TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Vaadete mõju päringute täitmisplaanide koostamisele kahe andmebaasisüsteemi näitel

Magistritöö

Üliõpilane: Darja Kašnikova
Üliõpilaskood: 132423IABMM
Juhendaja: Erki Eessaar

Tallinn
2015

Autorideklaratsioon

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

---------- ----------

(*kuupäev*) (*allkiri*)

Annotatsioon

Kašnikova D. (2015) Vaadete mõju päringute täitmisplaanide koostamisele kahe andmebaasisüsteemi näitel. Magistritöö, Tallinna Tehnikaülikool.

Käesolevas töös uuritakse, kuidas vaadete kasutamine mõjutab kahes SQL-andmebaasisüsteemis päringute täitmisplaanide valikut. Töö käigus võrreldakse päringute täitmisplaanide erinevusi ja uuritakse neid erinevusi põhjustavaid faktoreid. Lisaks võrreldakse päringute täitmise kiiruseid. Võrdlemiseks kasutatakse mitmeid päringute paare – igas paaris on üks päring tehtud vaate põhjal ja teine loogiliselt samaväärne päring on tehtud otse baastabelite põhjal. Töös kasutatavateks andmebaasisüsteemideks on Oracle Database ja PostgreSQL.

Töös käsitletakse probleemi, et päringute tegemine vaadete põhjal võib põhjustada andmebaasisüsteemi poolt mitteoptimaalsete ja ressursimahukate täitmisplaanide valimise, mille tulemusena muutub päringute täitmine märgatavalt aeglasemaks võrreldes otse baastabelite põhjal tehtud päringute täitmisega. Töö tulemusena võib väita, et see probleem on PostgreSQL andmebaasisüsteemis suurem kui Oracle andmebaasisüsteemis. Töö annab lugejatele infot selle kohta, kuidas mõjutab erinevate lausekonstruktsioonide kasutamine vaadete alampäringutes nende põhjal tehtavate päringute täitmist.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 122 leheküljel, 5 peatükki, 20 joonist, 19 tabelit.

Abstract

Kašnikova D. (2015) The Influence of Views to the Creation of Query Execution Plans in the Example of Two Database Management Systems. Master's Thesis, Tallinn University of Technology.

In this thesis a research will be conducted how the usage of views affects the creation of query execution plans in case of two SQL database management systems (DBMSs). Differences of query execution plans as well as factors that cause the differences are researched. In addition, execution speeds of the queries are compared. Multiple pairs of queries are analyzed – in each pair one query references a view and another logically equivalent query is made directly based on base table. Database management systems studied in the thesis are Oracle Database and PostgreSQL.

The thesis focuses on a problem that the usage of views may cause the selection of sub-optimal and resource-intensive query execution plans by DBMSs. As a result the query execution takes noticeably more time compared to the execution of logically equivalent queries made directly based on base tables. As a result of the conducted analysis, it can be argued that this problem is bigger in case of PostgreSQL compared to Oracle Database. The work gives information how the usage of different language constructs in the subqueries of views influences the execution of queries based on the views.

The thesis is in Estonian and contains 122 pages of text, 5 chapters, 20 figures, 19 tables.

Lühendite ja mõistete nimekiri

Aknafunktsioonid	***Window functions*** Funktsioon, mis arvutab iga päringu tulemusse kuuluva rea korral väärtuse tabeli teiste ridade põhjal (nende ridade hulka nimetatakse aknaks), mis on hetkel vaadeldava reaga mingil viisil seotud. Funktsiooni tulemus võib sõltuda ridade järjekorrast aknas. Aknafunktsioone nimetatakse ka analüütilisteks funktsioonideks.
DML	***Data Manipulation Language*** Andmekäitluskeel; andmebaasikeele alamkeel millesse kuuluvad laused on mõeldud andmete otsimiseks ja muutmiseks.
FK	***Foreign key*** Välisvõti.
Kokkuvõtte-funktsioonid	***Aggregate functions*** Funktsioon, mille kasutamisel read grupeeritakse ning funktsiooni abil leitakse väärtus iga grupi kohta.
Korreleeruv alampäring	***Correlated subquery*** Kontseptuaalselt täidetakse alampäringut uuesti iga peapäringu leitava rea korral [1].
Mittekorreleeruv alampäring	***Noncorrelated subquery*** Alampäring, mis pole põhipäringuga läbipõimunud ja mida saab seetõttu käivitada eraldiseisva lausena [1].
PK	***Primary key*** Primaarvõti.

SQL ***Structured Query Language***
Struktuurpäringukeel.

Täitmisplaan ***Execution plan***

Andmebaasisüsteemi poolt andmekäitluskeele lause kohta koostatav protseduur (algoritm), mis määrab, milliste andmebaasisüsteemi sisemiste tegevuste kaudu jõuda lausega soovitud tulemusteni.

UQ ***Unique constraint***
Unikaalsuse kitsendus.

VDL ***Virtual Data Layer***
Virtuaalne andmekiht.

Vaade ***View***

Nime omav tuletatud tabel, milles olevad andmed leitakse päringu tulemusena teiste tabelite põhjal sellel hetkel, kui keegi soovib sellest tabelist andmeid küsida.

Vaate mestimine ***View merging***

Andmebaasisüsteemi poolt vaate alampäringu ja päringu põhjal tehtud päringu kokkupanemine üheks päringuks nii, et selle päringu *FROM* klauslis pole alampäringuid.

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Andmebaasi kogutakse andmeid. Andmete kogumisel pole aga mõtet kui neid andmeid hiljem ei kasutata. Selleks, et andmebaasis olevaid andmeid kasutada, tuleb need andmebaasist üles leida ja selleks kirjutada andmebaasisüsteemile mõistetav päring. Kui andmebaasisüsteemis on kasutusel deklaratiivne päringukeel (nagu näiteks SQLi päringukeel), siis sellisel juhul deklareeritakse päringu lauses kasutaja soov selle kohta, millised andmed tahetakse leida. Päringu kirjutaja ei koosta andmete otsimise algoritmi. Selle algoritmi leidmine on andmebaasisüsteemi ülesanne. Sellist algoritmi nimetatakse täitmisplaaniks. Ühe ja sama lause täitmiseks saab andmebaasisüsteem enamasti kasutada mitut erinevat algoritmi (sageli isegi kümneid algoritme), mis kõik annavad kokkuvõttes sama tulemuse. Algoritmi valikust sõltub päringu täitmise efektiivsus (nt kiirus). Andmebaasisüsteemi ülesandeks on leida päringu käivitamise hetkele vastavatele oludele kõige sobivam algoritm, mille korral lause täidetakse võimalikult efektiivselt. Sellist algoritmi nimetatakse optimaalseks täitmisplaaniks. Sõltuvalt andmebaasisüsteemi toimimise loogikast, päringu keerukusest, arvutusressurssidest võib optimaalsus tähendada kõige paremat täitmisplaani kõigi võimalike plaanide seast või kõige paremat plaani piisavalt hulga võimalike plaanide seas. Käesolevas töös käsitletakse andmebaasikeelt SQL ja andmebaasisüsteeme kus saab seda keelt kasutada (SQL-andmebaasisüsteeme). SQL keel on mahukas keel. Selles on alamkeeled nagu andmekäitluskeel ja andmekirjelduskeel. Käesolev töö keskendub vaadetele (nende loomiseks on andmekirjelduskeele lause CREATE VIEW) ning nende põhjal tehtavatele päringutele (selleks on andmekäitluskeele lause SELECT). Vaade on nime omav tuletatud tabel, mis luuakse teiste andmebaasi tabelite põhjal. Nendeks tabeliteks võivad olla teised vaated või baastabelid. Vaate spetsifikatsioonis sisaldub vaate alampäring. See päring täidetakse ning leitakse vaates olevad andmed selle hetkel, kui keegi soovib vaates olevaid andmeid näha. Baastabel on nimega tabel, mis ei ole defineeritud teiste tabelite põhjal.

Andmebaasi vaated võimaldavad varjata andmebaasi skeemi keerukust ja lihtsustada rakenduse arendamist, kuid võivad ka põhjustada andmebaasisüsteemi poolt mitteoptimaalsete täitmisplaanide valikut.

1.1 Taust ja probleem

Antud töö abil on võimalik leida vastused küsimustele, mis puudutavad vaadete põhjal tehtavate päringute andmebaasisüsteemi poolse täitmise printsiipe. Töös uuritakse selleks muuhulgas päringute täitmisplaane, saamaks aru, kuidas andmebaasisüsteem sisemiselt ühte või teist lauset täidab.

Otsisin samateemalisi eksperimente, kuid leidis ainult väikesemahulisi katsetusi, mida kirjeldatakse veebipäevikute sissekannetes. Käesolevas töös soovitakse viia läbi mahukamaid eksperimente ning katsetada ka selliseid päringuid, mida olemasolevates uuringutes pole käsitletud.

