TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Hierarhiliste andmete esitamine SQL-andmebaasides kolme disainilahenduse näitel

Magistritöö

Üliõpilane: Katerina Krönström

Üliõpilaskood: 121831 IABMM

Juhendaja: dotsent Erki Eessaar

Tallinn

2015

Autodeklaratsioon

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.

....	....
(*kuupäev*)	(*allkiri*)

Annotatsioon

Krönström K. (2015) Hierarhiliste andmete esitamine SQL-andmebaasides kolme disainilahenduse näitel. Magistritöö, Tallinna Tehnikaülikool.

Käesoleva magistritöö põhieesmärgiks on võrrelda SQL-andmebaasides hierarhiliste andmete esitamiseks mõeldud disainilahendusi, keskendudes lahenduste uurimise põhjalikkuse huvides võimalike disainilahenduste ühele pärisalamhulgale. Uuritavateks disainilahendusteks on valitud külgnevusnimistu, pesatatud hulkade ja materialiseeritud tee disainilahendused, mis on autori hinnangul selle valdkonna disainilahendustest kõige populaarsemad. Kuigi neid disaine kasutatakse sageli, ei ole nende kohta piisavalt eksperimentidel põhinevat infot, et konkreetse andmebaasi kavandamise olukorras osata valida sobivaim disainilahendus. Uuringu käigus tutvutakse nii selle valkdonna kohta ilmunud materjalidega kui tehakse katseid. Töö esimeses osas analüüsitakse ning pannakse ühesugust struktuuri kasutades kirja hierarhilise struktuuriga andmete esitamist kirjeldavad disainilahendused. Magistritöö teises osas projekteeritakse eksperimendi andmebaasid. Uuringus kasutatavateks andmebaasisüsteemideks on Oracle Database ja PostgreSQL. Disainilahenduste võrdlemiseks viiakse kavandatud andmebaasides läbi andmete lugemise, muutmise ja kustutamise operatsioonid. Disainilahendusi võrreldakse operatsioonide kiiruste, erinevate disainilahenduste korral tekkivate andmemahtude ning päringute ja andmemuudatuse lausete keerukuse seisukohast. Magistritöö tulemusena analüüsitakse iga uuritava disainilahenduse eeliseid ja puuduseid.

Magistritöö on koostatud eesti keeles ning koosneb 119 leheküljest. Töö sisaldab 34 joonist ja 17 tabelit.

Abstract

Krönström K. (2015) Representing Hierarchical Data in SQL Databases in the Example of Three Design Solutions. Master's Thesis, Tallinn University of Technology.

The main purpose of the current Master's Thesis is to compare different design solutions for representing hierarchical data in SQL databases. In order to make this investigation more thorough the author concentrates to a proper subset of possible design solutions. The investigated design solutions are adjacency list, nested sets, and materialized path that are in the author's opinion the most popular in the field of this kind of design solutions. Despite their widespread usage there is not enough experimental data about them that would enable database designers to select the best design solution for a given database. The author investigates existing materials about this topic as well as conducts experiments. In the first part of the Thesis, the author analyzes different SQL database design solutions that enable us to represent hierarchical data and specifies these by using the same structure. In the second part of the Thesis, the author designs databases that will be used in the tests. The database management systems that the author uses for the investigation are Oracle Database and PostgreSQL. In order to compare different design solutions, the author carries out select, update, and delete operations in the designed databases. The author compares the design solutions in terms of performance of operations, data size, and complexity of query and data modification statements. As a result of the Thesis, the advantages and disadvantages of each considered design solution are analyzed.

Master's Thesis is written in Estonian and there are 119 pages. Thesis includes 34 figures and 17 tables.

Lühendite ja mõistete nimekiri

Andmebaasisüsteem – Database Management System (DBMS)

Anonüümne plokk – Anonymous block

Arhetüüp – Archetype

CTE – Common Table Expression

Graaf – Graph

Hierarhia – Hierarchy

IBM – Information Management System

Puu – Tree

SQL-andmebaas – SQL database

SQL – Structured Query Language

TTÜ – Tallinna Tehnikaülikool

Täitmisplaan – Execution Plan

Vihje - Hint

Jooniste nimekiri

Joonis 1. Graafi näide....14
Joonis 2. Puu näide....15
Joonis 3. Hierarhia näide (ametite hierarhia)....15
Joonis 4. Graafiteooria kirjeldav mõistekaart....16
Joonis 5. Jaotises 1.2. käsitlevaid mõisteid ja nende vahelisi seoseid kirjeldav mõistekaart....22
Joonis 6. Töötajate hierarhia....25
Joonis 7. Töötajate olemi-suhte diagramm.....26
Joonis 8. Külgnevusnimistu disaini näide konkreetse andmetega....28
Joonis 9. Alternatiivne võimalus hierarhia esitamiseks....30
Joonis 10. Sõlmede numereerimine....30
Joonis 11. Pesastatud hulkade mudeli näide konkreetsete andmetega....31
Joonis 12. Pesastatud intervallid....33
Joonis 13. Pesastatud intervallide mudeli näide konkreetsete andmetega....34
Joonis 14. Sulunditabeli mudeli näide konkreetsete andmetega....37
Joonis 15. Lametabeli mudeli näide konkreetsete andmetega....40
Joonis 16. Materialiseeritud tee mudeli näide konkreetsete andmetega....43
Joonis 17. Hulk pärilikkuse veerge mudeli näide konkreetsete andmetega....45
Joonis 18. Tase-tabeli mudeli näide konkreetsete andmetega....47
Joonis 19. Otseste järeltulijate grupid mudeli näide konkreetsete andmetega....48
Joonis 20. Struktureeritud puu mudeli näide konkreetsete andmetega....51
Joonis 21. Osapoolte näite olemi-suhte diagramm....59
Joonis 22. Külgnevusnimistu disaini andmebaasi diagramm....63
Joonis 23. Pesastatud hulkade disaini andmebaasi diagramm....64
Joonis 24. Materialiseeritud tee disaini andmebaasi diagramm....65
Joonis 25. S1A päringu täitmisplaan Oracle andmebaasisüsteemis (25 000 rida)....81
Joonis 26. S1A päringu täitmisplaan PostgreSQL andmebaasisüsteemis (25 000 rida)....82
Joonis 27. S2A päringu täitmisplaan Oracle andmebaasisüsteemis (25 000 rida)....84
Joonis 28. S2A päringu täitmisplaan PostgreSQL andmebaasisüsteemis (25 000 rida)....85
Joonis 29. S2M päringu täitmisplaan Oracle andmebaasisüsteemis (25 000 rida)....85
Joonis 30. S2M päringu täitmisplaan PostgreSQL andmebaasisüsteemis (25 000 rida)....86
Joonis 31. S2A päringu täitmisplaan Oracle andmebaasisüsteemis (9 000 rida)....87
Joonis 32. S2M päringu täitmisplaan Oracle andmebaasisüsteemis (9 000 rida)....88
Joonis 33. S2N päringu täitmisplaan Oracle andmebaasisüsteemis (9 000 rida)....89
Joonis 34. S2N päringu täitmisplaan PostgreSQL andmebaasisüsteemis (9 000 rida)....91

Tabelite nimekiri

Tabel 1. Disainide populaarsus Google otsingu tulemuste järgi (04.11.2014) (sulgudesse on kirjutatud täpne otsingustring) 55
Tabel 2. Eksperimendi andmemahud 66
Tabel 3. Eksperimendis kasutatavad päringud ja operatsioonid 70
Tabel 4. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle ja PostgreSQL andmebaasisüsteemides (sekundites) 77
Tabel 5. Andmebaasi andmemahud erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides (MB) 78
Tabel 6. Koodiridade arv erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides ...78
Tabel 7. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle ja PostgreSQL andmebaasisüsteemides (sekundites) 79
Tabel 8. Andmebaasi andmemahud erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides (MB) 79
Tabel 9. Koodiridade arv erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides ...79
Tabel 10. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle andmebaasisüsteemis (sekundites) 92
Tabel 11. Täitmisplaanide kasutamine erinevate andmemahtude korral Oracle andmebaasisüsteemis .92
Tabel 12. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused PostgreSQL andmebaasisüsteemis (sekundites) 94
Tabel 13. Tätmisplaanide kasutamine erinevate andmemahtude korral PostgreSQL andmebaasisüsteemis 94
Tabel 14. Võrdlustabel deklaratiivsete kitsenduste kohta erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides 96
Tabel 15. CHECK kitsendused erinevate disainide korral eksperimendi Oracle andmebaasi näitel 97
Tabel 16. Andmete salvestamiseks kuluv salvestusruum erinevate tüüpide korral PostgreSQL andmebaasisüsteemis 98
Tabel 17. Andmete salvestamiseks kuluv salvestusruum erinevate tüüpide korral Oracle andmebaasisüsteemis 98

Sisukord

Sissejuhatus 10
1. Teoreetiline taust 13
1.1. Graafiteooria 13
1.1.1. Graaf 13
1.1.2. Puu ja hierarhia 14
1.2. Hierarhilise struktuuriga andmete esitamine erinevate andmemudelite korral 17
1.2.1. Hierarhilise struktuuriga andmete esitamine SQL-andmebaasides 18
2. Erinevad disainilahendused hierarhiliste andmete esitamiseks SQL-andmebaasides 23
2.1. Adjacency List 27
2.2. Nested Sets 30
2.3. Nested Intervals 33
2.4. Closure Table 36
2.5. Flat Table 39
2.6. Materialized Path 42
2.7. Multiple Lineage Columns 44
2.8. Hardcoded Tree 46
2.9. Degenerate Node and Edge 48
2.10. Structured Tree 50
3. Eksperiment 53
3.1. Varasemad uuringud 53
3.2. Eksperimendi eesmärgid 55
3.3. Eksperimendi kirjeldus 57
3.3.1. Kasutatavad andmebaasisüsteemid 58
3.4. Eksperimendi andmebaasi projekteerimine 59
3.4.1. Kontseptuaalne andmemudel 59
3.5. Testandmed 66
3.6. Eksperimendi käigus katsetatavad operatsioonid 67
4. Mõõtmiste tulemused 77
5. Tulemuste analüüs ja järeldused 80
5.1. Päringute ja andmemuudatuse operatsioonide kiirused 80
5.2. Koodi keerukus 95
5.3. Hierarhiaga seotud kitsenduste deklaratiivne jõustamine 96
5.4. Andmete salvestamiseks kulunud salvestusruum 97
5.5. Järeldused 99
Kokkuvõte 102

Summary 104
Kasutatud kirjandus...106
Lisad..109
Lisa 1 – tabelite loomise laused Oracle andmebaasis 109
Lisa 2 – tabelite loomise laused PostgreSQL andmebaasis 111
Lisa 3 - Tabelite andmemaht megabaitides Oracle andmebaasis 113
Lisa 4 – Külgnevusnimistu mudeli jaoks hierarhia genereerimine................................113
Lisa 5 – Pesastatud hulkade mudeli jaoks hierarhia genereerimine...............................116
Lisa 6 – Materialiseeritud tee mudeli jaoks hierarhia genereerimine 119

Sissejuhatus

Hierarhilised struktuurid on maailmas küllaltki levinud. Sotsioloogias on tuntud mõiste meritokraatia ehk sotsiaalne hierarhia, mis tekib inimeste võimetest ja oskustest tuleneva sotsiaalse kihistumise tulemusel. Bioloogias kasutatakse taksonoomilist hierarhiat, mille tasemed on riik, hõimkond, klass, sugukond, perekond, liik (Wikipedia 2014a). Paljud organisatsioonid (valitsus, sõjavägi, ettevõtted) on hierarhilise struktuuriga, kus on selgelt defineeritud ülemus-alluvus suhted ja käsuliin. Võib väita, et hierarhiaid on igal pool meie ümber ning nendele vastavaid andmeid on vaja kindlasti andmebaasides säilitada.

Andmebaaside loomiseks on võimalik kasutada palju erinevaid andmebaasisüsteeme. 2014. aasta sügise seisuga on populaarseimad SQL-andmebaasisüsteemid, mille üldnimi tuleb sellest, et neis kasutatakse andmebaasikeelt SQL (*Structured Query Language*) (DB-Engines Ranking 2014). 2014. aasta detsembri seisuga on seal esimese kümne andmebaasisüsteemi hulgas seitse SQL-andmebaasisüsteemi.

Tänapäevaks on välja mõeldud palju erinevaid viise, kuidas hierarhilisi andmeid SQL-andmebaasides säilitada. Samuti on SQL standardisse (alates standardi versioonist *SQL:1999*) lisandunud keelekonstruktsioonid rekursiivsete päringute tegemiseks hierarhiate põhjal.

Hierarhiliste andmete esitamine SQL andmebaasides on aktuaalne teema, kuna tõenäoliselt enamus andmebaasi kavandajatest on kokku puutunud selliste ülesannetega, mille puhul on vaja säilitada andmebaasis andmeid hierarhiate kohta. Autori isiklik kogemus näitab, et andmebaaside kavandajad valivad tavaliselt kõige tuntuma disainilahenduse (edaspidi *disaini, mudeli*) milleks on külgnevusnimistu (vt Jaotis 2.1), kuid antud disain ei sobi kõige paremini kõikide ülesannete jaoks. Kui mõne aja pärast selgub, et päringute ja operatsioonide täitmine on muutnud liiga aeglaseks (näiteks tänu andmemahu kiirele suurenemisele), siis tuleb terve töötav süsteem ümber teha. See toob kindlasti kaasa lisakulu (aja ja raha osas). Iga andmebaasi kavandaja peaks enne disaini valimist selgelt läbi mõtlema, millised on oodatavad andmemahud, milliseid päringuid hakatakse tegema, kas hierarhiates tehakse tihti muudatusi. Sõltuvalt vastustest sellistele küsimustele tuleb teha õige disaini valik.

Kuna võimalusi hierarhiaid SQL andmebaasides esitada on palju ja tänaseks on need küllatki hästi teada, siis vajavad andmebaaside kavandajad informatsiooni, mille alusel valida kõige sobivam disain. Uuringuid, kus neid disainilahendusi erinevates aspektides (näiteks päringute ja andmemuudatuste

kiirus, lausete keerukus, andmemahud) võrreldakse, on aga kahjuks vähe (leitud analoogseid uuringuid tutvustatakse magistritöö teises pooles). Käesolevas töös üritatakse seda tühikut täita.

Magistritöö eesmärgiks on võrrelda disaine hierarhiliste andmete SQL-andmebaasides hoidmise kohta. Kuna kõigi uuritavate disainidega on vaja eksperimenteerida, tuleb töö mahtu arvestades valida uurimiseks välja üks pärisalamhulk võimalikest disainidest. Uurimiseks on valitud disainid, mille kohta Google otsing leiab kõige rohkem vastuseid. Sellised disainid on leidnud kirjanduses ja praktikas kõige rohkem käsitlemist ning kõige sobivama disaini valikul võiksid olla esimesed, mida kaaluda. Need disainid on külgnevusnimistu (ingl *Adjacency List),* pesastatud hulgad (ingl *Nested Sets)* ja materialiseeritud tee (ingl *Materialized Path)*. Töös kasutatavaid andmebaasisüsteeme (Oracle Database (edaspidi ka lihtsalt Oracle) ja PostgreSQL) kasutatakse TTÜs andmebaaside õpetamisel ning autor on nendega varem kokku puutunud. Need andmebaasisüsteemid kuuluvad 2014. aasta sügisel kõige populaarsemate andmebaasisüsteemide hulka (DB-Engines Ranking 2014).

Käesolev magistritöö võib pakkuda huvi SQL-andmebaaside kavandajatele, eriti nendele, kes kasutavad Oracle Database ja/või PostgreSQL andmebaase. Magistritöö võiks olla kasulik ka SQLi kursustes erinevate hierarhiatega seotud disainilahenduste illustreerimiseks. Kuna autor kasutab Oracle Database andmebaasisüsteemi oma tööülesannete täitmisel igapäevatöös, siis aitavad antud töö tulemused autori ettevõttel tulevikus potentsiaalselt paremaid andmebaase disainida.

Magistritöö koosneb sissejuhatusest, viiest peatükist, kokkuvõttest, kasutatud kirjandusest ja lisadest.

Töö esimene peatükk sisaldab teoreetilist tausta ehk annab lühiülevaate graafiteooriast, erinevatest andmemudelitest ja nendes hierarhiate esitamise võimalustest. Teoreetilises osas esitatakse kaks mõistekaarti (ingl *concept map*). Mõistekaardiks nimetatakse visualiseerimistehnikat, mille abil saab teadmust struktureerida ja mõistete vahelisi tähendusseoseid võrgu/graafi kujul esitada (Reiska 2008). Esimene mõistekaart iseloomustab graafiteooriat ning teine iseloomustab erinevaid andmemudeleid ja nendes hierarhiate esitamise võimalusi.

Teine peatükk sisaldab erinevate SQL-andmebaasides hierarhiate talletamiseks mõeldud disainilahenduste kirjeldusi, mis on võrreldavuse tagamiseks esitatud ühesugust struktuuri kasutades. Lisaks kirjeldusele esitatakse iga disaini kohta ka näited.

Kolmas peatükk tutvustab antud töös tehtavat eksperimenti ja selle eesmärke. Antud peatükis antakse ka ülevaate kasutavatest andmebaasisüsteemidest, testandmetes ja tehtavatest päringutest.

Neljas peatükk sisaldab tehtud eksperimendi tulemusi (mõõdetud väärtusi, mis näitavad lausete täitmiseks kulunud aega, andmemahtu, lausete keerukust).

Viiendas peatükis analüüsitakse saadud eksperimendi tulemusi ja esitakse järeldused.

1. Teoreetiline taust

Käesolevas peatükis antakse lühiülevaade graafiteooriast ning hierarhiliste andmete andmebaasides esitamise ning nende andmete kasutamise võimalustest. Graafiteooria leiab käsitlemist kuna hierarhiat esitav puustruktuur on graafi erivorm. Selleks, et saada aru kuidas tuleb andmebaasides hierarhiaid korrektselt esitada, peab teadma graafiteooria põhimõisteid. Seega on järgnevalt esitatud lühiülevaade graafidega seotud definitsioonidest, omadustest ja kitsendustest.

1.1. Graafiteooria

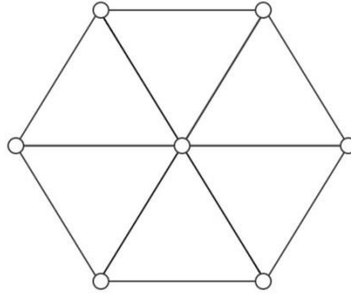
Jooniste kasutamine muudab ülesande lahendamise lihtsamaks ja selgemaks ja aitab juhtida mõtet õiges suunas. Sõltuvalt probleemist, on võimalik kujutada punktidena, näiteks, linnu, inimesi, tehaseid ning ühendada neid joontega, tähistades niiviisi vastavate elementide vahelisi seoseid. Selliseid skeeme nimetatakse **graafideks** (Koit 1968) (vt Joonis 1).

Graafiteooriale pani aluse šveitsi matemaatik ja füüsik Leonhard Euler 1736. aastal lahendades Königsbergi sildade ülesannet – kuidas läbida hargnevaid jõgesid ületavaid seitset silda, neist igat üht vaid üks kord läbides. L.Euler tõestas, et sellist marsruuti ei eksisteeri. Termini graaf võttis esimesena kasutusele inglise matemaatik J.J. Sylvester 1878. aastal (Petuhhov 2014a) .

Käesoleval ajal võib graafiteooriat nimetada iseseisvaks matemaatikaharuks, millel on palju praktilisi rakendusi (näiteks, keemia, bioloogia, transpordi ja majanduse planeerimise valdkonnas).

1.1.1. Graaf

Graaf (ka mittesuunatud graaf) G koosneb tippude (e sõlmede ingl *node*) hulgast V (tähistatakse vahel ka V(G)) ja servade (e kaarte ingl *edge*) hulgast E (tähistatakse vahel ka E(G)) (Buldas, Laud, Willemson 2002) (vt Joonis 1). Kui E on tühi hulk, siis öeldakse, et tegemist on tühja graafiga (Empty Graph).



Joonis 1. Graafi näide

Graaf võib olla suunatud, s.t et iga kaare jaoks on määratud, millisest sõlmest see algab ja millises sõlmes lõpeb. Suunamata graafi puhul on seos kahe sõlme vahel mõlemas suunas. Naabersõlmede jada kahe sõlme vahel on tee (Petuhhov 2014b). Teed, mis ei sisalda ühtegi serva rohkem kui üks kord, nimetatakse ahelaks (Graafidest). Ahel on kinnine, kui tema alg- ja lõpptipp langevad kokku (Graafid a). Sidus graaf on graaf, kus iga kahe tipu vahel leidub ahel (Graafid b).

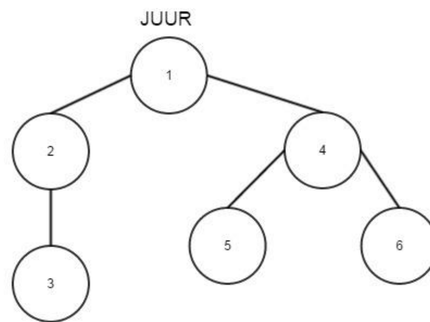
1.1.2. Puu ja hierarhia

Kinnine ahel, milles on vähemalt üks kaar/serv, on tsükkel (ingl *cycle*) (Graafid a). Graafi, milles pole tsükleid, nimetatakse metsaks. Sidusat metsa nimetatakse **puuks** (Penjam 2013) (vt Joonis 2). Puu on graafi erivorm.

Kaasik (1992) selgitab, et juurega puus fikseeritakse üks sõlm juurena ja orienteeritakse kaared selliselt, et iga lehe jaoks leidub parajasti üks juurest selleni viiv tee. Leht on puu rippuv sõlm (Kaasik 1992). Kõik puu mingist sõlmest lehtede poole jäävad sõlmed on sõlme järglased ehk järeltulijad. Kõik sõlmest juure poole jäävad sõlmed on sõlme eellased. Puu tippu koos tema kõigi (nii vahetute kui ka kaugemate) alluvatega nimetatakse alampuuks (ingl *subtree*) (Mittelineaarsed andmestruktuurid). Sõlme järk on sõlme alampuude arv. Sõlme, millele ei eelne mitte ühtegi sõlme, nimetatakse juureks. Sõlme, mille järk on 0 (sel ei ole ühtegi järeltulijat), nimetatakse leheks. Ülejäänud sõlmed on hargnevad tipud. Puu sõlmed jagunevad hierarhia alusel tasemetesse. Juur on tasemel 0 (Mittelineaarsed struktuurid 2014).

Puude omadused (Petuhhov 2014b):

- „puu on sidus graaf, kus pole tsükleid,
- eksisteerib üks, teistest erinev sõlm, mis on antud puu juur (ingl *root*),
- kui N on puu sõlmede arv, siis graafil on N-1 serva.“ Näiteks joonisel 2 oleval puul on kuus sõlme ja viis serva.

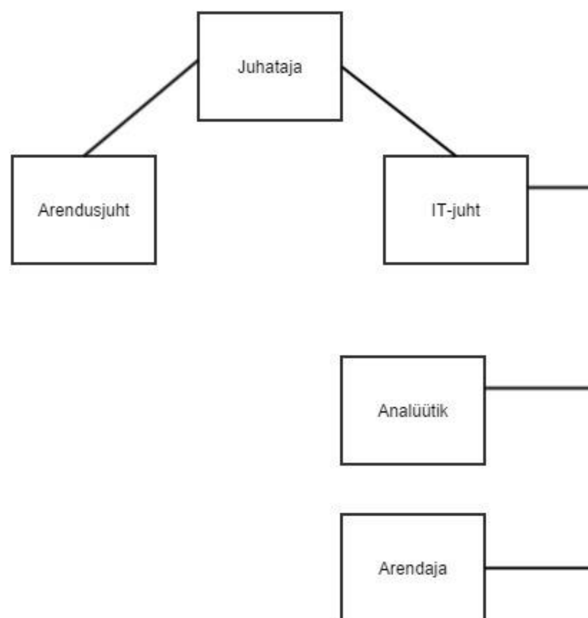


Joonis 2. Puu näide

Hierarhia on orienteeritud puu (vt Joonis 3), st hierarhilistel seostel on suund (Petuhhov 2014c). Hierarhia on puude erivorm (Celko 2012).

Hierarhial on lisaomadused (Celko 2012):

- pärimine
- alluvus



Joonis 3. Hierarhia näide (ametite hierarhia)

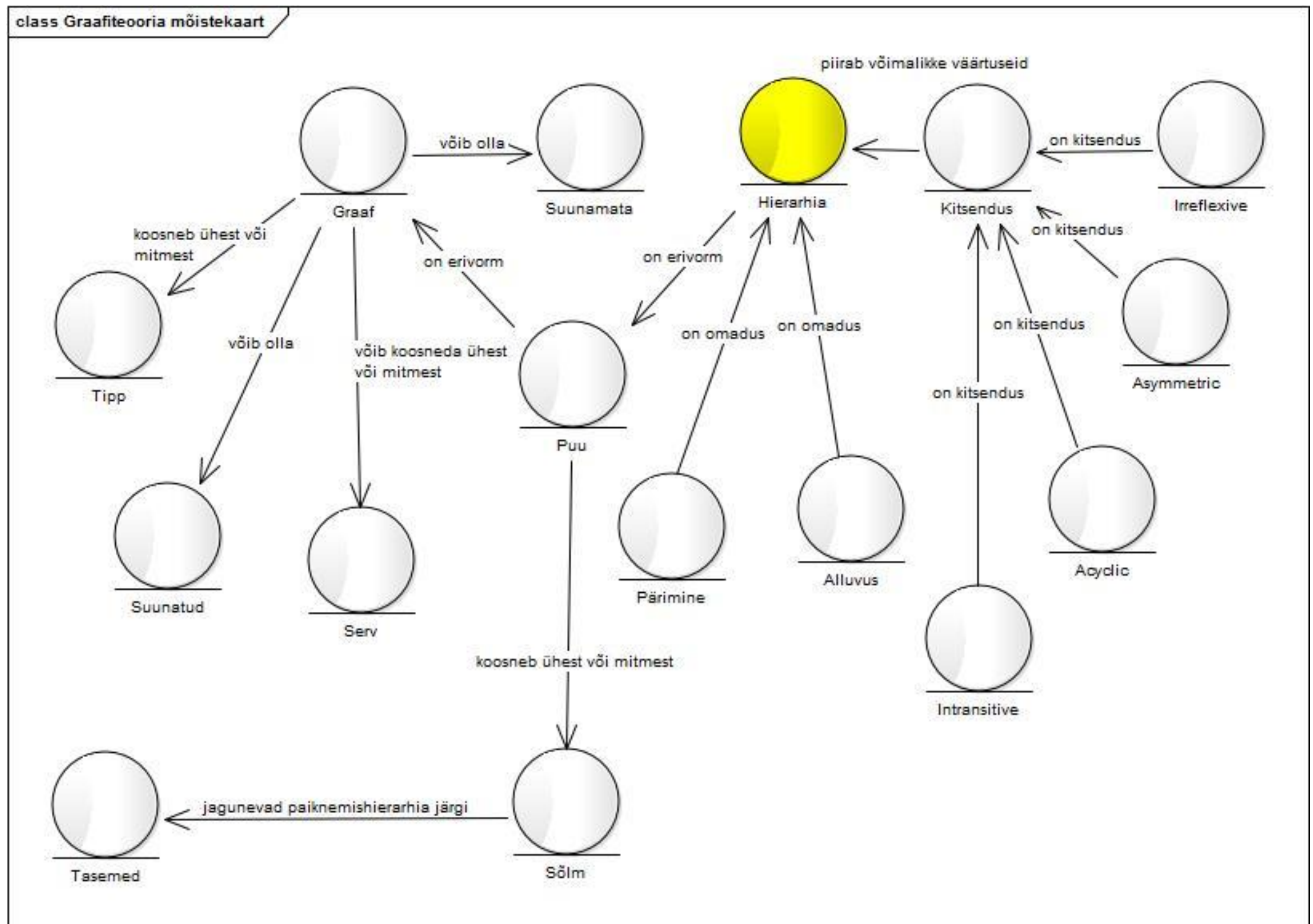
Hierarhiatega võivad olla seotud järgmised kitsendused (Miliauskaitė ja Nemuraitė 2005).

- **{Irreflexive}** – sõlm ei saa olla iseenda vanemaks või lapseks
- **{Asymmetric}** – kui sõlm A on sõlme B vanem, siis sõlm B ei saa olla sõlme A vanem
- **{Antisymmetric} -** kui sõlm A on sõlme B vanem, siis sõlm B ei saa olla sõlme A vanem (sõlm võib olla iseenda vanem)
- **{Acyclic}** nii irrefleksiivne kui asümmeetriline – sõlm ei saa olla enda otsene ega kaudne vanem või laps

- **{Intransitive}** – sõlme vanemad ei saa olla selle laste otsesteks vanemateks

Selliseid kitsendusi saab näiteks väljendada kontseptuaalses mudelis, kui seal soovitakse väljendada hierarhiat.

Järgnevalt on esitatud mõistekaart (vt Joonis 4), mis kirjeldab jaotises 1.1 (ka alamjaotistes) käsitletud mõisteid ja nende vahelisi seoseid. See mõistekaart on ühtlasi suunatud graafi näide.



Joonis 4. Graafiteooria kirjeldav mõistekaart

1.2. Hierarhilise struktuuriga andmete esitamine erinevate andmemudelite korral

Termin “andmemudel” tähistab kirjanduses mitut erinevat mõistet. Andmemudeli all mõeldakse ühelt poolt konkreetse andmebaasi lihtsustatud kirjeldust. Teisalt defineerib Edgar Frank (Ted) Codd (1981) andmemudelit kui järgneva kolme komponendi ühendit (Eessaar, 2008).

- Kogum andmestruktuuride tüüpe, mis on mistahes antud mudelile vastava andmebaasi ehitusplokkideks. Näiteks relatsioonilises mudelis on sellisteks andmestruktuurideks relatsioonilised baasmuutujad (ingl *relational variable*), SQL aluseks olevas andmemudelis aga baastabelid (luuakse CREATE TABLE lausega). Eristan relatsioonilist mudelit ja SQL andmemudelit, sest kuigi SQL põhineb relatsioonilisel mudelil ei järgi see kõiki relatsioonilise mudeli põhimõtteid.
- Kogum operaatoreid või tuletusreegleid, mida võib rakendada erinevat tüüpi andmestruktuuridele. Näiteks relatsioonilise ja SQL andmemudelite korral on üheks selliseks operaatoriks relatsioonide/tabelite ühendamise operaator.
- Kogum üldiseid terviklikkuse reegleid, mis kirjeldavad hulga terviklikke andmebaasi seisundeid, seisundi muutuseid või mõlemaid. Näiteks relatsiooniline mudel nõuab, et igas relatsioonilises muutujas peab olema vähemalt üks kandidaatvõti, SQL aluseks olev andmemudel aga lubab võtmeteta tabeleid.

Iga andmemudeli alusel võib luua ühe või mitu andmebaasikeelt. Iga andmebaasisüsteem toetab ühte või mitut andmebaasikeelt ning sellest tulenevalt ka nende andmebaasikeelte aluseks olevaid andmemudeleid. Tänapäevaks on välja mõeldud palju erinevaid andmemudeleid (Coddi defineeritud tähenduses) ja need kõik võimaldavad mingil kujul hierarhiliste andmete esitamist ja töötlemist – mõned paremini, mõned halvemini. Leidub andmemudeleid nagu hierarhiline andmemudel ja võrkandmemudel, mis on saanud otseselt inspiratsiooni puudest ja graafidest. Need andmemudelid olid populaarsed 1960ndatel ja 1970ndatel aastatel. Nende edasiarenduseks 21. sajandil võib pidada XML andmemudelit ja graafi-andmemudelit.

Graafipõhistes andmebaasides (mis on loodud mõne graafi-andmemudelit toetava andmebaasisüsteemi abil) on andmed sõlmedes, millel on omadused. Sõlmede vahel on seosed, millel on ka omadused (Eessaar 2014). Graafipõhised andmebaasisüsteemid on näiteks *Neo4j* ja *InfiniteGraph*. XML andmebaasides esitatakse andmeid XML dokumentides sisalduvate hierarhiatena. Saab eristada kahte liiki XML-andmebaasisüsteeme: *XML-võimelised* ja *päris XML-andmebaasisüsteemid* (Männiko

2008). *XML-võimeline andmebaasisüsteem* kujutab endast mitte-XML andmemudelil põhinevat andmebaasisüsteemi (nt relatsioonilist või objektorienteeritud), mida on laiendatud võimalustega hoida XML-formaadis andmeid. XML dokumendid salvestatakse selle andmebaasisüsteemi andmemudeli toetatud andmestruktuuridesse (nt SQL-andmebaasisüsteemide korral VARCHAR või XML tüüpi tabeli veergudesse). *Päris XML-andmebaasisüsteemi* põhiliseks andmestruktuuriks on XML dokument (Vallaste 2000). XML-andmebaasisüsteemid on näiteks *eXist* ja *Sedna*.

Käeolevas töös on tähelepanu keskendunud SQL aluseks olevale andmemudelile, kuna hierarhilise struktuuriga andmete esitamine ja kasutamine selle mudeli alusel loodud andmebaasides on suhteliselt keeruline. Samas on sellisel andmemudelil põhinevad andmebaasisüsteemid endiselt valdavad ning seetõttu on teema uurimiseks ja uute teadmiste saamiseks endiselt aktuaalne.

Miks peetakse hierarhiliste andmete talletamist ja kasutamist SQL andmebaasides keeruliseks ülesandeks? Probleem on selles, et tänapäeva SQL-andmebaasisüsteemides on hierarhilise struktuuriga andmete töötlemiseks piiratud hulk operaatoreid ning nende abil andmete töötlemine on suhteliselt aeglane. Samas, see ei ole relatsioonilise mudeli probleem, vaid SQLi ning SQL-andmebaasisüsteemide probleem (Eessaar 2014).

Järgnevas jaotises antakse ülevaade hierarhilise andmete esitamisest SQL-andmebaasides.

1.2.1. Hierarhilise struktuuriga andmete esitamine SQL-andmebaasides

1969. aastal pakkus IBM uurija Edgar Frank Codd välja relatsioonilise andmemudeli. 1970. aastal avaldas Codd teise sama teemalise artikli nimega “A Relational Model for Data for Large Shared Data Banks”, mis sai juba laiemalt tuntuks. Relatsiooniline andemudel näeb ette relatsiooniliste muutujate (ingl *relational variable*) kasutamise, mille iga võimalikku väärtust nimetatakse relatsiooniks (ingl *relation*). Iga relatsioon sisaldab null või rohkem korteeži (ingl *tuple*) (Eessaar 2014). See mudel põhineb matemaatilistel teooriatel – hulgateooria ja predikaatarvutus, aga ka tüüpide ja binaarsete relatsioonide teooria (Puutusmaa 2012).

Tänapäeva relatsioonilised andmebaasisüsteemid võimaldavad enamasti kasutada andmebaasikeelt SQL, mis põhineb relatsioonilisel andmemudelil, kuid ei järgi kõiki selle mudeli põhimõtteid. Seega nimetan kasutajatele SQL keelt pakkuvaid andmebaasisüsteeme edaspidi SQL-andmebaasisüsteemideks ning nende abil loodud andmebaase SQL-andmebaasideks.

SQL-andmebaaside põhiliseks ehitusplokiks on tabelid, mis omakorda koosnevad veergudest ja ridadest. Iga tabel võib olla välisvõtmete kaudu seotud teiste tabelitega. Andmete ning andmebaasiobjektide haldamiseks nendes andmebaasides kasutatakse andmebaasikeelt SQL.

