Key: lennuliin.sihtlennujaam_kood, (lennuliin.vahepeatusi)::integer, (((lennuliin.alguslennujaam_kood)::text || ','::text) || (lennuliin.sihtlennujaam_kood)::text)
              ->  Seq Scan on lennuliin  (cost=0.00..1410.60 rows=991 width=11)
                    Filter: ((vahepeatusi <= 2) AND ((alguslennujaam_kood)::text = 'KATL'::text))
        ->  HashAggregate  (cost=4609.22..4779.50 rows=8514 width=43)
              Group Key: uus_lend.sihtlennujaam_kood, ((eelmine_lend.vahepeatusi + uus_lend.vahepeatusi) + 1), ((eelmine_lend.teekond || ','::text) || (uus_lend.sihtlennujaam_kood)::text)
              ->  Hash Join  (cost=2216.25..4545.36 rows=8514 width=43)
                    Hash Cond: ((eelmine_lend.sihtlennujaam_kood)::text = (uus_lend.alguslennujaam_kood)::text)
                    Join Filter: ((eelmine_lend.teekond !~~ (('%'::text || (uus_lend.sihtlennujaam_kood)::text) || '%'::text)) AND (((eelmine_lend.vahepeatusi + uus_lend.vahepeatusi) + 1) <= 2))
                    ->  WorkTable Scan on marsruut eelmine_lend  (cost=0.00..20.00 rows=1000 width=56)
                    ->  Hash  (cost=1075.11..1075.11 rows=65611 width=11)
                          ->  Seq Scan on lennuliin uus_lend  (cost=0.00..1075.11 rows=65611 width=11)
 ->  CTE Scan on marsruut  (cost=0.00..1704.80 rows=85240 width=20)
```

Joonis 44. S2A päringu täitmisplaan PostgreSQL andmebaasisüsteemis ilma välisvõtme indeksiteta