Töö on vajalik infosüsteemide arhitektidele, sest aitab neil otsustada kas arhitektuuriline lähenemine, mille korral SQL-andmebaasist andmete otsimine toimub põhiliselt vaadete kaudu, on mõistlik kasutada või mitte. Sellisel arhitektuurilisel lähenemisel on minu hinnangul mitmeid eeliseid, kuid vajalik on aru saada, kuivõrd võib selle kasutamine mõjuda negatiivselt päringute täitmise kiirusele.

Töö on samuti vajalik andmebaasi administraatoritele, disaineritele ja arendajatele, kuna annab põhjalikku ülevaade sellest, mis juhtudel ja mis põhjustel võivad andmebaasisüsteemid valida vaadete põhjal tehtud päringutele mitteoptimaalse täitmisplaani. Kuigi töös vaadeldakse ainult kahte SQL-andmebaasisüsteemi on seda kasulik lugeda ka teiste andmebaasisüsteemide kasutajatel, sest pakub neile võrdlusmaterjali, mille alusel enda kasutatavat andmebaasisüsteemi hinnata ja selles eksperimente läbi viia.

1.2 Ülesande püstitus

Käesoleva töö uurimisküsimuseks on, kuidas mõjutab vaadete kasutamine päringute täitmisplaanide valikut erinevates andmebaasisüsteemides. Vaadates uurimisküsimust lähemalt, siis on tööl järgmised eesmärgid.

- Uurida erinevusi täitmisplaanides, kui päring on tehtud vaate põhjal ja kui loogiliselt samaväärne päring on tehtud otse baastabelite põhjal. Mõningatel juhtudel võivad nende lausete täitmisplaanid erineda. Seega on alameesmärgid:
    - uurida, millest see erinevus tuleneb, ehk erinevusi põhjustavaid faktoreid,

    - uurida, kuidas erineb täitmise kiirus loogiliselt samaväärsete lausete puhul, milles üks tehakse vaate põhjal ja teine otse baastabelite põhjal.

- PostgreSQL ja Oracle Database andmebaasisüsteemide võrdlemine vaadete põhjal päringute tegemise seisukohast. Need andmebaasisüsteemid olid ülesande püstitusele ette antud, sest tegemist on andmebaasisüsteemidega, mida olen TTÜs õppetöös kasutanud. Lisaks kasutan Oracle Database andmebaasisüsteemi oma igapäevases töös. Kuna töö eksperimendi osas tehakse katseid kahe andmebaasisüsteemiga ja nende kahe versiooniga, siis peaksid töö tulemused ka näitama seda, kuidas on arenenud andmebaasisüsteemide optimeerimismoodul (st kas probleemid varasema versiooni optimeerimismoodulis on uuema versiooni moodulis lahendatud või on hoopis tekkinud uusi probleeme).

Töö aitab aru saada, milliste vaadete ja päringute korral on töökiiruse probleemid kõige suuremad. See omakorda võimaldab vaadete ja päringute kirjutajatel luua paremaid vaateid ja päringuid vaadete põhjal. Veel üheks töö alameesmärgiks on iga teoreetilise alampeatükki kohta esitada mõistekaart, et aidata lugejatel teemavaldkonda paremini mõista.

1.3 Metoodika

Töö eesmärkide saavutamiseks ja uurimisküsimustele vastuste leidmiseks kavandatakse andmebaas – kavandatakse baastabelid, primaar-, välisvõtmed, unikaalsuse kitsendused, indeksid ja vaated. Seejärel andmebaas realiseeritakse, lisatakse testandmed, valitakse päringud, proovitakse teha päringuid otse baastabelite põhjal ja vaadete põhjal ja seejärel otsitakse erinevusi päringute täitmisplaanides ja nende põhjuseid.

Töö põhilisteks andmebaasisüsteemideks on Oracle Database 12.1 ja PostgreSQL 9.3. ning samas viiakse läbi eksperimendi ka nende andmebaasisüsteemide varasemates versioonides Oracle Database 11.1 ja PostgreSQL 9.1. Valitud on mitte vahetult eelmine versioon, vaid üle-eelmine versioon.

1.4 Ülevaade tööst

Peale sissejuhatust tutvustatakse teoreetilist tausta. Kõigepealt kirjutatakse lühidalt andmebaasikeelest SQL ja selle andmekäitluskeelest. Seejärel tutvustatakse andmekäitluskeele

lausete täitmisplaane, andmebaasi virtuaalset andmete kihti, vaateid ning nende kasutamise eeliseid ja puuduseid ning käsitletakse lühidalt peamiseid sarnaseid eksperimente.

Järgnevalt kirjeldatakse eksperimenti, selle uurimisküsimusi ja eesmärke. Tutvustatakse eksperimendis kasutatavaid andmebaasisüsteeme, nende versioone ning teisi tehnilisi andmeid. Antakse ülevaate andmebaasi ülesehitusest ja testandmete genereerimisest. Lisaks kirjeldatakse valitud vaadete sisu ja valiku põhjuseid.

Lõpuks kirjeldatakse eksperimendi tulemusi, tehakse järeldusi ja kokkuvõtteid.

2. Teoreetiline taust

Selles peatükis tutvustatakse töös käsitletavaid põhimõisteid, et muuta töö lugejale paremini arusaadavaks ning ta selle töö temaatikasse järk-järgult sisse juhatada.

2.1 SQL ja selle andmekäitluskeel

SQL (*Structured Query Language*) on andmebaasikeel, mille üheks alamkeeleks on DML (*Data Manipulation Language*) ehk andmekäitluskeel, mis võimaldab andmebaasist andmeid otsida, lisada, muuta ja kustutada. Sellesse alamkeelde kuuluvad *SELECT*, *INSERT*, *UPDATE*, *DELETE* ja *MERGE* laused [2].

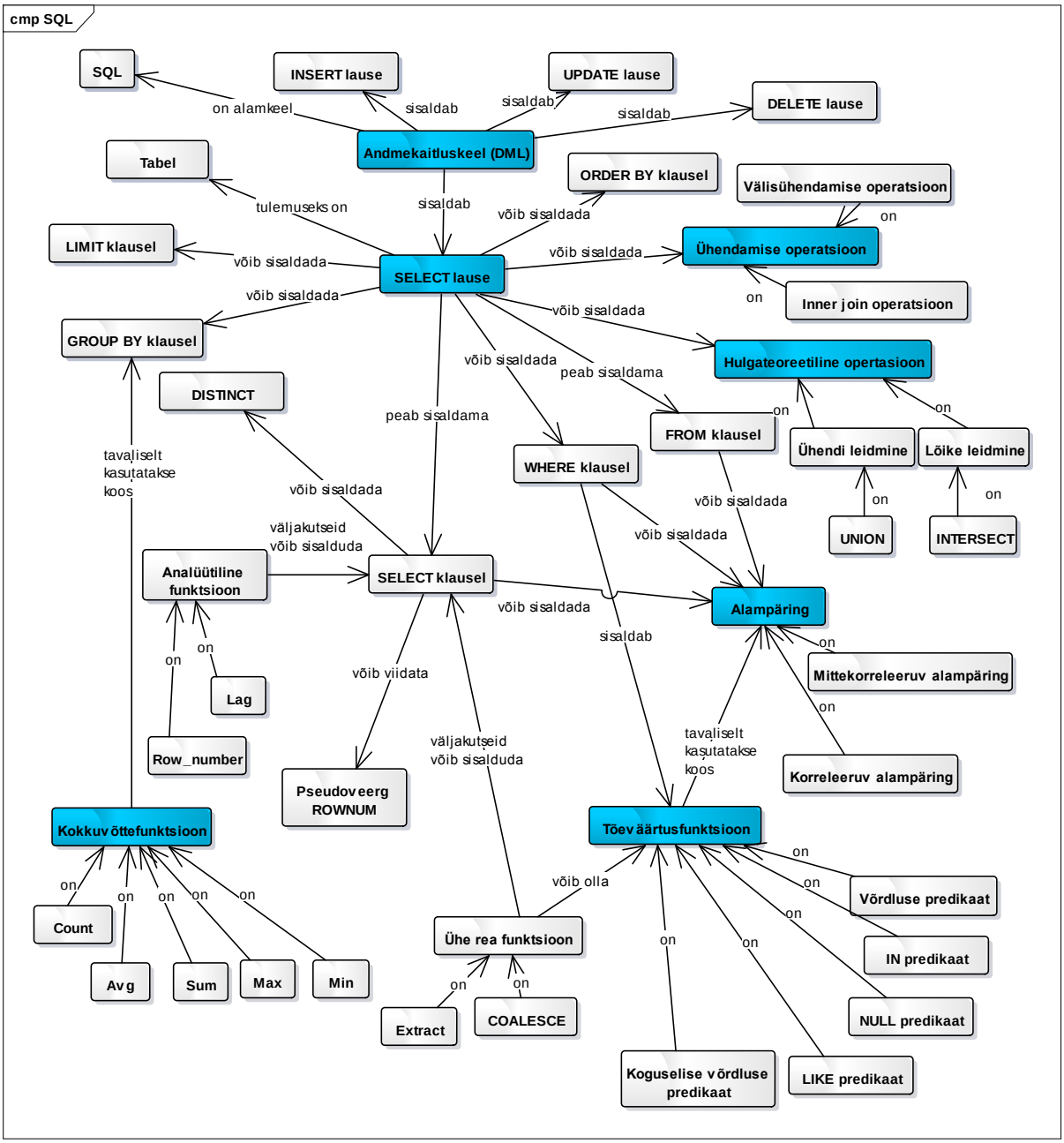
Andmete andmebaasist otsimiseks kasutatakse *SELECT* lauset (edaspidi kasutatakse *SELECT* lause asemel termini „päring“), mille põhilisteks osadeks on *SELECT – FROM – WHERE – GROUP BY – HAVING – ORDER BY* ja tulemuseks on tabel [3].