Läbi aegade on üheks vastuväiteks relatsioonilisele mudelile ning SQLile olnud, et need ei toeta piisavalt hästi hierarhiliste andmete andmebaasis esitamist ja haldamist. Nüüd on hakatud mõistma, et probleem pole mitte selles nagu ei saaks relatsioonilistes ja SQL-andmebaasides hierarhilisi andmeid esitada, vaid selles, et arendajad peavad õppima, kuidas seda teha (Celko 2004). Samuti on probleem selles, et paljudes SQL-andmebaasisüsteemides pole piisavalt vahendeid (näiteks funktsioone, lausekonstruktsioone), et hierarhilisi andmeid mugavalt ja efektiivselt töödelda. Seega, kui on näha ette vajadus hoida SQL-andmebaasis hierarhilisi andmeid, siis tuleb valida selleks otstarbeks võimalikult head tuge pakkuv andmebaasisüsteem. Käesoleva töö eksperimendi osas kasutatakse andmebaasisüsteeme Oracle Database ja PostgreSQL ning töö annab vastuse küsimusele, kui hästi need SQL-andmebaasisüsteemid toetavad hierarhiliste andmete kasutamist.

Järgmises jaotises antakse ülevaade hierarhiliste andmete töötlemise võimalustest Oracle Database ja PostgreSQL andmebaasisüsteemides. Igas SQL-andmebaasisüsteemis on kasutusel oma SQLi dialekt e mägimurrak. Tavaliselt ei kattu see täielikult SQL standardis kirjeldatud keelega.

1.2.1.1. Hierarhiliste andmete esitamise võimalused Oracle Database abil loodud andmebaasides

Oracle võimaldab teha hierarhiliste andmete põhjal rekursiivsed päringuid, kasutades CONNECT BY konstruktsiooni. „Protseduur või funktsioon on *rekursiivne*, kui ta kutsub iseennast välja.” (Rekursioon 2008) Hierarhiate kontekstis tähendab rekursiivne päring, et sellega on võimalik liikuda hierarhias soovitud tasemeni ning valida teel andmed, mida päringu tulemuses esitada. Oracle 9g ja varasemates versioonides on seda konstruktsiooni võimalik kasutada ainult atsükliliste hierarhiate jaoks, kuna tsükli leidmisel tagastas süsteem vea. Alates versioonist 10g on see viga parandatud (lisatud võimalus kasutada *NOCYCLE* võtmesõna, mille kasutamisel tagastab päring tulemuse isegi siis kui hierarhias esineb tsükkel) (Wikipedia 2014b).

Alternatiivne võimalus on kasutada rekursiivseid ühiseid tabeli avaldisi (CTE, *Common Table Expression*). Ühine tabeli avaldis võimaldab päringu alguses defineerida alampäringu, mida päringus saab korduvalt kasutada nagu vaadet. Oracle toetab seda konstruktsiooni alates versioonist 11.2.

Rekursiivne ühine tabeli avaldis on tabeli avaldis, milles viidatakse sellele samale avaldisele ja selle kaudu moodustatakse rekursiivne avaldis (Eessaar 2014) .

Ühise tabeli avaldise süntaks on järgmine:

```
WITH [RECURSIVE] with_query [, ...]
SELECT...
```

WITH_QUERY süntaks on järgmine:

```
WITH query_name [ (column_name [,...]) ] AS (SELECT ...)
```

Oracle võimaldab alates versioonis 9.0.1. kasutada andmetüüpi *XMLType*. See annab võimaluse hoida andmebaasis hierarhilise struktuuriga andmeid XML formaadis. Võib luua tabelite veerge, milles olevad väärtused on hierarhiaid esitavad XML dokumendid. Samuti võib XML tüüpi andmete hoidmiseks luua XML tüüpi tabeleid. Lisaks pakub Oracle funktsioone XML formaadis andmete töötlemiseks.

Hierarhilisi andmeid võib ka esitada JSON tüüpi väärtustena, kus iga selline väärtus esitab ühe hierarhia. JSON (*JavaScript Object Notation*) on andmevahetusformaat, mis põhineb JavaScripti programmeerimiskeele alamhulgal (Wikipedia 2014c). Alates Oracle versioonist 12.1.0.2 on selles võimalik kasutada funktsioone ja operaatoreid nagu JSON_TABEL, JSON_VALUE, JSON_EXISTS või IS_JSON, mis töötavad JSON formaadis olevate andmetega. Oracle andmebaasis võivad need andmed olla salvestatud veergudes, kus andmetüübiks on määratud näiteks VARCHAR2, NVARCHAR2 või CLOB.

1.2.1.2. Hierarhiliste andmete esitamise võimalused PostgreSQL andmebaasis

PostgreSQL võimaldab (alates versioonist 8.4) kasutada ühiseid tabeli avaldisi, sealhulgas ka rekursiivseid ühiseid tabeli avaldisi (täpsemalt kirjeldatud jaotises 1.2.1.1).

PostgreSQL võimaldab kasutada lisamoodulit *tablefunc*, mis sisaldab erinevaid funktsioone, mis tagastavad ridade hulka. Üks nendest funktsioonidest on *connectby* funktsioon, mis töötab analoogselt Oracle andmebaasisüsteemi *CONNECT BY* konstruktsiooniga. Funktsioonide kasutamiseks tuleb *tablefunc* moodul eraldi installeerida (PostgreSQL 9.1.14 Documentation a).

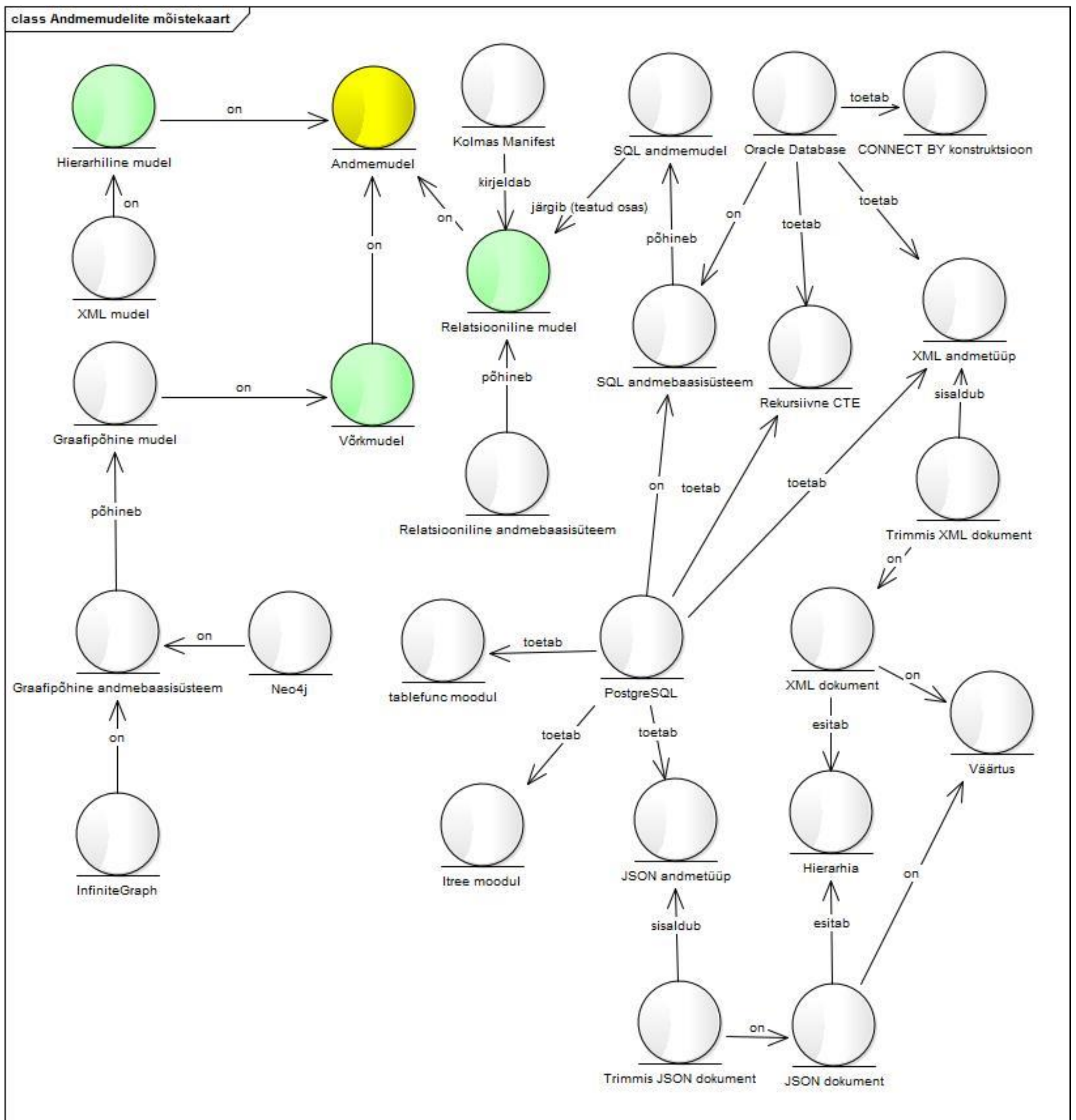
Alternatiivne võimalus hierarhiate andmebaasis esitamiseks on kasutada *ltree* moodulit, mis nõuab samuti eraldi installeerimist. *Ltree* on moodul, mis võimaldab hoida andmeid hierarhiate kohta märgiste kujul ning pakub erinevaid võimalusi nende andmete põhjal päringute tegemiseks (PostgreSQL 9.1.14 Documentation b). *Ltree* moodulit on võimalik kasutada näiteks *materialiseeritud tee* disaini korral (vt Jaotis 2.6).

Iga märgis (ingl *Label*) võib koosneda tähtedest, numbritest ja alakriipsudest. Märgistest saab koostada teekondi, millest igaüks on üksteisest punktidega eraldatud märgiste kogum. Teekondade andmeid saab hoida veergudes, kus andmetüübiks on määratud *ltree*. Teekondade põhjal päringute tegemiseks kasutatakse spetsiaalseid päringuid, mida nimetatakse *lquery*. Näiteks Tabelis 3 (jaotis 3.6) kasutatakse S1M päringu puhul lquery tingimust **.10796.** (leiab kõik read, kus teekonnas sisaldub *.10796.*). Lquery kasutatakse ka I1M ja D1M päringute puhul (vt Tabel 3). Alates PostgreSQL 9.1 saab moodulite installeerimiseks kasutada CREATE EXTENSION lauset.

Alates andmebaasisüsteemi versioonist 8.3 on võimalik andmebaasis kasutada XML andmetüüpi. Seega võib luua tabelite veerge, milles olevad väärtused on hierarhiaid esitavad XML dokumendid. PostgreSQL võimaldab kasutada XML tüüpi andmetega töötamiseks erinevaid funktsioone nagu XMPLPARSE, XMLCONCAT, XMLELEMENT jne. Selleks, et sellist tüüpi oleks võimalik PostgreSQL serveris kasutada, peab vastav valik olema tehtud serveri installeerimisel.

PostgreSQL võimaldab alates versioonist 9.2 kasutada JSON andmetüüpi. Seega võib luua tabelite veerge, milles olevad väärtused on hierarhiaid esitavad JSON dokumendid. Nende andmete töötlemiseks eksisteerib hulk erinevaid operaatoreid ja funktsioone nagu TO_JSON, ROW_TO_JSON jne.

Järgnevalt on esitatud mõistekaart (vt Joonis 5), mis kirjeldab jaotises 1.2 käsitletud mõisteid ja nende vahelisi seoseid.



Joonis 5. Jaotises 1.2. käsitlevaid mõisteid ja nende vahelisi seoseid kirjeldav mõistekaart

2. Erinevad disainilahendused hierarhiliste andmete esitamiseks SQL-andmebaasides

Selles peatükis antakse ülevaade erinevatest disainilahendustest, mis võimaldavad talletada andmeid hierarhiate kohta SQL-andmebaasides nii, et andmed hierarhia moodustavate objektide ja nende seoste kohta on „lahti lõhutud“ ja paigutatud eraldi veergudesse/tabelitesse. Nimekirjas ei tooda välja lahendusi, mille korral salvestatakse hierarhia väärtusena tabeli väljas (XML või JSON dokumendina). Selliste lahenduste korral salvestatakse hierarhiad (nii hierarhilised seosed kui seostes osalevate olemite andmed) andmeväärtustena, mitte erinevatesse tabelitesse ja väljadesse „laiali lõhutuna“. Selliste lahenduste korral on hierarhiate andmed väärtustesse kapseldatud ja läheb vaja keerukaid operaatoreid, et selliseid väärtuseid kasutada. Sellised disainid moodustavad autori arvates omaette disainide klassi ja võiks lõputöö mahtu arvestades vaatluse alt välja jätta.

Kokku esitatakse selles peatükis 11 erinevat disainilahendust. Kõik disainilahendused on leitud autori poolt raamatutest ja artiklitest. Need disainid on:

1) Adjacency List (Külgnevusnimistu)
2) Nested Sets (Pesastatud hulgad)
3) Nested Intervals (Pesastatud intervallid)
4) Closure Table, Tree changing over time (Sulunditabel)
5) Flat Table (Lametabel)
6) Materialized Path (Materialiseeritud tee)
7) Multiple Lineage Columns (Hulk pärilikkuse veerge)
8) Hardcoded Tree (Tase-tabel)
9) Degenerate node and edge (Otseste järeltulijate grupid)
10) Structured Tree (Struktureeritud puu)

Üleval toodud nimekiri koosneb põhilahendustest. Erinevate disainide kombineerimisel on võimalik saada uusi lahendusi, millest võib mõelda kui variatsioonidest.

Autor ei leidnud ühtegi allikat, kus kõiki neid disaine oleks üheskoos ja struktureeritud viisil kirjeldatud.

Külgnevusnimistu, Pesastatud hulgad, Pesastatud intervallid, Materialiseeritud tee, *Tase-tabel, Struktureeritud puu, Otseste järeltulijate grupid* – need disainid on leitud raamatutest (Celko 2012; Tropashko 2007; Blaha 2010).

Sulunditabel, Lametabel, Hulk pärilikkuse veerge – need disainid on leitud kasutades otsingumootorit *Google* (http://www.google.com). Otsingusõnad:

- *Sql hierarchies*
- *Hierarchy models*

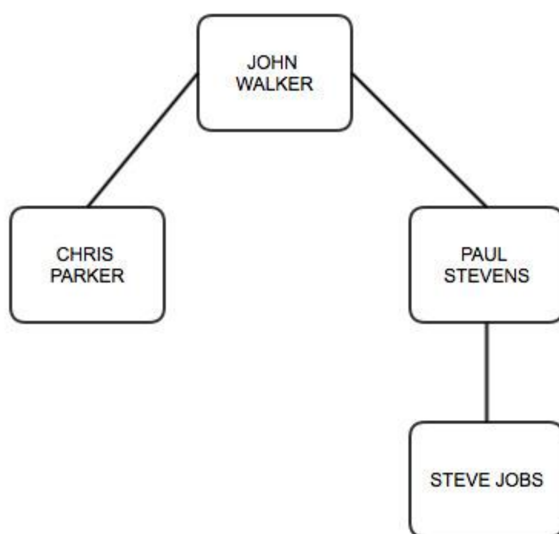
Võimaldamaks lugejal disaine paremini võrrelda ning mõista nende sarnasusi ja erinevusi, esitatakse kõikide disainide kirjeldused selles töös kasutades ühesugust struktuuri.
Nimi inglise keeles: Disaini iseloomustav nimi, mis aitab erinevaid disaine eristada ja nendele viidata. See nimi on inglise keeles.
Nimi eesti keeles: Nime tõlge eesti keelde, kui autor oskas seda teha. Tuleb mainida, et kuna eestikeelset kirjandust sellel teemal napib, siis tuli paljudele disainidele ise eestikeelsed nimed välja pakkuda.
Samuti teada kui: Alternatiivne nimi inglise keeles, mis on leitud teistest allikatest.
Allikad: Kirjandus, mis on võetud aluseks disainide kirjeldamisel.
Kirjeldus: Disainide tekstiline kirjeldus (sõnastatud definitsioon ja selgitus) ja graafiline esitus (joonised, mis kirjeldavad hierarhiat konkreetse näite põhjal).
Eelised: Disaini kasutamisest tingitud positiivsed tagajärjed nii kirjanduse põhjal kuid ka autori poolt mõtteharjutusena tuletatud. Eeliste nimekiri ei pretendeeri ammendavusele. Antud töös esitatud uuring ja teised analoogilise uuringud on aluseks edasisele eeliste määramisele.
Puudused: Disaini kasutamisest tingitud negatiivsed tagajärjel nii kirjanduse põhjal kuid ka autori poolt mõtteharjutusena tuletatud. Puuduste nimekiri ei pretendeeri ammendavusele. Antud töös esitatud uuring ja teised analoogilise uuringud on aluseks edasisele puuduste määramisele.
Näide: Disaini rakendamine konkreetse näite jaoks. Selles jaotises esitatakse tabelid koos näiteandmetega. Selleks, et erinevad disainid oleksid paremini võrreldavad, kasutatakse kõikides disainides ühesugust näite-hierarhiat, millele vastavad andmed tuleb andmebaasis talletada.
Strateegia: Disaini realisatsioon konkreetses andmebaasisüsteemis. Selles töös on kõik strateegiad loodud andmebaasisüsteemi Oracle 12c Enterprise Edition jaoks. Anud töös ei esitata strateegiaid mallina (mida saab kasutada koodi genereerimise realiseerimiseks), vaid esitatakse konkreetse ülesande lahendamiseks mõeldud laused. Lahendatav ülesanne on kõigi disainide puhul ühesugune.

Strateegias on kirjeldatud ainult need kitsendused, mida on võimalik esitada deklaratiivselt, ilma protseduurset koodi kirjutamata. Deklaratiivne kitsendus tähendab seda, et süsteemile öeldakse, *mida* on vaja kontrollida kuid mitte *kuidas* seda teha. Deklaratiivsed kitsendused pakuvad huvi kuna nende loomine on arendajate jaoks palju lihtsam võrreldes protseduurse koodi abil realiseeritud kitsendustega. Viimaste loomisel tuleb mõelda nii kõikvõimalikele sündmustele, mille tulemusena võivad andmebaasi jõuda kitsendustele mittevastavad andmed kui ka tabelite/ridade lukustamisele.

Strateegiate koostamisel lähtutakse eeldusest, et kui keegi peaks tahtma kustutada hierarhia sõlme, siis tuleks süsteemil automaatselt kustutada kõigi selle otseseks või kaudseks järglaseks olevate sõlmede andmed. Strateegias tuuakse selle saavutamise meede välja vaid siis, kui seda saab saavutada välisvõtmega seotud ON DELETE CASCADE kompenseeriva tegevuse abil. Protseduurset koodi, mida paljudel juhtudel selle saavutamiseks vaja läheb, strateegia kirjelduses välja ei tooda.

Strateegias on näidatud ka loodud indeksid. Indeksite defineerimisel on lähtutud tõsiasjast, et Oracle indekseerib automaatselt primaarvõtme ja unikaalsuse kitsendusega veerge, kuid ei indekseeri automaatselt välisvõtme veerge. Välisvõtme veerud oleks mõistlik ühendamisoperatsioonide kiirendamiseks ning tabelite lukustamise vajaduse vähendamiseks indekseerida, kuid need indeksid ei tohiks dubleerida automaatselt tekkivaid indekseid.

Erinevate disainilahenduste illustreerimiseks kasutatakse järgnevat töötajate hierarhiat (vt Joonis 6). Antud näide on välja mõeldud autori poolt ning ainult illustreerimise eesmärgil.



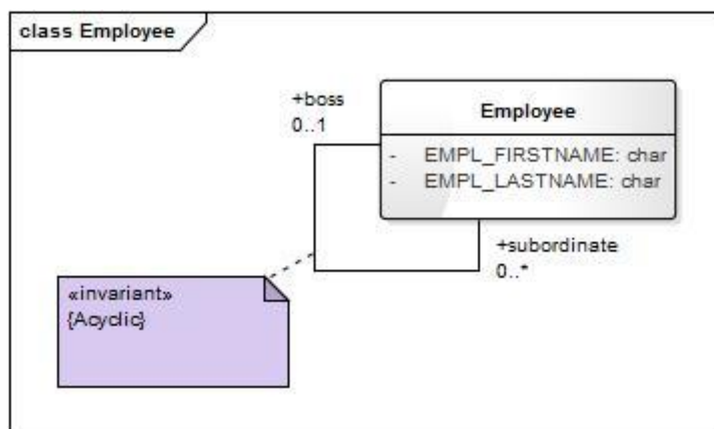
Joonis 6. Töötajate hierarhia

Töötajate vahel on alluvussuhted. Joonis 4 kohaselt ei ole John Walker'il ülemust. Chris Parker ja Paul Stevens on John Walker'i otsesed alluvad. Steve Jobs on Paul Stevens'i otsene alluv. Paul Stevens on Steve Jobs'i otsene ülemus ja John Walker kaudne ülemus.

Antud näite korral kehtivad järgmised ärireeglid.

- Igal töötajal on null või üks ülemus.
- Igal töötajal on null või rohkem alluvat.
- Töötaja ei saa olla enda alluv ega enda ülemus (ei otseselt ega kaudselt) (st pole lubatud tsüklid).

Järgnevalt (vt Joonis 7) on kujutatud olemi-suhte diagramm, mis on osa kontseptuaalsest andmemudelist. Olemi-suhte diagrammi loomiseks on kasutatud vahendit Enterprise Architect.



Joonis 7. Töötajate olemi-suhte diagramm.

Seosetüübil on nii irrefleksiivsuse kui asümmeetria kitsendus *{Acyclic}*, mis tähendab, et ükski töötaja ei saa olla seotud otseselt ega kaudselt iseendaga.

Kitsendusi saab andmebaasis jõustada kas deklaratiivselt (näiteks kasutades CHECK kitsendust) või protseduurselt (kasutades trigereid ja protseduure või teostades kontrolli andmebaasi kasutavate rakenduste koodis). PostgreSQL andmebaasisüsteem võimaldab kasutada CHECK kitsendustes kasutaja-defineeritud funktsioone. Oracle andmebaasisüsteem ei võimalda CHECK kitsendustes kasutaja-defineeritud funktsioonide kasutamist. Nii PostgreSQL kui ka Oracle ei luba kasutada CHECK kitsendustes alampäringuid. Disainide kirjelduses on välja toodud ainult need kitsendused, mida on võimalik esitada Oracle andmebaasis deklaratiivselt, ilma alampäringuteta ja funktsioonideta.

2.1. Adjacency List

Nimi inglise keeles: Adjacency List

Nimi eesti keeles: Külgnevusnimistu

Samuti teada kui: Parent-Child Model, Simple Tree

Allikad: (Karwin 2010), (Celko 2012), (Stadnik 2008), (Blaha 2010)

Kirjeldus:

Külgnevusnimistu kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides, mille puhul iga tabeli rida määrab ühe otseselt seotud sõlmede paari (ehk iga puu sõlm viitab otsesele vanem-sõlmele).

Karwin (2010) kirjeldab SQL antimustrite raamatus antimustrit *Naive Trees*, mille kohaselt kasutatakse hierarhiliste andmete hoidmiseks külgnevusnimistu mudelit. Iga antimuster kirjeldab ühte halba lahendust mingile probleemile.

Karwin kirjutab oma raamatus, et külgnevusnimistu kasutamine võib olla antimustriks, kui põhiülesandeks on leida sõlme kõik järeltulijad. Sellise ülesande lahendamisel tuleb N tasemelise hierarhia korral viia läbi N-1 ühendamise operatsioon (ingl *JOIN*). Kui me ei tea kui mitu taset hierarhias on või kui tasemete arv ajas muutub, siis me ei tea kui mitut ühendamise operatsiooni kasutada või peame tasemete arvu muutudes hakkama päringut ümber kirjutama.

Mõnede andmebaasisüsteemide SQL dialekt e mägimurrak toetab konstruktsioone, mis võimaldavad teha rekursiivseid päringuid tasemete arvu defineerimata (näiteks *CONNECT BY* konstruktsioon Oracle andmebaasisüsteemis). Seega külgnevusnimistu disaini kasutamist võib nimetada antimustriks nendes andmebaasisüsteemides, mis ei toeta rekursiivseid päringuid.

Kolmandas Manifestis (*The Third Manifesto*) (Date ja Darwen 2014), kus kirjeldatakse relatsioonilise mudeli põhimõtteid, öeldakse, et tõeliselt relatsioonilises andmebaasis oleks relatsiooniline muutuja, mille võimalikud väärtused esitavad hierarhilisi andmeid, kavandatud külgnevusnimistu disaini põhimõttel (kuid ilma NULLideta). Sellist andmemudelit toetav andmebaasisüsteem toetaks transitiivse sulundi leidmise (ingl *transitiive closure*) relatsioonilist operatsiooni ning nii saaks hierarhiaid esitada ja kasutada.

Eelised:

- Kiire andmete muutmine, lisamine ja kustutamine (Stadnik 2008).

- Lihtne esitada (selge struktuur).

Puudused:

- Ajakulukas vanemate ja järeltulijate leidmine (Stadnik 2008).
- Hierarhia on visuaalselt halvasti jälgitav.
- Päringu tegemiseks tuleb eelnevalt teada hierarhia tasemete arvu (kui süsteem ei võimalda kasutada konstruktsioone, mis toetavad rekursiivseid päringuid) (Karwin 2010).
- Hierarhilisi seoseid esitavas veerus peab lubama NULL-markereid (puu juure korral), mis võivad tingida loogiliselt ebakorrektseid päringu tulemusi.
    - Näide:

      Järgnev päring peab leidma ühendi kõigist töötajatest kelle ülemus on 1 ja kelle ülemus ei ole 1. Loogiliselt võttes peaks päringu tulemuses olema kõik töötajad. Kuid tegelikult ei ole selle päringu tulemuses töötajat 1 (vt Joonis 8).

```
SELECT empl_id, empl_firstname
FROM Employee
WHERE parent_id=1
UNION SELECT empl_id, empl_firstname
FROM Employee
WHERE parent_id<>1;
```

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	EMPL_FIRSTNAME	EMPL_LASTNAME	PARENT_ID	
1	John	Walker		John Walker'il ei ole ülemust (PARENT_ID väljas on NULL-marker)
2	Chris	Parker	1	Chris Parker on John Walker'i alluv
3	Paul	Stevens	1	Paul Stevens on John Walker'i alluv
4	Steve	Jobs	3	Steve Jobs on Paul Stevens'i alluv

Joonis 8. Külgnevusnimistu disaini näide konkreetse andmetega

Strateegia:

Tabeli ja indeksi loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     parent_id      NUMBER(10),
     CONSTRAINT pk_employee PRIMARY KEY (empl_id),
     CONSTRAINT fk_employee FOREIGN KEY(parent_id) REFERENCES
employee(empl_id) ON DELETE CASCADE,
     CONSTRAINT check_irreflexive CHECK(empl_id != parent_id)
  );

CREATE INDEX idx_parent_id
  ON employee (parent_id);
```

Loodi välisvõtme kitsendus *fk_employee*, mille puhul kasutati määrangut *ON DELETE CASCADE*. Loodi CHECK kitsendus *check_irreflexive*, millega kontrollitakse, et töötaja ei saa olla iseenda ülemus.

Loodi täiendav indeks *idx_parent_id* välisvõtme veerule *parent_id*.

2.2. Nested Sets

Nimi inglise keeles: Nested Sets

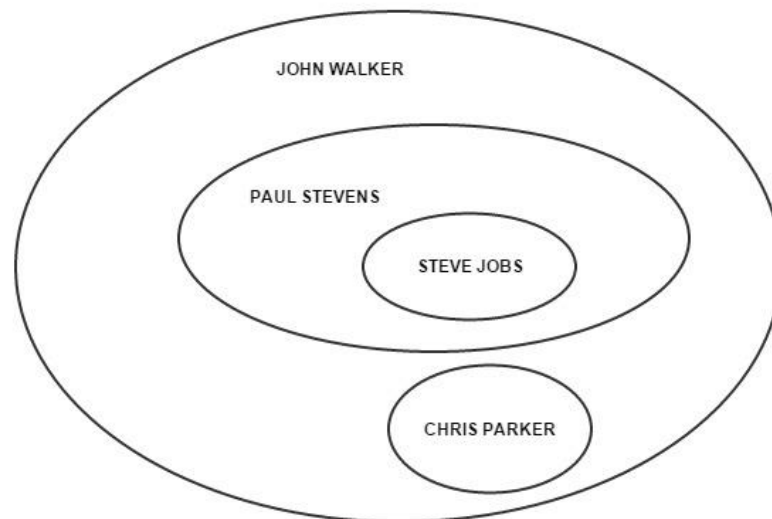
Nimi eesti keeles: Pesastatud hulgad

Samuti teada kui: Modified Preorder Tree Traversal

Allikad: (Celko 2012), (Stadnik 2008)

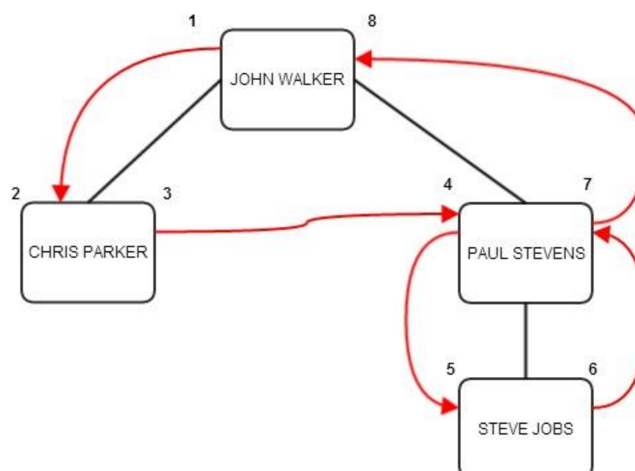
Kirjeldus:

Hierarhiat on võimalik esitada alternatiivselt (vt Joonis 9).



Joonis 9. Alternatiivne võimalus hierarhia esitamiseks

Pesastamine tähendab ühe või mitme ühetüübilise struktuuri samatüübilise struktuuri sisse paigutamist. (IT terministandardi projekti (1998-2001) sõnastik)



Joonis 10. Sõlmede numereerimine

Pesastatud hulkade mudel kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides, mille korral nummerdatakse kõik sõlmed kellaosuti liikumise vastupidises suunas (vt Joonis 10).

Ajalooliselt hoitakse sõlmi identifitseerivaid arve veergudes, mille nimedeks on *lft* ja *rgt* (*LEFT* ja *RIGHT* on reserveeritud sõnad paljudes SQL dialektides).

Eelised:

- Kiire hierarhia taseme, vanemate ja järeltulijate leidmine (Stadnik 2008) (sõlm *[lft, rgt]* on sõlme *[lft2,rgt2]* järeltulija, kui *lft > lft2 and lft < rgt2* (Celko 2012)).

Puudused (Stadnik 2008)**:**

- Võrreldes külgnevusnimistu disainiga ajakulukas andmete lisamine, muutmine, kustutamine, kuna iga muudatuse tegemisel tuleb ümber arvutada terve puu.

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	EMPL_FIRSTNAME	EMPL_LASTNAME	LFT	RGHT	
1	John	Walker	1	8	John Walker'il ei ole ülemust
2	Chris	Parker	2	3	Chris Parker on John Walker'i alluv
3	Paul	Stevens	4	7	Paul Stevens on John Walker'i alluv
4	Steve	Jobs	5	6	Steve Jobs on Paul Stevens'i alluv

Joonis 11. Pesastatud hulkade mudeli näide konkreetsete andmetega

Strateegia:

Tabeli ja indeksi loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10) ,
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     lft            NUMBER(11) NOT NULL,
     rght         NUMBER(11) NOT NULL,
     CONSTRAINT pk_employee PRIMARY KEY (empl_id),
     CONSTRAINT check_lft_rght CHECK( lft >= 0 AND rght > 0 AND lft <
rght),
     CONSTRAINT uni_lft_rght UNIQUE (lft, rght)
  );
```

Loodi CHECK kitsendus *check_lft_rght*, millega kontrollitakse, et veergude *lft* ja *rght* väärtused on positiivsed täisarvud või null (disain seda rangelt ei nõua, kuid lihtsuse mõttes on parem kasutada mittenegatiivseid arve). *lft* veeru väärtus võib olla null, kuid *rght* veeru väärtus on alati nullist suurem. *lft* veeru väärtus peab olema rangelt väiksem kui *rght* veeru väärtus.

Loodi unikaalne kitsendus *uni_lft_rght* veergudele *lft* ja *rght*, millest tekib automaatselt ka unikaalne liitindeks nendele veergudele.

2.3. Nested Intervals

Nimi inglise keeles: Nested Intervals

Nimi eesti keeles: Pesastatud intervallid

Samuti teada kui: Matrix Encoding

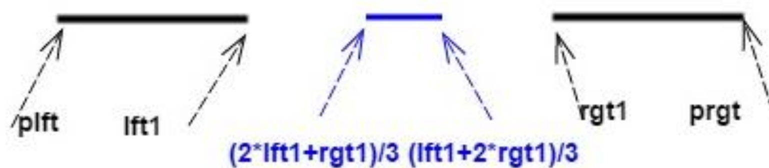
Allikad: (Tropashko 2005)

Kirjeldus:

Pesastatud intervallide mudel kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides, mille kasutamise korral kodeeritakse iga puu sõlm arvude paariga *head (lft)* ja *tail (rgt)*. Sõlmele *s* vastav intervall sisaldub alati *s* vanemaks oleva sõlme (kui selline eksisteerib) intervallis (Tropashko 2005). Tropashko (2005) kohaselt on sõlm *[clft, crgt]* sõlme *[plft, prgt]* järeltulija kui:

plft <= clft and crgt <= prgt

Kodeerimiseks saab kasutada mitte ainult täisarve, vaid kõiki reaalarve.



Joonis 12. Pesastatud intervallid

Joonisel 12 on näha, et *[plft, prgt]* intervalli sees on vaba interval *[lft1, rgt1],* kuhu saab paigutada sõlme intervalliga *[(2*lft1+rgt1)/3, (lft1+2*rgt1)/3]*. Peale lisamist on vabaks jäänud järgmised intervallid: *[lft1, (2*lft1+rgt1)/3]* ja *[(lft1+2*rgt1)/3, rgt1].*

Ülaltoodud näide ei ole kindel algoritm otspunktide arvutamiseks, vaid lihtne näide disaini põhimõtete illustreerimiseks. Üldjuhul pesastatud intervallide disain ei määra ette mingi kindla algoritmi, seda võib vabalt ise valida (üks olemasolevatest algoritmidest on näiteks *Farey* Fractions (Tropashko 2005)).

Eelised:

- Võrreldes pesastatud hulkade mudeliga ei ole uue sõlme lisamisel vaja ümber arvutada tervet puud, kuna pesastatud intervallide disaini korral sõlmed kodeeritakse (mitte nummerdatakse), ehk uue sõlme lisamisel arvutatakse selle sõlme intervall (teiste sõlmede intervallid jäävad samaks) (Tropashko 2005).