```
HashAggregate  (cost=41833.85..41835.85 rows=200 width=20)
 Group Key: marsruut.sihtlennujaam_kood
 CTE marsruut
   ->  Recursive Union  (cost=464.15..40105.85 rows=76800 width=43)
         ->  HashAggregate  (cost=464.15..465.73 rows=90 width=11)
               Group Key: lennuliin.sihtlennujaam_kood, (lennuliin.vahepeatusi)::integer, (((lennuliin.alguslennujaam_kood)::text || ','::text) || (lennuliin.sihtlennujaam_kood)::text)
               ->  Bitmap Heap Scan on lennuliin  (cost=19.24..457.43 rows=897 width=11)
                     Recheck Cond: ((alguslennujaam_kood)::text = 'KATL'::text)
                     Filter: (vahepeatusi <= 2)
                     ->  Bitmap Index Scan on lennuliin_alguslennujaam_idx  (cost=0.00..19.02 rows=897 width=0)
                           Index Cond: ((alguslennujaam_kood)::text = 'KATL'::text)
         ->  HashAggregate  (cost=3656.99..3810.41 rows=7671 width=43)
               Group Key: uus_lend.sihtlennujaam_kood, ((eelmine_lend.vahepeatusi + uus_lend.vahepeatusi) + 1), ((eelmine_lend.teekond || ','::text) || (uus_lend.sihtlennujaam_kood)::text)
               ->  Nested Loop  (cost=0.29..3599.46 rows=7671 width=43)
                     ->  WorkTable Scan on marsruut eelmine_lend  (cost=0.00..18.00 rows=900 width=56)
                     ->  Index Scan using lennuliin_alguslennujaam_idx on lennuliin uus_lend  (cost=0.29..3.80 rows=9 width=11)
                           Index Cond: ((alguslennujaam_kood)::text = (eelmine_lend.sihtlennujaam_kood)::text)
                           Filter: ((eelmine_lend.teekond !~~ (('%'::text || (sihtlennujaam_kood)::text) || '%'::text)) AND (((eelmine_lend.vahepeatusi + vahepeatusi) + 1) <= 2))
 ->  CTE Scan on marsruut  (cost=0.00..1536.00 rows=76800 width=20)
```

Joonis 45. S2A päringu täitmisplaan PostgreSQL andmebaasisüsteemis koos välisvõtme indeksitega

7.2 Päringute ja andmemuudatuse operatsioonide keerukuse võrdlus

Oracle ja PostgreSQL andmebaasisüsteemides on andmekäitluskeele lausete lähtekoodi keerukus sama. Päringute ning andmete sisestamisoperatsiooni korral on

külgnevusnimistu disainis päringus vähem koodiridu kui *seoste suunatud graafi* disaini korral.

7.3 Andmebaasi salvestamiseks kulunud ruum

PostgreSQL puhul kulub andmete salvestamiseks mõningal määral vähem ruumi kui Oracle puhul.

Külgnevusnimistu disaini tabelid võtavad vähem ruumi kui *seoste suunatud graafi* omad. See oligi eeldatav, kuna *seoste suunatud graafi* disaini puhul salvestatakse iga lennuliini kohta *lennuliin* tabelisse kaks rida.

7.4 Järeldused

Analüüsist lähtuvalt saab öelda, et *külgnevusnimistu* disaini korral on päringud süntaktiliselt mõnevõrra lihtsamad kui *seoste suunatud graafi* korral.

Päringute kiiruse seisukohalt on Oracle andmebaasisüsteemis kasulikum kasutada *külgnevusnimistu* disaini. PostgreSQL puhul on andmete muutmisoperatsioonid ja sõlme naabrite arvu leidmine samuti kiirem *külgnevusnimistu* disainiga. Rekursiivne päring on aga PostgreSQL andmebaasisüsteemis kiirem *seoste suunatud graafi* disaini korral.

Suurima naabrite arvuga sõlme leidmisel on kiirem Oracle andmebaasisüteem. Suure ridade hulga kustutamisel on samuti kiirem Oracle. Ridade lisamise operatsioonid on mõnevõrra kiiremad PostgreSQL andmebaasisüsteemis. Rekursiivsed päringud on kiiremad Oracle andmebaasisüsteemis. Samas kui CTE'sse ei taheta lisada korduvaid ridu, siis võimaldab PostgreSQL kasutada rekursioonis DISTINCT operatsiooni, mis võimaldab CTE'sse lisada ainult unikaalsed read. See võimaldab vähendada CTE's olevate ridade hulka, mida rekursiooni käigus tuleb lugeda ja see parandab töökiirust. Seega, kui rekursiooni käigus võib tekkida palju korduvaid ridu, siis on efektiivsem kasutada PostgreSQL andmebaasisüsteemi.

Andmemahu kasvades kasvab ka enamikele päringutele kuluv aeg. Erandiks on vaid ridade lisamine, mis on kõikide andmemahtude juures samas suurusjärgus. Tabelisse rea lisamisel tuleb täiendada ka tabelile loodud indekseid, kuid sellel pole vaatamata indeksite mahu kasvamisele koos tabelite mahuga märgatavat mõju üksiku rea lisamise

kiirusele. Kõige märkimisväärselt mõjutab andmemahu muutmine sügavuti rekursiivset päringut juhul, kui ühes andmemahu suurenemisega suureneb ka võimalike teede arv lähtesõlmest.

Kokkuvõtteks võib öelda, et kui kasutatakse Oracle andmebaasisüsteemi, siis oleks parim valik *külgnevusnimistu* disain. Kui aga kasutatakse PostgreSQL andmebaasisüsteemi ja sügavuti rekursiivseid päringud on vähekasutatavad, võiks kasutada samuti *külgnevusnimistu* disaini. Juhul kui rekursiivseid päringuid tehakse palju, siis võiks kasutada *seoste suunatud graafi* disaini.

Milline on minu subjektiivne hinnang SQL-andmebaasides graafidega töötamisele? Uuritud andmekäitluse lausete põhjal ei tundunud Oracle ja PostgreSQL andmebaasides graafidega töötamine väga keerukas. Rekursiivsete päringute mugavaks tegemiseks on võimalik kasutada CTEd. Oracle juures oli aga tehtud rekursiivse päringu puhul oluline puudujääk – see ei võimaldanud kasutada DISTINCT operaatorit rekursiivses osas. Suure hulga naabrite korral olid rekursiivsed päringud sügavusele kaks üsna aeglased. Ülejäänud operatsioonid olid aga võrdlemisi kiired. Seega üks suund graafide paremaks toetamiseks SQL-andmebaasides oleks optimeerida rekursiivseid päringuid. Selleks et öelda, kas ja milliste operatsioonide korral oleks otstarbekam kasutada graafipõhiseid andmebaasisüsteeme, peaks samad katsed läbi viima ka mõnes graafimudelit toetavas andmebaasisüsteemis.

8 Kokkuvõte

Graafidena esitatavad andmed on laialt levinud. Graafideks on näiteks sotsiaalvõrgud, transpordivõrgud, tehnilised infovahetuse võrgud. Seega on vaja neid kuidagi ka andmebaasis esitada. Graafi andmemudelil põhinevad andmebaasisüsteemid on disainitud selliste andmete hoidmiseks ja võimaldavad teha lihtsalt erinevaid graafipäringuid. Graafi andmemudelil põhinev andmebaasisüsteem pole aga ainuke võimalus graafide säilitamiseks. SQL-andmebaasisüsteemid on 2017. aasta kevade seisuga populaarseimad andmebaasisüsteemid. Need on olnud turul juba aastakümneid. On olemas mitmeid disainilahendusi, kuidas graafi kujul olevaid andmeid SQL-andmebaasis hoida. Samas pole ma leidnud allikat, kus need kõik oleks korraga välja toodud. Samuti pole ma leidnud eksperimente, kus oleks võrreldud omavahel erinevaid disainilahendusi.

Magistritöö üheks eesmärgiks oli süstematiseerida SQL-andmebaasides graafide esitamise disainilahendusi ning ühtlustada ja struktureerida nende esitust. Magistritöö teiseks eesmärgiks oli katsetada konkreetse probleemvaldkonna (lennumarsruutide andmebaas) näitel mõnda nendest disainilahendustest ning uurida nende lahenduste omadusi.