Järgnevalt nimetatakse olulisemaid *SELECT* lausete osi, mida kasutatakse käesoleva töö eksperimentide osas (vt Eksperimendi kirjeldus). Neid konstruktsioone saab kasutada nii baastabelite põhjal tehtavates nimetutes päringutes kui ka vaadete alampäringutes. Kui ei ole öeldud teisiti, siis saab nimetatud lausekonstruktsioone kasutada nii PostgreSQL kui Oracle Database andmebaasisüsteemis. See loetelu ei esita ammendavat SQL andmekäitluskeele kirjeldust, kuid peaks lugejani viima arusaamise, et SQL andmekäitluskeel on võimalusterohke ja ka keerukas.

- *SELECT* klausli tulemusest korduvaid ridu eemaldav *DISTINCT*;
- Pseudoveerus *ROWNUM* (Oracle Database andmebaasisüsteemi spetsiifiline) olev väärtus näitab iga rea korral tabelist rea lugemise järjekorranumbrit. Konstruktsiooni *FETCH FIRST … ROWS ONLY* on võimalik kasutada väljastavate ridade arvu piiramiseks (Oracle Database andmebaasisüsteemis alates versioonis 12c Release 1, PostgreSQL andmebaasisüsteemis alates versioonis 8.4). PostgreSQL andmebaasisüsteemis saab ridade hulga piiramiseks kasutada ka selle andmebaasisüsteemi-spetsiifilist *LIMIT* klauslit;
- *WHERE* klauslis saab kasutada predikaate (tõeväärtusfunktsioone) nagu võrdluse (*>; >= ; <=; <; =; <>*), in (*[NOT] IN*), like (*[NOT] LIKE*), null (*IS [NOT] NULL*), koguselise võrdluse (*>; >= ; <=; <; =; <> ALL* | *ANY* | *SOME*) predikaadid;

- Ridade sorteerimine (*ORDER BY*) kasvavas (*ASC*) ja kahanevas järjekorras (*DESC*);
- Funktsioonid:
    - Kokkuvõttefunktsioonid, mis leiavad väärtuse ridade hulga põhjal. Nende näideteks on funktsioonid *Avg* (aritmeetilise keskmise leidmine), *Sum* (summa leidmine), *Max* (maksimaalne väärtuse leidmine), *Min* (minimaalse väärtuse leidmine) ja *Count* (koguse/ ridade arvu leidmine). Sageli kasutatakse kokkuvõttefunktsioone koos *GROUP BY* klausliga;
    - Ühe rea funktsioonid, mille korral rakendatakse funktsiooni eraldi igas tabeli reas olevale väärtusele. Nende näideteks on funktsioon alamkomponentide eraldamiseks kuupäevast/kellaajast *EXTRACT(YEAR FROM {kuupäev})* ja *NULL*ide asendamisega seotud funktsioon *COALESCE* (selle poole pöördumisel peab ette andma üks või rohkem argumenti, funktsioon tagastab vasakult lugedes esimese mitte *NULL*ise argumendi); [3]
    - Analüütilised funktsioonid (aknafunktsioonid (*WINDOW*)). Nende näideteks on *LAG() OVER …*, mis võimaldab ligipääsu teatud nihkega reale (lähtudes jooksvast reast) ja *ROW_NUMBER() OVER …*, mis seob iga reaga millele see rakendatakse unikaalse väärtuse;
- Keerukamad *SELECT* laused:
    - Hulgateoreetilised operatsioonid nagu ühendi leidmine *UNION* ja lõike leidmine *INTERSECT*;
    - Tabelite ühendamine: *INNER JOIN*, välisühendamine *LEFT (RIGHT) OUTER JOIN*;
    - Korreleeruv ja mittekorreleeruv alampäring kasutades *IN* predikaati või võrdluse (*>; >= ; <=; <; =*) predikaati;
    - Rekursiivne päring *CONNECT BY*. [1]

Järgnevalt esitatakse mõningad teema põhimõisted (vt Joonis 1):



Joonis 1. Mõistekaart - SQL ja selle andmekäitluskeel

2.2 Andmekäitluskeele lause töötlemine ehk täitmisplaanid

Paljud andmebaasisüsteemid võimaldavad kasutajatel kirjutada päringuid kasutades deklaratiivset keelt nagu näiteks SQL. Kasutades sellist keelt kirjutatakse, *millist* tulemust soovitakse (mida peab süsteem leidma), kuid mitte, seda *kuidas* soovitud tulemuseni jõuda. Viimati mainitud ülesanne on delegeeritud päringu optimeerijale (optimeerimismoodulile) – see on andmebaasisüsteemi komponent, mis vastutab efektiivse täitmisplaani leidmise eest. Täitmisplaan kirjeldab lause täitmiseks kasutatavat algoritmi. Enamasti on iga lause korral

võimalik valida erinevate täitmisplaanide vahel (mis annavad kõik sama tulemuse). Optimeerija peab üritama leidma plaani, mis on optimeerimise eesmärki silmas pidades kõige parem (või vähemasti parim piisavalt suure hulga kaalutud plaanide seast). Optimeerimise eesmärgiks võib näiteks olla, et kogu lause tuleb täita võimalikult kiiresti. Andmebaasisüsteem võib aga ei pruugi võimaldada optimeerimise eesmärki muuta [4].

Päringute optimeerija toimib sellisel viisil, et koostab hulga alternatiivseid täitmisplaane, aga valib täitmiseks ühe täitmisplaani, mis on väikseima maksumusega. Maksumuse arvutamiseks on andmebaasisüsteemil vaja informatsiooni andmebaasis olevate andmete hulga, paigutuse ja jaotuse kohta. Selliseid andmeid nimetatakse andmebaasi statistikaks. Selleks, et andmebaasisüsteem oskaks valida parima plaani, peab selle käsutuses olev lähteinformatsioon olema võimalikult täpne, st statistika peab olema värske [4].

SQL andmekäitluskeele lausele vastamiseks läbib andmebaasisüsteem järgmised etapid.

- Dekompositsioon – analüüsitakse lause süntaksi ning semantikat ja viiakse lause andmebaasisüsteemile arusaadavamale kujule. Selle tulemuseks on lause esitus puustruktuurina, mida nimetatakse analüüsi puuks.

- Loogilise täitmisplaani koostamine ja parandamine – analüüsipuu põhjal koostatakse relatsioonialgebra puu ning üritatakse muuta selles operatsioonide järjestust (optimeerida) nii, et tulemus oleks loogiliselt samaväärne, kuid andmebaasisüsteem peaks operatsioonide läbiviimisel tegema võimalikult vähe tööd.

- Füüsiliste täitmisplaanide koostamine ja parima täitmisplaani valik – sisendiks on relatsioonialgebra puu [4] ning väljundiks on füüsiline täitmisplaan, mille alusel toimub lause täitmine. Kui edaspidi viidatakse mõistele *täitmisplaan*, siis peetakse silmas *füüsilist täitmisplaani*, kui ei ole öeldud teisiti.