Puudused:

- -

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	EMPL_FIRSTNAME	EMPL_LASTNAME	HEAD	TAIL	
1	John	Walker	1	10	John Walker'il ei ole ülemust
2	Chris	Parker	5	6	Chris Parker on John Walker'i alluv
3	Paul	Stevens	7	8	Paul Stevens on John Walker'i alluv
4	Steve	Jobs	7,2	7,5	Steve Jobs on Paul Stevens'i alluv

Joonis 13. Pesastatud intervallide mudeli näide konkreetsete andmetega

Strateegia:

Tabeli ja indeksi loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     HEAD             BINARY_FLOAT NOT NULL,
     TAIL               BINARY_FLOAT NOT NULL,
    CONSTRAINT pk_employee PRIMARY KEY (empl_id),
    CONSTRAINT check_head_tail CHECK(tail > head),
    CONSTRAINT uni_head_tail UNIQUE (head, tail)
  );
```

Loodi CHECK kitsendus, millega kontrollitakse, et *tail* veeru väärtus peab olema rangelt suurem kui *head* veeru väärtus.

Loodi unikaalne kitsendus *uni_head_tail* veergudele *head* ja *tail*, millest tekib automaatselt ka unikaalne liitindeks nendele veergudele.

Märkus: *FLOAT* andmetüüpide kasutamine võib olla antimustriks, kui on oluline, et veergudes salvestatav väärtus oleks võimalikult täpne. Karwin (2010) kirjeldab SQL antimustrite raamatus antimustrit *Rounding Errors*, mille kohaselt kasutatakse *FLOAT* andmetüüpe ratsionaalarvude salvestamiseks. Kuna *BINARY_FLOAT* andmetüüp võib põhjustada ümardamise probleemi, siis

alternatiivselt on võimalik kasutada *employee* tabeli loomisel *head* veeru asemel kahte veergu *left_num* (vasaku väärtuse lugeja), *left_den* (vasaku väärtuse nimetaja) ning *tail* veeru asemel kahte veergu: *rgt_num* (parema väärtuse lugeja) , *rgt_den* (parema väärtuse nimetaja). Need veerud oleksid täisarvu tüüpi (*NUMBER(10,0)*).

2.4. Closure Table

Nimi inglise keeles: Closure Table

Nimi eesti keeles: Sulunditabel

Samuti teada kui: Bridge Table, Overlapping Trees

Allikad: (The simplest(?) way to do tree-based queries in SQL 2010), (Chodnicki 2010), (Blaha 2010)

Kirjeldus:

Sulunditabel kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides, mille puhul lisaks põhitabelile luuakse lisatabel, kus hoitakse kõiki seoseid sõlmede vahel. Teiste sõnadega arvutatakse välja ja materialiseeritakse kõik võimalikud teed eellaste ja järglaste vahel. Selle tabeli kaudu saab realiseerida hierarhiat suvalise tasemete arvuga.

Eelised:

- Üks ja sama sõlm võib osaleda erinevates puudes (näiteks ühel töötajal võib olla korraga kaks otseset ülemust) (Blaha 2010).
- Kiire konkreetse sõlme kõikide (otseste ja kaudsete) järeltulijate leidmine (Chodnicki 2010).

Puudused (Chodnicki 2010)**:**

- Kõikide andmete kättesaamiseks tuleb ühendada kaks tabelit .

Kui lisada tabelisse, kus hoitakse kõiki seoseid sõlmede vahel, juurde veel kaks veergu – *effectiveDate* ja *expirationDate*, siis tulemuseks on disaini nimega *Tree changing over Time* (Blaha 2010). Sellest võib mõelda kui käesoleva disaini variatsioonist.

Need kaks veergu võimaldavad andmebaasis talletada infot selle kohta, kuidas muutub hierarhia aja jooksul. Igal ajahetkel võib iga sõlmel olla maksimaalselt üks vanem ühes puus.

Joonisel 14 on tabelis *Employee_relation* kirjeldatud ära kõik hierarhias osalevad eellase-järglase paarid (paar oma otsese järglasega ning paarid kõigi kaudsete järglastega) ning nende paaride vahelise tee pikkus. Tee pikkuseks on sõlmede vaheliste servade arv.

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	EMPL_FIRSTNAME	EMPL_LASTNAME
1	John	Walker
2	Chris	Parker
3	Paul	Stevens
4	Steve	Jobs

Luuakse järgmise struktuuriga tabel nimega *Employee_relation*:

PARENT_ID	EMPL_ID	DEPTH	
1	2	1	John Walker on Chris Parker'i ülemus sügavusega 1 (otsene ülemus)
1	3	1	John Walker on Paul Stevens'i ülemus sügavusega 1 (otsene ülemus)
3	4	1	Paul Stevens on Steve Jobs'i ülemus sügavusega 1 (otsene ülemus)
1	4	2	John Walker on Steve Jobs'i ülemus sügavusega 2 (kaudne ülemus)

Joonis 14. Sulunditabeli mudeli näide konkreetsete andmetega

Strateegia:

Tabelite ja indeksi loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     CONSTRAINT pk_employee PRIMARY KEY (empl_id)
  );

CREATE TABLE employee_relation
  (
     parent_id NUMBER(10) NOT NULL,
     empl_id   NUMBER(10) NOT NULL,
     depth     NUMBER(10) NOT NULL,
     CONSTRAINT pk_empl_relation PRIMARY KEY (parent_id, empl_id),
     CONSTRAINT fk_parent_id FOREIGN KEY(parent_id) REFERENCES
employee(empl_id) ON DELETE CASCADE,
     CONSTRAINT fk_empl_id FOREIGN KEY(empl_id) REFERENCES
employee(empl_id) ON DELETE CASCADE,
     CONSTRAINT check_depth_positive CHECK(depth >= 0)
  );
```

Loodi *CHECK* kitsendus *check_depth_positive*, millega kontrollitakse, et *depth* veerus olev väärtus on null või nullist suurem.
Loodi *fk_parent_id* välisvõtme kitsendus, mille puhul kasutatakse määrangut *ON DELETE CASCADE*.
Loodi *fk_empl_id* välisvõtme kitsendus, mille puhul kasutatakse määrangut *ON DELETE CASCADE*.

2.5. Flat Table

Nimi inglise keeles: Flat Table

Nimi eesti keeles: Lametabel

Samuti teada kui: -

Allikad: (Four Ways To Work With Hierarchical Data 2000)

Kirjeldus:

Lametabel kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides. Lametabeli mudel on sarnane külgnevusnimistu mudeliga ehk iga tabeli rida määrab ära ühe otseselt seotud sõlmede paari. Lametabeli mudeli korral tabelisse lisatakse tabelisse juurde kaks veergu (Four Ways To Work With Hierarchical Data, 2000):

- *rank* (display_order) – järjestus (järjekorranumber, mille järgi sorteerides saab väljastada hierarhiat soovitud järjekorras)
- *depth* (indent_level) – hierarhia tase

Uue alamsõlme *s* järjekorranumber peab olema suurem kui vanema sõlme järjekorranumber ja väiksem kui *s* alamsõlmede järjekorranumber.

Eelised (Four Ways To Work With Hierarchical Data 2000)**:**

- Elementide järjestamise võimalus.

Puudused (Four Ways To Work With Hierarchical Data 2000)**:**

- Ajakulukas muutmine ja kustutamine, kuna uue sõlme lisamisel või kustutamisel, tuleb vastavalt kas suurendada järeltulijate *rank* veeru väärtust (+1) või vähendada (-1).
- Hierarhilisi seoseid esitavas veerus peab lubama NULL-markereid (puu juure korral), mis võivad tingida loogiliselt ebakorrektseid päringu tulemusi.

Näide:

Luuakse järgmise struktuuriga tabel nimegа *Employee*:

EMPL_ID	EMP_FIRSTNAME	EMPL_LASTNAME	PARENT_ID	RANK	DEPTH	
1	John	Walker		1	0	John Walker'il ei ole ülemust
2	Chris	Parker	1	2	1	Chris Parker on John Walker'i alluv
3	Paul	Stevens	1	3	1	Paul Stevens on John Walker'i alluv
4	Steve	Jobs	3	4	2	Steve Jobs on Paul Stevens'i alluv

Joonis 15. Lametabeli mudeli näide konkreetsete andmetega

Märkus:

Chris Parker ja Paul Stevens on mõlemad hierarhia teisel tasemel ehk nendel on üks ja sama otsene ülemus John Walker. Ülaltoodud näite puhul ei ole järjekord oluline ja seega on määratud *rank* veeru väärtus nende töötajate jaoks juhuslikult. Need väärtused võivad olla vastupidi, ehk Paul Stevens'i *rank* veeru väärtus 2 ja Chris Parker'i vastava veeru väärtus 1.

Kui on oluline ühel tasemel olevate töötajate (sõlmede) järjekord, siis tuleb seda vastavalt määrata.

Strateegia:

Tabeli loomise lause:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     rank           NUMBER(10) NOT NULL,
     depth         NUMBER(5) NOT NULL,
     parent_id      NUMBER(10),
     CONSTRAINT pk_employee PRIMARY KEY (empl_id),
     CONSTRAINT fk_employee FOREIGN KEY(parent_id) REFERENCES
employee(empl_id) ON DELETE CASCADE,
     CONSTRAINT check_depth_positive CHECK (depth >= 0) ,
     CONSTRAINT check_irreflexive CHECK(empl_id != parent_id)
  );

CREATE INDEX idx_parent_id
  ON employee (parent_id);
```

Loodi välisvõtme kitsendus *fk_employee*, mille puhul kasutati määrangut *ON DELETE CASCADE*.

Loodi *CHECK* kitsendus *check_depth_positive*, millega kontrollitakse, et *depth* väärtus on positiivne väärtus.

Loodi CHECK kitsendus *check_irreflexive*, millega kontrollitakse, et olem ei saa olla iseendaga seotud.

Loodi täiendav indeks *idx_parent_id* välisvõtme veerule *parent_id*.

2.6. Materialized Path

Nimi inglise keeles: Materialized Path

Nimi eesti keeles: Materialiseeritud tee

Samuti teada kui: Path Enumaration, Lineage Column

Allikad: (Tropashko 2010), (Stadnik 2008), (Smusenok, Trubilko, Furmanov 2013)

Kirjeldus:

Materialiseeritud tee kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides, mille puhul iga tabeli rida hoiab kogu teekonda juursõlmeni (ingl *Root*) (Tropashko 2010). Teekond koosneb identifikaatoritest. Identifikaatorite (numbrite) eraldamiseks kasutatakse tavaliselt *“.”* või *“/”*.

PostgreSQLis saab kasutada *ltree* moodulit, mis võimaldab hoida andmeid hierarhiate kohta märgiste kujul (vt Jaotis 1.2.1.2). See moodul võimaldab realiseerida hierarhiate esituse selle disaini kohaselt.

Eelised:

- Kiire sõlme lisamine puu leheks, kuna see ei nõua teekonna muutmist alamsõlmedes (Stadnik 2008).
- Kiire ja lihtne taseme leidmine, sest ei ole vaja läbi vaadata tabeli mitut rida, vaid piisab teed esitava stringi töötlemisest (Stadnik 2008).
- Visuaalselt hästi loetav, kuna iga sõlme korral on esitatud terve teekond juursõlmeni.

Puudused:

- Ajakulukas sõlme lisamine puu keskele (ehk hargnevate sõlmede alla), kuna see nõuab teekonna muutmist alamsõlmedes (Smusenok, Trubilko, Furmanov 2013).
- Ajakulukas sõlme kustutamine, kuna see nõuab teekonna muutmist alamsõlmedes (Stadnik 2008).
- Põhimõtteliselt piirab tasemete arvu maksimaalne lubatud stringi pikkus, mida konkreetne andmebaasisüsteem toetab. PostgreSQL ja Oracle võimaldavad vastavalt kasutada tüüpe TEXT ja CLOB, millesse kuuluva väärtuse suurus on vastavalt maksimaalselt umbes 1GB ja 2GB. Seega praktikas nende andmebaasisüsteemide korral ei ole see reaalseks probleemiks.

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	EMPLOYEE_FIRSTNAME	EMPLOYEE_LASTNAME	PATH	
1	John	Walker	.	John Walker'il ei ole ülemust
2	Chris	Parker	.1	Chris Parker on John Walker'i alluv
3	Paul	Stevens	.1	Paul Stevens on John Walker'i alluv
4	Steve	Jobs	.1.3	Steve Jobs on Paul Stevens'i otsene alluv

Joonis 16. Materialiseeritud tee mudeli näide konkreetsete andmetega

Strateegia:

Tabeli ja indeksi loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     path           VARCHAR2(4000) NOT NULL,
     CONSTRAINT pk_employee PRIMARY KEY (empl_id),
     CONSTRAINT path_num_point CHECK(REGEXP_LIKE (path, '^
([.][[:digit::]])+ $') OR REGEXP_LIKE (path, '^ [.]$')),
    CONSTRAINT uni_path UNIQUE (path)
  );
```

Loodi unikaalne kitsendus *idx_path_unique path* veerule *path*.

Loodi CHECK kitsendus *path_num_point*, millega kontrollitakse, et *path* veerus olev väärtus võib sisaldada ainult numbreid ja punkti.

2.7. Multiple Lineage Columns

Nimi inglise keeles: Multiple Lineage Columns

Nimi eesti keeles: Hulk pärilikkuse veerge

Samuti teada kui: David Chandler Model

Allikad: (Method of organizing hierarchical data in a relational database 2002)

Kirjeldus:

Hulk pärilikkuse veerge kirjeldab võimaliku hierarhiate esituse SQL-andmebaasides. Selle mudeli põhiidee seisneb selles, et iga hierarhia taseme jaoks luuakse eraldi veerg. Kõige ülemise taseme jaoks veergu ei looda. Sellist disainilahendust nimetatakse veel *David Chandler'i* mudeliks, kuna see on tema poolt 2002. aastal patenteeritud (Method of organizing hierarchical data in a relational database 2002).

Eelised (Method of organizing hierarchical data in a relational database 2002):

- Kiire vanemate, järeltulijate ja taseme leidmine.
- Kiire puu lehtede lisamine, kustutamine ja muutmine, kuna see ei nõua hierarhia andmete muutmist alamsõlmedes.

Puudused:

- Piiratud tasemete arv. Oracle andmebaasisüsteemis on maksimaalne veergude arv ühes tabelis 1000 (Database Administration) ning PosgtreSQL andmebaasisüteemis 250-1600 (sõltub veergude andmetüüpidest) (About PostgreSQL).
- Ajakulukas sisesõlmede lisamine, kustutamine ja muutmine, kuna see nõuab hierarhia andmete muutmist alamsõlmedes (Method of organizing hierarchical data in a relational database 2002)
- Kui hierarhia läheb sügavamaks (lisandub uus tase), siis tuleb lisaks tabelis olevate andmete muutmisele muuta ka tabeli struktuuri.

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	LEVEL1_PR_ID	LEVEL2_PR_ID	EMPL_FIRSTNAME	EMPL_LASTNAME	
1	0	0	John	Walker	John Walker'il ei ole ülemust.
2	1	0	Chris	Parker	John Walker on Chris Parker'i ülemus.
3	1	0	Paul	Stevens	John Walker on Paul Stevens'i ülemus (otsene).
4	1	3	Steve	Jobs	Paul Stevens on Steve Jobs'i ülemus (otsene). John Walker on Steve Jobs'i ülemus (kaudne).

Joonis 17. Hulk pärilikkuse veerge mudeli näide konkreetsete andmetega

Strateegia:

Tabeli ja indeksi loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     level1_pr_id   NUMBER(10) NOT NULL,
     level2_pr_id   NUMBER(10) NOT NULL,
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     CONSTRAINT pk_employee PRIMARY KEY (empl_id),
     CONSTRAINT fk_level1_empl_id FOREIGN KEY(level1_pr_id) REFERENCES
employee(
     empl_id) ON DELETE CASCADE,
     CONSTRAINT fk_level2_empl_id FOREIGN KEY(level2_pr_id) REFERENCES
employee(
     empl_id) ON DELETE CASCADE
  );

CREATE INDEX idx_fk_level1_pr_id
  ON employee (level1_pr_id);

CREATE INDEX idx_fk_level2_pr_id
  ON employee (level2_pr_id);
```

Loodi välisvõtme kitsendused *fk_level1_empl_id* ja *fk_level2_empl_id*, mille puhul kasutatakse määrangut *ON DELETE CASCADE*.

Loodi täiendavad indeksid *idx_fk_level1_pr_id* ja *idx_fk_level2_pr_id* välisvõtme veergudele *level1_pr_id* ja *level2_pr_id.*

2.8. Hardcoded Tree

Nimi inglise keeles: Hardcoded Tree

Nimi eesti keeles: Tase-tabel

Samuti teada kui: -

Allikad: (Blaha 2010)

Kirjeldus:

Tase-tabel on hierarhiate esitus SQL-andmebaasides, mille puhul iga hierarhia taseme jaoks luuakse eraldi tabel.

Eelised (Blaha 2010)**:**

- Selge struktuur, kuna iga taseme sõlmed on eraldi tabelis.

Puudused (Blaha 2010)**:**

- Otseste ja kaudsete alluvate/ülemuste leidmise päringute puhul tuleb iga taseme jaoks esitada ilmutatud kujul ühendamisoperatsioon. Pole võimalik päringud, mis muutmata kujul annavad soovitud tulemuse sõltumata hierarhias olevate tasemete arvust. Kui tasemete arv muutub, siis tuleb kõik selletüübilised päringud ümber kirjutada.
- Piiratud tasemete arv, kuna iga hierarhia taseme jaoks luuakse eraldi tabel. Tasemete arvu muutudes tuleb muuta tabelite hulka.

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee_level_1:*

Level_1_ID	EMPLOYEE_FIRSTNAME	EMPLOYEE_LASTNAME
1	John	Walker

Luuakse järgmise struktuuriga tabel nimega *Employee_level_2:*

Level_2_ID	EMPLOYEE_FIRSTNAME	EMPLOYEE_LASTNAME	Level_1_ID	
1	Chris	Parker	1	Chris Parker on John Walker'i alluv
2	Paul	Stevens	1	Paul Stevens on John Walker'i alluv

Luuakse järgmise struktuuriga tabel nimega *Employee_level_3:*

Level_3_ID	EMPLOYEE_FIRSTNAME	EMPLOYEE_LASTNAME	Level_2_ID
1	Steve	Jobs	2

Steve Jobs on Paul Stevens'i alluv

Joonis 18. Tase-tabeli mudeli näide konkreetsete andmetega

Strateegia:

Tabelite ja indeksite loomise laused:

```
CREATE TABLE employee_level_1
  (
     level_1_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     CONSTRAINT pk_employee_lev_1 PRIMARY KEY (level_1_id)
  );

CREATE TABLE employee_level_2
  (
     level_2_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     level_1_id NUMBER(10) NOT NULL,
     CONSTRAINT pk_employee_lev_2 PRIMARY KEY (level_2_id),
     CONSTRAINT fk_level1_id FOREIGN KEY(level_1_id) REFERENCES
employee_level_1(level_1_id) ON DELETE CASCADE
  );

CREATE TABLE employee_level_3
  (
     level_3_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     level_2_id NUMBER(10) NOT NULL,
     CONSTRAINT pk_employee_lev_3 PRIMARY KEY (level_3_id),
     CONSTRAINT fk_level2_id FOREIGN KEY(level_2_id) REFERENCES
employee_level_2(level_2_id) ON DELETE CASCADE
  );

CREATE INDEX idx_level_1_id
  ON employee_level_2 (level_1_id);

CREATE INDEX idx_level_2_id
  ON employee_level_3 (level_2_id);
```

Loodi välisvõtme kitsendused *fk_level1_id* ja *fk_level2_id*, mille puhul kasutatakse määrangut *ON DELETE CASCADE*.

Loodi täiendavad indeksid *idx_level_1_id* ja *idx_level_2_id* välisvõtme veergudele *level_1_id* ja *level_2_id*.

2.9. Degenerate Node and Edge

Nimi inglise keeles: Degenerate Node and Edge

Nimi eesti keeles: Otseste järeltulijate grupid

Samuti teada kui: -

Allikad: (Blaha 2010)

Kirjeldus:

Otseste järeltulijate gruppide disain on kasulik siis, kui on vaja hoida andmeid "vanem-laps" grupeerimise kohta. Selle disaini puhul hoitakse eraldi tabelis andmeid ühe vanema ja kõikide tema laste kohta.

Eelised (Blaha 2010)**:**

- Võimaldab hoida andmeid "vanem-laps" grupeerimise kohta.

Puudused:

- Andmed hierarhiate kohta on visuaalselt halvasti loetavad.
- Hierarhilisi seoseid esitavas veerus peab lubama NULL-markereid (puu juure korral), mis võivad tingida loogiliselt ebakorrektseid päringu tulemusi.

Näide:

Luuakse järgmise struktuuriga tabel nimega *Employee*:

EMPL_ID	EMPLOYEE_FIRSTNAME	EMPLOYEE_LASTNAME	NODE_ID
1	John	Walker	
2	Chris	Parker	10
3	Paul	Stevens	10
4	Steve	Jobs	11

Luuakse järgmise struktuuriga tabel nimega *Employee_node*:

NODE_ID	PARENT_ID	
10	1	Chris Parker ja Paul Stevens on John Walker'i alluvad
11	3	Steve Jobs on Paul Stevens'i alluv

Joonis 19. Otseste järeltulijate grupid mudeli näide konkreetsete andmetega

Strateegia:

Tabelite ja indeksite loomise laused:

```
CREATE TABLE employee
  (
     empl_id        NUMBER(10),
     empl_firstname VARCHAR2(100) NOT NULL,
     empl_lastname  VARCHAR2(100) NOT NULL,
     node_id        NUMBER(10),
     CONSTRAINT pk_employee  PRIMARY KEY (empl_id)
  );

CREATE TABLE employee_node
  (
     node_id        NUMBER(10),
     parent_id NUMBER(10) NOT NULL,
     CONSTRAINT pk_employee_node PRIMARY KEY (node_id)
  );

ALTER TABLE employee ADD CONSTRAINT fk_node_id FOREIGN KEY(node_id)
REFERENCES employee_node( node_id) ON DELETE CASCADE;

ALTER TABLE employee_node ADD CONSTRAINT fk_parent_id FOREIGN
KEY(parent_id) REFERENCES employee (empl_id) ON DELETE CASCADE;

CREATE INDEX idx_node_id
  ON employee(node_id);

CREATE INDEX idx_parent_id
  ON employee_node (parent_id);
```

Loodi välisvõtme kitsendused *fk_node_id* ja *fk_parent_id*, mille puhul kasutatakse määrangut *ON DELETE CASCADE*.

Loodi täiendavad indeksid *idx_node_id* ja *idx_parent_id* välisvõtme veergudele *node_id* ja *parent_id.*

2.10. Structured Tree

Nimi inglise keeles: Structured Tree

Nimi eesti keeles: Struktureeritud puu

Samuti teada kui: -

Allikad: (Blaha 2010)

Kirjeldus:

Struktureeritud puu on üldisem disain, mida on võimalik kasutada kombineerides seda teiste disainidega.

Struktureeritud puu disaini põhiidee seisneb selles, et lehed (sõlmed, mille järk on 0) eristatakse hargnevatest sõlmedest. Selle disaini kasutamisel luuakse lehtede ja hargnevate sõlmede jaoks eraldi tabelid. Lisaks luuakse üks põhitabel, kus hoitakse nii lehtede kuid ka hargnevate sõlmede andmeid. Lehtede ja hargnevate sõlmede tabelites on välisvõtmed, mis viitavad põhitabeli kandidaatvõtmele. Välisvõtmesse kuuluvad veerud on ka vastava tabeli primaarvõtme veerud.

Põhitabelis hierarhia esitamiseks on võimalik kasutada erinevaid disaine: külgnevusnimistu, pesastatud hulgad, materialiseeritud tee jne.

Eelised:

- Päringud, kus on vaja väljastada ainult lehtede või hargnevate sõlmede andmeid, on lihtsamad.

Puudused:

- Päringud, kus on vaja väljastada korraga nii lehtede kuid ka hargnevate sõlmede andmeid, on keerukamad ja raskelt mõistetavad.
- Hierarhilisi seoseid esitavas veerus peab lubama NULL-markereid (puu juure korral), mis võivad tingida loogiliselt ebakorrektseid päringu tulemusi.

Näide:

Luuakse järgmise struktuuriga tabel nimegа *Person*:

PRS_ID	PRS_FIRSTNAME	PRS_LASTNAME	PARENT_ID	
1	John	Walker		John Walker'il ei ole ülemust
2	Chris	Parker	1	Chris Parker on John Walker'i alluv
3	Paul	Stevens	1	Paul Stevens on John Walker'i alluv
4	Steve	Jobs	3	Steve Jobs on Paul Stevens'i alluv

Luuakse järgmise struktuuriga tabel nimega *Manager,* kus on hargnevate sõlmede identifikaatorid:

MAN_ID
1
3

Luuakse järgmise struktuuriga tabel nimega *Individual_contributor,* kus on lehtede identifikaatorid:

CONTR_ID
2
4

Joonis 20. Struktureeritud puu mudeli näide konkreetsete andmetega

Strateegia:

Tabelite ja indeksite loomise laused:

```
CREATE TABLE person
  (
     prs_id        NUMBER(10),
     prs_firstname VARCHAR2(100) NOT NULL,
     prs_lastname  VARCHAR2(100) NOT NULL,
     parent_id      NUMBER(10),
     CONSTRAINT pk_person PRIMARY KEY (prs_id),
     CONSTRAINT fk_parent_id FOREIGN KEY(parent_id) REFERENCES
person(prs_id) ON DELETE CASCADE,
     CONSTRAINT check_irreflexive CHECK(prs_id != parent_id)
  );

CREATE INDEX idx_parent_id
  ON person (parent_id);
```

```
CREATE TABLE manager
  (
     man_id        NUMBER(10),
     CONSTRAINT pk_manager PRIMARY KEY (man_id),
     CONSTRAINT fk_man_id FOREIGN KEY(man_id) REFERENCES person(prs_id)
     ON DELETE CASCADE
  );