Tõin mustrite formaadis välja leitud disainilahendused graafide esitamiseks SQL-andmebaasides. Jätsin vaatluse alt välja disainid, kus graafe hoitakse XML ja JSON dokumentide kujul, kuna need on oma olemuselt pigem dokumendimudeli disainid ja SQL-andmebaas on üks dokumentide hoidmise võimalustest. Samuti ei esita ma disaine, mis võimaldavad säilitada informatsiooni servade ja sõlmede eksisteerimise kohta kindlal ajahetkel. Disainide kataloogi lõpus esitasin kokkuvõte disainidest, kust on näha, milline disain sobib milliste omadustega graafide esitamiseks. Töö teises osas kavandasin ja viisin läbi eksperimendi lennuliinide andmebaasi näitel. Graafiks on antud juhul suunatud kaalutud tsükliline multigraaf. Tegin eksperimendi kahes populaarses SQL-andmebaasisüsteemis – PostgreSQL's (9.5) ja Oracle's (12c Enterprise Edition, Release 1). Eksperimendi käigus võrdlesin omavahel kaht lennuliinide süsteemi esitamiseks sobilikku disaini – *külgnevusnimistu* ja *seoste suunatud graafi* disaini. Selleks projekteerisin kõigepealt andmebaasid ning täitsin need testandmetega. Seejärel viisin läbi lugemise, muutmise ja kustutamise operatsioone. Eksperimendi tulemuste põhjal

analüüsisin disainilahenduste sobivust konkreetse ülesande lahendamiseks erinevate päringute, andmehulkade ning andmebaasisüsteemide korral.

Eksperimendi tulemused näitasid, et Oracle andmebaasisüsteemis on kiiruse seisukohalt kasulikum kasutada *külgnevusnimistu* disaini. PostgreSQL andmebaasisüsteemis on *külgnevusnimistu* disaini soovitavam kasutada, kui tehakse palju andmete lisamise ja muutmise operatsioone. Samuti on naabrite arvu leidmine kiirem *külgnevusnimistu* disaini puhul. Kui aga on tarvis teha palju rekursiivseid päringuid, siis võiks kasutada *seoste suunatud graafi* disaini.

Andmekäitluskeele lausete koodi keerukuse seisukohalt Oracle ja PostgreSQL vahel erinevusi eriti pole. Minu subjektiivsel hinnangul on *külgnevusministu* disain mõnevõrra lihtsam kui *seoste suunatud graafi* oma. Ka salvestusruumi kasutuse seisukohast on *külgnevusnimistu* disain parem. PostgreSQLis võtab andmete salvestamine vähem ruumi kui Oracle's.

Andmemahu kastades kasvab enamikel päringutel päringu tegemiseks kulunud aeg. Ainult rea lisamisel ei sõltu päringu tegemiseks kuluv aeg andmemahu suurusest.

Eksperimenteerisin magistritöös ainult kahe disainiga, sest need olid ainsad SQL-andmebaasi jaoks mõeldud graafide esitamise disainid, mis sobisid lähteülesande lahendamiseks. Üks võimalik töö edasiarendus oleks valida mingi teistsugune lähteülesanne, kus graafi omadused oleksid erinevad lennuliinide omast, ja võrrelda omavahel uue lähteülesande jaoks sobivaid disaine. Graafistruktuuriga andmete esitamisega edasi minnes tundub Oracle puhul perspektiivikam keskenduda *külgnevusnimistu* disainile ning üritada seda optimeerida (st uurida erinevaid jõudluse parandamise võimalusi). PostgreSQL puhul pole aga võimalik nii selgesõnaliselt öelda, millist disaini eelistada. Nii et seal võiks teha veel katseid erinevate disainidega. Teiseks võimalikuks edasiarenduseks on võrrelda omavahel graafide esitamist SQL-admebaasi erinevate disainide korral ja graafi andmemudelitel põhinevates andmebaasisüsteemides loodud andmebaasides.

Kasutatud kirjandus

1. „DB-Engines Ranking," [Võrgumaterjal] https://db-engines.com/en/ranking. (03.05.2017).
2. P. J. Sadalage ja M. Fowler, NoSQL distilled : a brief guide to the emerging world of polyglot persistence, 2013.
3. M. Fowler, „PolyglotPersistence," 2011. [Võrgumaterjal] https://martinfowler.com/bliki/PolyglotPersistence.html. (03.05.2017).
4. K. Krönström, „Hierarhiliste andmete esitamine SQL-andmebaasides kolme disainilahenduse näitel," Magistritöö, TTÜ Informaatikainstituut, 2015.
5. A. Buldas, P. Laud ja J. Villemson, Graafid, 2 toim., Tartu: Tartu Ülikooli Kirjastus, 2008.
6. J. Celko, Joe Celko's Trees and Hierarchies in SQL for Smarties, 2, Toim., Elsevier Science, 2012.
7. „Introduction to Graph Theory," [Võrgumaterjal] http://www.math.fsu.edu/~pkirby/mad2104/SlideShow/s6_1.pdf. (22.04.2017).
8. I. Petuhhov, „Graaf," [Võrgumaterjal] http://www.cs.tlu.ee/~inga/alg_andm/graph_2010.pdf. (03.05.2017).
9. R. Angles ja C. Gutierrez, „Survey of graph database models," *ACM Computing Surveys,* kd. 40, nr 1, 2008.
10. M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O'Reilly Media, Inc., 2017.
11. T. Haigh, „How Charles Bachman invented the DBMS, a foundation of our digital world," *Communications of the ACM,* kd. 59, nr 7, pp. 25-30, 2016.
12. M. Neunhöffer, „Graphs in data modeling—is the emperor naked?," 2015. [Võrgumaterjal] https://medium.com/@neunhoef/graphs-in-data-modeling-is-the-emperor-naked-2e65e2744413#.761cmqiko. (28.03.2017).
13. „Model One-to-Many Relationships with Document References," [Võrgumaterjal] https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/. (03.05.2017).
14. „Data Model Design," [Võrgumaterjal] https://docs.mongodb.com/manual/core/data-model-design/#data-modeling-referencing. (03.05.2017).
15. R. Hull ja C. K. Yap, „The Format Model: A Theory of database Organization," *Journal of the ACM,* kd. 31, nr 3, pp. 518-544, 1984.
16. G. M. Kuper, „The logical data model: a new approach to database logic," Doktoritöö, Stanford University, 1986.
17. M. Gyssens, J. Paredaens, J. v. d. Bussche ja D. v. Gucht, „A graph-oriented object database model," *IEEE Transactions on Knowledge and Data Engineering,* kd. 6, nr 4, 1994.

18. R. Angles, „A Comparison of Current Graph Database Models," *ICDEW '12 Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops,* pp. 171-177, 2012.
19. O. v. Rest, S. Hong, J. Kim, X. Meng ja H. Chafi, „PGQL: a Property Graph Query Language," *GRADES '16 Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems,* 2016.
20. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu ja Y. Wu, „Graph Pattern Matching: From Intractable to Polynomial Time," *Proceedings of the VLDB Endowment,* kd. 10, nr 1-2, pp. 264-275 , 2010.
21. „ArangoDB," [Võrgumaterjal] https://www.arangodb.com/. (07.05.2017).
22. „OrientDB," [Võrgumaterjal] http://orientdb.com/. (07.05.2017).
23. S. Pimentel, „The rise of the multimodel database," 2015. [Võrgumaterjal] http://www.infoworld.com/article/2861579/database/the-rise-of-the-multimodel-database.html. (22.04.2017).
24. E. F. Codd, „Derivability, redundancy and consistency of relations stored in large data banks," 1969.
25. „Relational model," [Võrgumaterjal] https://en.wikipedia.org/wiki/Relational_model. (28.03.2017).
26. C. J. Date ja H. Darwen, Databases, types, and the relational model, 3 toim., China Machine Press, 2007.
27. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea ja B. Bhattacharjee, „Building an Efficient RDF Store Over a Relational Database," *SIGMOD '13 Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data,* pp. 121-132, 2012.