Dekompositsiooni etapis toimub ka päringus viidatud vaadete lahtikirjutamine ja asendamine baastabelite poole pöördumistega. Kui täidetakse päring, mis viitab vaatele, siis toimub vaate dekompositsiooni protsess, mille käigus asendatakse päring samaväärse päringuga, mis viitab ainult baastabelitele. Sisuliselt asendatakse päringus kasutatavad viited vaate definitsiooniga (alampäringuga) [5]:

Vaade:

```
CREATE VIEW vw_Organization_Info ( party_id, organization_name, registration_code )
AS
SELECT party_id, name, registration_code
FROM Organization;
```

Päring vaate põhjal:

```
SELECT organization_name, registration_code
FROM vw_Organization_Info
WHERE registration_code LIKE '1%';
```

Viide vaatele asendatakse vaate definitsiooniga (alampäringuga):

```
SELECT organization_name, registration_code
FROM (SELECT party_id, name, registration_code
      FROM Organization)
WHERE registration_code LIKE '1%';
```

Vaate alampäring ja päring vaate põhjal „surutakse kokku“ e mestitakse – nendest moodustatakse uus päring, kus FROM klauslis pole alampäringut (*inline view*). Päring, mida tegelikult täidetakse:

```
SELECT name, registration_code
FROM Organization
WHERE registration_code LIKE '1%';
```

Vaatele viitava päringu füüsilise täitmisplaani näide Oracle Database andmebaasisüsteemis (vt Joonis 2) :

```
Plan hash value: 2518513461

-----------------------------------------------------------------------------------------------------------
| Id  | Operation         | Name         | Starts | E-Rows |E-Bytes| Cost (%CPU)| A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |              |      1 |        |       |    68 (100)|   3279 |00:00:00.01 |     377 |
|*  1 |  TABLE ACCESS FULL| ORGANIZATION |      1 |   3031 | 63651 |    68   (0)|   3279 |00:00:00.01 |     377 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("REGISTRATION_CODE" LIKE '1%')
```

Joonis 2. Päringu füüsiline täitmisplaan Oracle Database andmebaasisüsteemi näitel

Andmekäitluskeele lauset täidetakse üldiselt järgmises *järjekorras FROM – WHERE – GROUP BY – HAVING – SELECT – ORDER BY* [3].

Joonisel esitatakse täitmisplaan, kus punasega märgitud on hinnangulised väärtused, rohelisega – tegelikud ja sinisega – kumulatiivsed ehk kogunenud (vt Joonis 2).

Täitmisplaan sisaldab :

- *E-Rows* – hinnang iga operatsiooni tulemusena tagastatavate ridade arvule;
- *E-Bytes* – hinnang iga operatsiooni tulemusena leitavate ridade keskmisele suurusele baitides;
- *A-Rows* – iga operatsiooni tulemusena tagastatav tegelik ridade arv;
- *Cost* – hinnang operatsiooni (sammu) maksumusele. Täitmisplaani kogumaksumus esitatakse reas identifikaatoriga 0;
- *Starts* – operatsiooni (sammu) käivitamiste arv;
- *A-Time* (kumulatiivne) – tegelik summaarne aeg, mis on kulutud antud operatsiooni ja kõigi sellele eelnenud operatsioonide peale (kasutatakse antud töös päringute täitmise kiiruse võrdlemiseks);
- Operatsioon *TABLE ACCESS FULL* tähendab baastabeli kõikide allpool kõrgveemärki olevate (st kasutuses olevate) plokkide lugemist [4]. Sellest võib mõelda kui raamatu kogu sisu otsast peale läbilugemisest ilma raamatu indeksit lugemata.

Enamasti saavad kasutajad ühe ja sama ülesande lahendamiseks kirjutada SQLis rohkem kui ühe päringu, mis seda ülesannet lahendab. Vähegi keerulisema ülesande puhul võib selliseid päringuid olla kümneid. Üheltpoolt annab taoline keele „paindlikus“ päringute kirjutajatele võimaluse kavalaid lahendusi nuputada, kuid teiselt poolt muudab optimeerimismooduli töö raskemaks. Igale sellisele päringule koostab andmebaasisüsteem täitmisplaani. Ideaalis võiks andmebaasisüsteem aru saada, et hulk erinevaid päringuid lahendavad tegelikult sama ülesannet ja pakkuda neile välja ühesuguse täitmisplaani. Praktikas seda ei juhtu ja erinevatel lausetel on erinevad täitmisplaanid. Seega peab päringute kirjutaja hakkama uurima, millisel viisil tuleks tema valitud andmebaasisüsteemis päringuid kirjutada, et lauseid võimalikult kiiresti täidetakse. See muudab päringute kirjutaja töö raskemaks ja ebaefektiivsemaks.

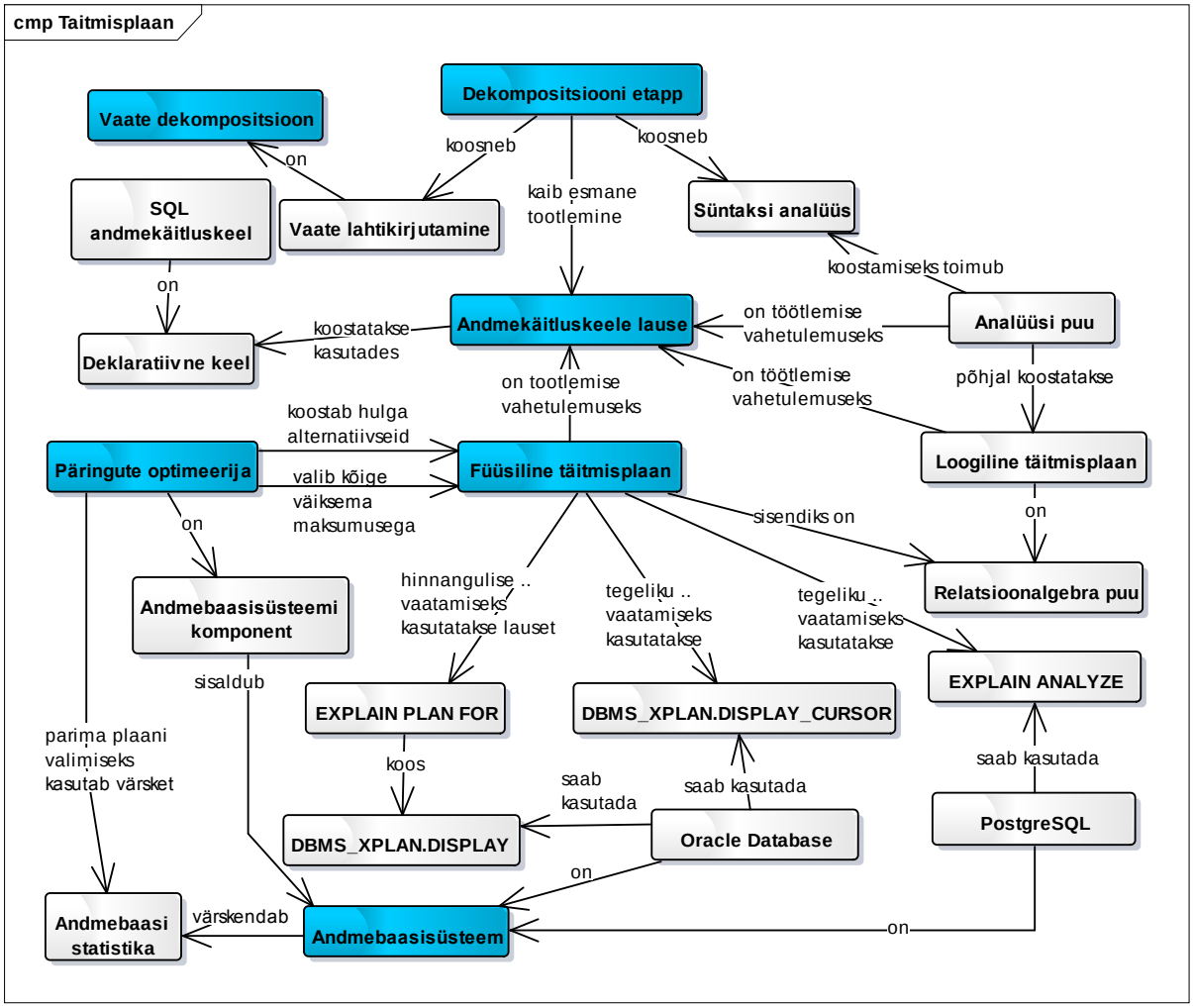
Selle töö kontekstis mõeldakse ühe ja sama ülesande erinevate lahenduste all päringute tegemist otse baastabelite põhjal ja päringute tegemist vaadete põhjal. Mõnikord proovitakse ka ühe ja sama ülesande lahendamiseks mitut erinevat päringut kas baastabelite või vaadete põhjal.

Täitmisplaanide vaatamiseks tuleb teha järgnevat.