CREATE TABLE individual_contributor
  (
     contr_id        NUMBER(10),
     CONSTRAINT pk_ind_contributor PRIMARY KEY (contr_id),
     CONSTRAINT fk_contr_id FOREIGN KEY(contr_id) REFERENCES person(prs_id)
     ON DELETE CASCADE
  );
```

Loodi välisvõtme kitsendused *fk_parent_id, fk_employee_id, fk_contr_id*, mille puhul kasutatakse määrangut *ON DELETE CASCADE*.

Loodi CHECK kitsendus, millega kontrollitakse, et olem ei ole iseendaga seotud.

Loodi täiendav indeks *idx_parent_id* välisvõtme veerule *parent_id*.

3. Eksperiment

Käesolevas peatükis kirjeldatakse detailselt käesoleva töö käigus läbiviidavat eksperimenti, mida hakatakse käesolevas töös läbi viima.

3.1. Varasemad uuringud

SQL-andmebaasides on sageli vaja säilitada hierarhilisi andmeid. Autor eeldas, et kindlasti on hierarhiliste andmete SQL-andmebaasides hoidmist varem uuritud. Autor leidis järgmised analoogsed uuringud.

- Atanassov, I. (2007). An Improvement of an Approach for Representation of Tree Structures in Relational Tables. – *Proceedings of the 2007 international conference on Computer systems and technologies: 14-15 June 2007, Rousse, Bulgaria.* II.9-1 - II.9-6

  Antud töös vaadeldi andmebaasisüsteeme Oracle 8i ja InterBase (versioon ei ole täpsustatud). Hierarhiate andmebaasis hoidmise realiseerimisel on disainiks valitud materialiseeritud tee. Eksperimendis viiakse läbi andmete lugemise operatsioon (teostatakse ainult üks päring). Eksperimendis uuritakse, kuidas mõjutab indeksite kasutamine päringu kiirust. Lisaks uuritakse teekonna veerus eraldajate kasutamise mõju päringute täitmise kiirusele (teekonna veerus hoitakse tervet teed juurelemendini).

  Tulemused näitasid, et indekseeritud tabelites on päringud kiiremad. Samas eraldajate kasutamine teekonna veerus muudab päringud aeglasemaks.

- Smusenok, S., Trubilko, A., Furmanov, A. (2013). Анализ способов представления иерархических структур в реляционных базах данных с использованием стресс-тестов. – *Открытые информационные и компьютерные интегрированные технологии. Выпуск 62.* Харьков: Нац. аэрокосм. ун-т "ХАИ", 107-11.

  Antud töös uuritakse järgmisi disaine: külgnevusnimistu, pesastatud hulgad, sulunditabel ja materialiseeritud tee. Eksperimendi operatsioonideks on andmete lugemine, salvestamine ja kustutamine. Eksperimendid viiakse läbi erinevate andmemahtudega. Kasutatavaid andmebaasisüsteeme ei ole täpsustatud (on öeldud, et tegemist on relatsiooniliste andmebaasisüsteemidega – ilmselt siis SQL-andmebaasisüsteemidega).

Selle töö tulemusena jõudsid autorid loogilise järelduseni, et erinevate ülesannete jaoks sobivad erinevad lahendused. Kui kriitilise tähtsusega kriteeriumiks on päringute kiirus, siis tuleb kasutada pesastatud hulkade mudelit. Kui on vaja tihti teha muudatusi puus, siis sobib paremini külgnevusnimistu disain. Sulunditabeli disain võimaldab teha kiiremaid päringuid konkreetse alampuude kohta, kuid uue sõlme lisamiseks kulub rohkem aega, kui teiste disainide korral.

- Stadnik, M. (2008). Иерархические структуры данных и производительность . [WWW] http://habrahabr.ru/post/47280/ (24.09.2014)

Antud töös kasutatakse MySQL 5.0 andmebaasisüsteemi. Disainideks on valitud külgnevusnimistu, pesastatud hulgad ja materialiseeritud tee. Eksperimendis viiakse läbi andmete lugemise, salvestamise ja kustutamise operatsioonid. Eksperimendid viiakse läbi erinevate andmemahtudega.

Selle töö tulemusena jõudis autor järelduseni, et pesastatud hulkade mudel ja materialiseeritud tee mudel sobivad kõigepealt selliste ülesannete jaoks, kus on vaja teha päringuid suuremate andmemahtude korral. Samal ajal on materialiseeritud tee mudeli kasutamise korral andmete muutmise operatsioonid (lisamine, kustutamine, muutmine) kiiremad võrreldes pesastatud hulkade disainiga.

3.2. Eksperimendi eesmärgid

Olemasolevates uuringutes on mitmeid puudujääke.

- Ei ole käsitletud PostgreSQL andmebaasisüsteemi (vähemasti pole seda andmebaasisüsteemi nimeliselt mainitud)
- Oracle Database andmebaasisüsteemi korral on kasutatud 2014. aasta sügiseks vananenud versiooni – praegu on turul versioon 12.1
- Pole välja toodud täitmisplaane, et uurida operatsioonide töökiiruse erinevuste tagamaid. Täitmisplaanide vaatamine annab võimaluse teada saada, kas erinevad andmebaasisüsteemid kasutavad ühe ja sama ülesande lahendamiseks sarnaseid või erinevaid strateegiaid.
- Ei ole analüüsitud ülesannete lahendamiseks vajalike lausete keerukust.
- Ei ole uuritud erinevate disainilahenduste mõju kettal salvestatavate andmete mahule.
- Ei ole uuritud deklaratiivsete kitsenduse esitamise võimalust erinevate disainide korral.

Antud töö eesmärgiks on laiendada eksperimente ning täita eelnevates uuringutes täheldatud tühikuid. Selleks on valitud kolm erinevat SQL-andmebaasi disaini, mis lahendavad erineval viisil hierarhiliste andmete esitamise ülesannet. Valiti ainult kolm disaini, sest plaanis on läbi viia küllaltki mahukad katsed ning liiga suure hulga disainide korral läheb töö maht liiga suureks. Eksperimendis uuritavateks disainilahendusteks on valitud *külgnevusnimistu, pesastatud hulgad* ja *materialiseeritud tee*. Need disainid on Google otsingu tulemuste järgi (vt Tabel 1) kõige populaarsemad ning loodetavasti pakub nende uurimine huvi võimalikult suurele inimeste hulgale. Materialiseeritud tee disain on valitud lametabeli disaini asemel, kuna lametabel on väga sarnane külgnevusnimistu mudeliga, mis on populaarsuse järgi esimesel kohal. Otsingutulemusi vaadates võib kohe välja pakkuda veel ühe lõputöö teema, et uurida disaine, mis on kõige vähem tuntud ning kõrvutada nende omadusi kõige populaarsemate disainide omadega.

Tabel 1. Disainide populaarsus Google otsingu tulemuste järgi (04.11.2014) (sulgudesse on kirjutatud täpne otsingustring)

Disain	**Otsingutulemusi**
Külgnevusnimistu (*"adjacency list" sql*)	28 700
Pesastatud hulgad (*"nested sets" sql*)	26 800
Lametabel (*"flat table" sql*)	18 800
Materialiseeritud tee (*"materialized path" sql*)	16 600
Struktureeritud puu (*"structured tree" sql*)	8 360
Sulunditabel (*"closure table" sql*)	4 620
Pesastatud intervallid (*"nested intervals" sql*)	1 770

Disain	Otsingutulemusi
Otseste järeltulijate grupid (*"degenerate node and edge" sql*)	329
Tase-tabel (*"hardcoded tree" sql*)	252
Hulk pärilikkuse veerge (*"multiple lineage columns" sql*)	43

Uuritavateks andmebaasisüsteemideks on autori poolt valitud Oracle Database (Oracle 12c Enterprise Edition Release 1) ja PostgreSQL (PostgreSQL 9.3.0.), kuna tegemist on populaarsete süsteemidega (DB-Engines Ranking 2014). DB-Engines Ranking (2014) kohaselt on 2014. aasta detsembris Oracle Database ja PostgreSQL populaarsuselt vastavalt esimesel ja neljandal kohal. Need andmebaasisüsteemid on kättesaadavad Tallinna Tehnikaülikooli serveris ja autor on nendega varasemalt kokku puutunud. Lisaks sellele kasutab autor Oracle Database andmebaasisüsteemi oma tööülesannete täitmisel igapäevatöös. Antud töö eksperimendi tulemused aitavad autori ettevõttel tulevikus potentsiaalselt paremaid andmebaase disainida. PostgreSQL korral ei saa jätta mainimata, et tegemist on tasuta pakutava ja avatud lähtekoodiga andmebaasisüsteemiga, mis ei jää arendajatele pakutavatelt võimalustelt maha parimatest kommertssüsteemidest. Selle kasutamine võiks aidata ettevõtetel olulisel määral kulusid kokku hoida. Valiku tegemisel tuleb teada kuidas see süsteem erinevates olukordades käitub.

Põhiküsimused, millele selle töö eksperimentidega vastust otsitakse, on järgnevad.

- Kuidas erinevad päringute ja andmemuudatuste kiirused *erinevates andmebaasisüsteemides* ühe ja sama disaini korral ja sama andmemahuga?
- Kuidas erinevad päringute ja andmemuudatuste kiirused ühes ja samas andmebaasisüsteemis *erinevate disainide* korral sama andmemahuga?
- Kuidas erinevad päringute ja andmemuudatuste kiirused ühes andmebaasisüsteemis sama disaini korral ja *erinevate andmemahtudega*?
- Milline on päringute koodi keerukus samas andmebaasisüsteemis erinevate disainide korral?
- Kas sama ülesande lahendamiseks mõeldud koodi keerukus on erinevates andmebaasisüsteemides ühesugune?
- Millised disainid toetavad millisel määral hierarhiatega seotud kitsenduste deklaratiivse esitamist?
- Milline on tabelite andmemaht erinevate disainide korral samas andmebaasisüsteemis?

Kavas on ka käsitleda andmebaasisüsteemide kasutatavaid täitmisplaane, selleks, et aru saada, millest tulenevad kiiruslikud erinevused lausete täitmisel.

3.3. Eksperimendi kirjeldus

Eksperimendi käigus loob autor ühe kontseptuaalse andmemudeli alusel (vt Joonis 16) kolm erinevat andmebaasi disaini kahes andmebaasisüsteemis (Oracle ja PostgreSQL) (vt Lisa 1 ja Lisa 2). PostgreSQL andmebaasis loob autor iga disaini jaoks eraldi skeemi, mille nimedeks on vastavalt *adjacency_list, nested_sets* ja *materialized_path*. Oracle andmebaasis paigutatakse kõik disainilahendused ühte skeemi nimega *C##TUD15.* Kuna Oracle andmebaasis on tabelid ühes skeemis, siis pannakse iga tabeli nime ette nimekonfliktide vältimiseks disaini inglisekeelne nimetus (näiteks *ADJACENCY_OSAPOOL*).

Vaadeldava kolme erineva disaini võrdlemiseks kahes erinevas andmebaasisüsteemides viib autor läbi järgnevad eksperimendid.

- Päringute täitmise kiiruse mõõtmine.
- Ridade lisamise kiiruse mõõtmine.
- Ridade kustutamise kiiruse mõõtmine.
- Päringute ja operatsioonide keerukuse mõõtmine (koodiridade arvu metoodika).
- Tabelite ja neile loodud indeksite andmemahu mõõtmine.

Kõiki päringuid ja operatsioone käivitatakse 5 korda ning arvutatakse saadud tulemuste põhjal aritmeetiline keskmine väärtus. Töökiiruse hindamisel kasutatakse PostgreSQL andmebaasisüsteemis käsku *EXPLAIN ANALYZE* ning Oracle andmebaasisüsteemis kasutatakse TOAD programmi moodulit *Auto Optimize SQL*, mis näitab päringu täitmiseks kulunud aega.

Päringute ja operatsioonide keerukuse mõõtmiseks kasutatakse koodiridade arvu metoodikat. Koodiridade arvu metoodika (ingl *Source lines of Code SLOC*) kirjeldab tarkvara meetrikat, mida kasutatakse tarkvara programmi koodi suuruse ja keerukuse hindamiseks. Selleks leitakse koodiridade arv valitud programmi lähtekoodis (Bhatt, Tarey, Patel 2012). Koodiridade arvu leidmiseks antud eksperemendi käigus kasutatakse *LocMetrics* tarkvara. Eksperemendi käigus uuritakse füüsiliselt käivitatavaid ridu (ingl *SLOC-P, Executable Physical*) (füüsilised read v.a tühjad read ja kommentaarid). Selleks, et tulemused oleksid võrreldatavad mõlemas andmebaasisüsteemis ja erinevate disainide korral päringute ja operatsioonide kirjutamisel, kasutatakse järgmisi põhimõtteid.

- Reserveeritud sõnad on kirjutatud suurtähtedega (näiteks SELECT, WITH)
- Põhilised võtmesõnad algavad uuel real (SELECT, FROM, WHERE, ORDER BY)
- SELECT klauslis algab iga veeru nimetus uuelt realt

- Iga tabeli nimi FROM klauslis algab uuelt realt
- Iga WHERE alamtingimus algab uuelt realt

PostgreSQL andmebaasisüsteemis saab tabeli andmemahu baitides teada kasutades funktsiooni *pg_total_relation_size(regclass)* (System Administration Functions, 16.10.2014). Antud funktsioon arvutab tabeli, tabeliga seotud indeksite ning süsteemisiseselt tabelist eraldi salvestatud liiga suurte väärtuste andmete andmemahu baitides. Oracle andmebaasisüsteemis saab käivitada päringu, mis vastavalt valitud skeemile arvutab iga tabeli (k.a indeksite) andmemahu baitides. Päringu skript on lisatud dokumendi lõppu (vt Lisa 3) (Stadnik 2013).

Kitsenduste deklareerimise võimaluste hindamine toimus vastavate tabelite loomise käigus.

3.3.1. Kasutatavad andmebaasisüsteemid

Eksperimendid viiakse läbi kahes andmebaasisüsteemis: PostgreSQL 9.3.0 ja Oracle 12c Enterprise Edition Release 1. Oracle andmebaasis päringute käivitamiseks, nende täitmiseks kulunud aja mõõtmiseks ja täitmisplaane uurimiseks kasutatakse programmi TOAD for Oracle Xpert Version 12.1.0.22. PostgreSQLis kasutatakse graafilist kasutajaliidest pakkuvat programmi PgAdmin 1.18.1.

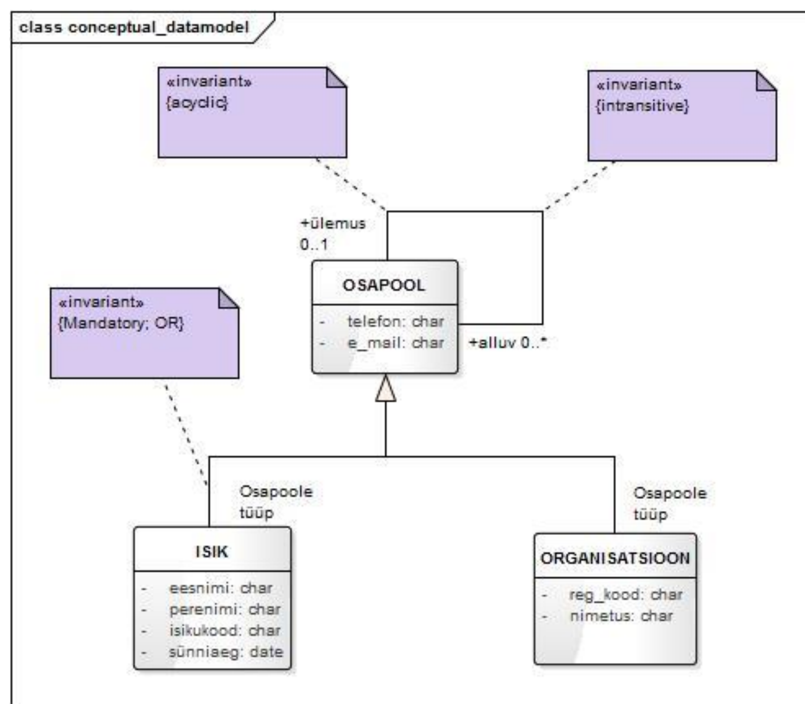
Eksperimentide läbiviimiseks kasutatakse Tallinna Tehnikaülikooli serverit *apex.ttu.ee* (edaspidi apex), kus tehakse katseid PostgreSQL 9.3.0 ja Oracle 12c andmebaasisüsteemides loodud andmebaaside põhjal. Tehnilised andmed: 40 GB RAM, 15 CPU-d, CentOS 6.4.

3.4. Eksperimendi andmebaasi projekteerimine

Antud töös kasutatakse osapoole mustrit (ingl *Party*) (Fowler 1995). Party muster on üks *arhetüüpmustritest*, mille alusel võimalik projekteerida ja realiseerida süsteeme väga erinevates valdkondades. Väga paljud väga erinevad süsteemid peavad ühel või teisel viisil pidama arvet sellega seotud osapoolte kohta. Piho (2011) kirjeldab oma doktoritöös arhetüüpidel põhinevat süsteemiarenduse metoodikat. Labori infosüsteemi (LIMS) loomise näitel demonstreeritakse selle edukat kasutamist. Lisaks Osapoole arhetüüpmustrile kasutatakse selles töös järgnevaid arhetüüpe: Toode, Osapoolte seos, Tellimus, Ladu, Ärireegel, Kogus, Raha.

3.4.1. Kontseptuaalne andmemudel

Kontseptuaalse andmemudeli esitamiseks kasutatakse antud töös olemi-suhte diagrammi (vt Joonis 21). Joonise loomiseks kasutati CASE vahendit Enterpise Architect.



Joonis 21. Osapoolte näite olemi-suhte diagramm

Joonis 21 kohaselt on iga isik osapool ja iga organisatsioon on osapool. *Osapool* on ülatüüp ning *Isik* ja *Organisatsioon* on selle alamtüübid.

Oletame, et tegemist on kontserniga, kus iga kontserni ettevõte on organisatsioon ja ettevõttes töötav inimene (töötaja) on isik. Töötajate vahel on alluvussuhted. Organisatsioonide vahel on ka

alluvussuhted. Organisatsiooni alluvaks ei saa olla töötaja. Töötaja alluvaks ei saa olla organisatsioon. Organisatsiooni ja töötajate vahel on töösuhe. Töösuhete kohta selles andmebaasis andmeid ei hoita.

Iga ülatüüpi kuuluv olem peab kuuluma vähemalt ühte alamtüüpi (sellele viidab osaluskohutus {Mandatory}). Iga ülatüüpi olem võib kuuluda maksimaalselt ühte alamtüüpi ehk iga osapool võib olla kas isik või organisatsioon, aga mitte mõlemat korraga (sellele viidab kuuluvus {Or}).

Tulenevalt kitsendusest *{Mandatory; Or}* võib kaaluda andmebaasi disaini, mille kohaselt luuakse tabelid *Isik* ja *Organisatsioon*, kuid eraldi tabelit *Osapool* ei looda. Tabelites *Isik* ja *Organiatsioon* oleks nii nende kui osapoolte üldised andmed kui ka vastavalt isikute ja organisatsioonide spetsiifilised andmed. Reaalsetes andmebaasides oleks küllaltki suure tõenäosusega andmemudelis seoseid *Osapool* tasemel (näiteks osapooltel on aadressid, lepingutega on seotud erinevad osapooled, kes võivad olla nii isikud kui organisatsioonid jne). Sellisel juhul oleks mõistlikum kasutada disainilahendust, mille kohaselt luuakse nii tabelid *Osapool*, *Isik* kui ka *Organisatsioon*. Tabelis *Osapool* on osapoolte üldandmed. Tabelites *Isik* ja *Organisatsioon* on vastavalt isikute ja organisatsioonide jaoks spetsiifilised andmed. Sellist lahendust kasutatakse ka käesoleva projekti eksperimentides. *{Mandatory; Or}* kitsenduse jõustamiseks mõeldud trigerite realiseerimine pole käesoleva töö eesmärgiga otseselt seotud ning seetõttu seda ei tehta.

Osapoolte vahel on nii irrefleksiivne kui asümmeetriline seosetüüp *{Acyclic}* ehk osapool ei saa olla seotud otseselt ega kaudselt iseendaga. Seosetüüp on ka intransitiivne *{Intransitive}* ehk sõlme vanemad ei saa olla selle laste otsesteks vanemateks.

Järgnevalt on esitatud Oracle abil realiseeritava SQL-andmebaasi disaini kirjeldavad diagrammid erinevate disainide korral. PostgreSQL abil realiseeritavas andmebaasis tuleb kasutada *VARCHAR2* andmetüübi asemel *VARCHAR* andmetüüpi ning *NUMBER(10)* andmetüübi asemel *INTEGER* andmetüüpi. Materialiseeritud tee disaini kasutamise korral kasutatakse PostgreSQL-is *ltree* moodulit. Sellel juhul tuleb osapool tabeli *teekond* veerule määrata *LTREE* andmetüüp. Nii Oracle kuid ka PostgreSQL andmebaasides kirjeldab ja loob autor ainult neid kitsendusi, mida saab esitada deklaratiivselt kasutades PRIMARY KEY, UNIQUE, FOREIGN KEY, NOT NULL ja CHECK kitsendusi. PostgreSQL võimaldab luua CHECK kitsendusi kasutades kasutaja-defineeritud funktsioone. PostgreSQL ei luba CHECK kitsendustes alampäringuid. Oracle ei luba kasutada CHECK kitsendustes alampäringuid ja funktsioone. Selleks, et eksperimendid oleksid läbi viidud

identsete tingimustega, luuakse mõlemas andmebaasis ainult need CHECK kitsendused, mida on võimalik realiseerida ilma alampäringuteta ja funktsioonideta.

Indeksite loomisel lähtutakse sellest, et PostgreSQLis ja Oracles ei indekseerita automaatselt välisvõtmeid. Ühendamisoperatsioonide kiirendamiseks ja andmete lukustamise vähendamiseks on soovitav välisvõtmed siiski indekseerida. Indekseerimisel tuleb jälgida, et loodavad indeksid ei hakkaks dubleerima PRIMARY KEY ja UNIQUE kitsenduste põhjal automaatselt loodavaid indekseid.

Kuidas muutub hierarhia peale kustutamist sõltub konkreetsest olukorrast. Üldiselt on kolm erinevat varianti:

- kõik osapooled, kes olid seotud kustutava osapoolega, lähevad kustutatava osapoole jaoks otseselt vanema osapoole alla,
- kõik alluvad osapooled (otsesed ja kaudsed), kes olid seotud kustutava osapoolega, kustutatakse ära,
- alluvate osapoolte ülemuseks saab teine suvaline osapool (uus või olemasolev).

Neid variante tuleb üldiselt realiseerida kas andmebaasis loodavate trigerite või protseduuride kaudu või rakenduse tasemel. Käitumist, mille kohaselt kõik kustutatavale osapoolele otseselt või kaudselt alluvad osapooled kustutatakse ära, saab külgnevusnimistu disaini korral realiseerida välisvõtmega seotud kompenseeriva tegevusega *ON DELETE CASCADE*.

Käesolevas töös realiseeritakse kõikide uuritavate mudelite korral selline käitumine, et kõik osapooled, kes olid kustutava osapoole *o* otseseks või kaudseks alluvaks lähevad *o* otsese vanema osapoole alla. Selleks ei loo autor trigereid, vaid realiseerib muudatust otseselt kustutamise operatsioonis. Eksperimendi käigus kustutatakse neid osapooli, mis ei ole kõige ülemisel tasemel. Kui kustutav osapool *o* on kõige ülemisel tasemel ja see kustutatakse, siis on vaja realiseerida käitumine, et peale kustutamist otsesed alluvad jäävad ilma ülemuseta või kustutatava osapoole asemele lisatakse uus osapool, kes saab kustutava osapoole otseste alluvate ülemuseks.

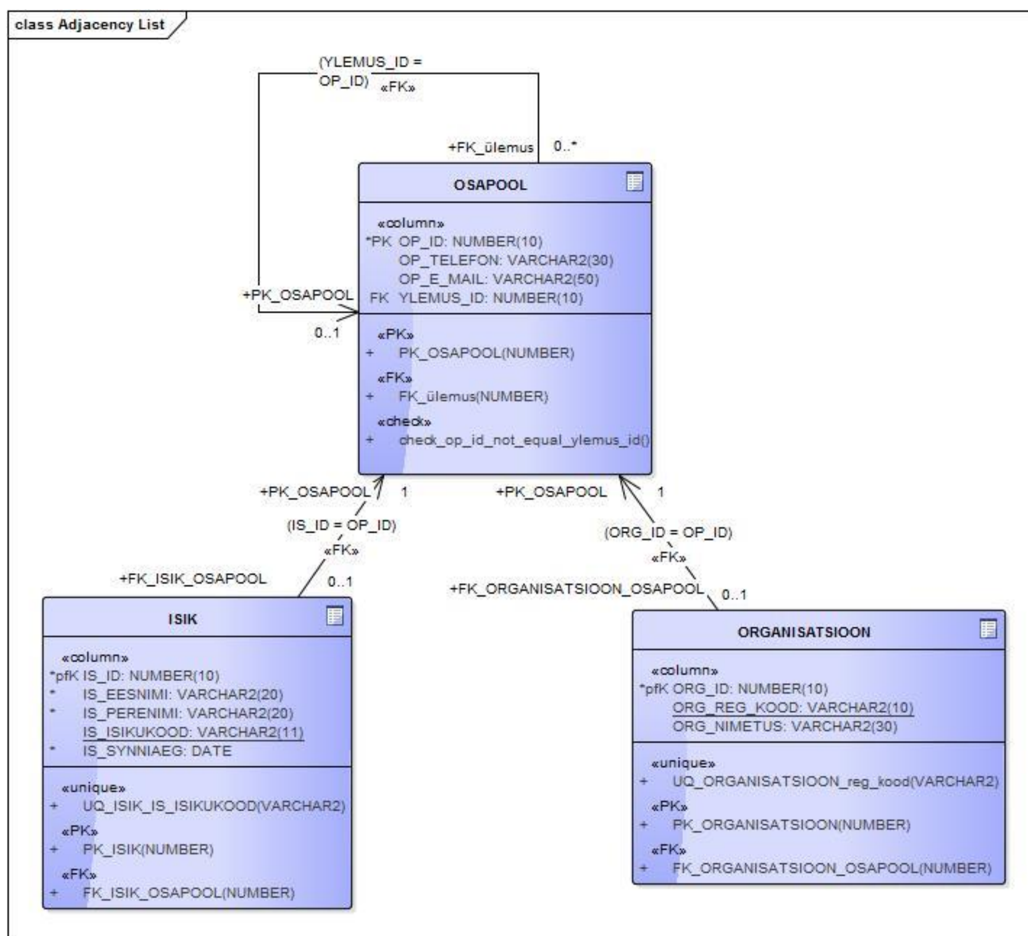
Isik ja *Organisatsioon* on osapoole peenendused. Süsteemis realiseeritakse käitumine, et kui kustutatakse ära konkreetse osapoole andmed, siis kustutatakse andmed ka tema peenenduste kohta. Selleks on järgmistele välisvõtmetele vaja deklareerida kaskaades kustutamise omadus (*ON DELETE CASCADE*).

- Tabel *isik* välisvõti (*IS_ID*)

  Tabel *organisatsioon* välisvõti (*ORG_ID*)

3.4.1.1. Andmebaasi disaini mudel külgnevusnimistu disaini kasutamise korral

Joonisel 22 on kujutatud külgnevusnimistu disaini andmebaasi diagramm. Tabelite arv on kolm, veergude arv 12 ja välisvõtmete arv kolm.



Joonis 22. Külgnevusnimistu disaini andmebaasi diagramm

PostgreSQL ja Oracle loovad primaarvõtme ja unikaalsuse kitsenduste alusel automaatselt indeksi. Täiendav indeks luuakse *Osapool* tabeli välisvõtme veerule *ylemus_id*. Täiendav indeks luuakse *isik* tabeli veerule *is_synniaeg*.

Tabelis *osapool* luuakse CHECK kitsendus, millega kontrollitakse, et olem ei saa olla iseenda alluv/ülemus (*CHECK (op_id!=ylemus_id)*).

Välisvõtme määrangut *ON DELETE SET NULL* kasutatakse järgmise välisvõtme puhul.

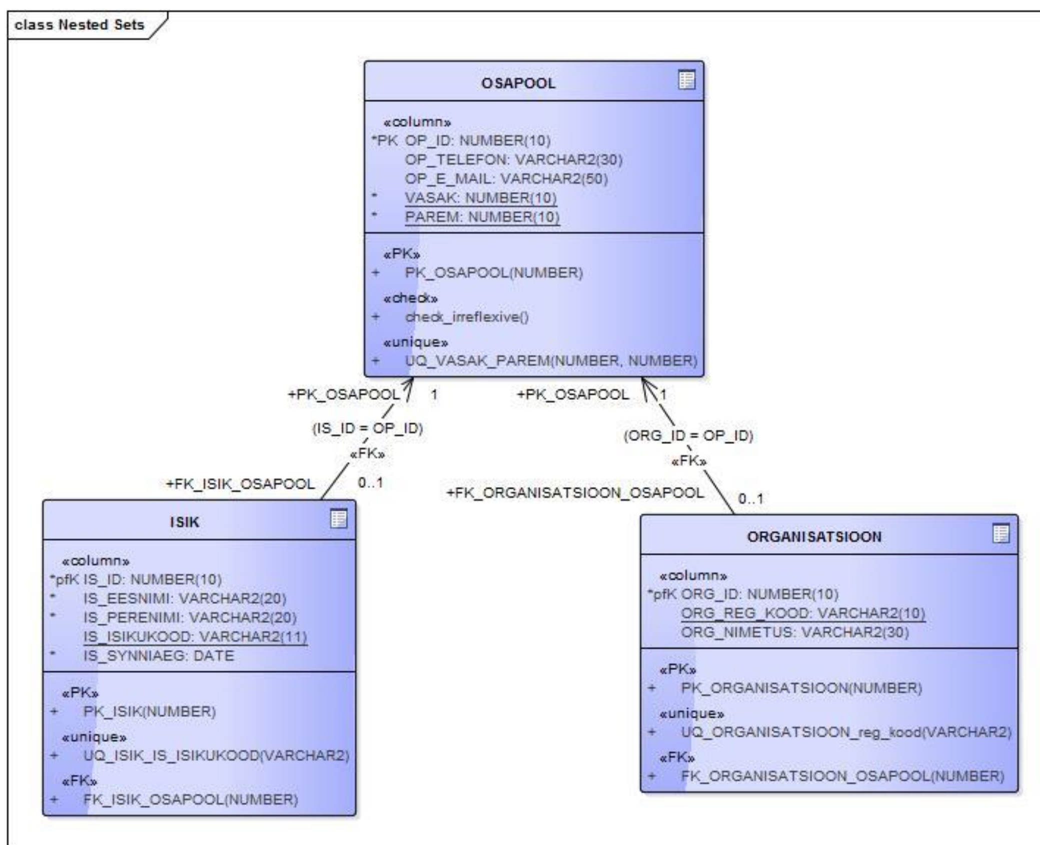
- Tabel *osapool* välisvõti (*Ylemus_ID*)

Välisvõtme määrangut *ON DELETE CASCADE* kasutatakse järgmiste välivõtmete puhul.

- Tabel *isik* välisvõti (*is_id*)
- Tabel *organisatsioon* välisvõti (*org_id)*

3.4.1.2. Andmebaasi disain pesastatud hulkade disaini kasutamise korral

Joonisel 23 on kujutatud pesastatud hulkade disaini andmebaasi diagramm. Tabelite arv on kolm, veergude arv 13 ja välisvõtmete arv kolm.



Joonis 23. Pesastatud hulkade disaini andmebaasi diagramm

Luuakse unikaalne kitsendus *Osapool* tabeli veergude *vasak* ja *parem* kombinatsioonile. PostgreSQL ja Oracle loovad primaarvõtme ja unikaalsuse kitsenduste alusel automaatselt indeksi. Täiendav indeks luuakse *Isik* tabeli veerule *is_synniaeg*.

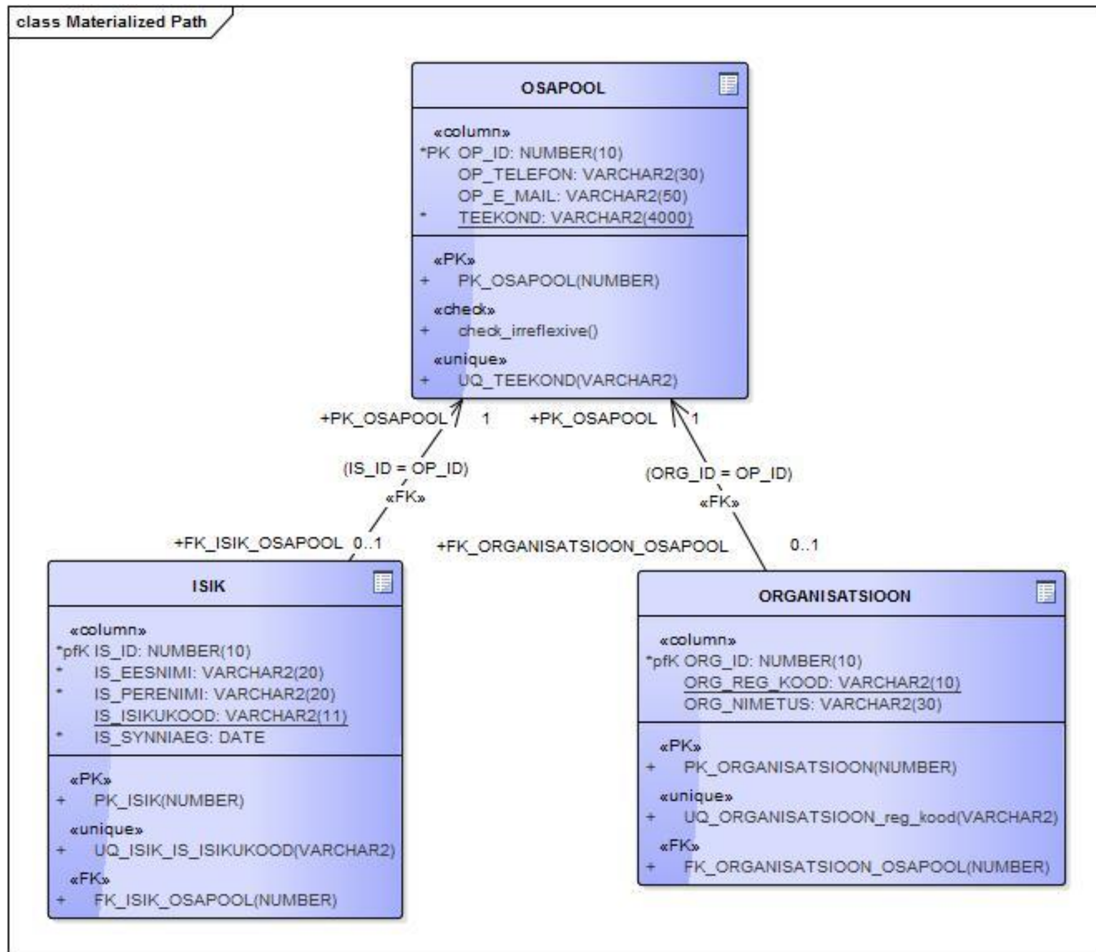
Tabelis *Osapool* luuakse CHECK kitsendus, millega kontrollitakse, et *vasak* veerus olev väärtus on väiksem kui *parem* veerus olev väärtus, *vasak* väärtus on kas nulliga võrdne või nullist suurem ning *parem* veeru väärtus on rangelt nullist suurem (otseselt seda antud disain ei nõua, kuid antud juhul on lihtsustamise mõttes kasulik selline kitsendus luua) (*CHECK(vasak < parem AND vasak >= 0 AND parem > 0)*).

Välisvõtme määrangut *ON DELETE CASCADE* kasutatakse järgmiste välivõtmete puhul.

- Tabel *Isik* välisvõti (*is_id*)
- Tabel *Organisatsioon* välisvõti (*org_id*)

3.4.1.3. Andmebaasi disain materialiseeritud tee disaini kasutamise korral

Joonisel 24 on kujutatud materialiseeritud tee disaini andmebaasi diagramm. Tabelite arv on kolm, veergude arv 12 ja välisvõtmete arv kolm.



Joonis 24. Materialiseeritud tee disaini andmebaasi diagramm

Luuakse unikaalne kitsendus *Osapool* tabeli *teekond* veerule. PostgreSQL ja Oracle loovad primaarvõtme ja unikaalsuse kitsenduste alusel automaatselt indeksi. Täiendav indeks luuakse *Isik* tabeli veerule *is_synniaeg*.

Luuakse CHECK kitsendus, millega kontrollitakse, et olem *osapool* ei oleks iseendaga otseselt ega kaudselt seotud (*CHECK (teekond NOT LIKE '%.' || op_id || '.%' AND teekond NOT LIKE op_id || '.' || op_id)*).

Välisvõtme määrangut *ON DELETE CASCADE* kasutatakse järgmiste välivõtmete puhul.

- Tabel *isik* välisvõti (*is_id*)
- Tabel *organisatsioon* välisvõti (*org_id*)

3.5. Testandmed

Kõik eksperimendi käigus loodud andmebaasid täidetakse testandmetega. Põhiandmete (isikute ja organisatsiooni andmed, osapoole telefoni ja e-posti aadress) genereerimiseks kasutatakse veebirakendust *Mockaroo* (http://www.mockaroo.com). Hierarhia genereerimiseks luuakse PL/SQL skript (Oracle andmebaasi jaoks), mis ehitab iga disaini jaoks hierarhia. Nii organisatsioonide kuid ka isikute hierarhiate maksimaalne sügavus on kõikide andmemahtude korral neli (ehk neli taset). Dokumendi lõpus on lisatud skript külgnevusnimistu disaini jaoks (vt Lisa 4), mis genereerib hierarhia kõige suurema andmemahu jaoks. Väiksemate andmemahtude jaoks vähendatakse igal tasemel olevate osapoolte arvu (*ROWNUM*) ning hierarhia maksimaalne sügavus jääb samaks kõikide andmemahtude korral. Skriptid esitatakse Lisa 4, Lisa 5 ja Lisa 6. PotgreSQL andmebaasi täidetakse andmetega, mida eksporditakse Oracle andmebaasist. Andmete eksportimiseks kasutatakse TOAD'i sisseehitatud funktsionaalsust *Export Dataset,* millega luuakse andmete lisamise skript (ingl *INSERT statements*). Seda skripti korrigeeritakse vastavalt vajadustele (näiteks tabelite ja veergude nimetused pannakse jutumärkidesse) ning käivitatakse saadud skript PosgtreSQL andmebaasis. Seega sama andmemahu korral on mõlema andmebaasisüsteemi andmebaasis kõigi kolme disaini korral andmed täpselt ühesuguste hierarhiate kohta.

Genereeritav andmete hulk on esitatud Tabelis 2.

Tabel 2. Eksperimendi andmemahud

	Isikuid	**Organisatsioone**	**Osapooli**
Andmemaht 3	150	100	250
Andmemaht 2	5 000	4 000	9 000
Andmemaht 1	15 000	10 000	25 000

Eksperiment algab kõige suurema andmemahuga. Kui eksperiment on läbi viidud, siis kustutatakse tabelitest kõik andmed, lisatakse väiksem hulk andmeid, genereeritakse hierarhiad ning värskendatakse andmebaasi statistikat (et andmebaasisüsteemil oleks andmete hulga ja jaotuse kohta täpne informatsioon). Kolmanda andmemahuga eksperimendiks korratakse samu tegevusi.

Kõik genereeritavad andmed on juhusliku pikkuse ja sisuga.

- Telefon: kuni 30 tähemärki, ainult numbrid ja kriipsud
- E-posti aadress: kuni 50 tähemärki, tähed, numbrid, punktid ja @ märk

- Eesnimi inglise keeles: kuni 20 tähemärki, ainult tähed
- Perenimi inglise keeles: kuni 20 tähemärki, ainult tähed
- Isikukood: 11 tähemärki, ainult numbrid
- Sünniaeg: vahemikus 01.01.1900 kuni 30.09.2014
- Registratsiooni kood: 8 tähemärki, ainult numbrid
- Nimetus: kuni 30 tähemärki, tähed ja numbrid

3.6. Eksperimendi käigus katsetatavad operatsioonid

Eksperimendi läbiviimiseks on valitud järgmised operatsioonid:

- andmete lugemine (kaks päringut),
- lisamine (üks operatsioon),
- kustutamine (üks operatsioon).

Esimese päringu (S1) abil soovitakse leida konkreetse isiku alluvad (otsesed ja kaudsed), kes on sündinud peale 01.01.1990. Osapoole kaudne alluv on mõne tema otsese alluva otsene või kaudne alluv. Päringu tulemusena väljastatakse leitud isikute kohta kõik andmed, mis on tabelis *Osapool* ja *Isik* (osapoole numbriline identifikaator väljastatakse ühekordselt). Tulemused järjestatakse perekonnanime, eesnime ja sünniaja järgi kasvavalt.

Teise päringuga (S2) soovitakse leida iga organisatsiooni kohta kõigi tema otseste ja kaudsete alluvate organisatsioonide arv.

Kolmanda operatsiooniga (I1) lisatakse organisatsiooni alla uus organisatsioon. Uus organisatsioon lisatakse hierarhia keskele.

Neljanda operatsiooniga (D1) kustutatakse puu keskelt organisatsioon. Peale kustutamist lähevad kõik osapooled, kes olid seotud kustutava osapoolega, otsese vanema osapoole alla (kõik käitumise variandid on kirjeldatud jaotises 3.4.1.1).

Päringus S1 ja operatsioonides I1 ja D1 kasutatakse konkreetseid osapooli. Kuna iga andmemahu jaoks genereeritakse hierarhiad juhuslikult, siis igas katses kasutatakse erinevaid osapooli. Tabelis 2 on välja toodud päringute ja muutmis operatsioonide laused, mida kasutatakse kõige suurema andmemahu jaoks. Järgnevalt esitatakse teiste andmemahtudega tehtavates kasutatavates kasutatavad osapooled.

Andmemaht 2:

- S1 - otsitava isiku ID on 22230
- I1 - uus organisatsioon lisatakse organisatsiooni ID-ga 3385 alla
- D1 - kustutakse organisatsioon ID-ga 3385

Andmemaht 3:

- S1 - otsitava isiku ID on 24852
- I1 - uus organisatsioon lisatakse organisatsiooni ID-ga 9909 alla
- D1 - kustutakse organisatsioon ID-ga 9909

Järgnevalt (vt Tabel 3) on esitatud eksperimendi päringud ja operatsioonid.

A – külgnevusnimistu (ingl *Adjacency List*)
N – pesastatud hulgad (ingl *Nested Sets*)
M – materialiseeritud tee (ingl *Materialized Path*)

TOAD võimaldab kasutada *Auto Optimize SQL* moodulit, mis otsib valitud päringule alternatiivvariandid, ning võrdleb neid omavahel (päringu täitmiseks kuluv aeg). Antud eksperimendi käigus analüüsitakse sellisel viisil kõik Oracle andmebaasisüsteemis katsetavaid päringuid ning parema alternatiivi leidmisel kasutatakse seda. *Auto Optimize SQL* ei pakkunud nende päringute korral paremat lahendust. Alternatiivide leidmiseks on võimalik ka kasutada TOAD *SQL Optimizer* moodulit, mis oskab paremini päringuid ümber kirjutada. Antud eksperimendi käigus ei kasutanud autor seda moodulit, kuna see on tasuline ja nõuab eraldi installeerimist.

Igal andmebaasiobjektil (nt tabel, vaade) on identifikaator e nimi. Piiritletud identifikaator tähendab, et identifikaator on pandud jutumärkidesse (nt "ISIK"), piiritlemata identifikaator ei ole jutumärkides. PosgtreSQL andmebaasisüsteemis kasutatakse tabelite ja veergude nimedena piiritletud identifikaatoreid. Oracle andmebaasi puhul seda ei tehta. Miks see on niimoodi? Autoril on mugavam andmebaasiga töötada (nt vaadata tabelite nimekirja) kui kõik identifikaatorid on suurtähtedega. Oracle andmebaasi süsteemikataloogis salvestatakse piiritlemata identifikaatorid suurtähtedega, PostgreSQL andmebaasi süsteemikataloogis väiketähtedega. Selleks, et näha PostgreSQL korral väljundis tabelite ja veergude nimesid suurtähtedega (autorile meeldib nii rohkem) on kõik need identifikaatorid loodud piiritletud identifikaatoritena, mis sisaldavad ainult suurtähti.

PL/SQL keele põhistruktuuriks on plokk (Feuerstein 2011). Oracle andmebaasisüsteemis on muutmisoperatsioonid (I1 ja D1), mis on kirjutatud PL/SQL keeles, koondatud anonüümsetesse plokkidesse (*BEGIN ... END* vahele). Iga plokki algus on määratud võtmesõnaga *BEGIN* ning lõpp on

määratud *END* võtmesõnaga. Iga muutmislause (näiteks *INSERT, UPDATE, DELETE*) plokkis algatab uue transaktsiooni. TOAD on seadistatud niimoodi, et peale iga lause täitmist tehakse automaatselt COMMIT (*AUTOCOMMIT ON*), millega lõpetatakse transaktsioon. Samas, kui mõne lause täitmisel tekib viga, siis lükatakse automaatselt tagasi kõik vastavas plokis tehtud muudatused.

PostgreSQL andmebaasisüsteemis on muutmisoperatsioonide (I1 ja D1) laused samuti koondatud anonüümsesse plokki. PostgreSQL käsitleb anonüümses plokis tehtud muudatusi ühe transaktsioonina. Kui vähemalt üks muudatus ebaõnnestub, siis ebaõnnestub kogu transaktsioon (st kõik selles sisalduvad muudatused jäävad tegemata).

Antud töös kasutatakse anonüümseid plokke, kuna reaalses rakenduses pandaks sellised muudatuse protseduuridesse (Oracle) või funktsioonidesse (PostgreSQL), mis koosnevad plokkidest.

PostgreSQL andmebaasisüsteemis kasutatakse S1M päringus ja I1M, D1M muutmisoperatsioonides *ltree* mooduli operaatorit ~, mis töötab analoogselt *LIKE* operaatoriga, ehk kontrollib kas *ltree* väärtuses sisaldub otsitav string (*ltree ~ lquery*). S2M päringus kasutatakse operaatorit *LIKE*, kuna otsitav string sisaldab tabeli veerus olevat väärtust (sellel juhul muudetakse ka *ltre*e veerus olev väärtus stringiks, kasutades tüübiteisenduse operaatorit *::text*).

Tabel 3. Eksperimendis kasutatavad päringud ja operatsioonid

	Oracle	**PostgreSQL**
S1 A	WITH r(op_id, ylemus_id, op_telefon, op_e_mail) AS (SELECT op_id, ylemus_id, op_telefon, op_e_mail FROM adjacency_osapool WHERE ylemus_id = 10796 UNION ALL SELECT o.op_id, r.op_id, o.op_telefon, o.op_e_mail FROM adjacency_osapool o, r WHERE r.op_id = o.ylemus_id) SELECT is_id, is_perenimi, is_eesnimi, is_synniaeg, op_telefon, op_e_mail FROM r, adjacency_isik WHERE is_id = op_id AND is_synniaeg >= To_date('01.01.1990', 'DD.MM.YYYY') ORDER BY is_perenimi, is_eesnimi, is_synniaeg;	WITH recursive r("OP_ID", "YLEMUS_ID", "OP_TELEFON", "OP_E_MAIL") AS (SELECT "OP_ID", "YLEMUS_ID", "OP_TELEFON", "OP_E_MAIL" FROM "OSAPOOL" WHERE "YLEMUS_ID" = 10796 UNION ALL SELECT "O"."OP_ID", "r"."OP_ID" , "O"."OP_TELEFON", "O"."OP_E_MAIL" FROM "OSAPOOL" "O", "r" WHERE "r"."OP_ID" = "O"."YLEMUS_ID") SELECT "IS_ID", "IS_PERENIMI", "IS_EESNIMI", "IS_SYNNIAEG", "OP_TELEFON", "OP_E_MAIL" FROM "r", "ISIK" WHERE "IS_ID" = "OP_ID" AND "IS_SYNNIAEG" >= to_date('01.01.1990', 'DD.MM.YYYY') ORDER BY "IS_PERENIMI", "IS_EESNIMI", "IS_SYNNIAEG";
S1 N	SELECT is_id, is_perenimi, is_eesnimi, is_synniaeg, T2.op_telefon, T2.op_e_mail FROM nsets_isik, nsets_osapool T1, nsets_osapool T2 WHERE T2.op_id = is_id AND T2.vasak > T1.vasak AND T2.vasak <= T1.parem AND T1.op_id = 10796 AND is_synniaeg >= To_date('01.01.1990', 'DD.MM.YYYY') ORDER BY is_perenimi, is_eesnimi, is_synniaeg;	SELECT "IS_ID", "IS_PERENIMI", "IS_EESNIMI", "IS_SYNNIAEG", "T2"."OP_TELEFON", "T2"."OP_E_MAIL" FROM "ISIK", "OSAPOOL" "T1", "OSAPOOL" "T2" WHERE "T2"."OP_ID" = "IS_ID" AND "T2"."VASAK" > "T1"."VASAK" AND "T2"."VASAK" <= "T1"."PAREM" AND "T1"."OP_ID" = 10796 AND "IS_SYNNIAEG" >= To_date('01.01.1990', 'DD.MM.YYYY') ORDER BY "IS_PERENIMI", "IS_EESNIMI", "IS_SYNNIAEG";

<table>
<tr><th></th><th>Oracle</th><th>PostgreSQL</th></tr>
<tr><td>S1 M</td><td><pre>
SELECT is_id,
            is_perenimi,
            is_eesnimi,
            is_synniaeg,
            op_telefon,
            op_e_mail
FROM   mpath_osapool,
       mpath_isik
WHERE is_id = op_id
       AND teekond LIKE
'%.10796.%'
       AND is_synniaeg >=
To_date('01.01.1990',
'DD.MM.YYYY')
ORDER BY is_perenimi, is_eesnimi,
is_synniaeg;
</pre></td><td><pre>
SELECT "IS_ID",
            "IS_PERENIMI",
            "IS_EESNIMI",
            "IS_SYNNIAEG",
            "OP_TELEFON",
            "OP_E_MAIL"
FROM   "OSAPOOL",
       "ISIK"
WHERE "IS_ID" = "OP_ID"
       AND "TEEKOND" ~ '*.10796.*'
       AND "IS_SYNNIAEG" >=
To_date('01.01.1990', 'DD.MM.YYYY')
ORDER BY "IS_PERENIMI", "IS_EESNIMI",
"IS_SYNNIAEG";
</pre></td></tr>
<tr><td>S2A</td><td><pre>
  SELECT op1.op_id,
       (SELECT Count(org.org_id)
        FROM   adjacency_osapool
op,

adjacency_organisatsioon org
        WHERE op.op_id =
org.org_id
        START WITH op.ylemus_id =
op1.op_id
        CONNECT BY PRIOR op.op_id
= op.ylemus_id) AS org
FROM   adjacency_osapool op1,
       adjacency_organisatsioon
WHERE op_id = org_id;
</pre></td><td><pre>
WITH recursive r("OP_ID", "YLEMUS_ID",
"PATH") AS
(SELECT "OP_ID",
              "YLEMUS_ID",
              cast (ARRAY["OP_ID"] AS
VARCHAR(100)[])
       FROM   "OSAPOOL",
              "ORGANISATSIOON"
       WHERE "OP_ID" = "ORG_ID"
       AND   "YLEMUS_ID" IS NULL
       UNION ALL
       SELECT   "O"."OP_ID",
                  "r"."OP_ID" ,
                  cast("r"."PATH"
                             ||
"O"."OP_ID"::VARCHAR AS VARCHAR(100) [])
       FROM   "OSAPOOL" "O"
       inner join "r"
       ON   "O"."YLEMUS_ID" =
"r"."OP_ID"
       inner join "ORGANISATSIOON"
       ON   "ORG_ID"= "O"."OP_ID"
)
SELECT "ORG_ID",
       (SELECT count(r."OP_ID")
              FROM   r
              WHERE "ORG_ID"::VARCHAR =
ANY("PATH")
              AND   "ORG_ID" !=
r."OP_ID")
FROM   "ORGANISATSIOON";
</pre></td></tr>
</table>

<table>
<tr><th></th><th>Oracle</th><th>PostgreSQL</th></tr>
<tr><td>S2 N</td><td>SELECT T4.op_id,
(SELECT Count(T3.org_id)
FROM nsets_osapool T1,
nsets_osapool T2,

nsets_organisatsioon T3
WHERE T2.vasak >
T1.vasak
AND T2.vasak <=
T1.parem
AND T4.op_id =
T1.op_id
AND T3.org_id =
T2.op_id) AS org
FROM nsets_osapool T4,
nsets_organisatsioon ORG
WHERE T4.op_id = ORG.org_id;</td><td>SELECT "T4"."OP_ID",
(SELECT Count("T3"."ORG_ID")
FROM "OSAPOOL" "T1",
"OSAPOOL" "T2",
"ORGANISATSIOON" "T3"
WHERE "T2"."VASAK" >
"T1"."VASAK"
AND "T2"."VASAK" <=
"T1"."PAREM"
AND "T4"."OP_ID" =
"T1"."OP_ID"
AND "T3"."ORG_ID" =
"T2"."OP_ID") AS org
FROM "OSAPOOL" "T4",
"ORGANISATSIOON"
WHERE "ORG_ID" = "OP_ID";</td></tr>
<tr><td>S2 M</td><td>SELECT mp.op_id,
(SELECT Count(org_id)
FROM mpath_osapool mp2,

mpath_organisatsioon
WHERE mp2.op_id = org_id
AND mp2.teekond
LIKE

mp.teekond

|| '.%') AS org
FROM mpath_osapool mp,
mpath_organisatsioon org
WHERE org.org_id = mp.op_id;</td><td>SELECT "mp"."OP_ID",
(SELECT Count("ORG_ID")
FROM "OSAPOOL" "mp2",
"ORGANISATSIOON"
WHERE "ORG_ID" = "mp2"."OP_ID"
AND "mp2"."TEEKOND"::text
LIKE "mp"."TEEKOND"::text || '.%') AS
org
FROM "OSAPOOL" "mp",
"ORGANISATSIOON" "ORG_ID"
where "ORG_ID" = "OP_ID";</td></tr>
</table>

	Oracle	PostgreSQL

I1 A	`BEGIN`
	`    INSERT INTO adjacency_osapool`
	`                (op_id,`
	`                 op_telefon,`
	`                 op_e_mail,`
	`                 ylemus_id)`
	`    VALUES      (25001,`
	`                 '111111',`
	
	`'insert_organisatsioon@google.com`
	`',`
	`                 667);`
	
	`    INSERT INTO`
	`adjacency_organisatsioon`
	`                (org_id,`
	`                 org_reg_kood,`
	`                 org_nimetus)`
	`    VALUES      (25001,`
	`                 '1111111',`
	`                 'Insert`
	`Organisatsion');`
	
	`    UPDATE adjacency_osapool`
	`    SET    ylemus_id = 25001`
	`    WHERE  ylemus_id = 667`
	`           AND op_id != 25001;`
	`END;`
		`DO $$`
		`BEGIN`
		`    INSERT INTO "OSAPOOL"`
		`                ("OP_ID",`
		`                 "OP_TELEFON",`
		`                 "OP_E_MAIL",`
		`                 "YLEMUS_ID")`
		`    VALUES      (25001,`
		`                 '111111',`
		
		`'insert_organisatsioon@google.com',`
		`                 667);`
		
		`    INSERT INTO "ORGANISATSIOON"`
		`                ("ORG_ID",`
		`                 "ORG_REG_KOOD",`
		`                 "ORG_NIMETUS")`
		`    VALUES      (25001,`
		`                 '1111111',`
		`                 'Insert`
		`Organisatsion');`
		
		`    UPDATE "OSAPOOL"`
		`    SET    "YLEMUS_ID" = 25001`
		`    WHERE  "YLEMUS_ID" = 667`
		`           AND "OP_ID" != 25001;`
		`END;`
		`$$ LANGUAGE plpgsql;`

	Oracle	PostgreSQL

I1 N

Oracle