28. „Apache Jena," [Võrgumaterjal] https://jena.apache.org/index.html. (04.04.2017).
29. „3Store," [Võrgumaterjal] http://threestore.sourceforge.net/. (07.05.2017).
30. A. Chebotkoa, S. Lub ja F. Fotouhib, „Semantics preserving SPARQL-to-SQL translation," *Data & Knowledge Engineering,* kd. 68, nr 10, pp. 973-1000, 2009.
31. „Ontop," [Võrgumaterjal] http://ontop.inf.unibz.it/. (04.04.2017).
32. „D2RQ," [Võrgumaterjal] http://d2rq.org/. (07.05.2017).
33. „Hierarchical and recursive queries in SQL," [Võrgumaterjal] https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL. (07.05.2017).
34. „Using Common Table Expressions," [Võrgumaterjal] https://technet.microsoft.com/en-us/library/ms190766(v=sql.105).aspx. (05.05.2017).
35. „Recursive Queries Using Common Table Expressions," [Võrgumaterjal] https://technet.microsoft.com/en-us/library/ms186243(v=sql.105).aspx. (04.05.2017).
36. „Recursive WITH, part II: Hierarchical queries," 04 10 2014. [Võrgumaterjal] http://rdbms-insight.com/wp/2014/recursive-with-part-ii-hierarchical-queries/. (07.05.2017).
37. „WITH Queries (Common Table Expressions)," [Võrgumaterjal] https://www.postgresql.org/docs/9.6/static/queries-with.html. (21.03.2017).
38. L. Schneider, „Connect By Loop," [Võrgumaterjal] http://www.dba-oracle.com/t_advanced_sql_connect_by_loop.htm. (21.03.2017).

39. „tablefunc," [Võrgumaterjal] https://www.postgresql.org/docs/9.6/static/tablefunc.html. (04.05.2017).
40. „XML Type," [Võrgumaterjal] https://www.postgresql.org/docs/current/static/datatype-xml.html. (25.03.2017).
41. „JSON in Oracle Database," [Võrgumaterjal] https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6275. (25.03.2017).
42. „JSON Types," [Võrgumaterjal] https://www.postgresql.org/docs/9.6/static/datatype-json.html. (25.03.2017).
43. „Using XMLType," [Võrgumaterjal] https://docs.oracle.com/cd/B10501_01/appdev.920/a96620/xdb04cre.htm. (25.03.2017).
44. J. Celko, Joe Celko's SQL for Smarties, 4 toim., 2010.
45. M. Blaha, Patterns of Data Modeling, 2010.
46. R. Angles, A. Prat-Pérez, D. Dominguez-Sal ja J.-L. Larriba-Pey, „Benchmarking database systems for social network applications," *GRADES '13 First International Workshop on Graph Data Management Experiences and Systems,* pp. 1-7, 2013.
47. C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen ja D. Wilkins, „A comparison of a graph database and a relational database: a data provenance perspective," *ACM SE '10 Proceedings of the 48th Annual Southeast Regional Conference,* 2010.
48. A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi ja J. Banerjee, „Graph analysis: do we have to reinvent the wheel?," *GRADES '13 First International Workshop on Graph Data Management Experiences and Systems,* pp. 1-6, 2013.
49. A. Gubichev ja M. Then, „Graph Pattern Matching: Do We Have to Reinvent the Wheel?," *GRADES'14 Proceedings of Workshop on GRAph Data management Experiences and Systems,* pp. 1-7, 2014.
50. A. Jindal, S. Madden, M. Castellanos ja M. Hsu, „Graph analytics using vertica relational database," *BIG DATA '15 Proceedings of the 2015 IEEE International Conference on Big Data,* pp. 1191-1200, 2015.
51. „Google Maps," [Võrgumaterjal] https://www.google.ee/maps. (03.05.2017).
52. B. Karwin, SQLAntipatterns. Avoiding the Pitfalls of Database Programming, 2010.
53. V. Tropashko, „Nested Intervals Tree Encoding in SQL," [Võrgumaterjal] https://sigmodrecord.org/publications/sigmodRecord/0506/p47-article-tropashko.pdf. (11.02.2017).
54. V. Tropashko, „Nested Intervals with Farey Fractions," 2004. [Võrgumaterjal] https://arxiv.org/html/cs/0401014. (02.04.2017).
55. „Farey Sequence," [Võrgumaterjal] http://mathworld.wolfram.com/FareySequence.html. (17.04.2017).
56. „Problems with Floating-Point Values," [Võrgumaterjal] https://dev.mysql.com/doc/refman/5.7/en/problems-with-float.html. (07.05.2017).
57. K. Fling, „Four Ways To Work With Hierarchical Data," detsember 2000. [Võrgumaterjal] http://evolt.org/node/4047. (11.02.2017).

58. D. Chandler, „Method of organizing hierarchical data in a relational database,“ november 2002. [Võrgumaterjal] http://www.google.com/patents/US6480857. (12.02.2017).
59. K. Y. J. F. Chong, „Method of implementing an acyclic directed graph structure using a relational data-base,“ oktoober 2003. [Võrgumaterjal] https://www.google.com/patents/US6633886. (21.02.2017).
60. R. N. Goldberg ja G. A. Jirak, „Relational database management system and method for storing, retrieving and modifying directed graph data structures,“ aprill 1993. [Võrgumaterjal] https://www.google.com/patents/US5201046. (21.02.2017).
61. „The simplest(?) way to do tree-based queries in SQL,“ 2010. [Võrgumaterjal] http://dirtsimple.org/2010/11/simplest-way-to-do-tree-based-queries.html. (26.03.2017).
62. K. Erdogan, „A Model to Represent Directed Acyclic Graphs (DAG) on SQL Databases,“ 2008. [Võrgumaterjal] https://www.codeproject.com/Articles/22824/A-Model-to-Represent-Directed-Acyclic-Graphs-DAG-o. (26.03.2017).
63. „Markup Languages: Comparison and Examples,“ [Võrgumaterjal] http://www.isi.edu/expect/web/semanticweb/comparison.html. (04.05.2017).
64. „Defining a graph with node distances in OWL,“ 2014. [Võrgumaterjal] http://stackoverflow.com/questions/23393515/defining-a-graph-with-node-distances-in-owl. (04.05.2017).
65. „Airport, airline and route data,“ [Võrgumaterjal] http://openflights.org/data.html. (19.02.2017).
66. C. Gallant, „What is the difference between arithmetic and geometric averages?,“ [Võrgumaterjal] http://www.investopedia.com/ask/answers/06/geometricmean.asp. (03.05.2017).
67. V. Nguyen, S. Deeds-Rubin, T. Tan ja B. Boehm, „A SLOC Counting Standard,“ [Võrgumaterjal] http://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf. (01.05.2017).
68. „System Administration Functions,“ [Võrgumaterjal] https://www.postgresql.org/docs/9.5/static/functions-admin.html. (03.05.2017).
69. „How do you find the disk size of a Postgres / PostgreSQL table and its indexes,“ [Võrgumaterjal] http://stackoverflow.com/questions/2596624/how-do-you-find-the-disk-size-of-a-postgres-postgresql-table-and-its-indexes. (03.05.2017).
70. S. Stadnik, „How to get size of all tables in an Oracle database schema,“ 2013. [Võrgumaterjal] http://ozmoroz.com/2013/08/how-to-get-size-of-all-tables-in-oracle.html#.WQotN4VOKM8. (03.05.2017).
71. „International Civil Aviation Organization airport code,“ [Võrgumaterjal] https://en.wikipedia.org/wiki/International_Civil_Aviation_Organization_airport_code. (01.05.2017).
72. „Pearson Correlation Coefficient Calculator,“ [Võrgumaterjal] http://www.socscistatistics.com/tests/pearson/Default2.aspx. (03.05.2017).
73. M. Winand, „Operations,“ [Võrgumaterjal] http://use-the-index-luke.com/sql/explain-plan/postgresql/operations. (07.05.2017).

Lisa 1 – tabelite loomise laused Oracle andmebaasis