- Oracle Database andmebaasisüsteemis võib kasutada lauset *EXPLAIN PLAN FOR* ja *DBMS_XPLAN.DISPLAY* funktsiooni. Tuleb teada, et *EXPLAIN PLAN* võib näidata ebaõiget päringu täitmisplaani [6], kuna *EXPLAIN PLAN* täitmisel ei käivitata tegelikult päringut. *EXPLAIN PLAN* annab tulemuseks andmebaasisüsteemi *hinnangu* (prognoosi), milline peaks täitmisplaan olema, kuid selle prognoosi koostamisel pole andmebaasisüsteemil kogu seda infot, mida see kasutab tegelikult täidetava lause täitmisplaani koostamisel. Selle tulemusena võivad prognoositav täitmisplaan ja tegelik täitmisplaan üksteisest erineda. Seetõttu kasutatakse antud töös *DBMS_XPLAN.DISPLAY_CURSOR* funktsiooni*;*

- PostgreSQL andmebaasisüsteemis kasutatakse *EXPLAIN ANALYZE* käsu. *ANALYZE* valik põhjustab päringu tegeliku täitmise ning siis näidatakse lause tegeliku täitmise statistikat, sealhulgas iga täitmisplaanis oleva operatsiooni peale kulutatud koguaega (millisekundites) ja tagastatud ridade arvu [7].

Järgnevalt esitatakse mõningad teema põhimõisted (vt Joonis 3):



Joonis 3. Mõistekaart - Täitmisplaanid

2.3 Virtuaalne andmekiht

Andmebaasi (ja vastavalt ka andmebaasisüsteemi) arhitektuuri võib jagada kolmeks kihiks: sisemine, kontseptuaalne ja väline. SQL-andmebaasi kontseptuaalses kihis paikneb andmebaasi kontseptuaalne skeem, kuhu kuuluvad baastabelid ja nendele rakenduvad kitsendused ning mis sisaldab andmebaasi loogilist kirjeldust [8].

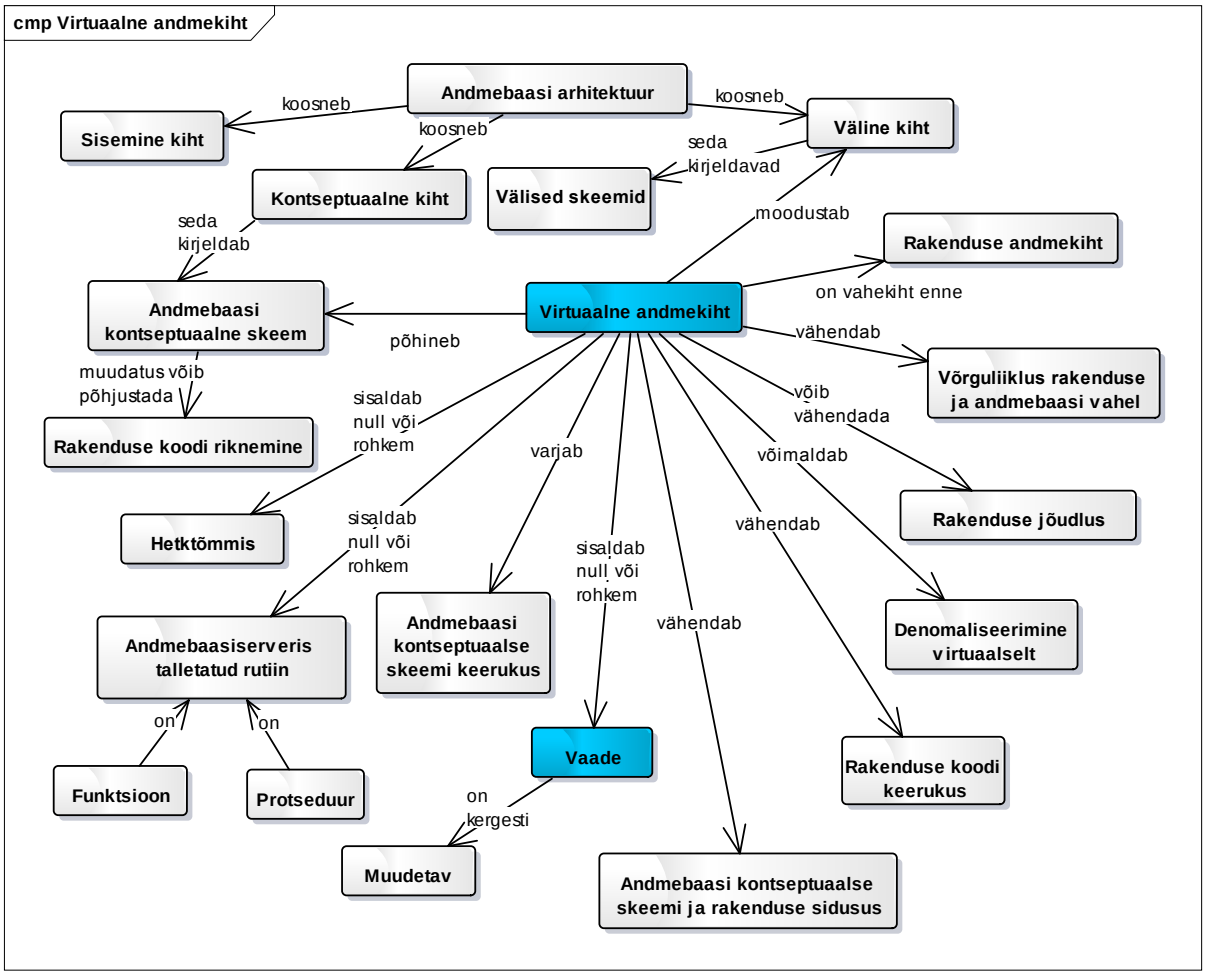
Iga andmebaasi kontseptuaalse skeemi muudatus (SQL-andmebaasi korral muudatus baastabelite struktuuris) võib põhjustada rakenduse koodi riknemise (st muutub see, kuidas kasutajad peavad andmebaasi poole pöörduma. Kasutajaks on tavaliselt rakendus ning selliseid kasutajaid võib olla rohkem kui üks). Teisalt ei tohi andmebaasi skeemi muuta rakendus-spetsiifiliseks, kuna see tähendaks andmebaasi skeemi muutmist liiga sõltuvaks ainult ühe kasutaja nõudmistest. Need on kaks olulist faktorit, mis raskendavad andmebaasi

administraatorite ja rakenduse ning andmebaasi arendajate tööd. Seega tuleb kontseptuaalse skeemi disainimisel arvestada kahe asjaoluga: milline on vastuvõetav rakenduse andmekihi ja andmebaasi skeemi sidestuse aste, ning kui lihtsalt peab olema võimalik teha muudatusi [9, p. 112].

Burns [9, p. 110] väidab, et need probleemid on võimalik lahendada virtuaalse andmekihi (*Virtual Data Layer VDL*) abil, mis paikneb andmebaasi (kontseptuaalse) skeemi (baastabelite) ja rakenduse andmekihi vahel. Virtuaalne andmekiht vastab eelnevalt nimetatud arhitektuuris olevale välisele kihile. Virtuaalne andmekiht koosneb andmebaasiobjektidest, mille eesmärk on varjata rakenduste eest andmebaasi kontseptuaalse skeemi keerukust ning võimaldada rakendusel kergesti andmeid kasutada ja uuendada [9, p. 110]. Virtuaalse andmekihi moodustavad vaated, hetktõmmised (materialiseeritud vaated) ning andmebaasiserveris talletatud rutiinid (funktsioonid ja protseduurid). Käesolevas töös keskendutakse vaadetele („tavalistele“, mitte „materialiseeritud“ vaadetele). Vaade võimaldab saavutada (osalise) loogilise andmete sõltumatuse [8], st andmebaasi kontseptuaalses skeemis tehtud muudatused mõjutavad rakenduse koodi võimalikult vähe. Burns [9, p. 112] märgib, et sageli saab teha andmebaasi struktuuris muudatusi nii, et kui andmebaasi kontseptuaalne skeem muutub, siis muudetakse ainult vaate definitsiooni (alampäringut), kuid see ei mõjuta vaate kasutajaid (rakendusi). Teiste sõnadega, kasutaja näeb läbi muudetud vaate endiselt samu andmeid [8]. Kui vaatesse on vajalik lisada uus veerg või esitada vaate kaudu teisendatud väärtuseid, siis seda saab teha jooksvalt, mõjutamata rakendusi, mis juba kasutavad seda vaadet. Andmebaasi väline kiht kirjeldatakse ühe või mitme välise skeemi abil, st igale kasutajale (nt rakendusele) saab luua oma vaadete ja teiste välise skeemi objektide komplekti, mille kaudu kasutajatele andmeid serveeritakse ja lubatakse andmebaasis tegevusi teha.