```sql
DECLARE
    p_parent_parem NUMBER(10);
    p_parent_vasak NUMBER(10);
BEGIN
    SELECT vasak,
           parem
    INTO   p_parent_vasak,
p_parent_parem
    FROM   nsets_osapool
    WHERE  op_id = 667;

    UPDATE nsets_osapool
    SET    vasak = CASE
                     WHEN vasak >
p_parent_parem THEN vasak + 2
                     ELSE vasak
                   END,
           parem = CASE
                     WHEN parem
>= p_parent_parem THEN parem + 2
                     ELSE parem
                   END
    WHERE  parem >=
p_parent_parem;

    UPDATE nsets_osapool
    SET    vasak = vasak + 1,
           parem = parem + 1
    WHERE  vasak > p_parent_vasak
           AND parem <
p_parent_parem;

    INSERT INTO nsets_osapool
                (op_id,
                 op_telefon,
                 op_e_mail,
                 vasak,
                 parem)
    VALUES      (25001,
                 '111111',

'insert_organisatsioon@google.com
',
                  p_parent_vasak
+ 1,
                 ( p_parent_parem
+ 1 ));

       INSERT INTO
nsets_organisatsioon
                (org_id,
                 org_reg_kood,
                 org_nimetus)
      VALUES      (25001,
                 '1111111',
                 'Insert
Organisatsion');
END;
```

PostgreSQL

```sql
DO $$
DECLARE
    p_parent_parem INTEGER;
    p_parent_vasak INTEGER;
BEGIN
    SELECT "VASAK",
           "PAREM"
    INTO   p_parent_vasak,
p_parent_parem
    FROM   nested_sets."OSAPOOL"
    WHERE  "OP_ID" = 667;

    UPDATE nested_sets."OSAPOOL"
    SET "PAREM" = CASE
                       WHEN "PAREM" >=
p_parent_parem THEN "PAREM" + 2
                       ELSE "PAREM"
                     END
    WHERE  "PAREM" >= p_parent_parem;

    UPDATE nested_sets."OSAPOOL"
    SET    "VASAK" = CASE
                       WHEN "VASAK" >
p_parent_parem THEN "VASAK" + 2
                       ELSE "VASAK"
                     END
    WHERE  "PAREM" >= p_parent_parem;

    UPDATE nested_sets."OSAPOOL"
    SET    "VASAK" = "VASAK" + 1,
           "PAREM" = "PAREM" + 1
    WHERE  "VASAK" > p_parent_vasak
           AND "PAREM" < p_parent_parem;

    INSERT INTO nested_sets."OSAPOOL"
                ("OP_ID",
                 "OP_TELEFON",
                 "OP_E_MAIL",
                 "VASAK",
                 "PAREM")
    VALUES      (25001,
                 '111111',

'inset_organisatsioon@google.com',
                 p_parent_vasak + 1,
                 ( p_parent_parem + 1
));

    INSERT INTO
nested_sets."ORGANISATSIOON"
                ("ORG_ID",
                 "ORG_REG_KOOD",
                 "ORG_NIMETUS")
    VALUES      (25001,
                 '111111',
                 'Insert
Organisatsion');
END;
$$ LANGUAGE plpgsql;
```

	Oracle	PostgreSQL

I1 M	BEGIN
	INSERT INTO mpath_osapool
	(op_id,
	op_telefon,
	op_e_mail,
	teekond)
	VALUES (25001,
	'111111',
	'insert_organisatsioon@google.com',
	'1150.667.25001');
	
	INSERT INTO mpath_organisatsioon
	(org_id,
	org_reg_kood,
	org_nimetus)
	VALUES (25001,
	'1111111',
	'Insert Organisatsion');
	
	UPDATE mpath_osapool
	SET teekond = '1150.667.25001.' || substr(teekond, instr(teekond,'1150.667.')+length('1150.667.'))
	WHERE teekond LIKE '1150.667.%'
	AND op_id != 25001;
	END;
		DO $$
		BEGIN
		INSERT INTO "OSAPOOL"
		("OP_ID",
		"OP_TELEFON",
		"OP_E_MAIL",
		"TEEKOND")
		VALUES (25001,
		'111111',
		'inset_organisatsioon@google.com',
		'1150.667.25001');
		
		INSERT INTO "ORGANISATSIOON"
		("ORG_ID",
		"ORG_REG_KOOD",
		"ORG_NIMETUS")
		VALUES (25001,
		'1111111',
		'Insert Organisatsion');
		
		UPDATE "OSAPOOL"
		SET "TEEKOND" = text2ltree('1150.667.25001.' || substring(ltree2text("TEEKOND"), position('1150.667.' IN ltree2text("TEEKOND"))+length('1150.667.')))
		WHERE "TEEKOND" ~ '1150.667.*'
		AND "OP_ID" != 25001;
		END;
		$$ LANGUAGE plpgsql;

D1 A	BEGIN
	UPDATE adjacency_osapool
	SET ylemus_id = 1150
	WHERE ylemus_id = 667;
	
	DELETE FROM adjacency_osapool
	WHERE op_id = 667;
	END;
		DO $$
		BEGIN
		UPDATE "OSAPOOL"
		SET "YLEMUS_ID" = 1150
		WHERE "YLEMUS_ID" = 667;
		
		DELETE FROM "OSAPOOL"
		WHERE "OP_ID" = 667;
		END;
		$$ LANGUAGE plpgsql;

	Oracle	PostgreSQL

D1 N	DECLARE
	    p_parent_parem NUMBER(10);
	    p_parent_vasak NUMBER(10);
	    deleted_vasak NUMBER(10);
	    deleted_parem NUMBER(10);
	BEGIN
	    SELECT vasak,
	           parem
	    INTO p_parent_vasak,
	p_parent_parem
	    FROM nsets_osapool
	    WHERE op_id = 1150;
	
	    UPDATE nsets_osapool
	    SET vasak = CASE
	                     WHEN vasak >
	p_parent_parem THEN vasak - 2
	                     ELSE vasak
	                   END,
	           parem = CASE
	                     WHEN parem
	>= p_parent_parem THEN parem - 2
	                     ELSE parem
	                   END
	    WHERE parem >=
	p_parent_parem;
	
	    SELECT vasak,
	           parem
	    INTO deleted_vasak,
	deleted_parem
	    FROM nsets_osapool
	    WHERE op_id = 667;
	
	    UPDATE nsets_osapool
	    SET vasak = vasak - 1,
	           parem = parem - 1
	    WHERE vasak > deleted_vasak
	           AND parem <
	deleted_parem;
	
	    DELETE FROM nsets_osapool
	    WHERE op_id = 667;
	END;
		DO $$
		DECLARE
		    p_parent_parem INTEGER;
		    p_parent_vasak INTEGER;
		    deleted_vasak INTEGER;
		    deleted_parem INTEGER;
		BEGIN
		    SELECT "VASAK",
		           "PAREM"
		    INTO p_parent_vasak,
		p_parent_parem
		    FROM "OSAPOOL"
		    WHERE "OP_ID" = 1150;
		
		    UPDATE "OSAPOOL"
		    SET "VASAK" = CASE
		                       WHEN "VASAK" >
		p_parent_parem THEN "VASAK" - 2
		                       ELSE "VASAK"
		                     END
		    WHERE "PAREM" >= p_parent_parem;
		
		    UPDATE "OSAPOOL"
		    SET "PAREM" = CASE
		                       WHEN "PAREM" >=
		p_parent_parem THEN "PAREM" - 2
		                       ELSE "PAREM"
		                     END
		    WHERE "PAREM" >= p_parent_parem;
		
		    SELECT "VASAK",
		           "PAREM"
		    INTO deleted_vasak, deleted_parem
		    FROM "OSAPOOL"
		    WHERE "OP_ID" = 667;
		
		    UPDATE "OSAPOOL"
		    SET "VASAK" = "VASAK" - 1,
		           "PAREM" = "PAREM" - 1
		    WHERE "VASAK" > deleted_vasak
		           AND "PAREM" < deleted_parem;
		
		    DELETE FROM "OSAPOOL"
		    WHERE "OP_ID" = 667;
		END;
		$$ LANGUAGE plpgsql;

D1 M	BEGIN
	    UPDATE mpath_osapool
	    SET teekond =
	Replace(teekond, '.667', NULL)
	    WHERE teekond LIKE
	'%.667.%';
	
	    DELETE FROM mpath_osapool
	    WHERE op_id = 667;
	END;
		DO $$
		BEGIN
		    UPDATE "OSAPOOL"
		    SET "TEEKOND" =
		text2ltree(Replace(ltree2text("TEEKOND")
		, '.667', NULL))
		    WHERE "TEEKOND" ~ '*.667.*';
		
		    DELETE FROM "OSAPOOL"
		    WHERE "OP_ID" = 667;
		END;
		$$ LANGUAGE plpgsql;

4. Mõõtmiste tulemused

Selles jaotises esitatakse päringute ja andmemuudatuse operatsioonide kiiruse mõõtmised tulemused, andmemahud erinevate disainide korral ja lausete keerukuse mõõtmise tulemused (leitud koodiridade arvu metoodika alusel). Võimalikult hea ülevaatlikuse huvides esitatakse need koondtabelitena. Tulemuste analüüs ja järelduste tegemine toimub järgmises peatükis (peatükk 5).

Tabelis 4 esitatakse päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle ja PostgreSQL andmebaasisüsteemides. Kiiruse mõõtmise tulemused esitatakse sekundites. Veergudeks on operatsioonide identifikaatorid (nt S2N tähendab operatsiooni S2 pesastatud hulkade (ingl *nested sets*) disaini abil loodud tabelite põhjal). Ridadeks on andmebaasisüsteemid ja andmemahud (nt „25 000 Oracle" tähendab Oracle andmebaasis loodud tabeleid, kus on 25 000 rida testandmeid).

A – *adjacency list* (külgnevusnimistu)
N – *nested sets* (pesastatud hulgad)
M – *materialized path* (materialiseeritud tee)

Järgnevates tabelites on parema loetavuse huvides Oracle kohta käivad andmed sinisel taustal ja PostgreSQL kohta käivad andmed valgel taustal. Tabelites 4, 5 ja 6 on rasvaselt tähistatud tulemused, mis on PostgreSQL ja Oracle samatüübiliste mõõtmiste võrdluses paremad (kõik mõõdikud on praegu sellised, et väiksem tulemus on ka parem tulemus). See tõstab esile ühe või teise andmebaasisüsteemi paremuse.

Tabel 4. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle ja PostgreSQL andmebaasisüsteemides (sekundites)

	S1 A	**S1 N**	**S1 M**	**S2 A**	**S2 N**	**S2 M**	**I1 A**	**I1 N**	**I1 M**	**D1 A**	**D1 N**	**D1 M**
250 Oracle	0,038	0,029	0,039	0,071	0,110	0,061	0,035	0,075	0,027	0,030	0,049	0,050
250 PostgreSQL	**0,012**	**0,006**	**0,006**	**0,023**	**0,020**	**0,044**	**0,010**	**0,020**	**0,011**	**0,007**	**0,014**	**0,02**
9 000 Oracle	0,043	0,038	0,040	**4**	**0,930**	**4**	0,045	0,327	0,102	0,045	0,427	0,113
9 000 PostrgeSQL	**0,016**	**0,020**	**0,020**	9,397	0,99	24,73	**0,011**	**0,055**	**0,018**	**0,013**	**0,060**	**0,059**
25 000 Oracle	0,19	0,2	0,21	**0,185**	2	**10**	0,041	1	0,237	0,040	0,96	0,290
25 000 PostgreSQL	**0,090**	**0,033**	**0,050**	49	**1,43**	183,4	**0,016**	**0,771**	**0,024**	**0,020**	**0,770**	**0,110**

Järgnevalt (vt Tabel 5) on esitatud erinevate disainide korral mõõdetud andmemahud Oracle ja PostgreSQL andmebaasisüsteemides. Tulemused esitatakse megabaitides.

Tabel 5. Andmebaasi andmemahud erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides (MB)

	Külgnevusnimistu	**Pesastatud hulgad**	**Materialiseeritud tee**
250 Oracle	0,63	0,63	0,63
250 PostgreSQL	**0,16**	**0,16**	**0,19**
9 000 Oracle	2,44	2,50	4,37
9 000 PostgreSQL	**2,20**	**2,23**	**2,80**
25 000 Oracle	6,07	6,26	**7,26**
25 000 PostgreSQL	**5,90**	**5,98**	8,14

Järgnevalt (vt Tabel 6) on esitatud lausete keerukuse mõõtmise tulemused. Iga operatsiooni jaoks esitatakse füüsiliselt käivitatavate ridade arv.

Tabel 6. Koodiridade arv erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides

	S1 A	**S1 N**	**S1 M**	**S2 A**	**S2 N**	**S2 M**	**I1 A**	**I1 N**	**I1 M**	**D1 A**	**D1 N**	**D1 M**
SLOC-P Oracle	**26**	**15**	**12**	**10**	**12**	11	**22**	**43**	**23**	**7**	**34**	**7**
SLOC-P PostgreSQL	27	**15**	**12**	24	**12**	**9**	24	47	24	9	38	9

SLOC-P, Physical executable source lines of code – Füüsiliselt käivitatavad read.

Tabelites 7, 8 ja 9 on samad andmed kui tabelites 4, 5 ja 6. Tabelites 7 ja 8 tähistatakse rasvaselt mõõtmistulemused, mis on konkreetse ülesande korral ühes ja samas andmebaasisüsteemis ja samade andmemahtudega kõige paremad. Tabelis 9 tähistatakse rasvaselt mõõtmistulemused, mis on konkreetse ülesande korral ühes ja samas andmebaasisüsteemis kõige paremad. See tõstab esile disainide paremuse.

Tabel 7. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle ja PostgreSQL andmebaasisüsteemides (sekundites)

	S1 A	S1 N	S1 M	S2 A	S2 N	S2 M	I1 A	I1 N	I1 M	D1 A	D1 N	D1 M
250 Oracle	0,038	**0,029**	0,039	0,071	0,110	**0,061**	0,035	0,075	**0,027**	**0,030**	0,049	0,050
250 PostgreSQL	0,012	**0,006**	**0,006**	0,023	**0,020**	0,044	**0,010**	0,020	0,011	**0,007**	0,014	0,02
9 000 Oracle	0,043	**0,038**	0,040	4	**0,930**	4	**0,045**	0,327	0,102	**0,045**	0,427	0,113
9 000 PostrgeSQL	**0,016**	0,020	0,020	9,397	**0,99**	24,73	**0,011**	0,055	0,018	**0,013**	0,060	0,059
25 000 Oracle	**0,19**	0,2	0,21	**0,185**	2	10	**0,041**	1	0,237	**0,040**	0,96	0,290
25 000 PostgreSQL	0,090	**0,033**	0,050	49	**1,43**	183,4	**0,016**	0,771	0,024	**0,020**	0,770	0,110

Tabel 8. Andmebaasi andmemahud erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides (MB)

	Külgnevusnimistu	Pesastatud hulgad	Materialiseeritud tee
250 Oracle	**0,63**	**0,63**	**0,63**
250 PostgreSQL	**0,16**	**0,16**	0,19
9 000 Oracle	**2,44**	2,50	4,37
9 000 PostgreSQL	**2,20**	2,23	2,80
25 000 Oracle	**6,07**	6,26	7,26
25 000 PostgreSQL	**5,90**	5,98	8,14

Tabel 9. Koodiridade arv erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides

	S1 A	S1 N	S1 M	S2 A	S2 N	S2 M	I1 A	I1 N	I1 M	D1 A	D1 N	D1 M
SLOC-P Oracle	26	15	**12**	10	12	**11**	**22**	43	23	**7**	34	**7**
SLOC-P PostgreSQL	27	15	**12**	24	12	**9**	**24**	47	**24**	**9**	38	**9**

5. Tulemuste analüüs ja järeldused

Järgnevalt on esitatud saadud tulemuste analüüs ja tulemuste põhjal tehtud järeldused. Järgnevates jaotistes esitatakse ülesande püstituses (vt Jaotis 3.2) kirjutatud küsimused ning analüüsi käigus nendele küsimustele saadud vastused.

5.1. Päringute ja andmemuudatuse operatsioonide kiirused

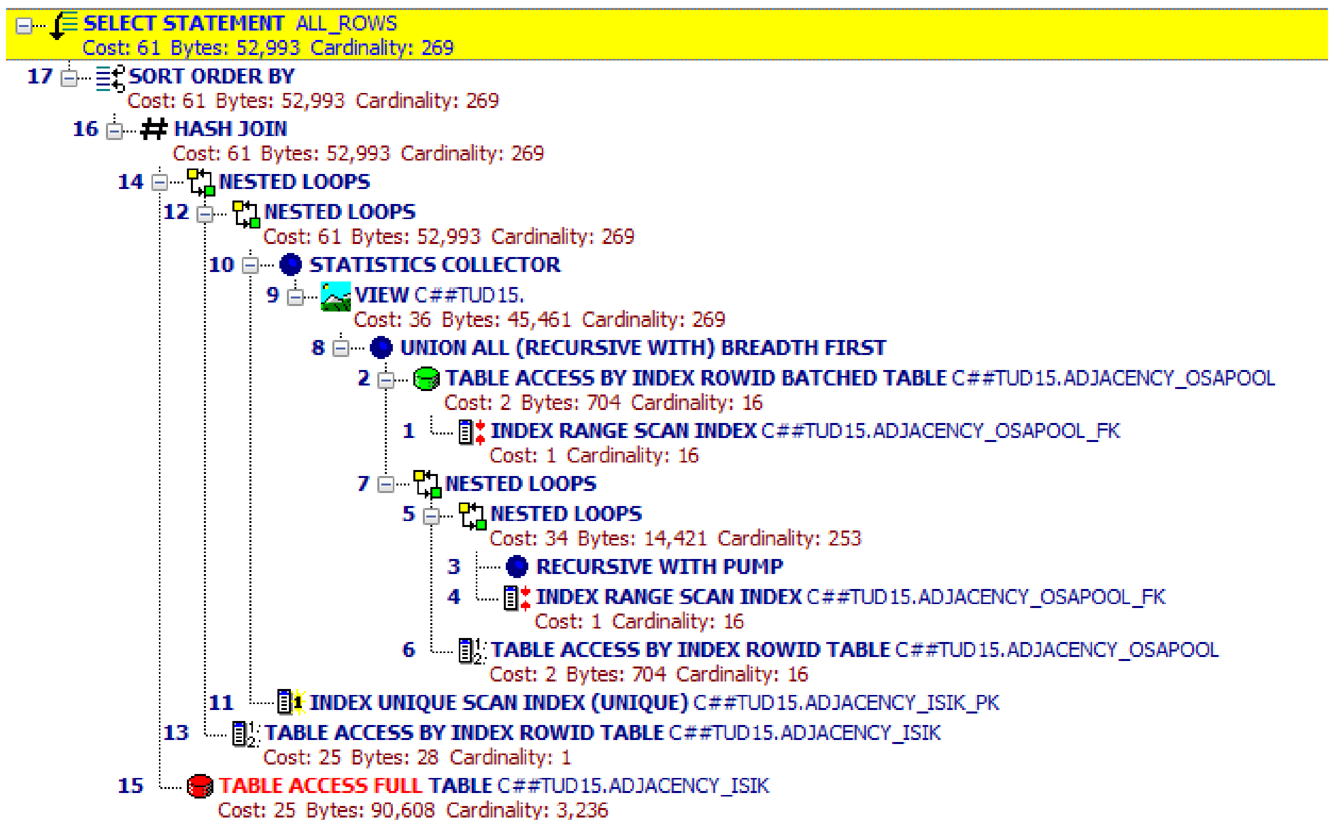
Kuidas erinevad päringute ja andmemuudatuste kiirused erinevates andmebaasisüsteemides ühe ja sama disaini korral ja sama andmemahuga? (vt Tabel 4)

Esimese päringu (S1) puhul on kõigi kolme disaini korral kiiruste erinevus PostgreSQL ja Oracle andmebaasisüsteemides suhteliselt väike (päringutele kuluv aeg on alla ühe sekundi). Kuid on siiski nähtav, et kõikide andmemahtude ja disainide korral täidetakse PostgreSQL andmebaasisüsteemis päring kiiremini kui Oracle andmebaasisüsteemis. Selleks, et aru saada, kust tulevad kiiruslikud erinevused, tuleb uurida päringu täitmisplaane. Täitmisplaan kirjeldab andmebaasisüsteemi poolt lause täitmiseks kasutatavat algoritmi. Kui kasutaja kirjutab SQL lause, siis selle alusel deklareerib ta süsteemile oma soovi. Selle soovi alusel ning arvestades andmebaasis olevate andmetega, peab andmebaasisüsteem koostama parima võimaliku protseduuri (algoritmi) selle soovi täitmiseks.

Uuritavaks andmemahuks on võetud 25 000 rida (vt Tabel 2) ja külgnevusnimistu disain, kuna sellel juhul on kiiruste erinevus kõige suurem.

Järgnevalt on esitatud S1A päringu (25 000 rida) täitmisplaan Oracle andmebaasisüsteemis.

```
  WITH r(op_id, ylemus_id, op_telefon, op_e_mail)
     AS (SELECT op_id,  ylemus_id, op_telefon, op_e_mail
 FROM   adjacency_osapool
  WHERE  ylemus_id = 10796
         UNION ALL
         SELECT o.op_id,  r.op_id, o.op_telefon,  o.op_e_mail
         FROM   adjacency_osapool o, r
         WHERE  r.op_id = o.ylemus_id)
SELECT is_id,   is_perenimi, is_eesnimi,   is_synniaeg, op_telefon,  op_e_mail
FROM    r,  adjacency_isik
WHERE  is_id = op_id  AND is_synniaeg >= To_date('01.01.1990', 'DD.MM.YYYY')
ORDER BY  is_perenimi, is_eesnimi, is_synniaeg;
```



Joonis 25. S1A päringu täitmisplaan Oracle andmebaasisüsteemis (25 000 rida)

Jooniselt 25 on näha, et andmebaasisüsteem kasutab rekursiivse päringu jaoks operatsiooni *UNION ALL (RECURSIVE) BREADTH WITH*. *VIEW* operaator tähendab, et andmebaasisüsteem „peatab“ operatsioonid ning loob vahetulemuse, käivitades *VIEW* operaatori all oleva alamplaani (Lewis 2009). Rida *STATISTICS COLLECTOR* tähendab, et täitmisplaani koostades tekitab andmebaasisüsteemi optimeerimismoodul (optimeerija) nii põhiplaani, kui ka alternatiivvariandid, mis algavad statistika kogumise punktist. See strateegia tähendab, et kui optimeerija poolt põhiplaani koostamisel kasutatud statistika osutub ebatäpseks, siis saab optimeerija statistika kogumise punkti järel valida alternatiivplaani (Männil 2014). Andmebaasisüsteem kasutab täitmisplaani koostamisel süsteemikataloogi talletatud andmebaasi statistikat. Nagu igasuguste andmetega võib ka statistikaga juhtuda, et see ei peegelda tegelikkust õieti. Valedel eeldustel koostatud täitmisplaani täitmine võib osutuda väga ebaefektiivseks (nt kulutada liiga palju aega). Alternatiivvariandid jätavad andmebaasisüsteemile võimaluse lause täitmiseks kasutatavat algoritmi jooksvalt parandada. Infot statistika paikapidavuse kohta saab andmebaasisüsteem seda lauset esmakordselt täites.

Seejärel ei vii andmebaasisüsteem otse läbi tabeli ja vaate ühendamist, vaid alguses sisemises tsüklis ühendab vaate tabeli *adjacency_isik* indeksiga *adjacency_isik_pk* ning välises tsüklis ühendab selle põhjal ülejäänud *adjacency_isik* tabeli. Viimasena tehakse räsiühendamine *adjacency_isik* tabeliga

(kusjuures *adjacency_isik* tabelile tehakse täisskaneering ning ei kasutata *is_synniaeg* veerule loodud indeksit). Tabeli täisskaneering (*full table scan* Oracle; *seq scan* – PostgreSQL) tähendab kõigi kasutuses olevate selle tabeli plokkide lugemist. Tuues analoogia raamatu lugemisega on see sama kui lugeda raamatu sisu osa, ilma indeksit kasutamata. Mõnikord on selline lugemine efektiivne, mõnikord mitte.

Järgnevalt on esitatud S1A päringu (25 000 rida) täitmisplaan PostgreSQL andmebaasisüsteemis.