```
CREATE TABLE KYLGNEVUS_LENNUFIRMA
(
  LENNUFIRMA_ID             NUMBER(5),
  NIMETUS                   VARCHAR2(150 CHAR) NOT NULL,
  CONSTRAINT pk_k_lennufirma PRIMARY KEY (lennufirma_id)
);

CREATE TABLE KYLGNEVUS_LENNUJAAM
(
  LENNUJAAM_KOOD            VARCHAR2(4 CHAR),
  NIMETUS                   VARCHAR2(100 CHAR) NOT NULL,
  LINN                      VARCHAR2(50 CHAR),
  RIIK                      VARCHAR2(50 CHAR) NOT NULL,
  CONSTRAINT pk_k_lennujaam PRIMARY KEY (lennujaam_kood)
);
COMMENT ON COLUMN kylgnevus_lennujaam.lennujaam_kood is 'ICAO
kood';
CREATE INDEX idx_k_lennujaam_riik ON kylgnevus_lennujaam(riik);

CREATE TABLE KYLGNEVUS_LENNULIIN
(
  LENNULIIN_ID              NUMBER(5) GENERATED BY DEFAULT AS
IDENTITY (START WITH 1 INCREMENT BY 1),
  ALGUSLENNUJAAM_KOOD       VARCHAR2(4 CHAR) NOT NULL,
  SIHTLENNUJAAM_KOOD        VARCHAR2(4 CHAR) NOT NULL,
  LENNUFIRMA_ID             NUMBER(5) NOT NULL,
  VAHEPEATUSI               NUMBER(1) NOT NULL,
  CONSTRAINT pk_k_lennuliin PRIMARY KEY (lennuliin_id),
  CONSTRAINT fk_k_lennuliin_algus FOREIGN KEY
(alguslennujaam_kood) REFERENCES
kylgnevus_lennujaam(lennujaam_kood) ON DELETE CASCADE,
  CONSTRAINT fk_k_lennuliin_siht FOREIGN KEY
(sihtlennujaam_kood) REFERENCES
kylgnevus_lennujaam(lennujaam_kood) ON DELETE CASCADE,
  CONSTRAINT fk_k_lennuliin_lennufirma FOREIGN KEY
(lennufirma_id) REFERENCES kylgnevus_lennufirma(lennufirma_id)
ON DELETE CASCADE,
  CONSTRAINT lennuliin_lennujaam_check CHECK
(alguslennujaam_kood != sihtlennujaam_kood)
```