Virtuaalse andmekihi loomisel on mitmeid eeliseid. Näiteks aitab see isoleerida rakendusi andmebaasi kontseptuaalses skeemis tehtud muudatustest; vähendab andmebaasi kontseptuaalse skeemi ja rakenduse vahelist sidusust; vähendab rakenduse koodi keerukust; vähendab võrguliiklust, kuna rakenduse ja andmebaasi vahel peab liikuma vähem andmeid [9, p. 111]. Vaated võimaldavad esitada rakendustele andmeid denormaliseeritult – vormis ja viisil, mis on kõige sobivamad konkreetsele rakendusele, kuid samas hoida andmebaasi kontseptuaalset skeemi kõrge tasemeni normaliseerituna koos kõigi sellest tulenevate eelistega.

Järgnevalt esitatakse mõningad teema põhimõisted (vt Joonis 4):



Joonis 4. Mõistekaart - Virtuaalne andmekiht

2.4 Vaated

Järgnevalt tutvustatakse selle töö keskset mõistet – vaade. Vaadete loomine on võimalik erinevate andmemudelite korral, kuid käesolevas töös mõeldakse vaateid SQL-andmebaasides.

2.4.1 Vaadete definitsioon

Andmebaasi kontseptuaalset skeemi ja väliseid skeeme moodustavad tabelid jagunevad tuletatud tabeliteks ja baastabeliteks. Sellistest tabelitest võib mõelda kui muutujatest, mis omavad erinevatel ajahetkedel erinevat väärtust. Baastabel on nimega tabel, mis ei ole defineeritud teiste tabelite põhjal. Vaade e virtuaalne tabel on nime omav tuletatud tabel. „Tuletatud“ tähendab seda, et vaade defineeritakse teiste tabelite põhjal. Andmebaasisüsteem arvutab vaate väärtuse välja peale seda, kui kasutaja seda väärtust küsib. Selle poolest erinevad vaated hetktõmmistest (tuntakse ka kui materialiseeritud vaateid). Hetktõmmise väärtus

arvutatakse välja (ja salvestatakse eraldiseisva füüsilise koopiana) juba enne, kui kasutaja seda väärtust küsib. See võimaldab väärtuse kasutajale kiiresti tagastada, mis parandab süsteemi jõudlust. Hetktõmmiseid (erinevatest vaadetest) kasutatakse süsteemi jõudluse parandamiseks. Käesolevas töös hetktõmmiseid ei uurita.

Vaate definitsioonis kirjeldatav alampäring on päring. Päring täidetakse kui vaatest küsitakse andmeid. Päringu tulemusena leitakse virtuaalses tabelis olevad read [8].

Groff ja Weinberg sõnul „Vaade loob baastabeli kasutamise illusiooni kuna vaatel on nimi nagu baastabelilgi ning andmebaasis hoitakse ainult vaate definitsiooni.“ [10, p. 410]. Seega andmebaasi kasutaja ei tea, kas ta kasutab virtuaalset tabelit või baastabelit [8].

Järgnevalt esitatakse vaate loomise näide Oracle Database andmebaasisüsteemis. Vaate nimi on *vw_Organization_Info* ja selle alampäringu tulemuseks on kõikide organisatsioonide identifikaatorid, nimed ja registrikoodid:

```
CREATE VIEW vw_Organization_Info ( party_id, organization_name, registration_code )
AS
SELECT party_id, name, registration_code
FROM Organization
WITH CHECK OPTION;
```

WITH CHECK OPTION kitsendus tähendab, et andmete uuendamise ja lisamise puhul hakatakse kontrollima andmete vastavust vaate alampäringu tingimustele [9]. Andmemuudatus sellise vaate kaudu peab rahuldama vaate alampäringu tingimust.

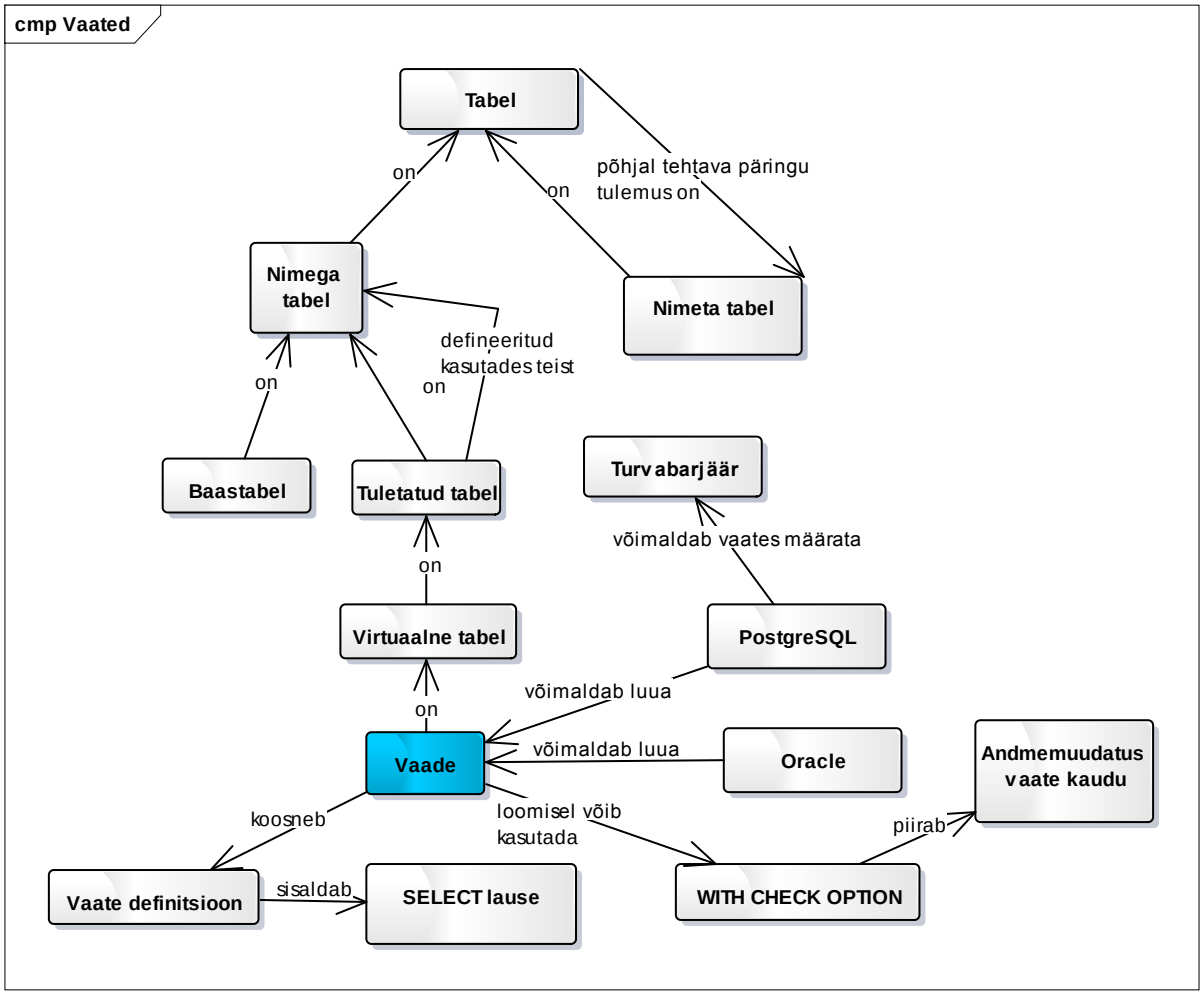
Kuna eksperimendis ei teostata andmete uuendamisi ja lisamisi läbi vaadete, siis seda valikut vaadete loomisel ei kasutata. Samas on seda väga soovitatav kasutada vaadete puhul, mille kaudu soovitakse andmeid muuta.

Vaateid saab kasutada selleks, et näidata kasutajatele neid ja ainult neid andmeid, mis kasutajatel on oma tööks vajalikud. PostgreSQLis tuleb selle juures arvestada võimalusega, et vaikimisi saab vaatesse „sisse häkkida“ ja näha ka selliseid andmeid, mida vaade välja ei too. Selle vältimiseks tuleb PostgreSQLis luua vaade *WITH (security_barrier)* määranguga. Kuid see omakorda seab piirangud „*view merging*“ (vaate mestimise) protsessi läbiviimisele (ehk vaate alampäringuid täidetakse eraldi). *WITH (security_barrier)* määrangu mõjust vaadete põhjal tehtavatele päringutele tuleb edaspidi juttu.

Andmebaasisüsteem võib kasutada vaadete põhjal tehtud päringute täitmiseks erinevaid meetodeid.

- Luuakse päringu ja vaate definitsioonis kirjeldatud päringu põhjal uus päring;
    - Päring ja vaate alampäring surutakse „täielikult kokku“ e mestitakse (*view merging*)
    - Kui mestimine pole võimalik, siis võib andmebaasisüsteem suruda päringu tingimuses oleva predikaadi vaatest tulenevasse alampäringusse (*predicate pushing*) [11] [12]. Selle taga on üldine strateegia, mille kohaselt oleks andmete otsimisel mõistlik rakendada piiravad tingimused võimalikult varakult, et töödeldavate andmete hulka kohe ja palju vähendada.
- Vaate põhjal luuakse ajutine tabel ning päring täidetakse kasutades loodud ajutist tabeli.