```
WITH recursive r("OP_ID", "YLEMUS_ID", "OP_TELEFON", "OP_E_MAIL") AS
( SELECT "OP_ID", "YLEMUS_ID", "OP_TELEFON", "OP_E_MAIL"
       FROM   "OSAPOOL"
       WHERE  "YLEMUS_ID" = 10796
       UNION ALL
       SELECT "O"."OP_ID", "r"."OP_ID" , "O"."OP_TELEFON","O"."OP_E_MAIL"
       FROM   "OSAPOOL" "O", "r"
       WHERE  "r"."OP_ID" = "O"."YLEMUS_ID" )
SELECT "IS_ID", "IS_PERENIMI", "IS_EESNIMI", "IS_SYNNIAEG", "OP_TELEFON", "OP_E_MAIL"
FROM   "r", "ISIK"
WHERE  "IS_ID" = "OP_ID"
AND      "IS_SYNNIAEG" >= to_date( '01.01.1990', 'DD.MM.YYYY')
ORDER BY "IS_PERENIMI", "IS_EESNIMI", "IS_SYNNIAEG";
```

```
Sort  (cost=23201.59..23254.21 rows=21047 width=217)
  Sort Key: "ISIK"."IS PERENIMI", "ISIK"."IS EESNIMI", "ISIK"."IS SYNNIAEG"
  CTE r
    ->  Recursive Union  (cost=4.76..16429.25 rows=97171 width=41)
          ->  Bitmap Heap Scan on "OSAPOOL"  (cost=4.76..142.86 rows=61 width=41)
                Recheck Cond: ("PARENT ID" = 10796)
                ->  Bitmap Index Scan on "OSAPOOL FK"  (cost=0.00..4.74 rows=61 width=0)
                      Index Cond: ("PARENT ID" = 10796)
          ->  Hash Join  (cost=1018.45..1434.30 rows=9711 width=41)
                Hash Cond: (r 1."OP ID" = "O"."PARENT ID")
                ->  WorkTable Scan on r r 1  (cost=0.00..12.20 rows=610 width=4)
                ->  Hash  (cost=481.20..481.20 rows=25220 width=41)
                      ->  Seq Scan on "OSAPOOL" "O"  (cost=0.00..481.20 rows=25220 width=41)
  ->  Hash Join  (cost=266.81..3028.02 rows=21047 width=217)
        Hash Cond: (r."OP ID" = "ISIK"."IS ID")
        ->  CTE Scan on r  (cost=0.00..1943.42 rows=97171 width=200)
        ->  Hash  (cost=226.20..226.20 rows=3249 width=21)
              ->  Bitmap Heap Scan on "ISIK"  (cost=65.47..226.20 rows=3249 width=21)
                    Recheck Cond: ("IS SYNNIAEG" >= to date('01.01.1990'::text, 'DD.MM.YYYY'::text))
                    ->  Bitmap Index Scan on "IDX SYNNIAEG"  (cost=0.00..64.66 rows=3249 width=0)
                          Index Cond: ("IS SYNNIAEG" >= to date('01.01.1990'::text, 'DD.MM.YYYY'::text))
```

Joonis 26. S1A päringu täitmisplaan PostgreSQL andmebaasisüsteemis (25 000 rida)

Rekursiivse päringu tegemiseks kasutab optimeerija *Recursive Union* operaatorit. *Osapool* tabelist isiku otseste ja kaudsete alluvate leidmiseks kasutatakse rekursiivset seost realiseerivale välisvõtme veerule loodud indeksi *Bitmap index scan* operatsiooni abil lugemist kombineerituna sama tabeli erinevate ridade räsiväärtuste alusel ühendamisega (*Hash join* operatsioon). *Bitmap index scan* operatsioon tähendab, et andmebaasisüsteem loeb indeksist kõik viited ridadele, sorteerib need mälus

oleva „bitikaarti“ andmestruktuuri abil ja seejärel loeb ridu selles järjekorras nagu need on füüsiliselt kettale kirjutatud (seda teeb *Bitmap Heap Scan* operatsioon).

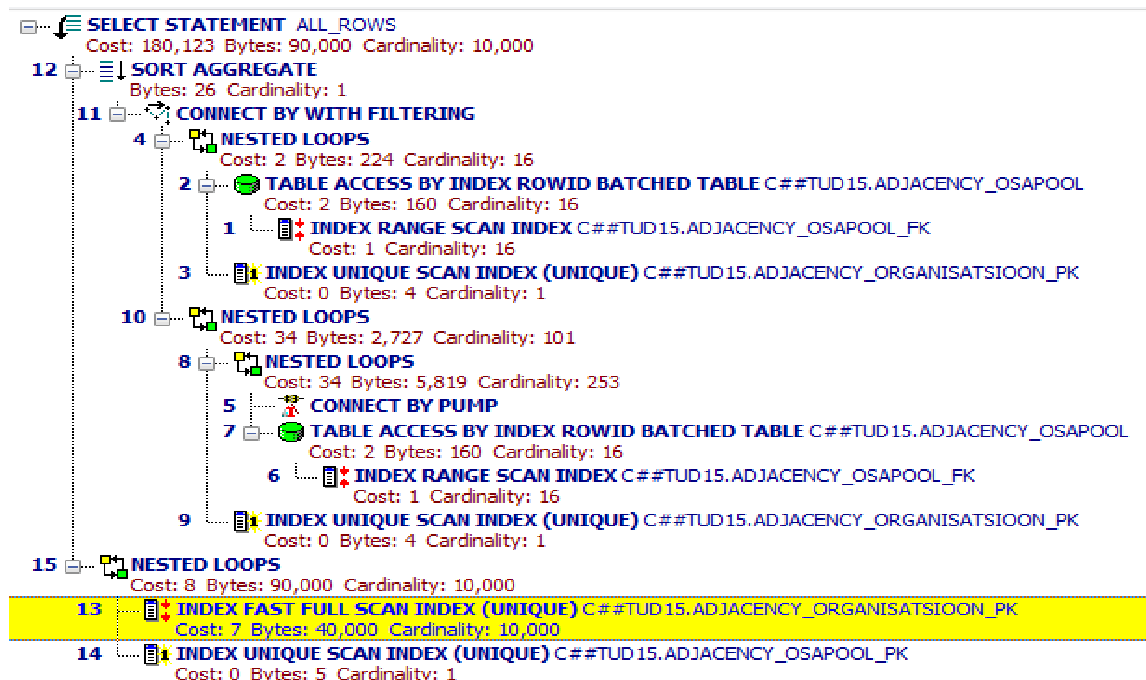
Põhipäringus ühendatakse rekursiivse tabeli avaldise (CTE) abil *Osapool* tabelist leitud read tabeli *Isik* ridadega kasutades räsiväärtuste alusel ühendamist. Tabelist *Isik* leitud ridadest, milles sünnikuupäev rahuldab tingimust (leitakse *Bitmap index scan + Bitmap Heap Scan* operatsioonide abil), moodustatakse mälus olev räsitabel, mis ühendatakse leitud ridadega tabelist *Osapool*.

Oracle andmebaasisüsteemis ei kasuta opetimeerija andmete lugemiseks *adjacency_isik* tabeli *is_synniaeg* veerule loodud indeksit. See *võib* olla põhjuseks, miks Oracle andmebaasisüsteemis täidetakse päring aeglasemalt kui PostgreSQL andmebaasisüsteemis. Oracles on päringu strateegia muutmiseks võimalik kasutada vihjeid, millega saab sundida optimeerija lause täitmisplaani koostamisel teatud kindlat otsust tegema. Antud juhul saab päringus kasutada vihjet */*+ index(adjacency_isik IDX_AD_SYNNIAEG) */*, mille puhul andmebaasisüsteem peab kasutama *adjacency_isik* tabeli *is_synniaeg* veerule loodud indeksit. Autor proovis seda vihjet S1A päringu korral ning selle tõttu täitmisplaan muutus. Andmebaasisüsteem hakkas vastavat indeksit kasutama, kuid päringu täitmise kiiruse seisukohalt midagi oluliselt ei muutunud. Muutus lugemisoperatsioonide arv ehk ilma indeksita on lugemisoperatsioonide arv (ingl *consistent gets*) 448, ning kasutades indeksit - 3565. Mida rohkem lugemisoperatsioone, seda rohkem võtab vastuse leidmine aega. Tõepoolest, indeks ei ole võluvits, mis alati ja kõik päringud kiiremaks teeb. Seega antud juhul tegi andmebaasisüsteem indeksi mitte kasutamisel õieti.

Kõige suuremad erinevused kiirustes on siiski teise päringu korral (S2) kõige suurema andmemahuga (25 000 rida). Külgnevusnimistu disaini korral täidetakse S2 päringut PostgreSQL andmebaasisüsteemis 49 sekundit, Oracle andmebaasisüsteemis annab sama päring tulemuse 0,185 sekundiga. Pesastatud hulkade disaini korral leitakse PostgreSQL andmebaasisüsteemis S2 päringu tulemus 1,43 sekundiga ning Oracle andmebaasisüsteemis kahe sekundiga. Materialiseeritud tee disaini korral leitakse PostgreSQL andmebaasisüsteemis S2 päringu tulemus 183,4 sekundiga ning Oracle andmebaasisüsteemis 10 sekundiga. Nagu on näha, siis kõige suurem erinevus on külgnevusnimsitu ja materialiseeritud tee disainide korral. Järgnevalt on esitatud nende päringute täitmisplaanid.

Järgnevalt on esitatud S2A päringu (25 000 rida) täitmisplaan Oracle andmebaasisüsteemis.

```
SELECT op1.op_id,
      (SELECT Count(org.org_id)
       FROM   adjacency_osapool op, adjacency_organisatsioon org
       WHERE  op.op_id = org.org_id START WITH op.ylemus_id = op1.op_id
       CONNECT BY PRIOR op.op_id = op.ylemus_id) AS org
FROM   adjacency_osapool op1, adjacency_organisatsioon
WHERE  op_id = org_id;
```



Joonis 27. S2A päringu täitmisplaan Oracle andmebaasisüsteemis (25 000 rida)

Joonisel 27 on näha, et andmebaasisüsteem ei tee ühelegi tabelile täisskaneeringut (võrreldav raamatu sisu otsast lõpuni läbilugemisega), vaid kasutab ära indekseid. Rekursiivse päringu tegemiseks kasutab optimeerija operaatori *CONNECT BY WITH FILTERING*, mille puhul andmed filtreeritakse (*START WITH op.ylemus_id = op1.op_id*) ja seejärel tehakse rekursiivne ühendamine. Tabelid *adjacency_organisatsioon* ja *adjacency_osapool* ühendatakse *NESTED LOOPS* algoritmi abil.

Järgnevalt on esitatud S2A päringu (25 000 rida) täitmisplaan PostgreSQL andmebaasisüsteemis.

```
WITH recursive r("OP_ID", "YLEMUS_ID", "PATH") AS
(SELECT "OP_ID", "YLEMUS_ID", cast (ARRAY["OP_ID"] AS VARCHAR(100)[])
       FROM   "OSAPOOL", "ORGANISATSIOON"
       WHERE  "OP_ID" = "ORG_ID"   AND    "YLEMUS_ID" IS NULL
       UNION ALL
       SELECT     "O"."OP_ID",  "r"."OP_ID" , cast("r"."PATH" || "O"."OP_ID"::VARCHAR AS
VARCHAR(100) [])
       FROM       "OSAPOOL" "O" inner join "r" ON "O"."YLEMUS_ID" = "r"."OP_ID"
       inner join "ORGANISATSIOON" ON   "ORG_ID"= "O"."OP_ID" )
```

```
SELECT "ORG_ID",  ( SELECT count(r."OP_ID") FROM r WHERE  "ORG_ID"::VARCHAR = ANY("PATH")
AND    "ORG_ID" != r."OP_ID")  FROM   "ORGANISATSIOON";
```

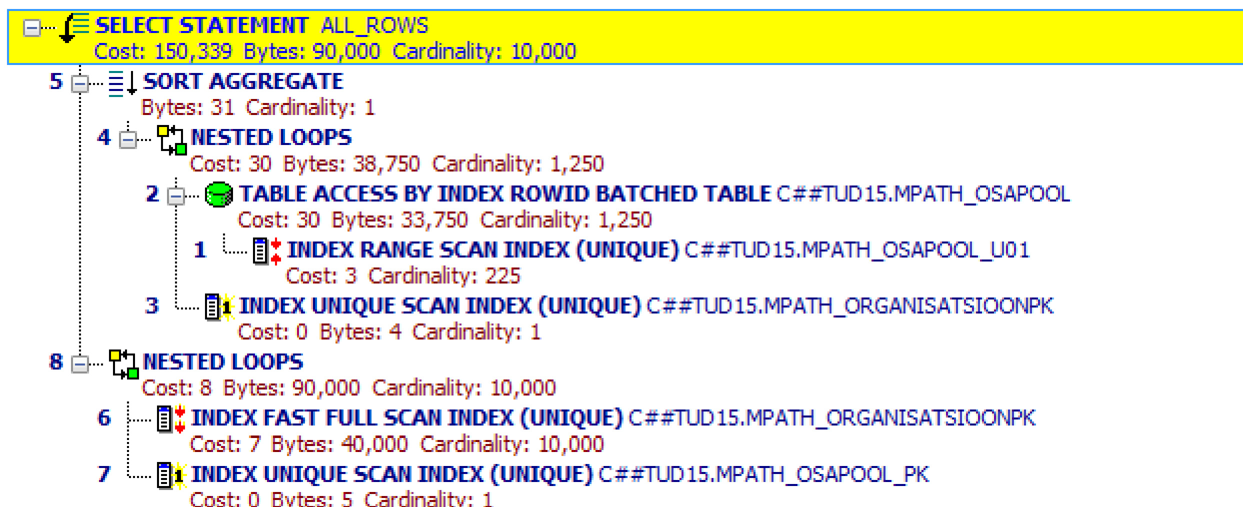
```
Seq Scan on "ORGANISATSIOON"  (cost=4886.56..258326.56 rows=10000 width=4)
  CTE r
    ->  Recursive Union  (cost=0.57..4886.56 rows=631 width=40)
          ->  Nested Loop  (cost=0.57..28.10 rows=1 width=8)
                ->  Index Scan using "OSAPOOL FK" on "OSAPOOL"  (cost=0.29..11.46 rows=2 width=8)
                      Index Cond: ("PARENT ID" IS NULL)
                ->  Index Only Scan using "ORGANISATSIOON pkey" on "ORGANISATSIOON" "ORGANISATSIOON 1"  (cost=0.29..8.30 rows=1 width=4)
                      Index Cond: ("ORG ID" = "OSAPOOL"."OP ID")
          ->  Nested Loop  (cost=0.57..484.58 rows=63 width=40)
                ->  Nested Loop  (cost=0.29..430.37 rows=158 width=40)
                      ->  WorkTable Scan on r  (cost=0.00..0.20 rows=10 width=36)
                      ->  Index Scan using "OSAPOOL FK" on "OSAPOOL" "O"  (cost=0.29..42.86 rows=16 width=8)
                            Index Cond: ("PARENT ID" = r."OP ID")
                ->  Index Only Scan using "ORGANISATSIOON pkey" on "ORGANISATSIOON" "ORGANISATSIOON 2"  (cost=0.29..0.32 rows=1 width=4)
                      Index Cond: ("ORG ID" = "O"."OP ID")
  SubPlan 2
    ->  Aggregate  (cost=25.32..25.33 rows=1 width=0)
          ->  CTE Scan on r r 1  (cost=0.00..25.24 rows=31 width=0)
                Filter: (("ORGANISATSIOON"."ORG ID" <> "OP ID") AND ((("ORGANISATSIOON"."ORG ID")::character varying)::text = ANY (("PATH")::text[])))
```

Joonis 28. S2A päringu täitmisplaan PostgreSQL andmebaasisüsteemis (25 000 rida)

Joonisel 28 on näha, et päringul on põhiosa ja alamosa *SubPlan2*, mida tuleb korduvalt käivitada (Explaining the unexplainable – part 4 2013). Põhiosas käiakse läbi tabeli *Osapool* indekseid ning ühendatakse ridu rekursiivselt. Seejärel käiakse läbi *Organisatsioon* tabel ning käivitatakse korduvalt alamplaani (SubPlan 2). Sellega on seletav antud päringu täitmiskiiruse halb tulemus.

Järgnevalt on esitatud S2M päringu (25 000 rida) täitmisplaan Oracle andmebaasisüsteemis.

```
SELECT mp.op_id,(SELECT Count(org_id)
         FROM   mpath_osapool mp2, mpath_organisatsioon
         WHERE  mp2.op_id = org_id
                AND mp2.teekond LIKE mp.teekond|| '.%') AS org
FROM   mpath_osapool mp, mpath_organisatsioon org
WHERE  org.org_id = mp.op_id;
```



Joonis 29. S2M päringu täitmisplaan Oracle andmebaasisüsteemis (25 000 rida)

S2M päring kasutab suhteliselt lihtsat lähenemist. Vaadates joonist 29, siis on näha, et alampäringu puhul ühendatakse tabelid *mpath_osapool* ja *mpath_organisatsioon* kasutades *NESTED LOOPS* algoritmi. Tabeli *mpath_organisatsioon* korral piisab andmebaasisüsteemil primaarvõtmele loodud indeksi lugemisest, kuid tabeli *mpath_osapool* korral tuleb lugeda ka tabeliplokke (*TABLE ACCESS BY INDEX ROWID*), sest alampäringus on tingimus, millega otsitakse osapoole alluvaid.

Põhipäring täidetakse ainult indeksite abil ning tabelite plokke ei loeta. Tabelid *mpath_organisatsioon* ja *mpath_osapool* ühendatakse *NESTED LOOPS* algoritmi kasutades.

Järgnevalt on esitatud S2M päringu (25 000 rida) täitmisplaan PostgreSQL andmebaasisüsteemis.

```
SELECT "mp"."OP_ID",  (SELECT Count("ORG_ID")
        FROM   "OSAPOOL" "mp2", "ORGANISATSIOON"
        WHERE  "ORG_ID" = "mp2"."OP_ID"
               AND "mp2"."TEEKOND"::text LIKE  "mp"."TEEKOND"::text || '.%') AS org
FROM   "OSAPOOL" "mp", "ORGANISATSIOON" "ORG_ID"
where "ORG_ID" = "OP_ID";
```

```
Hash Join  (cost=893.50..12233233.50 rows=10000 width=4)
  Hash Cond: ("ORG_ID"."ORG_ID" = mp."OP_ID")
  ->  Seq Scan on "ORGANISATSIOON" "ORG_ID"  (cost=0.00..165.00 rows=10000 width=4)
  ->  Hash  (cost=581.00..581.00 rows=25000 width=4)
        ->  Seq Scan on "OSAPOOL" mp  (cost=0.00..581.00 rows=25000 width=4)
  SubPlan 1
    ->  Aggregate  (cost=1223.19..1223.20 rows=1 width=0)
          ->  Hash Join  (cost=1020.06..1223.06 rows=50 width=0)
                Hash Cond: ("ORGANISATSIOON"."ORG_ID" = mp2."OP_ID")
                ->  Seq Scan on "ORGANISATSIOON"  (cost=0.00..165.00 rows=10000 width=4)
                ->  Hash  (cost=1018.50..1018.50 rows=125 width=4)
                      ->  Seq Scan on "OSAPOOL" mp2  (cost=0.00..1018.50 rows=125 width=4)
                            Filter: (("TEEKOND")::text ~~ (('%.'::text || (mp."OP_ID")::text) || '.%'::text))
```

Joonis 30. S2M päringu täitmisplaan PostgreSQL andmebaasisüsteemis (25 000 rida)

Jooniselt 30 on näha, et päringul on põhiosa ja üks alamosa *SubPlan1*, mida tuleb päringu käigus korduvalt käivitada. Põhiosas käiakse läbi tabel *Osapool* ja luuakse selle põhjal räsitabel. Seejärel käiakse järjest läbi tabel *Organisatsioon* ja tehakse üle vastavate võtmeveergude räsiühendamine. Nagu S1A päringu puhul, on selles päringus probleem alamplaanis (ehk alamplaani korduv käivitamisel kasvab selle päringule kuluv aeg) ning ka selles, et põhipäringus toimub tabelite *Organisatsioon* ja *Osapool* täisiskaneering.

Muutmisoperatsioonid (I1 ja D1) täidetakse mõlemas andmebaasisüsteemis kõikide disainide korral üsna kiiresti (maksimaalne aeg on üks sekund pesastatud hulkade disaini korral 25 000 reaga). Kuid üldjuhul täidetakse PostgreSQL andmebaasisüsteemis muutmisoperatsioonid natuke kiiremini.

Kuidas erinevad päringute ja andmemuudatuste kiirused ühes andmebaasisüsteemis erinevate disainide korral sama andmemahuga? (vt Tabel 7)

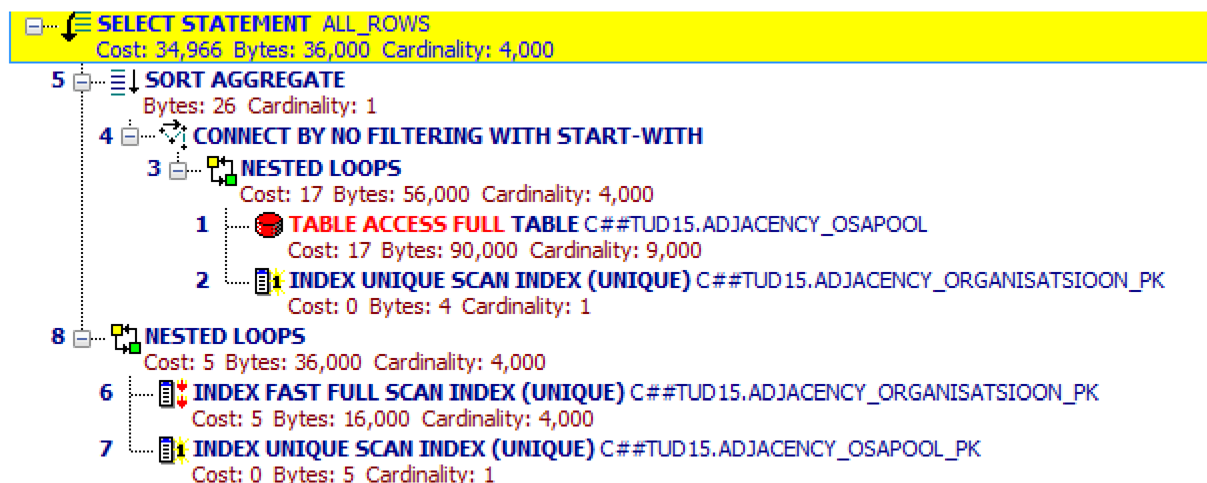
Oracle

Väikese andmemahuga (250 rida) on kõikide päringute ja operatsioonide korral tulemused üsna head ning kiiruste vahe ei ole suur ja eriti märgatav (kõik tulemused on alla ühe sekundi).

Probleemid hakkavad tekkima 9 000 reaga eksperimendis. Teise päringu korral (S2) näitasid külgnevusnimistu ja materialiseeritud tee disainid tulemust neli sekundit, samas pesastatud hulkade disaini korral oli tulemus 0,93 sekundit.

Järgnevalt on esitatud S2A päringu täitmisplaan 9 000 reaga eksperimendis Oracle andmebaasisüsteemis.

```
SELECT op1.op_id,
      (SELECT Count(org.org_id)
       FROM   adjacency_osapool op, adjacency_organisatsioon org
       WHERE  op.op_id = org.org_id START WITH op.ylemus_id = op1.op_id
       CONNECT BY PRIOR op.op_id = op.ylemus_id) AS org
FROM   adjacency_osapool op1, adjacency_organisatsioon
WHERE  op_id = org_id;
```

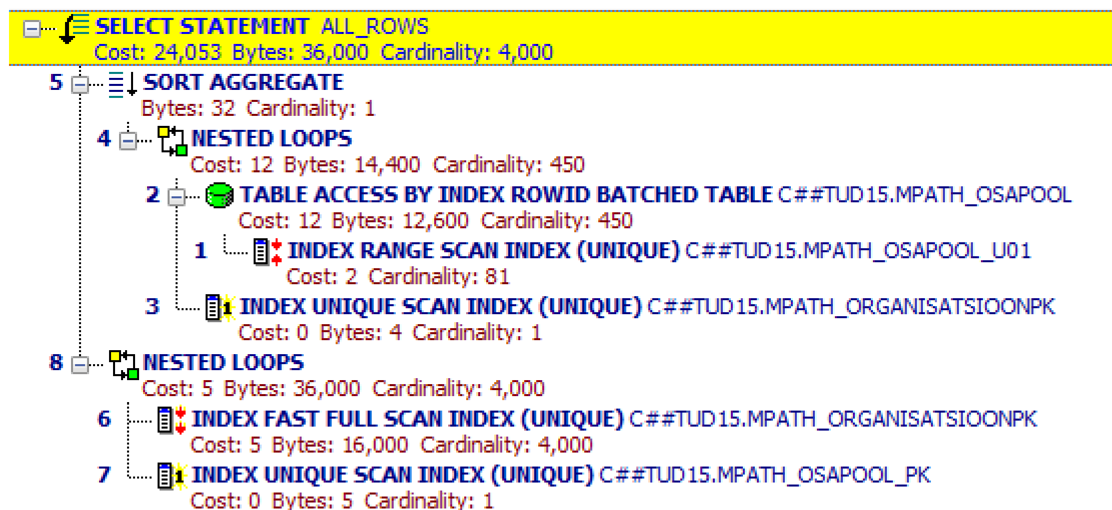


Joonis 31. S2A päringu täitmisplaan Oracle andmebaasisüsteemis (9 000 rida)

Joonisel 31 on näha, et 9 000 reaga eksperimendis kasutab andmebaasisüsteem rekursiivse päringu tegemiseks *CONNECT BY NO FILTERING WITH START-WITH* operaatorit, mille puhul filtreerimist ei tehta. Täpsemalt, tehakse täisskaneering tabelile *adjacency_osapool,* ühendatakse tulemus *adjacency_organisatsioon* tabeliga kasutades *NESTED LOOPS* algoritmi ning tabeli *adjacency_organisatsioon* primaarvõtme alusel automaatselt loodud indeksit, rakendatakse operaatorit *CONNECT BY NO FILTERING WITH START-WITH* ja alles siis pannakse peale filter *START WITH op.ylemus_id = op1.op_id.* 25 000 reaga eksperimendis otsustas andmebaasisüsteem kasutada *CONNECT BY WITH FILTERING* operaatorit. Seepärast on 9000 reaga eksperimendis tulemus halvem kui 25 000 reaga eksperimendis. Päringu kiiruse parandamiseks antud andmemahu korral võib kasutada päringus vihjet (ingl *hint*) */*+ CONNECT_BY_FILTERING */*. Kasutades seda muutus lause täitmisplaan ning lause täimise kiirus paranes oluliselt ehk uus tulemus on üks sekund (originaalpäringu tulemus oli neli sekundit).

Järgnevalt on esitatud S2M päringu täitmisplaan 9 000 reaga Oracle andmebaasisüsteemis.

```
SELECT mp.op_id,(SELECT Count(org_id)
        FROM   mpath_osapool mp2, mpath_organisatsioon
        WHERE  mp2.op_id = org_id
               AND mp2.teekond LIKE mp.teekond|| '.%') AS org
FROM   mpath_osapool mp, mpath_organisatsioon org
WHERE  org.org_id = mp.op_id;
```

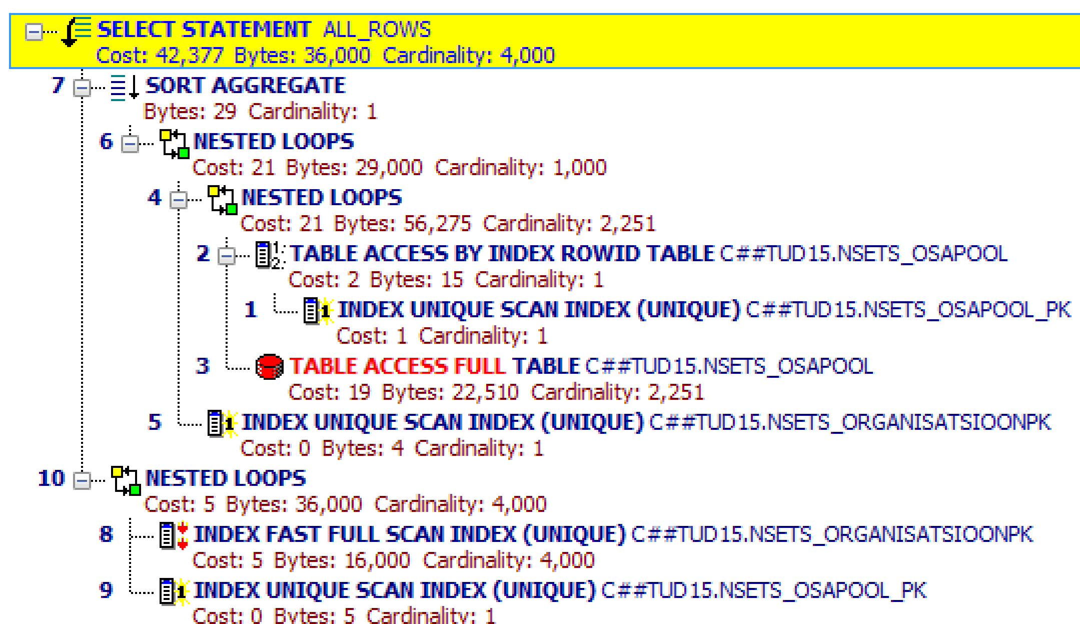


Joonis 32. S2M päringu täitmisplaan Oracle andmebaasisüsteemis (9 000 rida)

Joonisel 32 on näha, et ühelegi tabelile andmebaasisüsteem täisskaneeringut ei tee, vaid kasutab ära indekseid. Tegemist on üsna hea täitmisplaaniga, seega võib teha järelduse, et probleem on pigem teekonna stringi töötlemises (päringus kasutatakse *LIKE* operaatorit).

Järgnevalt on esitatud S2N päringu täitmisplaan 9 000 reaga eksperimendis Oracle andmebaasisüsteemis.

```
SELECT T4.op_id,
       (SELECT Count(T3.org_id)
        FROM   nsets_osapool T1, nsets_osapool T2, nsets_organisatsioon T3
        WHERE  T2.vasak > T1.vasak
               AND T2.vasak <= T1.parem
               AND T4.op_id = T1.op_id
               AND T3.org_id = T2.op_id) AS org
FROM   nsets_osapool T4, nsets_organisatsioon ORG
WHERE  T4.op_id = ORG.org_id;
```



Joonis 33. S2N päringu täitmisplaan Oracle andmebaasisüsteemis (9 000 rida)

Jooniselt 33 on näha, et alampäringus kasutab andmebaasisüsteem *nsets_osapool* tabeli puhul ühel juhul indeksit, ning teisel puhul teeb täisskaneeringut ning ühendab tulemuse kasutades *NESTED_LOOPS* algoritmi. Seejärel ühendab saadud tulemuse *nsets_organisatsioon* tabeliga kasutades ka *NESTED_LOOPS* algoritmi.

Kõige suurema andmemahuga (25 000 rida) hakkavad ilmuma kiiruslikud erinevused disainide vahel muutmisoperatsioonide I1 ja D1 puhul. I1 operatsiooni puhul on kõige halvem tulemus pesastatud hulkade disaini korral (üks sekund). Kõige halvema tulemuse andis pesastatud hulkade disain ka D1 operatsiooni korral (0,96 sekundit). See on tingitud sellest, et iga muudatuse tegemisel nõuab sõlmede ümberarvutamist, mis on kindlasti aeglane.

Kuid kõige suurem probleem antud andmemahu korral (25 000 rida) on päringuga S2. Oracle andmebaasis täidetakse S2 päring materialiseeritud tee disaini korral 10 sekundiga, mis on valitud kolmest disainist kõige halvem tulemus. Pesastatud hulkade disaini korral täidetakse päring kahe sekundiga, ning külgnevusnimistu disaini korral 0,185 sekundiga.

Vastavad täitmisplaanid on esitatud eespool (vt Joonis 27 ja Joonis 29). Külgnevusnimsitu disaini korral valib optimeerija *CONNECT BY WITH FILTERING* operaatori kasutamise, mis töötab üsna kiiresti. Lisaks sellele ei tee andmebaasisüsteem ühelegi tabelile täisskaneeringut. Materialiseeritud tee disaini korral on täitmisplaan korralik, kuid probleem on teekonna stringi töötlemises, mis tõstab päringule kuluvat aega. Pesastatud hulkade disaini korral on 25 000 reaga eksperimendis täitmisplaan sama mis 9 000 reaga eksperimendis (vt Joonis 33). Sellel joonisel on näha, et *nsets_osapool* tabelile tehakse täisskaneeringut ning kindlasti andmemahu kasvades kasvab ka selle päringule kuluv aeg.

PostgreSQL

Väikese andmemahuga (250 rida) on kõik päringute ja operatsioonide tulemused on üsna head ning suuri erinevuse kiiruste vahel ei esine (kõik tulemused on alla ühe sekundi).

Analoogselt Oracle andmebaasisüsteemiga hakkavad probleemid tekkima juba 9 000 reaga eksperimendis. S2 päringu korral leiti külgnevusnimistu disaini korral päringu tulemus 9,397 sekundiga ja materialiseeritud tee disaini korral 24,73 sekundiga. Nii 25 000 reaga eksperimendis kui ka 9 000 reaga eksperimendis on täitmisplaanid külgnevusnimistu ja materialiseeritud disaini korral samad. Need on esitatud eespool (vt Joonis 28 ja Joonis 30).

S2 päringu korral leidis PostgreSQL 9000 reaga eksperimendis kõige kiiremini S2N päringu tulemuse. Järgnevalt on esitatud selle päringu täitmisplaan.