```
);

CREATE TABLE SEOS_LENNUFIRMA
(
  LENNUFIRMA_ID            NUMBER(5),
  NIMETUS                  VARCHAR2(150 CHAR) NOT NULL,
  CONSTRAINT pk_s_lennufirma PRIMARY KEY (lennufirma_id)
);

CREATE TABLE SEOS_LENNUJAAM
(
  LENNUJAAM_KOOD           VARCHAR2(4 CHAR),
  NIMETUS                  VARCHAR2(100 CHAR) NOT NULL,
  LINN                     VARCHAR2(50 CHAR),
  RIIK                     VARCHAR2(50 CHAR) NOT NULL,
  CONSTRAINT pk_s_lennujaam PRIMARY KEY (lennujaam_kood)
);
COMMENT ON COLUMN seos_lennujaam.lennujaam_kood is 'ICAO kood';
CREATE INDEX idx_s_lennujaam_riik ON seos_lennujaam(riik);

CREATE TABLE SEOS_LEND
(
  LEND_ID                  NUMBER(5) GENERATED BY DEFAULT AS
IDENTITY (START WITH 1 INCREMENT BY 1),
  LENNUFIRMA_ID            NUMBER(5) NOT NULL,
  VAHEPEATUSI              NUMBER(1) NOT NULL,
  CONSTRAINT pk_s_lend PRIMARY KEY (lend_id),
  CONSTRAINT fk_s_lend_lennufirma FOREIGN KEY (lennufirma_id)
REFERENCES seos_lennufirma(lennufirma_id) ON DELETE CASCADE
);

CREATE TABLE SEOS_SEOSE_SUUND_R
(
  SEOSE_SUUND_KOOD         NUMBER(1),
  NIMETUS                  VARCHAR2(10 CHAR) NOT NULL,
  CONSTRAINT pk_s_seose_suund PRIMARY KEY (seose_suund_kood),
  CONSTRAINT uk_s_seose_suund_nimetus UNIQUE (nimetus)
);

CREATE TABLE SEOS_LENNULIIN
(
```

```
  LENNULIIN_ID              NUMBER(6) GENERATED BY DEFAULT AS
IDENTITY (START WITH 1 INCREMENT BY 1),
  LENNUJAAM_KOOD            VARCHAR2(4 CHAR) NOT NULL,
  LEND_ID                   NUMBER(5) NOT NULL,
  SEOSE_SUUND_KOOD          NUMBER(1) NOT NULL,
  CONSTRAINT pk_s_lennuliin PRIMARY KEY (lennuliin_id),
  CONSTRAINT fk_s_lennuliin_lennujaam FOREIGN KEY
(lennujaam_kood) REFERENCES seos_lennujaam(lennujaam_kood) ON
DELETE CASCADE,
  CONSTRAINT fk_s_lennuliin_lend FOREIGN KEY (lend_id)
REFERENCES seos_lend(lend_id) ON DELETE CASCADE,
  CONSTRAINT fk_s_lennuliin_seos FOREIGN KEY (seose_suund_kood)
REFERENCES seos_seose_suund_r(seose_suund_kood)
);
```

Lisa 2 – tabelite loomise laused PostgreSQL andmebaasis