Järgnevalt esitatakse mõningad teema põhimõisted (vt Joonis 5):



Joonis 5. Mõistekaart - Vaated

2.4.2 Vaadete kasutamise eelised

Burns [9, pp. 172-173] nimetab vaateid kõige olulisemaks virtualiseerimise vahendiks ja kirjeldab mitmeid vaadete kasutamisest tulenevaid eeliseid.

- Võimalik on luua rakenduse-spetsiifilisi vaateid. Sellisel juhul ei ole vaja andmebaasi kontseptuaalset skeemi rakenduste vajadustest sõltuvat denormaliseerida. See aitab muuhulgas tagada andmete terviklikkust. Lisaks on vaate loomisel võimalik kasutada *WITH CHECK OPTION* kitsendust, mis ei luba teha vaate kaudu andmemuudatusi, mis lähevad vastuollu vaate alampäringu tingimustega.

- Aitavad vähendada rakenduse koodi ja andmebaasi kontseptuaalse skeemi vahelist sidustust (*coupling*). See tähendab, et kuna rakenduse koodis pöördutakse vaadete, mitte baastabelite poole, siis on võimalik teha muudatusi andmebaasi kontseptuaalses skeemis, mõjutamata sealjuures rakenduse koodi. Peale baastabelite struktuuri

muutmist tuleb teha vastavaid täiendusi muudetud baastabelitega seotud vaadetes. Näiteks, kui andmebaasi kontseptuaalses skeemis muudetakse baastabeli veeru nime, siis tuleb ka seda veergu kasutavas vaates muuta vaate alampäringus seda nime. Selline võimalus on eriti kasulik siis, kui rakenduse koodile pole ligipääsu ja seda ei saa muuta.

- Vaadete veergude nimed võivad olla rakenduse-spetsiifilised, mis võimaldab ära peita baastabelite veergude tegelikud nimed.

- Vaadete abil näeb kasutaja tema jaoks terviklikku ja muutumatut pilti andmebaasi struktuurist isegi siis, kui vaadete aluseks olevate baastabelite struktuur on muutunud [8].

- Vaated võimaldavad kasutajatel kirjutada lihtsaid päringuid vaadete põhjal, selle asemel, et kirjutada keerulisemaid päringuid baastabelite põhjal. Selles mõttes on vaated nagu makrod, mis võimaldavad keerukaid ülesandeid kiiresti täita. Vaate alampäring võib olla keerukas, kuid vaade varjab kasutaja ees selle keerukuse. Keerukamad päringud, kus näiteks ühendatakse kokku mitmes erinevas baastabelis olevad andmed, saab defineerida vaadetena. Kokkuvõttes võimaldab see kasutajatel kiiremini päringuid kirjutada. See võib ka vähendada võrguliiklust andmebaasi ja rakenduse vahel. See juhtub siis kui enne vaadete kasutusele võtmist teeb rakendus palju päringuid baastabelite põhjal ja ühendab erinevates tabelites olevad andmed ise kokku. Vaadete kasutuselevõtu järel saab rakendus teha lihtsa päringu vaate põhjal kuid erinevates baastabelites olevate andmete kokku otsimine ja rakendusele sobiva vastuse formeerimine jääb andmebaasisüsteemi ülesandeks.

- Läbi teatud tingimustele vastavate vaadete on võimalik (ilma vaate täiendava programmeerimiseta) muuta baastabelites olevaid andmeid. SQL seab sellistele vaadetele küllaltki palju piiranguid. Samas pakub SQL ka võimalust kuidas programmeerida vaade ümber nii, et selle kaudu saab andmeid muuta. Selleks saab nii Oracle Database kui PostgreSQL andmebaasisüsteemis kasutada *INSTEAD OF* trigereid. PostgreSQL andmebaasisüsteemis on lisaks võimalik kasutada andmebaasisüsteemi-spetsiifilisi andmekäitluskeele lausete ümberkirjutamise reegleid (*DO INSTEAD* reegleid).

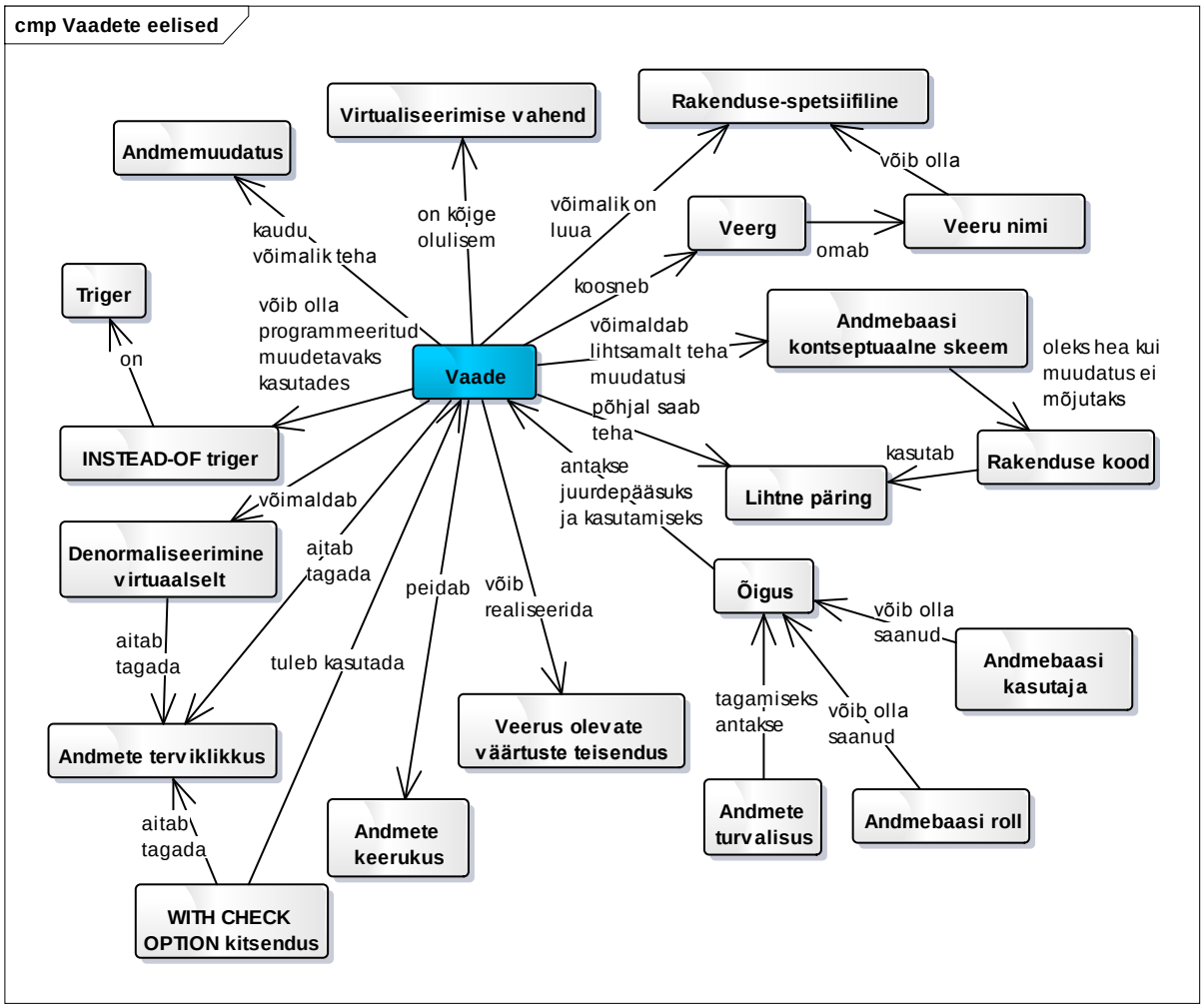
- Vaateid saab kasutada andmebaasile juurdepääsu piiramiseks ja selle kaudu andmete turvalisuse tagamiseks. Vaade võib näidata ja lubada muuta ainult neid andmeid, mis

kasutajale on tema tööks vajalikud ja peita teisi andmeid, mille nägemiseks ja muutmiseks kasutajal ei ole volitusi [8]. Siinkohal on jälle oluline mainida *WITH CHECK OPTION* kitsendust, mis keelab vaadete kaudu muudatused, mis pole kooskõlas vaate alampäringu tingimustega. Andmebaasi kasutajale või rollile saab anda õiguse kasutada vaadet, aga mitte vaate viidatavaid baastabeleid. Andmebaasi rollidele vastavad infosüsteemis defineeritud turvalisuse grupid, mis sisaldavad individuaalseid rakendusi ja kasutajate kontosid.