```
SELECT "T4"."OP_ID",
       (SELECT Count("T3"."ORG_ID")
        FROM   "OSAPOOL" "T1", "OSAPOOL" "T2", "ORGANISATSIOON" "T3"
        WHERE  "T2"."VASAK" > "T1"."VASAK"
               AND "T2"."VASAK" <= "T1"."PAREM"
               AND "T4"."OP_ID" = "T1"."OP_ID"
               AND "T3"."ORG_ID" = "T2"."OP_ID") AS org
FROM   "OSAPOOL" "T4", "ORGANISATSIOON"
WHERE  "ORG_ID" = "OP_ID";
```

```
Hash Join  (cost=116.00..963989.50 rows=4000 width=4)
  Hash Cond: ("T4"."OP ID" = "ORGANISATSIOON"."ORG ID")
  ->  Seq Scan on "OSAPOOL" "T4"  (cost=0.00..176.00 rows=9000 width=4)
  ->  Hash  (cost=66.00..66.00 rows=4000 width=4)
        ->  Seq Scan on "ORGANISATSIOON"  (cost=0.00..66.00 rows=4000 width=4)
  SubPlan 1
    ->  Aggregate  (cost=240.89..240.90 rows=1 width=0)
          ->  Hash Join  (cost=154.34..239.78 rows=444 width=0)
                Hash Cond: ("T3"."ORG ID" = "T2"."OP ID")
                ->  Seq Scan on "ORGANISATSIOON" "T3"  (cost=0.00..66.00 rows=4000 width=4)
                ->  Hash  (cost=141.84..141.84 rows=1000 width=4)
                      ->  Nested Loop  (cost=22.82..141.84 rows=1000 width=4)
                            ->  Index Scan using "OSAPOOL pkey" on "OSAPOOL" "T1"  (cost=0.29..8.30 rows=1 width=8)
                                  Index Cond: ("T4"."OP ID" = "OP ID")
                            ->  Bitmap Heap Scan on "OSAPOOL" "T2"  (cost=22.54..123.53 rows=1000 width=8)
                                  Recheck Cond: (("VASAK" > "T1"."VASAK") AND ("VASAK" <= "T1"."PAREM"))
                                  ->  Bitmap Index Scan on "UNI VASAK PAREM"  (cost=0.00..22.29 rows=1000 width=0)
                                        Index Cond: (("VASAK" > "T1"."VASAK") AND ("VASAK" <= "T1"."PAREM"))
```

Joonis 34. S2N päringu täitmisplaan PostgreSQL andmebaasisüsteemis (9 000 rida)

Jooniselt 34 on näha, et päringul on põhiosa ja alamosa *SubPlan1*, mida tuleb päringu käigus korduvalt käivitada. Põhiosas käiakse läbi tabel *Organisatsioon* ja luuakse selle põhjal räsitabel. Seejärel käiakse järjest läbi tabel *Osapool* ja tehakse üle vastavate võtmeveergude räsiühendamine.

Alampäringus käiakse läbi tabeli *Osapool* indeksid mitmekordselt läbi (nagu oleks tegemist kahe eraldi tabeliga *t1* ja *t2*) ning ühendatakse need kasutades *NESTED LOOPS* algoritmi. Saadud tulemuste põhjal luuakse räsitabel ning tehakse räsiühendamine *Organisatsioon* tabeliga.

Ilmselt on nende täitmisplaanide puhul probleem alamplaanides (*SubPlan*), mida tuleb korduvalt käivitada. Külgnevusnimistu ja materialiseeritud tee disainide alusel loodud tabelite põhjal toimuvaid operatsioone teeb ka aeglasemaks teekonna stringi töötlemine (küglevusnimistu disaini korral luuakse teekonna string otse päringus *cast (ARRAY["OP_ID"] AS VARCHAR(100)[])* ning pealse seda kasutatakse saadud stringi töötlemiseks funktsiooni *ANY*, materialiseeritud tee disaini korral kasutatakse *LIKE* operaatorit). Pesastatud hulkade disaini korral stringi töötlemist ei toimu ja seega päring täidetakse kiiremini kui teiste disainide korral.

Kõige suurema andmemahuga (25 000 rida) hakkavad ka PostgreSQL andmebaasisüsteemis ilmnema kiiruslikud erinevused muutmisoperatsioonides I1 ja D1. Pesastatud hulkade disaini korral on näiteks I1 operatsiooni täitmise kiirus 0,771 sekundit (mis iseenesest ei ole halb tulemus), samas külgnevusnimistu disaini korral kulus täitmiseks 0,016 sekundit, mis on 48 korda kiirem kui pesastatud hulkade disaini korral. Ka D1 operatsiooni puhul oli pesastatud hulkade disainil kõige halvem tulemus.

See on tingitud sellest, et pesastatud disaini korral on iga muudatuse tegemisel vaja ümber arvutada terve puu (ka kustutamisel).

Kõige suurema andmemahu korral ilmnesid probleemid ka S2 päringuga. Külgnevusnimistu disaini korral kulus päringu täitmiseks 49 sekundit ning materialiseeritud tee disaini korral 183,4 sekundit. Kõige parem tulemus saadi pesastatud hulkade disaini korral, milleks oli 1,43 sekundit. Nende tulemuste põhjused on samad, mis 9 000 reaga eksperimendis. Esimeseks probleemiks on alamplaanide korduv käivitamine ning lisaks sellele teekonna stringide töötlemine.

Kuidas erinevad päringute ja andmemuudatuste kiirused ühes andmebaasisüsteemis sama disaini korral ja erinevate andmemahtudega?

Oracle

Tabel 10. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused Oracle andmebaasisüsteemis (sekundites)

	S1 A	S1 N	S1 M	S2 A	S2 N	S2 M	I1 A	I1 N	I1 M	D1 A	D1 N	D1 M
250 Oracle	0,038	0,029	0,039	0,071	0,110	0,061	0,035	0,075	0,027	0,030	0,049	0,050
9 000 Oracle	0,043	0,038	0,040	4	0,930	4	0,045	0,327	0,102	0,045	0,427	0,113
25 000 Oracle	0,19	0,2	0,21	0,185	2	10	0,041	1	0,237	0,040	0,96	0,290

Tabel 11. Täitmisplaanide kasutamine erinevate andmemahtude korral Oracle andmebaasisüsteemis

Operatsioon / Andmemaht	S1A	S1M	S1N	S2A	S2M	S2N
250 rida				+	+	
9000 rida		+	+	+	+	+
25 000 rida		+	+		+	+

Tabelis 10 tuuakse veelkord välja operatsioonide kiiruste mõõtmise tulemused Oracles. Tabelis 11 on välja toodud milliste andmemahtude korral kasutas Oracle andmebaasisüsteem ühesugust täitmisplaani ja milliste andmemahtude korral erinevat. Plussiga (+) on märgitud need andmemahud, mille korral andmebaasisüsteem kasutas ühesugust täitmisplaani. Näiteks S1A päringu puhul kasutas andmebaasisüsteem kõikide andmemahtude korral erinevat täimisplaani. S2M päringu korral kasutas andmebaasisüsteem kõikide andmemahtude korral ühesugust täitmisplaani. Tabel 11 on ühtlasi heaks illustratsiooniks sellele, et erinev andmete hulk võib tingida sama ülesande lahendamiseks erineva

algoritmi valiku. Selleks, et andmebaasisüsteem saaks sellist valikut teha peab andmebaasi statistika olema kooskõlas tegelikult andmebaasis olevate andmetega.

Esimese päringu (S1) puhul on kõikide disainide korral kiirused üsna stabiilsed, ehk andmemahu kasvamisel päringu kiirus ei kasva suurelt.

Teise päringu (S2) puhul on kõige suurem probleem materialiseeritud tee disaini (M) korral. 250 rea korral täidetakse päring 0,061 sekundiga, kuid 25 000 reaga võtab tulemuse leidmine juba 10 sekundit. Nagu on juba välja toodud eespool, siis on probleem teekonna stringi töötlemises.

S2 päringu puhul on külgnevusnimistu disaini (A) korral probleem 9 000 reaga – päringu tulemuse leidmine võtab sellel juhul neli sekundit. 25 000 reaga eksperimendis kulub päringu täitmiseks samas 0,185 sekundit. Kui vaadata täitmisplaane (vt Joonis 27 ja Joonis 31), siis on näha, et optimeerija kasutab erinevate andmemahtude korral erinevat strateegiat. 9 000 reaga eksperimendis kasutab andmebaasisüsteem *CONNECT BY WITH NO FILTERING WITH START-WITH*, mille puhul filtreerimist ei tehta ning teeb tabeli *Osapool* täisskaneeringut. 25 000 reaga eksperimendis suudab optimeerija leida paremat täitmisplaani ning kasutab juba *CONNECT BY WITH FILTERING* ning ei tee ühelegi tabelile täisskaneeringut. Vajadusel saab 9 000 reaga eksperimendis käsitsi strateegiat muuta, määrates päringus vihje */*+ CONNECT_BY_FILTERING */.*

Pesastatud hulkade disaini (N) korral suureneb S2 päringu puhul andmemahu kasvades ka päringule kuluv aeg ehk 250 reaga eksperimendis on vastav tulemus 0,110 sekundit ning 25 000 reaga eksperimendis on sama tulemus juba kaks sekundit. Selle päringu puhul teeb andmebaasisüsteem kõikide andmemahtude korral tabeli täisskaneeringut – seega andmemahu kasvades kasvab kindlasti ka päringule kuluv aeg.

Muutmisoperatsioonide (I1 ja D1) seisukohalt on kõige problemaatilisem pesastatud hulkade disain (N). 250 reaga eksperimendis on näiteks I1 operatsiooni puhul tulemus 0,075 sekundit, kuid 25 000 reaga see tulemus on juba üks sekund. Nagu on juba üleval välja toodud, et selle disaini probleem on sõlmede nummerdamises ehk iga muutmisel tuleb ümber arvutada terve puu sõlmede identifikaatorid. Teiste disainide korral ei ole andmemahu suurenedes kiiruslikud erinevused kuigi suured.

PostgreSQL

Tabel 12. Päringute ja andmemuudatuse operatsioonide kiiruse mõõtmise tulemused PostgreSQL andmebaasisüsteemis (sekundites)

	S1 A	S1 N	S1 M	S2 A	S2 N	S2 M	I1 A	I1 N	I1 M	D1 A	D1 N	D1 M
250 PostgreSQL	0,012	0,006	0,006	0,023	0,020	0,044	0,010	0,020	0,011	0,007	0,014	0,02
9 000 PostrgeSQL	0,016	0,020	0,020	9,397	0,99	24,73	0,011	0,055	0,018	0,013	0,060	0,059
25 000 PostgreSQL	0,090	0,033	0,050	49	1,43	183,4	0,016	0,771	0,024	0,020	0,770	0,110

Tabel 13. Tätmisplaanide kasutamine erinevate andmemahtude korral PostgreSQL andmebaasisüsteemis

Operatsioon / Andmemaht	S1A	S1M	S1N	S2A	S2M	S2N
250 rida	+	+	+	+	+	+
9000 rida	+	+	+	+	+	+
25 000 rida		+	+	+	+	

Tabelis 12 tuuakse veelkord välja operatsioonide kiiruste mõõtmise tulemused PostgreSQLis. Tabelis 13 on välja toodud milliste andmemahtude korral kasutas PostgreSQL andmebaasisüsteem ühesugust täitmisplaani ning milliste andmemahtude korral erinevat.

Esimese päringu (S1) puhul on kõikide disainide korral kiirused analoogselt Oracle andmebaasisüsteemiga stabiilsed ja andmemahu kasvamisel ei kasva päringu kiirus suurelt.

Kõige suuremad probleemid on teise päringuga (S2) külgnevusnimistu (A) ja materialiseeritud tee (M) disainide korral.

Külgnevusnimistu disaini (A) korral leitakse 25 000 rea korral päringu tulemus 49 sekundiga, kuid 250 reaga leitakse tulemus 0,023 sekundiga. Materialiseeritud tee disaini (M) korral leitakse 25 000 reaga päringu tulemus 183,4 sekundiga, kuid 250 rea korral 0,044 sekundiga. Pesastatud hulkade disaini (N) korral on olukord üsna stabiilne, ehk maksimaalne saadud tulemus 25 000 reaga on 1,43 sekundit, mis on võrreldes teiste disainidega hea tulemus. Eespool on juba välja toodud, et probleem on selles, et optimeerija kasutab alamplaane ning lisaks on aeglane ka teekonna stringi töötlemine.

Muutmisoperatsioonide korral (I1 ja D1) on probleem pesastatud hulkade disainiga (N). Andmemahu suurenedes kasvab ka väga kiiresti muutmisoperatsioonile kuluv aeg. Põhjused on juba eespool välja toodud, ehk pesastatud hulkade disaini korral võtab üsna palju aega puu ümberarvutamine.

Külgnevusnimistu ja materialiseeritud tee disaini näitasid nende operatsioonide korral head tulemust, ehk andmemahu suurenedes ei kasva operatsioonidele kuluv aeg väga suurelt

5.2. Koodi keerukus

Koodi keerukust hinnatakse antud eksperimendis katsetavate päringute ja operatsioonide lausete põhjal. Ühe ja sama ülesande saab SQLis enamasti lahendada mitmel erineval viisil. Seega kehtivad antud järeldused ainult selles töös katsetavate operatsioonide ja päringute lausete jaoks.

Milline on koodi keerukus samas andmebaasisüsteemis erinevate disainide korral?
Koodiridade arvu metoodika järgi on kõige lihtsamad päringud (S1 ja S2) materialiseeritud tee disaini korral nii Oracle kui ka PostgreSQL andmebaasisüsteemides (välja arvatud S2 päring – Oracle andmebaasisüsteemis on külgnevusnimistu disaini korral päringu keerukus 10 rida, mis on üks rida vähem, kui materialiseeritud tee disaini korral).

Kõige keerulisemad päringud (S1 ja S2) on külgnevusnimistu disaini korral (välja arvatud S2 päring – Oracle andmebaasisüsteemis on pesastatud hulkade disaini korral koodiridade arv kahe võrra suurem, võrreldes külgnevusnimistu disainiga). Põhjus on selles, et külgnevusnimistu disaini korral tuleb kasutada küllaltki keerukaid rekursiivseid päringuid.

Muutmisoperatsioonid I1 ja D1 on mõlemas andmebaasisüsteemis kõige keerulisemad pesastatud hulkade disaini korral. Põhjus on selles, et andmemuudatuse tulemusel on vaja hierarhia ümber arvutada. Külgnevusnimistu ja materialiseeritud tee disaini korral on muutmisoperatsioonide koodi keerukus peaaegu sama (paar rida vahet).

Kas sama ülesande lahendamiseks mõeldud koodi keerukus on erinevates andmebaasisüsteemides ühesugune?
Sama ülesannete lahendamiseks mõeldud koodi keerukus on nii Oracle kuid ka PostgreSQL andmebaasisüsteemides peaaegu sama (vahe on paar rida). Erinevus on tingutud sellest, et PostgreSQL andmebaasisüsteemis on muutmisoperatsioonid realiseeritud anonüümse plokina (sõnade DO ja END vahel).

Kõige suurem erinevus on teise päringu puhul külgnevusnimistu disaini korral (S2 A). Põhjuseks on, et PostgreSQL ei võimalda vaikimisi kasutada *CONNECT BY* konstruktsiooni (installeerides mooduli *tablefunc* on siiski võimalik), ning selle asemel tuleb kasutada ühist tabeli avaldist, mis on koodiridade arvu seisukohalt keerulisem.

5.3. Hierarhiaga seotud kitsenduste deklaratiivne jõustamine

Tabelis 14 on välja toodud kõik jaotises 1.1.2 kirjeldatud hierarhiatega seotud kitsendused ning iga ühe juures on esitatud hinnang, kas seda saab vastavas disainis deklaratiivselt (ilma kasutaja-defineeritud funktsioonide ja alampäringuteta) jõustada või mitte.

Kuna kitsenduste deklaratiivse esitamise võimalused (ilma kasutaja-defineeritud funktsioonide ja alampäringuteta) on nii Oracle kuid ka PostgreSQL andmebaasisüsteemides samad, siis on tulemused esitatud ühe tabelina. Autor märgib siinjuures, et kui andmebaasisüsteemid võimaldaksid tabeliga seotud CHECK kitsendustes kasutada alampäringuid ja/või luua üldiseid kitsendusi (ingl ASSERTION), siis oleks võimalik kõiki neid kitsendusi deklaratiivselt esitada. Seega pole küsimus selles nagu oleks mõni disainidest loomu poolest selline, et seal ei ole võimalik kitsendusi jõustada, vaid küsimus on tänapäeva andmebaasisüsteemide piiratuses ja selles, milline disain võimaldab selliste piirangute tingimustes „maksimumi võtta“ ja ikkagi võimalikult palju kitsendusi võimalikult lihtsalt jõustada.

Tabel 14. Võrdlustabel deklaratiivsete kitsenduste kohta erinevate disainide korral Oracle ja PostgreSQL andmebaasisüsteemides

Disain / Kitsendus	Külgnevusnimistu	Pesastatud hulgad	Materialiseeritud tee
{Irreflexive}	Jah	Jah	Jah
{Assymmetric}	Ei	Ei	Jah
{Antisymmetric}	Ei	Ei	Ei
{Acyclic}	Ei	Ei	Jah
{Intransitive}	Ei	Ei	Ei

Tabelis 15 on esitatud iga ülaltoodud tabelis oleva JAH kohta CHECK kitsendus eksperimendi andmebaasi näitel (Oracle Database).

Tabel 15. CHECK kitsendused erinevate disainide korral eksperimendi Oracle andmebaasi näitel

<table>
<tr><th>Disain ja kitsendus</th><th>CHECK kitsendus (kood)</th></tr>
<tr><td>Külvnevusnimistu {Irreflexive}</td><td>CHECK (OP_ID != YLEMUS_ID)</td></tr>
<tr><td>Materialiseeritud tee {Irreflexive}</td><td>CHECK (TEEKOND NOT LIKE OP_ID || '.' || OP_ID)</td></tr>
<tr><td>Pesastatud hulgad {Irreflexive}</td><td>CHECK (VASAK < PAREM)</td></tr>
<tr><td>Materialiseeritud tee {Asymetric}</td><td>CHECK (TEEKOND NOT LIKE '%.' || OP_ID || '.%')</td></tr>
<tr><td>Materialiseeritud tee {Acyclic}</td><td>CHECK (TEEKOND NOT LIKE '%.' || OP_ID || '.%'
AND TEEKOND NOT LIKE OP_ID || '.' || OP_ID)</td></tr>
</table>

5.4. Andmete salvestamiseks kulunud salvestusruum

Tabelitest 5 ja 8 on näha, et nii Oracle kuid ka PostgreSQL andmebaasisüsteemides osutus andmemahult kõige väiksemaks külgnevusnimistu disain. Teiseks tuli mõlemas andmebaasisüsteemides pesastatud hulkade disain ning viimaseks jäi materialiseeritud tee disain.

Külgnevusnimistu disaini korral on tabelis *Osapool* välisvõti ja temasse kuuluv veerg on *NUMBER(10)* tüüpi (PostgreSQL andmebaasis *INTEGER* tüüpi). Pesastatud hulkade disaini korral on välisvõtme asemel kaks *NUMBER(10)* tüüpi veergu (PostgreSQL andmebaasis *INTEGER* tüüpi). Materialiseeritud tee disaini korral kasutatakse ühte veergu, mis on *VARCHAR2* tüüpi (PostgreSQLis *ltree* tüüpi).

Kui võrrelda andmemahtude alusel andmebaasisüsteeme, siis peaaegu alati oli väiksem andmemaht PostgreSQL korral v.a materjaliseeeritud tee disain 25000 reaga, kui Oracle andmemaht oli väiksem.

Järgnevalt (vt Tabel 16 ja Tabel 17) on esitatud kui palju salvestusruumi kulutavad Oracle (Database concepts) ja PosgtreSQL (PosgtreSQL 9.1.14 Documentation e) andmebaasisüsteemid erinevate tüüpide andmete salvestamiseks.

Tabel 16. Andmete salvestamiseks kuluv salvestusruum erinevate tüüpide korral PostgreSQL andmebaasisüsteemis

Andmetüüp	**Salvestusruum (baitides)**
Integer	4
Date	4
Varchar(n)	Muutuv pikkus (sõltub märkidest, märkide arvust ja kasutatavast kodeeringust)

Tabel 17. Andmete salvestamiseks kuluv salvestusruum erinevate tüüpide korral Oracle andmebaasisüsteemis

Andmetüüp	**Salvestusruum (baitides)**	
Number	ROUND((length(p)+s)/2))+1	*p* - vastav arv, *s* - 0, kui *p* on positiivne arv, -1, kui *p* on negatiivne arv
Date	7	
Varchar2(n)	Muutuv pikkus (sõltub märkidest, märkide arvust ja kasutatavast kodeeringust)	

5.5. Järeldused

Vastavalt ülaltoodud analüüsile, võib teha järelduse, et ei eksisteeri absoluutselt parimat disaini, vaid disaini valik sõltub konkreetsest kontekstist ja määratud nõudmistest. Kuna väikese andmemahu korral on valitud disainid töökiiruse seisukohalt peaaegu võrdsed, siis tehakse järgmised järeldused kõige suurema andmemahu põhjal.

Koodi keerukuse seisukohalt on üldjuhul kõige lihtsamad päringud ja operatsioonid materialiseeritud tee disaini korral. Samas materialiseeritud tee disaini kasutades loodud tabelid võtavad nii Oracle kuid ka PostgreSQL andmebaasides võrreldes teiste disainidega kõige rohkem ruumi. See on loogiline, sest iga olemi korral registreeritakse tema tee juureni.

Suhteliselt häid tulemusi näitas materialiseeritud tee disain muutmisoperatsioonide korral mõlemas andmebaasisüsteemis kõikide andmemahtude korral. Kuid keerulise koondandmete päringu täitmine osutus selle disaini korral liiga aeglaseks.

Veel üheks positiivseks näitajaks materialiseeritud tee disaini korral on, et see võimaldab deklaratiivselt esitada suhteliselt palju hierarhiatega seotud kitsendusi (nii irreflektiivsuse kuid ka asümmeetria kitsendusi).

Suure andmemahuga tekkivad külgnevusnimistu disaini korral PostgreSQL andmebaasisüsteemis probleemid koondpäringuga S2. Oracle andmebaasisüsteemis suudab andmebaasisüsteemi optimeerimismoodul sellele küllaltki hea täitmisplaani leida ning lause täitmine on üsna kiire. Muutmisoperatsioonide korral näitas külgnevusnimistu disain mõlemas andmebaasisüsteemis kõige paremat töökiirust. Samuti võtab külgnevusnimistu disaini alusel loodud andmebaas kõige vähem salvestusruumi. Teisalt võimaldab külgnevusnimistu realiseerida deklaratiivselt ainult irreflektiivsuse kitsendust.

Pesastatud hulkade disain näitas mõlemas andmebaasisüsteemis kõige paremat tulemust keerulise koondpäringu täitmise kiiruse osas. Kuid muutmisoperatsioonide korral on pesastatud hulkade disaini korral kõige halvem tulemus. Lisaks on koodi keerukuse seisukohalt muutmisoperatsioonid kõige keerulisemad just pesastatud hulkade disaini korral. Kahte viimast tähelepanekut saab selgitada sellega, et muudatuse tulemus tuleb sõlmede vasak-parem paarid ümber arvutada. Pesastatud hulkade

disaini alusel loodavad tabelid võtavad natuke rohkem ruumi kui küglevusnimistu, kuid vähem kui materialiseeritud tee disaini alusel loodud tabelid.

Üldjuhul töötavad antud eksperimendis katsetatavad päringud ja operatsioonid PosgtreSQL andmebaasisüteemis kiiremini, kui Oracle andmebaasisüsteemis (v.a päring S2A, S2M, kus kasutatakse alampäringuid). Koodi keerukuse seisukohalt on PostgreSQL andmebaasisüsteemis katsetatavad päringud ja operatsioonid keerulisemad, kuid üldjuhul kulub kõikide disainide puhul andmete salvestamiseks PosgtreSQL andmebaasisüsteemis vähem ruumi, kui Oracle andmebaasisüsteemis (v.a materialiseeritud tee disain kõige suurema andmemahu korral).

Kokkuvõtteks võib öelda, et kui disaini valikul on kõige olulisemaks kriteeriumiks näiteks päringute kiirus ning muutmisoperatsioone (andmete lisamine, muutmine ja kustutamine) tehakse andmebaasis väga harva, siis sobib kõige paremini pesastatud hulkade disain. Selline disain sobib hästi näiteks andmeaida jaoks, sest ka andmete suuruse seisukohalt on see parem kui materialiseeritud tee disain (kuid veidi kehvem külgnevusnimistu disainist). Andmeaitades on teatavasti väga suured andmemahud.

Kui on oluline, et muutmisoperatsioonid töötaksid kiiresti, siis kõige parem valik on külgnevusnimistu disain. Operatiivandmete andmebaasides otsitakse ja muudetakse andmeid üksikute olemite kohta. See disain võiks taolise andmebaasi jaoks sobida. Samas tuleb sellise disaini valimisel valida ka andmebaasisüsteem, mis toetab rekursiivsete päringute tegemist (nii PostgreSQL kui Oracle viimased versioonid sobivad).

Kui on oluline, et disaini alusel loodav andmebaas võtaks võimalikult vähe ruumi, siis tuleb valida külgnevusnimistu disain. Kui olulisemaks kriteeriumiks on võimalikult lihtne kood, siis tuleb valida materialiseeritud tee disain.

Jaotises 3.1 on nimetatud analoogsed uuringud. Otseselt ei saa antud eksperimendis saadud tulemusi nimetatud uuringute tulemustega võrrelda, kuna tegemist on erinevate andmebaasisüsteemidega (või versioonidega) ning ülaltoodud uuringutes pole selgelt ja täpselt defineeritud, milliseid päringuid ja operatsioone tehti. Kuid siiski üldpilt näitab, et sama disain käitub erinevates andmebaasisüsteemides võrreldes teiste disainidega sarnaselt.

Näiteks Stadnik (2008) jõudis oma eksperimendis samuti järelduseni, et pesastatud hulkade disain ei sobi selliste hierarhiate jaoks, kus tihti tehakse muudatusi (lisamine ja kustutamine). Materialiseeritud tee disain näitas analoogselt käesoleva töö eksperimendiga head tulemust lisamise ja kustutamise operatsioonides. Päringute tulemusi on raske hinnata ja võrrelda, kuna pole täpsustatud milliseid andmeid täpselt küsiti.

Smusenok, Trubilko ja Furmanov (2013) jõudsid oma eksperimendis järelduseni, et kui on oluline, et päringud töötaksid võimalikult kiiresti, sobib kõige paremini pesastatud hulkade disain. Külgnevusnimistu kasutamine sobib nende arvates siis, kui hierarhias tehakse tihti muudatusi. Käesoleva eksperimendi tulemused näitasid samuti, et külgnevusnimistu disain on tõhus muutmisoperatsioonide korral. Samas pesastatud hulkade disain näitas kõige paremat tulemust keerulise koondpäringu täitmise kiiruse osas.

Atanassov (2007) keskendus oma eksperimendis sellele, kuidas mõjutab indeksite kasutamine ning teekonna veerus eraldajate kasutamine päringu kiirust materialiseeritud tee disaini korral. Nimetatud eksperimendi käigus saadud tulemusi ei saa võrrelda selle magistritöö eksperimendi tulemustega.

Kokkuvõte

Hierarhiad on igal pool meie ümber. Hierarhilise struktuuriga andmeid on vaja kindlasti andmebaasides säilitada. 2014. aasta sügise seisuga on andmebaasisüsteemidest kõige populaarsemad SQL-andmebaasisüsteemid, mille üldnimi tuleneb sellest, et neis kasutatakse SQL andmebaasikeelt (DB-Engines Ranking 2014). Hierarhilise struktuuriga andmete töötlemiseks on nendes andmebaasisüsteemides piiratud hulk operaatoreid ning nende abil andmete töötlemine nõuab kasutajalt suhteliselt keerulise koodi kirjutamist. Tänapäevaks on välja mõeldud palju erinevaid viise, kuidas hierarhilisi andmeid SQL-andmebaasides säilitada, kuid eksperimente, kus neid viise erinevates aspektides võrreldakse, on vähe.

Käesoleva magistritöö põhieesmärgiks oli võrrelda SQL-andmebaasides hierarhiliste andmete esitamise disainilahendusi, keskendudes lahenduste uurimise põhjalikkuse huvides võimalike disainilahenduste ühele pärisalamhulgale. Töö esimeses osas pandi struktureeritult kirja hierarhiliste andmete esitamiseks mõeldud disainilahendused, mida autor leidis raamatutest ja teistest allikatest. Autor pani kirja disainilahendused, mis näevad ette hierarhiliste andmete „lahti lõhkumise“ erinevatesse tabelitesse ja väljadesse ning jättis vaatluse alt välja disainid, mis näevad ette hierarhiate (nii seoste kui seotes osalevate olemite andmete) salvestamist andmeväärtustena (nt XML või JSON dokumentidena). Kuna autor ei tea ühtegi allikat, kus kõiki neid disainlahendusi oleks üheskoos ja struktureeritud viisil kirjeldatud, siis võib ka seda lugeda üheks käesoleva töö kõrvaltulemuseks. Töös uuritavateks disainilahendusteks valiti Google otsingumootori andmete alusel populaarseimad disainid, milleks on küglevusnimistu, pesastatud hulgad ja materialiseeritud tee. Magistritöö teises osas projekteeriti ja loodi eksperimendi andmebaasid ning kirjeldati eksperimentide sisu. Andmebaasid realiseeriti PostgreSQL ja Oracle andmebaasisüsteemides. Hierarhiliste andmete genereerimiseks kõikide disainilahenduste jaoks kasutati olemasolevat testandmete generaatorit kombinatsioonis enda loodud PL/SQL keele skriptidega. Eksperimendi käigus viidi läbi lugemise, muutmise ja kustutamise operatsioone. Töö kolmandas osas võrreldi operatsioonide täitmise kiiruseid, disainilahenduste alusel loodavate andmebaaside andmemahtu, kitsenduste deklaratiivse jõustamise võimalusi ning hierarhiliste andmete lugemise ja muutmise lausete keerukust. Selleks, et aru saada, millest tulenevad päringute täitmisel kiiruslikud erinevused, uuriti lausete täitmisplaane.

Eksperimendi tulemused näitasid (oodatult), et ei eksisteeri absoluutselt parimat disainilahendust. Disainilahenduse valik sõltub konkreetsekt kontekstist e olukorrast. Valikut mõjutavad kontekstist tulenevad nõudmised ja valikukriteeriumite olulisus. Seega enne valiku tegemist tuleb selgelt läbi

mõelda, mis on tähtis ja kriitiline antud ülesande korral ning mis on vähem oluline. Järgnevalt nimetatakse vaid uuringus käsitletud disainlahendusi. Näiteks, kui on tähtis, et päringud töötaksid väga kiiresti, siis tuleb valida pesastatud hulkade disaini. Kui hierarhias tehakse tihti muudatusi, siis pesastatud hulkade disain kõige paremini ei sobi, vaid parem oleks valida külgnevusnimistu disain. Andmete lugemise ja muutmise koodi lihtsuse seisukohalt on kõige mõistlikum kasutada materialiseeritud tee disaini. Kuid näiteks andmebaasi ruumikasutuse seisukohalt see on kõige halvem valik. Kui on oluline, et disaini alusel loodav andmebaas võtaks võimalikult vähe ruumi, siis sobib kõige rohkem külgnevusnimistu disain. Erinevaid variante võib nimetada palju, kuid on selge, et andmebaasi kavandajad peavad ise enne disainilahenduse valikut määrata kriteeriumite suhtelise olulisuse.

Kui võrrelda omavahel Oracle ja PostgreSQL andmebaasisüsteeme, siis üldjuhul töötavad antud eksperimendis katsetatavad päringud ja operatsioonid kiiremini PostgreSQL andmebaasisüsteemis, kuid koodi keerukuse seisukohalt on need keerulisemad. Keerulise koondpäringu (kus kasutatakse alampäringuid) korral ei suuda PostgreSQL andmebaasisüsteemi optimeerimismoodul leida head täitmisplaani ning selle lause täitmine on üsna aeglane. PostgreSQL andmebaasisüsteemis kulub kõikide disainide salvestamiseks üldjuhul vähem ruumi (v.a materialiseeritud tee disain kõige suurema andmemahu korral), kui Oracle andmebaasisüsteemis.

Kuna antud töös uuriti ainult kolme disaini, siis on kindlasti võimalik eksperimenti edasi arendada ja laiendada. Üks võimalik suund on näiteks uurida teisi disaine kasutades sama eksperimenti. Teiseks võimaluseks on laiendada skoopi ehk näiteks uurida kuidas mõjutab päringute ja operatsioonide kiirust indeksite ning muude töökiirust parandavate meetmete (näiteks ridade füüsilisel tasemel sorteerimise) kasutamine erinevate disaini korral ja erinevates andmebaasisüsteemides. Skoopi võib ka laiendada kaasates teistel andmemudelitel põhinevaid andmebaasisüsteeme. Puud on graafide erijuhtumid. Seega tasub kindlasti uurida erinevaid võimalusi graafide esitamiseks SQL-andmebaasides. Kui soovitakse leida sobivaimat disaini konkreetse konteksti jaoks, siis saab kasutada analüütiliste hierarhiate meetodit (Saaty meetodit), mis võimaldab kriteeriumite paariviisiliste võrdluste ja alternatiivide paariviisiliste võrdluste tulemusena leida alternatiivide suhtelise headuse.

Summary

Hierarchies are everywhere around us. Certainly it must be possible to store hierarchical data in databases. In the autumn of 2014, the most popular type of database management systems (DBMSs) is SQL DBMS (DB-Engines Ranking 2014). This general name is derived from the SQL database language that these systems provide to their users. However, there are a limited number of operators that can be used in order to process hierarchical data in these databases and this processing requires writing of quite complex database language code. Nowadays developers and researchers have proposed many different ways how to store hierarchical data in SQL databases. However, there are a limited number of experiments for comparinf different designs in the various aspects.

The main purpose of the current Master's Thesis was to compare different design solutions for representing hierarchical data in SQL databases. In order to make this investigation more thorough the author concentrated to a proper subset of possible design solutions. In the thirst part of this Thesis, we specified different design solutions, which were found from books and other sources. We specified these by using the same structure. We described design solutions that require "breaking up" hierarchies and storing values in different tables and fields. We did not consider design solutions that mean storing hierarchies (both relationships and data of entities they relate) as data values (for instance, as XML or JSON documents). The author does not know a single material that specifies all these design solutions together by using a structured form. Hence, it can be considered as a result of the thesis as well. The design solutions that were further investigated in the Thesis were adjacency list, nested sets, and materialized path. These were the most popular designs according to the data from the Google search engine. In the second part of this Thesis, we designed and created databases for the experiment. Databases were implemented in PostgreSQL and Oracle database management systems. In order to generate hierarchical data for the created databases, we used an existing test-data generator and created a PL/SQL script. During the experiment, we carried out select, update, and delete operations. In the third part of the Thesis, we compared these design solutions in terms of performance of operations, data size, possibilities of enforcing constraint declaratively, and complexity of queries and data modification statements. We investigated execution plans to find out, why there are differences in the performance of operations.

The experiments demonstrated (as expected) that there is no the best design solution. The design choice depends on the specific context. The choice is affected by requirements and relative importance of criteria that depends on the context. Next, only the design solutions of this investigation are

mentioned. For example, if it is important that queries must be fast, then nested sets design has to be selected. If there are frequent changes in the hierarchies, then nested sets design is not the best choice. In this case adjacency list model is better. From the point of view of the code (queries and data modification statements) simplicity, the most sensible choice is materialized path. If it is important that the designed database takes as little space as possible, then adjacency list is the best choice. It is possible to give more examples, but it is clear, that database developers have to specify the relative importance of criteria before making the choice.

In general, queries that were carried out during this experiment are faster in PostgreSQL database management system (compared to Oracle), but from the point of view of code complexity these are more complicated. In case of complex query (where subqueries are used), PostgreSQL database management system can not find a good execution plan and that is why the query execution is fairly slow. In PostgreSQL, the data size in case of all design solutions is smaller (except materialized path design solution 25 000 rows) than in Oracle.

Certainly it is possible to expand current experiment, because only three design solutions were investigated. One possible direction is to investigate other design solutions using the same experiment structure. The second direction would be extension of the scope. For example, it is possible to investigate how indexes and other means of improving query performance (for instance, sorting rows at the physical level) affects performance of operations in case of different design solutions and different database management systems. The scope of the investigation can also be extended by involving database management systems with different underlying data models. Trees are a special case of graphs. Hence, it would be interesting to investigate how to represent graphs in SQL databases. If there is a need to find the best design solution for a given context, then it would be possible to use analytic hierarchy process (Saaty's Method) that uses pairwise comparison of criteria and alternatives in order to find the relative goodness of alternatives.

Kasutatud kirjandus

1. About PostgreSQL, [WWW] http://www.postgresql.org/about/ (01.12.2014)
2. Atanassov, I. (2007). An Improvement of an Approach for Representation of Tree Structures in Relational Tables. – *Proceedings of the 2007 international conference on Computer systems and technologies: 14-15 June 2007, Rousse, Bulgaria.* II.9-1 - II.9-6.
3. Bhatt, K., Tarey, V., Patel, P. (2012). Analysis of source lines of Code (SLOC) metric. – International Journal of Emerging Technology and advanced engineering. [WWW] http://www.ijetae.com/files/Volume2Issue5/IJETAE_0512_25.pdf (01.12.2014)
4. Blaha, M. (2010). *Patterns of Data Modeling*. Boca Raton, CRC press.
5. Buldas, A., Laud, P., Willemson, J. (2002). Graafid. [WWW] http://research.cyber.ee/~peeter/teaching/graafid02s/graafid.pdf (19.11.2014)
6. Celko, J. (2004). *Hierarchical SQL*.[WWW] http://www.onlamp.com/pub/a/onlamp/2004/08/05/hierarchical_sql.html (11.09.2014)
7. Celko, J. (2012). *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Second Edition. Waltham, Elsevier. [WWW] http://books.google.ee/books?id=8NdQY8TuEDUC&printsec=frontcover&dq=sql+hierarchi es&hl=en&sa=X&ei=DgUPVMbVCoHuyQO7j4HoDA&ved=0CB8Q6AEwAA#v=one page&q=sql%20hierarchies&f=false (01.10.2014)
8. Chodnicki, S. (2010). Analyzing Hierarchical Data Using Bridge Tables. [WWW] http://type-exit.org/adventures-with-open-source-bi/2010/12/analyzing-hierarchical-data-using-bridge-tables/ (01.10.2014)
9. Database Administration, [WWW] https://docs.oracle.com/database/121/REFRN/refrn0043.htm#REFRN0043 (30.11.2014)
10. Database Concepts, [WWW] http://docs.oracle.com/cd/B19306_01/server.102/b14220/datatype.htm (06.12.2014)
11. Date, C.J., Darwen, H. (2014). *Databases, Types and the Relational Model*. 3rd edn. Addison Wesley. [WWW] http://www.thethirdmanifesto.com/ (05.11.2014)
12. DB-Engines Ranking (2014), [WWW] http://db-engines.com/en/ranking (12.09.2014)
13. Empty Graph, [WWW] http://mathworld.wolfram.com/EmptyGraph.html (01.12.2014)
14. Eessaar, E. (2008). *Andmebaaside projekteerimine*. Tallinn, Tallinna Tehnikaülikooli Kirjastus.
15. Eessaar, E. (2014). Teema 2. Hierarhilise-, võrk- ja relatsioonilise andmemudeli põhimõisteid [WWW] http://maurus.ttu.ee/ained/IDU0220_2014/doc/3/Teema_IDU0220_2_2014_ver3.pdf 20.10.2014
16. Eessaar, E. (2014). SQL ja hierarhilised andmed [WWW] http://maurus.ttu.ee/ained/IDU0220_2014/doc/15/SQL_ja_hierarhilised_andmed_IDU0220_2014.pdf (15.10.2014)
17. Explaining the unexplainable – part 4 (2013), [WWW] http://www.depesz.com/2013/05/19/explaining-the-unexplainable-part-4/ (05.12.2014)
18. Feuerstein, S. (2011). Technology: PL/SQL. Building with Blocks. [WWW] http://www.oracle.com/technetwork/issue-archive/2011/11-mar/o21plsql-242570.html (03.12.2014)

19. Four Ways To Work With Hierarchical Data (2000), [WWW] http://evolt.org/node/4047/ (16.09.2014)
20. Graafid, [WWW] http://www.cs.ut.ee/~nestra/mat/inf/p/aa/08s/s/graafid.pdf (01.12.2014 a)
21. Graafid, [WWW] http://www.staff.ttu.ee/~jmajak/Graafid.doc (01.12.2014 b)
22. Graafidest, [WWW] http://www.math.olympiaadid.ut.ee/arhiiv/oppemat/eesti/graafid.pdf (01.12.2014)
23. Halpin T., Morgan T. (2010) *Information Modeling and Relational Databases*. Morgan Kaufmann.
24. IT terministandari sõnastik (2014), [WWW] http://eki.ee/dict/its/ (17.09.2014)
25. Kaasik, Ü. (1992) Matemaatika Leksikon. Eesti Entsüklopeediakirjastus. Tallinn.
26. Karwin, B. (2010). *SQL Antipatterns. Avoiding the Pitfalls of Database Programming.* [WWW] http://it-ebooks.info/book/70/ (10.09.2014)
27. Koit, M. (1968). *Pilk graafiteooriasse.* Matemaatika ja kaasaeg, XIV. Tartu
28. Lewis, J. (2009). Explain VIEW. [WWW] http://jonathanlewis.wordpress.com/2009/06/25/explain-view/ (10.11.2014)
29. Method of organizing hierarchical data in a relational database (2002), [WWW] http://www.google.com/patents/US6480857 (12.10.2014)
30. Miliauskaitė, E., Nemuraitė, L. (2005). Representation of integrity constraints in conceptual models . - *Information Technology and Control, Vol.34, No.4, 2005.* [E-ajakiri] (http://itc.ktu.lt/) (01.12.2014)
31. Mittelineaarsed andmestruktuurid, [WWW] http://www.ttkool.ut.ee/comp/kaug/prog1/prog009.html (01.12.2014)
32. Mittelineaarsed struktuurid (2014), [WWW] http://www.cs.tlu.ee/~inga/alg_andm/Mittelineaarsed_struktuurid_puud_2014.pdf (05.11.2014)
33. Männiko, K. (2008). *XML-andmebaasi kasutamine terminihaldussüsteemis*. Bakalaureusetöö. Tallinn. Tallinna Ülikool, Informaatika Instituut. [WWW] http://www.cs.tlu.ee/instituut/opilaste_tood/magistri_tood/2007_sygis/Kaur_Manniko/Kaur_Manniko_Bakalaureuse_Too.pdf (20.10.2014)
34. Männil, M. (2014). *Mõnede SQL-andmebaasisüsteemide võimekusest SQLi keelelise liiasuse silumisel.* Magistritöö. Tallinn, Tallinna Tehnikaülikool, Informaatikainstituut.
35. Penjam, J. (2013). Puud. Minimaalsed graafi katvad puud. Rändkaupmehe ülesanne. [WWW] http://www.cs.ioc.ee/DM2/Handouts/8_Puud.pdf (08.09.2014)
36. Petuhhov, I. (2014a). Mittelineaarsed struktuurid. Graaf. Põhialgoritmid graafidel. [WWW] http://www.cs.tlu.ee/~inga/alg_andm/Graafid_2014.pdf (30.11.2014)
37. Petuhhov, I. (2014b). Mittelineaarsed struktuurid. Puu. Kahendpuu. Puude läbimine ja realisatsioon. [WWW] http://www.cs.tlu.ee/~inga/alg_andm/Mittelineaarsed_struktuurid_puud_2014.pdf (10.09.2014)
38. Petuhhov, I. (2014 c). Puud.Kahendpuud. [WWW] http://www.cs.tlu.ee/~inga/alg_andm/tree_gen_2011.pdf (04.12.201404.12.2014)
39. Piho, G. (2011). *Archetypes Based Techniques for Development of Domains, Requirements and Software Towards LIMS Software factory.* Doktoritöö. Tallinn. Tallinna Tehnikaülikool, Informaatikainstituut. TTÜ kirjastus
40. PostgreSQL 9.1.14 Documentation, [WWW] http://www.postgresql.org/docs/9.1/static/tablefunc.html (02.11.2014 a)

41. PostgreSQL 9.1.14 Documentation, [WWW] http://www.postgresql.org/docs/9.1/static/ltree.html (05.10.2014 b)
42. PosgtreSQl 9.1.14 Documentation, [WWW] http://www.postgresql.org/docs/9.1/static/sql-begin.html (03.12.2014 c)
43. PostgreSQL 9.1.14 Documentation, [WWW] http://www.postgresql.org/docs/9.1/static/sql-end.html (03.12.2014 d)
44. PostgreSQL 9.1.14 Documentation, [WWW] http://www.postgresql.org/docs/9.1/static/datatype.html (06.12.2014 e)
45. Puutusmaa, S. (2012). Sisulise tähendusega võtmete ja surrogaatvõtmete kasutamise eelised ning puudused SQL-andmebaasides. Bakalaureusetöö. [WWW] http://maurus.ttu.ee/ained/IDU0230_2014/doc/20/Votmed_SQL_andmebaasides.pdf (11.09.2014)
46. Rekursioon (2008), [WWW] http://www.cs.tlu.ee/~inga/alg_andm/rekursioon_2008.pdf (06.11.2014)
47. Reiska, P. (2008). Mõistekaardid IKT abil [WWW] http://www.htk.tlu.ee/tiigriope/index.php?title=M%C3%B5istekaardid_IKT_abil (16.10.2014)
48. Smusenok, S., Trubilko, A., Furmanov, A. (2013). Анализ способов представления иерархических структур в реляционных базах данных с использованием стресс-тестов. – Открытые информационные и компьютерные интегрированные технологии. Выпуск 62. Харьков: Нац. аэрокосм. ун-т "ХАИ", 107-11.
49. Stadnik, M. (2008). Иерархические структуры данных и производительность . [WWW] http://habrahabr.ru/post/47280/ (24.09.2014)
50. Stadnik, S. (2013). How to get size of all tables in an Oracle database schema. [WWW] http://www.ozmoroz.com/2013/08/how-to-get-size-of-all-tables-in-oracle.html (15.10.2014)
51. System Administration Functions. PostgreSQL 9.3.5 Documentation. [WWW] http://www.postgresql.org/docs/9.3/static/functions-admin.html (10.10.2014)
52. The simplest(?) way to do tree-based queries in SQL (2010), [WWW] http://dirtsimple.org/2010/11/simplest-way-to-do-tree-based-queries.html
53. Tropashko, V. (2005). Nested Intervals Tree Encoding in SQL. [WWW] http://www.sigmod.org/publications/sigmod-record/0506/p47-article-tropashko.pdf (16.09.2014)
54. Tropashko, V. (2007). *SQL Design Patterns: Expert Guide To SQL Programming*. Kittrell, Rampant TechPress.
55. Vallaste, H. (2000). e-Teatmik: IT ja sidetehnika seletav sõnaraamat. [WWW] http://www.vallaste.ee (01.12.2014)
56. Wikipedia. (2014a). Hierarhia. Wikimedia Foundation, Inc. [WWW] http://et.wikipedia.org/wiki/Hierarhia (15.09.2014)
57. Wikipedia. (2014b). Hierarchical and recursive queries in SQL. Wikimedia Foundation, Inc. [WWW] http://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL (01.11.2014)
58. Wikipeadia. (2014c). JSON. [WWW] http://et.wikipedia.org_/wiki/JSON (05.11.2014)

Lisad

Lisa 1 – tabelite loomise laused Oracle andmebaasis