```
CREATE SCHEMA kylgnevus;

CREATE TABLE kylgnevus.LENNUFIRMA
(
  LENNUFIRMA_ID             INTEGER PRIMARY KEY,
  NIMETUS                   VARCHAR(150) NOT NULL
);

CREATE TABLE kylgnevus.LENNUJAAM
(
  LENNUJAAM_KOOD            VARCHAR(4) PRIMARY KEY,
  NIMETUS                   VARCHAR(100) NOT NULL,
  LINN                      VARCHAR(50),
  RIIK                      VARCHAR(50) NOT NULL
);
CREATE INDEX lennujaam_riik_idx ON kylgnevus.lennujaam(riik);

CREATE SEQUENCE kylgnevus.lennuliin_id_seq;
CREATE TABLE kylgnevus.LENNULIIN
(
```

```
  LENNULIIN_ID              INTEGER PRIMARY KEY DEFAULT
NEXTVAL('kylgnevus.lennuliin_id_seq'),
  ALGUSLENNUJAAM_KOOD       VARCHAR(4) NOT NULL,
  SIHTLENNUJAAM_KOOD        VARCHAR(4) NOT NULL,
  LENNUFIRMA_ID             INTEGER,
  VAHEPEATUSI               SMALLINT NOT NULL,
  CONSTRAINT lennuliin_alguskoht_kood_fkey FOREIGN KEY
(alguslennujaam_kood) REFERENCES
kylgnevus.lennujaam(lennujaam_kood) ON DELETE CASCADE,
  CONSTRAINT lennuliin_sihtkoht_kood_fkey FOREIGN KEY
(sihtlennujaam_kood) REFERENCES
kylgnevus.lennujaam(lennujaam_kood) ON DELETE CASCADE,
  CONSTRAINT lennuliin_lennufirma_id_fkey FOREIGN KEY
(lennufirma_id) REFERENCES kylgnevus.lennufirma(lennufirma_id)
ON DELETE CASCADE,
  CONSTRAINT lennuliin_lennujaam_check CHECK
(alguslennujaam_kood != sihtlennujaam_kood)
);
ALTER SEQUENCE kylgnevus.lennuliin_id_seq OWNED BY
kylgnevus.lennuliin.lennuliin_id;

CREATE SCHEMA seos;

CREATE TABLE seos.LENNUFIRMA
(
  LENNUFIRMA_ID             INTEGER PRIMARY KEY,
  NIMETUS                   VARCHAR(150) NOT NULL
);

CREATE TABLE seos.LENNUJAAM
(
  LENNUJAAM_KOOD            VARCHAR(4) PRIMARY KEY,
  NIMETUS                   VARCHAR(100) NOT NULL,
  LINN                      VARCHAR(50),
  RIIK                      VARCHAR(50) NOT NULL
);
CREATE INDEX lennujaam_riik_idx ON seos.lennujaam(riik);

CREATE SEQUENCE seos.lend_id_seq;
CREATE TABLE seos.LEND
(
```

```
  LEND_ID                  INTEGER PRIMARY KEY DEFAULT
NEXTVAL('seos.lend_id_seq'),
  LENNUFIRMA_ID            INTEGER NOT NULL,
  VAHEPEATUSI              SMALLINT NOT NULL,
  CONSTRAINT lend_lennufirma_id_fkey FOREIGN KEY (lennufirma_id)
REFERENCES seos.lennufirma(lennufirma_id) ON DELETE CASCADE
);
ALTER SEQUENCE seos.lend_id_seq OWNED BY seos.lend.lend_id;

CREATE TABLE seos.SEOSE_SUUND_R
(
  SEOSE_SUUND_KOOD         SMALLINT PRIMARY KEY,
  NIMETUS                  VARCHAR(10) NOT NULL,
  CONSTRAINT seose_suund_r_nimetus_key UNIQUE (nimetus)
);

CREATE SEQUENCE seos.lennuliin_id_seq;
CREATE TABLE seos.LENNULIIN
(
  LENNULIIN_ID             INTEGER PRIMARY KEY DEFAULT
NEXTVAL('seos.lennuliin_id_seq'),
  LENNUJAAM_KOOD           VARCHAR(4) NOT NULL,
  LEND_ID                  INTEGER NOT NULL,
  SEOSE_SUUND_KOOD         SMALLINT NOT NULL,
  CONSTRAINT lennuliin_lennujaam_id_fkey FOREIGN KEY
(lennujaam_kood) REFERENCES seos.lennujaam(lennujaam_kood) ON
DELETE CASCADE,
  CONSTRAINT lennuliin_lend_id_fkey FOREIGN KEY (lend_id)
REFERENCES seos.lend(lend_id) ON DELETE CASCADE,
  CONSTRAINT lennuliin_seose_suund_fkey FOREIGN KEY
(seose_suund_kood) REFERENCES
seos.seose_suund_r(seose_suund_kood) ON UPDATE CASCADE
);
ALTER SEQUENCE seos.lennuliin_id_seq OWNED BY
seos.lennuliin.lennuliin_id;
```

Lisa 3 – andmete sisestamine seoste suunatud graafi disaini tabelitesse