```
CREATE TABLE ADJACENCY_OSAPOOL
(
  OP_ID       NUMBER(10) PRIMARY KEY,
  OP_TELEFON  VARCHAR2(30 CHAR),
  OP_E_MAIL   VARCHAR2(50 CHAR),
  YLEMUS_ID   NUMBER(10),
  CONSTRAINT ADJACENCY_OSAPOOL_R01 FOREIGN KEY (YLEMUS_ID) REFERENCES
ADJACENCY_OSAPOOL (OP_ID)ON DELETE SET NULL,
  CONSTRAINT CHECK_OP_ID_NOT_EQ_YLEMUS_ID CHECK (OP_ID != YLEMUS_ID)
);

CREATE INDEX ADJACENCY_OSAPOOL_FK ON ADJACENCY_OSAPOOL (YLEMUS_ID);

CREATE TABLE ADJACENCY_ORGANISATSIOON
(
  ORG_ID        NUMBER(10) PRIMARY KEY,
  ORG_REG_KOOD  VARCHAR2(10 CHAR)               NOT NULL,
  ORG_NIMETUS   VARCHAR2(30 CHAR)               NOT NULL,
  CONSTRAINT ADJACENCY_ORGANISATSIOON_U01 UNIQUE (ORG_REG_KOOD),
  CONSTRAINT ADJACENCY_ORGANISATSIOON_R01 FOREIGN KEY (ORG_ID) REFERENCES
ADJACENCY_OSAPOOL (OP_ID) ON DELETE CASCADE
);

CREATE TABLE ADJACENCY_ISIK
(
  IS_ID         NUMBER(10) PRIMARY KEY,
  IS_EESNIMI    VARCHAR2(20 CHAR)               NOT NULL,
  IS_PERENIMI   VARCHAR2(20 CHAR)               NOT NULL,
  IS_ISIKUKOOD  VARCHAR2(11 CHAR),
  IS_SYNNIAEG   DATE                            NOT NULL,
  CONSTRAINT ADJACENCY_ISIK_U01 UNIQUE (IS_ISIKUKOOD),
  CONSTRAINT ADJACENCY_ISIK_R01 FOREIGN KEY (IS_ID) REFERENCES ADJACENCY_OSAPOOL
(OP_ID) ON DELETE CASCADE
);

CREATE INDEX IDX_AD_SYNNIAEG ON ADJACENCY_ISIK(IS_SYNNIAEG);

CREATE TABLE MPATH_OSAPOOL
(
  OP_ID       NUMBER(10) PRIMARY KEY,
  OP_TELEFON  VARCHAR2(30 CHAR),
  OP_E_MAIL   VARCHAR2(50 CHAR),
  TEEKOND     VARCHAR2(4000 BYTE),
  CONSTRAINT MPATH_OSAPOOL_C01 CHECK (TEEKOND NOT LIKE '%.' || OP_ID  || '.%' AND
TEEKOND NOT LIKE OP_ID || '.' || OP_ID),
  CONSTRAINT MPATH_OSAPOOL_U01 UNIQUE (TEEKOND),
  CONSTRAINT CHECK_TEEKOND CHECK(REGEXP_LIKE(teekond, '\d') AND
REGEXP_LIKE(teekond, '.'))
);

CREATE TABLE MPATH_ORGANISATSIOON
(
  ORG_ID        NUMBER(38) PRIMARY KEY,
  ORG_REG_KOOD  VARCHAR2(10 CHAR)               NOT NULL,
  ORG_NIMETUS   VARCHAR2(30 CHAR)               NOT NULL,
```

```
  CONSTRAINT MPATH_ORGANISATSIOON_U01 UNIQUE (ORG_REG_KOOD),
  CONSTRAINT MPATH_ORGANISATSIOON_R01 FOREIGN KEY (ORG_ID) REFERENCES
MPATH_OSAPOOL (OP_ID) ON DELETE CASCADE
);

CREATE TABLE MPATH_ISIK
(
  IS_ID         NUMBER(10) PRIMARY KEY,
  IS_EESNIMI    VARCHAR2(20 CHAR)               NOT NULL,
  IS_PERENIMI   VARCHAR2(20 CHAR)               NOT NULL,
  IS_ISIKUKOOD  VARCHAR2(11 CHAR),
  IS_SYNNIAEG   DATE                            NOT NULL,
  CONSTRAINT MPATH_ISIK_U01 UNIQUE (IS_ISIKUKOOD),
  CONSTRAINT MPATH_ISIK_R01 FOREIGN KEY (IS_ID) REFERENCES MPATH_OSAPOOL (OP_ID)
ON DELETE CASCADE
);

CREATE INDEX IDX_MPATH_SYNNIAEG ON MPATH_ISIK(IS_SYNNIAEG);

CREATE TABLE NSETS_OSAPOOL
(
  OP_ID       NUMBER(10) PRIMARY KEY,
  OP_TELEFON  VARCHAR2(30 CHAR),
  OP_E_MAIL   VARCHAR2(50 CHAR),
  VASAK       NUMBER(10),
  PAREM       NUMBER(10),
  CONSTRAINT CHECK_IRREFLEXIVE CHECK (VASAK < PAREM AND VASAK >= 0  AND PAREM >
0),
  CONSTRAINT NSETS_OSAPOOL_U01 UNIQUE (PAREM, VASAK)
);

CREATE TABLE NSETS_ORGANISATSIOON
(
  ORG_ID        NUMBER(10) PRIMARY KEY,
  ORG_REG_KOOD  VARCHAR2(10 CHAR)               NOT NULL,
  ORG_NIMETUS   VARCHAR2(30 CHAR)               NOT NULL,
  CONSTRAINT NSETS_ORGANISATSIOON_U01 UNIQUE (ORG_REG_KOOD),
  CONSTRAINT NSETS_ORGANISATSIOON_R01 FOREIGN KEY (ORG_ID) REFERENCES
NSETS_OSAPOOL (OP_ID) ON DELETE CASCADE
);

CREATE TABLE NSETS_ISIK
(
  IS_ID         NUMBER(10) PRIMARY KEY,
  IS_EESNIMI    VARCHAR2(20 CHAR)               NOT NULL,
  IS_PERENIMI   VARCHAR2(20 CHAR)               NOT NULL,
  IS_ISIKUKOOD  VARCHAR2(11 CHAR),
  IS_SYNNIAEG   DATE                            NOT NULL,
  CONSTRAINT NSETS_ISIK_U01 UNIQUE (IS_ISIKUKOOD),
  CONSTRAINT NSETS_ISIK_R01 FOREIGN KEY (IS_ID) REFERENCES NSETS_OSAPOOL (OP_ID)
ON DELETE CASCADE
);

CREATE INDEX IDX_NSETS_SYNNIAEG ON NSETS_ISIK(IS_SYNNIAEG);
```

Lisa 2 – tabelite loomise laused PostgreSQL andmebaasis

```
CREATE SCHEMA adjacency_list;

CREATE TABLE adjacency_list."OSAPOOL"
(
  "OP_ID"       INTEGER PRIMARY KEY,
  "OP_TELEFON"  VARCHAR(30),
  "OP_E_MAIL"   VARCHAR(50),
  "YLEMUS_ID"   INTEGER,
  CONSTRAINT OSAPOOL_YLEMUS_ID_fkey FOREIGN KEY ("YLEMUS_ID") REFERENCES
adjacency_list."OSAPOOL" ("OP_ID") ON DELETE SET NULL,
  CONSTRAINT CHECK_OP_ID_NOT_EQ_YLEMUS_ID CHECK ("OP_ID" != "YLEMUS_ID")
);

CREATE INDEX "OSAPOOL_FK" ON adjacency_list."OSAPOOL" USING btree ("YLEMUS_ID");

CREATE TABLE adjacency_list."ORGANISATSIOON"
(
  "ORG_ID"        INTEGER PRIMARY KEY,
  "ORG_REG_KOOD"  VARCHAR(10)               NOT NULL,
  "ORG_NIMETUS"   VARCHAR(30)               NOT NULL,
  CONSTRAINT ORGANISATSIOON_ORG_REG_KOOD_key UNIQUE ("ORG_REG_KOOD"),
  CONSTRAINT ORGANISATSIOON_ORG_ID_fkey FOREIGN KEY ("ORG_ID") REFERENCES
adjacency_list."OSAPOOL" ("OP_ID") ON DELETE CASCADE
);

CREATE TABLE adjacency_list."ISIK"
(
  "IS_ID"         INTEGER PRIMARY KEY,
  "IS_EESNIMI"    VARCHAR(20)               NOT NULL,
  "IS_PERENIMI"   VARCHAR(20)               NOT NULL,
  "IS_ISIKUKOOD"  VARCHAR(11),
  "IS_SYNNIAEG"   DATE                      NOT NULL,
  CONSTRAINT ISIK_IS_ISIKUKOOD_key UNIQUE ("IS_ISIKUKOOD"),
  CONSTRAINT ISIK_IS_ID_fkey FOREIGN KEY ("IS_ID") REFERENCES
adjacency_list."OSAPOOL" ("OP_ID") ON DELETE CASCADE
);

CREATE INDEX "IDX_SYNNIAEG" ON adjacency_list."ISIK" USING btree ("IS_SYNNIAEG");

CREATE SCHEMA nested_sets;
CREATE TABLE nested_sets."OSAPOOL"
(
  "OP_ID"       INTEGER PRIMARY KEY,
  "OP_TELEFON"  VARCHAR(30),
  "OP_E_MAIL"   VARCHAR(50),
  "VASAK"       INTEGER,
  "PAREM"       INTEGER,
  CONSTRAINT CHECK_IRREFLEXIVE CHECK ("VASAK" < "PAREM" AND "VASAK" >= 0  AND
"PAREM" > 0),
  CONSTRAINT UNI_VASAK_PAREM UNIQUE ("PAREM", "VASAK")
);

CREATE TABLE nested_sets."ORGANISATSIOON"
(
  "ORG_ID"        INTEGER PRIMARY KEY,
  "ORG_REG_KOOD"  VARCHAR(10)               NOT NULL,
  "ORG_NIMETUS"   VARCHAR(30)               NOT NULL,
  CONSTRAINT ORGANISATSIOON_ORG_REG_KOOD_key UNIQUE ("ORG_REG_KOOD"),
```

```
  CONSTRAINT FK_OSAPOOL_ID FOREIGN KEY ("ORG_ID") REFERENCES nested_sets."OSAPOOL"
("OP_ID") ON DELETE CASCADE
);

CREATE TABLE nested_sets."ISIK"
(
  "IS_ID"         INTEGER PRIMARY KEY,
  "IS_EESNIMI"    VARCHAR(20)               NOT NULL,
  "IS_PERENIMI"   VARCHAR(20)               NOT NULL,
  "IS_ISIKUKOOD"  VARCHAR(11),
  "IS_SYNNIAEG"   DATE                      NOT NULL,
  CONSTRAINT ISIK_IS_ISIKUKOOD_key UNIQUE ("IS_ISIKUKOOD"),
  CONSTRAINT FK_OSAPOOL_ID FOREIGN KEY ("IS_ID") REFERENCES nested_sets."OSAPOOL"
("OP_ID") ON DELETE CASCADE
);

CREATE INDEX "IDX_SYNNIAEG" ON nested_sets."ISIK" USING btree ("IS_SYNNIAEG");

CREATE SCHEMA materialized_path;

CREATE EXTENSION ltree WITH SCHEMA materialized_path;

CREATE TABLE materialized_path."OSAPOOL"
(
  "OP_ID"       INTEGER PRIMARY KEY,
  "OP_TELEFON"  VARCHAR(30),
  "OP_E_MAIL"   VARCHAR(50),
  "TEEKOND"     materialized_path.LTREE,
  CONSTRAINT CHECK_ACYCLIC CHECK ("TEEKOND"::text NOT LIKE '%.' || "OP_ID"  ||
'.%' AND "TEEKOND"::text NOT LIKE "OP_ID" || '.' || "OP_ID"),
  CONSTRAINT OSAPOOL_TEEKOND_key UNIQUE ("TEEKOND")
);

CREATE TABLE materialized_path."ORGANISATSIOON"
(
  "ORG_ID"        INTEGER PRIMARY KEY,
  "ORG_REG_KOOD"  VARCHAR(10)               NOT NULL,
  "ORG_NIMETUS"   VARCHAR(30)               NOT NULL,
  CONSTRAINT ORGANISATSIOON_ORG_REG_KOOD_key UNIQUE ("ORG_REG_KOOD"),
  CONSTRAINT ORGANISATSIOON_ORG_ID_fkey FOREIGN KEY ("ORG_ID") REFERENCES
nested_sets."OSAPOOL" ("OP_ID") ON DELETE CASCADE
);

CREATE TABLE materialized_path."ISIK"
(
  "IS_ID"         INTEGER PRIMARY KEY,
  "IS_EESNIMI"    VARCHAR(20)               NOT NULL,
  "IS_PERENIMI"   VARCHAR(20)               NOT NULL,
  "IS_ISIKUKOOD"  VARCHAR(11),
  "IS_SYNNIAEG"   DATE                      NOT NULL,
  CONSTRAINT ISIK_IS_ISIKUKOOD_key UNIQUE ("IS_ISIKUKOOD"),
  CONSTRAINT ISIK_IS_ID_fkey FOREIGN KEY ("IS_ID") REFERENCES
nested_sets."OSAPOOL" ("OP_ID") ON DELETE CASCADE
);

CREATE INDEX "IDX_SYNNIAEG" ON materialized_path."ISIK" USING btree
("IS_SYNNIAEG");
```

Lisa 3 - Tabelite andmemaht megabaitides Oracle andmebaasis

```
SELECT owner, table_name, ROUND(sum(bytes)/1024/1024, 2) MB
    FROM
    (SELECT segment_name table_name, owner, bytes
       FROM dba_segments
      WHERE segment_type IN ('TABLE', 'TABLE PARTITION', 'TABLE SUBPARTITION')
     UNION ALL
     SELECT i.table_name, i.owner, 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.table_name, l.owner, 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.table_name, l.owner, 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 in UPPER('C##TUD15')
    GROUP BY table_name, owner
       ORDER BY SUM(bytes) DESC;
```

Lisa 4 – Külgnevusnimistu mudeli jaoks hierarhia genereerimine

```
alter table ADJACENCY_OSAPOOL DISABLE constraint CHECK_OP_ID_NOT_EQ_YLEMUS_ID;

DECLARE
    CURSOR null_level_org IS
      SELECT org_id
      FROM   adjacency_organisatsioon
      WHERE  ROWNUM = 1;

    CURSOR first_level_org IS
      SELECT org_id
      FROM   adjacency_organisatsioon,
           adjacency_osapool
      WHERE  ROWNUM <= 100
             AND OP_ID = org_id
             AND ylemus_id IS NULL;

    CURSOR second_level_org IS
      SELECT org_id
      FROM   adjacency_organisatsioon,
             adjacency_osapool
      WHERE  ylemus_id IS NULL
             AND op_id = org_id
             AND ROWNUM <= 50;

    CURSOR third_level_org IS
      SELECT org_id
      FROM   adjacency_organisatsioon,
             adjacency_osapool
```

```
      WHERE  ylemus_id IS NULL
             AND op_id = org_id
             AND ROWNUM <= 50;

    CURSOR null_level_persons IS
      SELECT is_id
      FROM   adjacency_isik,
             adjacency_osapool
      WHERE  ROWNUM = 1
             AND op_id = is_id
             AND ylemus_id IS NULL;

    CURSOR first_level_persons IS
      SELECT is_id
      FROM   adjacency_isik,
             adjacency_osapool
      WHERE  op_id = is_id
             AND ylemus_id IS NULL
             AND ROWNUM <= 50;

    CURSOR second_level_persons IS
      SELECT is_id
      FROM   adjacency_isik,
             adjacency_osapool
      WHERE  op_id = is_id
             AND ylemus_id IS NULL
             AND ROWNUM <= 50;

     CURSOR third_level_persons IS
      SELECT is_id
      FROM   adjacency_isik,
             adjacency_osapool
      WHERE  op_id = is_id
             AND ylemus_id IS NULL
             AND ROWNUM <= 10;
BEGIN
    -- organisatsioonide hierarhia juurelemendi määramine
    FOR rec_null IN null_level_org LOOP
        UPDATE adjacency_osapool
        SET    ylemus_id = op_id
        WHERE  op_id = rec_null.org_id;

        COMMIT;

        -- 1 taseme organisatsioonide määramine
        FOR rec IN first_level_org LOOP
            UPDATE adjacency_osapool
            SET    ylemus_id = rec_null.org_id
            WHERE  op_id = rec.org_id;

            COMMIT;

            -- 2. taseme organisatsioonide määramine
            FOR rec_1 IN second_level_org LOOP
                UPDATE adjacency_osapool
                SET    ylemus_id = rec.org_id
                WHERE  op_id = rec_1.org_id;

                COMMIT;

                -- 3. taseme organisatsioonide määramine
```

```
                FOR rec_2 IN third_level_org LOOP
                    UPDATE adjacency_osapool
                    SET    ylemus_id = rec_1.org_id
                    WHERE  op_id = rec_2.org_id;

                    COMMIT;

                END LOOP;
            END LOOP;
        END LOOP;
    END LOOP;

    -- töötajate hierarhia juurelemendi määramine
   FOR rec IN null_level_persons LOOP
      UPDATE adjacency_osapool
      SET ylemus_id = op_id
      WHERE op_id = rec.is_id;

      COMMIT;

      -- 1. taseme töötajate määramine
         FOR rec_1 IN first_level_persons LOOP
          UPDATE ADJACENCY_OSAPOOL
          SET ylemus_id = rec.is_id
          WHERE op_id = rec_1.is_id;

          COMMIT;

          -- 2. taseme töötajate määramine
             FOR rec_2 IN second_level_persons LOOP
               UPDATE ADJACENCY_OSAPOOL
               SET ylemus_id = rec_1.is_id
               WHERE op_id = rec_2.is_id;

                COMMIT;

                -- 3.taseme töötajate määramine
                   FOR rec_3 IN third_level_persons LOOP
                     UPDATE ADJACENCY_OSAPOOL
                     SET ylemus_id = rec_2.is_id
                     WHERE op_id = rec_3.is_id;

                           COMMIT;

                   END LOOP;
             END LOOP;
         END LOOP;
   END LOOP;

   UPDATE adjacency_osapool
   SET      ylemus_id = NULL
   WHERE ylemus_id = op_id;

END;

alter table ADJACENCY_OSAPOOL ENABLE constraint CHECK_OP_ID_NOT_EQ_YLEMUS_ID;
```

Lisa 5 – Pesastatud hulkade mudeli jaoks hierarhia genereerimine

```
-- Tree tabelis hoitakse andmed hierarhia kohta
CREATE TABLE Tree
(TR_ID INTEGER,
 TR_PARENT_ID INTEGER);

 -- Stack tabelis hoitakse andmed pestatud hulkade mudeli kohta
CREATE TABLE Stack
(stack_top INTEGER NOT NULL,
 ST_TR_ID INTEGER,
 lft INTEGER,
 rgt INTEGER);

INSERT INTO Tree
SELECT op_id,
   CASE WHEN ylemus_id IS NULL THEN op_id ELSE ylemus_id END
FROM adjacency_osapool,
     adjacency_organisatsioon
WHERE org_id  = op_id;


DECLARE
counter INTEGER;
max_counter INTEGER;
current_top INTEGER;
p_cnt INTEGER;
p_check INTEGER;

BEGIN
SELECT COUNT(TR_ID) INTO p_cnt FROM Tree;

counter := 2;
max_counter := 2 * p_cnt;
current_top := 1;

INSERT INTO Stack
SELECT 1, TR_ID, 1, NULL
 FROM Tree
 WHERE TR_PARENT_ID = TR_ID;

DELETE FROM Tree
 WHERE TR_PARENT_ID = TR_ID;

WHILE counter <= (max_counter - 2)
LOOP

SELECT count(T1.TR_ID)
   into p_check
 FROM Stack  S1, Tree  T1
 WHERE S1.ST_TR_ID = T1.TR_PARENT_ID
 AND S1.stack_top = current_top;

 IF p_check > 0
 THEN
 BEGIN
 INSERT INTO Stack
 SELECT (current_top + 1), MIN(T1.TR_ID), counter, NULL
 FROM Stack  S1, Tree  T1
 WHERE S1.ST_TR_ID = T1.TR_PARENT_ID
 AND S1.stack_top = current_top;
```

```
 DELETE FROM Tree
 WHERE TR_ID = (SELECT ST_TR_ID
 FROM Stack
 WHERE stack_top = current_top + 1);

counter := counter + 1;
 current_top := current_top + 1;
 END;
 ELSE
 BEGIN
       UPDATE Stack
          SET rgt = counter,
         stack_top = -stack_top
          WHERE stack_top = current_top
              AND rgt IS NULL;
            counter := counter + 1;
            current_top := current_top - 1;
    END;
 END IF;
 END LOOP;
 UPDATE Stack
   set rgt = lft +1
   where rgt is null
     and lft != 1;
 UPDATE Stack
    set rgt = max_counter
  WHERE rgt is null;
END;

DECLARE
CURSOR cur IS
SELECT ST_TR_ID, LFT, RGT
FROM STACK;
BEGIN
FOR rec IN cur
LOOP
  UPDATE NSETS_OSAPOOL
       SET VASAK = rec.LFT,
              PAREM = rec.RGT
          WHERE OP_ID =rec.ST_TR_ID;
END LOOP;
END;

DELETE FROM STACK;
DELETE FROM Tree;

INSERT INTO Tree
SELECT op_id,
 CASE WHEN ylemus_id IS NULL THEN op_id ELSE ylemus_id END
FROM adjacency_osapool,
     adjacency_isik
WHERE is_id = op_id;

DECLARE
counter INTEGER;
max_counter INTEGER;
current_top INTEGER;
p_cnt INTEGER;
p_check INTEGER;
org_count NUMBER;
BEGIN
```

```
SELECT COUNT(op_id) INTO p_cnt FROM adjacency_osapool;

SELECT count(org_id)
INTO org_count
FROM adjacency_organisatsioon;

counter := org_count*2 +2;
max_counter := 2 * p_cnt;
current_top := 1;

INSERT INTO Stack
SELECT 1, TR_ID, org_count *2 +1, NULL
FROM Tree
WHERE TR_PARENT_ID = TR_ID;

DELETE FROM Tree
WHERE TR_PARENT_ID = TR_ID;

WHILE counter <= (max_counter - 2)
LOOP

SELECT count(T1.TR_ID)
   into p_check
 FROM Stack  S1, Tree  T1
 WHERE S1.ST_TR_ID = T1.TR_PARENT_ID
 AND S1.stack_top = current_top;

 IF p_check > 0
 THEN
 BEGIN
 INSERT INTO Stack
 SELECT (current_top + 1), MIN(T1.TR_ID), counter, NULL
 FROM Stack  S1, Tree  T1
 WHERE S1.ST_TR_ID = T1.TR_PARENT_ID
 AND S1.stack_top = current_top;

 DELETE FROM Tree
 WHERE TR_ID = (SELECT ST_TR_ID
 FROM Stack
 WHERE stack_top = current_top + 1);

counter := counter + 1;
 current_top := current_top + 1;
 END;
 ELSE
 BEGIN
       UPDATE Stack
          SET rgt = counter,
         stack_top = -stack_top
          WHERE stack_top = current_top
              AND rgt IS NULL;
            counter := counter + 1;
            current_top := current_top - 1;
    END;
 END IF;
 END LOOP;
 UPDATE Stack
   set rgt = lft +1
   where rgt is null
     and lft != org_count *2 +1;
 UPDATE Stack
```

```
    set rgt = max_counter
  WHERE rgt is null;
END;

DECLARE
CURSOR cur IS
SELECT ST_TR_ID, LFT, RGT
FROM STACK;
BEGIN
FOR rec IN cur
LOOP
  UPDATE NSETS_OSAPOOL
       SET VASAK = rec.LFT,
              PAREM = rec.RGT
          WHERE OP_ID =rec.ST_TR_ID;
END LOOP;
END;
```

Lisa 6 – Materialiseeritud tee mudeli jaoks hierarhia genereerimine

```
DECLARE
CURSOR cur IS

WITH
  recursed_tree(OP_ID, PATH) AS
(
  SELECT
    OP_ID,
    cast(OP_ID as varchar(100))  AS Path
  FROM
   ADJACENCY_OSAPOOL
  WHERE
    YLEMUS_ID IS NULL

  UNION ALL

  SELECT
    next.OP_ID,
    concat(prev.Path || '.', cast(next.OP_ID as varchar(100)))   AS Path
  FROM
    recursed_tree   prev
  INNER JOIN
    ADJACENCY_OSAPOOL        next
       ON prev.OP_ID = next.YLEMUS_ID
)

SELECT
  op_id, path
FROM
  recursed_tree;
  BEGIN

  FOR rec IN cur
  LOOP
     UPDATE MPATH_OSAPOOL
        SET TEEKOND = rec.path
           WHERE OP_ID = rec.OP_ID;
  END LOOP;

  END;
```