```
INSERT INTO seos_seose_suund_r (seose_suund_kood, nimetus)
VALUES (1, 'LÄHE');
INSERT INTO seos_seose_suund_r (seose_suund_kood, nimetus)
VALUES (2, 'SIHT');

INSERT INTO seos_lennufirma (lennufirma_id, nimetus)
SELECT lennufirma_id, nimetus FROM kylgnevus_lennufirma;

INSERT INTO seos_lennujaam (lennujaam_kood, nimetus, linn, riik)
SELECT lennujaam_kood, nimetus, linn, riik FROM
kylgnevus_lennujaam;

INSERT INTO seos_lend (lend_id, lennufirma_id, vahepeatusi)
SELECT lennuliin_id, lennufirma_id, vahepeatusi FROM
kylgnevus_lennuliin;

INSERT INTO seos_lennuliin(lennujaam_kood, seose_suund_kood,
lend_id)
SELECT m.alguslennujaam_kood, 1, m.lennuliin_id FROM
kylgnevus_lennuliin m;
INSERT INTO seos_lennuliin(lennujaam_kood, seose_suund_kood,
lend_id)
SELECT m.sihtlennujaam_kood, 2, m.lennuliin_id FROM
kylgnevus_lennuliin m;
```

Lisa 4 – välisvõtme kitsenduste lisamine Oracle andmebaasi

```
CREATE INDEX idx_k_lennuliin_algus_lj_kood ON
kylgnevus_lennuliin(alguslennujaam_kood);
CREATE INDEX idx_k_lennuliin_siht_lj_kood ON
kylgnevus_lennuliin(sihtlennujaam_kood);
CREATE INDEX idx_k_lennuliin_lennufirma_id ON
kylgnevus_lennuliin(lennufirma_id);
```

```
CREATE INDEX idx_s_lend_lennufirma_id ON
seos_lend(lennufirma_id);
CREATE INDEX idx_s_lennuliin_lj_kood ON
seos_lennuliin(lennujaam_kood);
CREATE INDEX idx_s_lennuliin_lend_id ON seos_lennuliin(lend_id);
CREATE INDEX idx_s_lennuliin_seose_suund ON
seos_lennuliin(seose_suund_kood);
```

Lisa 5 – välisvõtme kitsenduste lisamine PostgreSQL andmebaasi

```
CREATE INDEX lennuliin_alguslennujaam_idx ON
kylgnevus.lennuliin(alguslennujaam_kood);
CREATE INDEX lennuliin_sihtlennujaam_idx ON
kylgnevus.lennuliin(sihtlennujaam_kood);
CREATE INDEX lennuliin_lennufirma_id_idx ON
kylgnevus.lennuliin(lennufirma_id);


CREATE INDEX lend_lennufirma_id_idx ON seos.lend(lennufirma_id);
CREATE INDEX lennuliin_lennujaam_idx ON
seos.lennuliin(lennujaam_kood);
CREATE INDEX lennuliin_lend_idx ON seos.lennuliin(lend_id);
CREATE INDEX lennuliin_seose_suund_kood_idx ON
seos.lennuliin(seose_suund_kood);
```

Lisa 6 – tabelite andmemahu leidmine Oracle andmebaasis

```
SELECT
  table_name,
  ROUND(SUM(bytes)/1024/1024, 2) AS mb
FROM (
  SELECT owner, segment_name AS table_name, bytes
    FROM   dba_segments
```

```
    WHERE  segment_type IN ('TABLE', 'TABLE PARTITION', 'TABLE
SUBPARTITION')
  UNION ALL
  SELECT i.owner, i.table_name, s.bytes
    FROM   dba_indexes i, dba_segments s
    WHERE  s.segment_name = i.index_name
    AND    s.owner = i.owner
    AND    s.segment_type IN ('INDEX', 'INDEX PARTITION', 'INDEX
SUBPARTITION')
  UNION ALL
  SELECT l.owner, l.table_name, s.bytes
    FROM   dba_lobs l, dba_segments s
    WHERE  s.segment_name = l.segment_name
    AND    s.owner = l.owner
    AND    s.segment_type = 'LOBSEGMENT'
  UNION ALL
  SELECT l.owner, l.table_name, s.bytes
    FROM   dba_lobs l, dba_segments s
    WHERE  s.segment_name = l.index_name
    AND    s.owner = l.owner
    AND    s.segment_type = 'LOBINDEX'
)
WHERE owner = 'C##TUD10'
AND REGEXP_LIKE(table_name, '^KYLGNEVUS*|^SEOS*')
GROUP BY table_name;
```

Lisa 7 – tabelite andmemahu leidmine PostgreSQL andmebaasis

```
SELECT
    schema_name,
    round(sum(total_size)/1024/1024, 2) mb
FROM (
    SELECT
        schema_name,
        pg_total_relation_size(table_name) AS total_size
    FROM (
```

```
        SELECT ('"' || table_schema || '"."' || table_name ||
'"') AS table_name,
        table_schema AS schema_name
        FROM information_schema.tables
        WHERE table_schema IN ('kylgnevus', 'seos')
    ) AS all_tables
    ORDER BY total_size DESC
) AS sizes
GROUP BY schema_name
```