- PostgreSQL andmebaasisüsteemis on võimalik luua vaade koos turvabarjääriga, et tagada andmete turvalisust (vaade luuakse koos *WITH (security_barrier)* määranguga). PostgreSQL 9.3 ei saa sellisel juhul läbi vaadete muuta automaatselt andmeid baastabelites. Selle asemel pakutakse kasutada *INSTEAD OF* trigereid. Alates PostgreSQL 9.4 on võimalik automaatselt andmeid muuta ka läbi turvabarjääriga vaadete. Oracle Database andmebaasisüsteemis on kasutusel *optimizer_secure_view_merging* parameeter, mille väärtus on vaikimisi määratud tõeseks (*TRUE*). Antud parameeter aitab kontrollida kõiki päringute teisendusi (sealhulgas vaate mestimist (*view merging*)), mis võivad põhjustada turvaprobleeme [13, p. 290].

- Võimaldavad esitada andmeid kasutajale sobivale kujule teisendatuna. Näiteks võivad vaate alampäringud teisendada gramme kilogrammideks, meetreid sentimeetriteks või esitada andmeid koondatuna massiividesse, *XML (Extensible Markup Language)* või *JSON (JavaScript Object Notation)* dokumentidesse.

Järgnevalt esitatakse mõningad teema põhimõisted (vt Joonis 6):



Joonis 6. Mõistekaart - Vaadete eelised

2.4.3 SQLis vaadete kasutamisega seotud probleemid

Vaateid on võimalik luua teiste vaadete põhjal (*views of views*), kuid mitmetasemelised viitamised võivad muuta optimeerijale optimaalse (või vähemasti enamikest teistest plaanidest parema) täitmisplaani koostamise raskeks. Seega soovitab Burns [9, p. 175] vaadete alampäringutes viidata baastabelitele ning mitte vaadetele. Võimalike problemaatiliste olukordade eest hoiatab sellega seoses ka „Oracle Database Performance Tuning Guide“ [14], kus väidetakse, et ebasoovitav viis vaadete kasutamiseks on kui vaade viitab teistele vaadetele ja kui need on päringutes kokku ühendatud. Antud eksperimendis tehakse ka selliseid vaateid, et kontrollida kas need võivad põhjustada mitteoptimaalseid, ressursimahukaid päringuid.

SQLis on läbi vaadete andmete muutmisele (liiga) palju piiranguid.

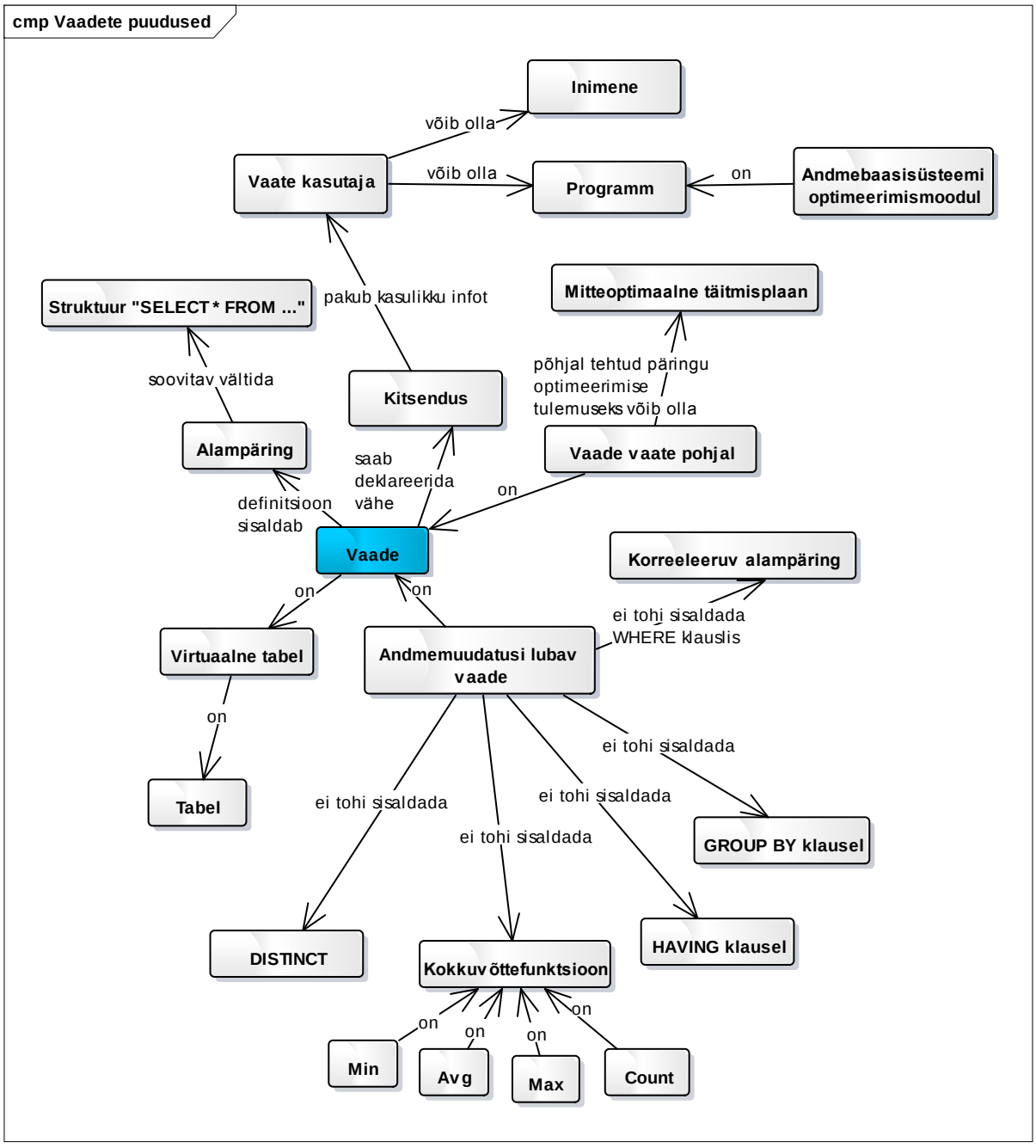
- Vaade ei tohi sisaldada *DISTINCT* klauslit;
- Ei tohi sisaldada kokkuvõttefunktsioone – *Avg(), Max(), Min()* ja teised;
- Ei tohi kasutada ühendi leidmist (*UNION DISTINCT*), lõike leidmist (*INTERSECT*), vahe leidmist (*EXCEPT*);
- Päringu *WHERE* klauslis ei tohi sisaldada korreleeruvat alampäringut;
- Ei tohi sisaldada grupeerimise *GROUP BY* või *HAVING* klauslit [8].

Lähtudes põhimõttest, et vaade peab kasutajale paistma nagu iga teine tabel, peaks palju suurem hulk vaateid lubama andmemuudatusi.

Vaatele on mitmesuguseid struktuurseid piiranguid. Kui defineerida vaate alampäring kujul *„SELECT * FROM …“,* siis * viitab veergudele, mis olid baastabelites vaate loomise hetkel. Kui hiljem lisatakse mõnda vaates viidatud baastabelisse uus veerg, siis see ei ilmu vaatesse enne, kui vaade kustutatakse ja taasluuakse [15, p. 186]. * kasutamist peaks vältima, sest vaate loomise lausest ei tule selgelt välja, millised veerud hakkavad vaates sisalduma ning erinevatel ajahetkedel käivitudes võib tulemuseks olla erineva struktuuriga vaade.

Vaade on tabel. Baastabelitele saab SQLis deklareerida mitmesuguseid kitsendusi. SQL standard ei näe ette vaadetele selliste kitsenduste (nt primaar-, välisvõtme või unikaalsuse kitsendused) deklareerimist. Tõsi, *WITH CHECK OPTION* määrangust võib mõelda kui vaadete-spetsiifilisest kitsendusest, mis on sarnane baastabelite *CHECK* kitsendusega. Oracle Database lubab vaadetele deklareerida baastabelitega analoogilisi kitsendusi, kuid ei jõusta neid. Vaate kitsendused luuakse Oracles ainult *DISABLE NOVALIDATE* režiimis, teisi režiime ei võimalda andmebaasisüsteem vaadete korral kasutada [16]. Nendest kitsendustest võib olla kasu programmidele (sh andmebaasisüsteemile endale), mis neid vaateid kasutavad, sest need annavad infot vaate kohta. Käesoleva töö eksperimentide osas loodavatele vaadetele katsetan selliste kitsenduste deklareerimist uurimaks nende mõju täitmisplaanide koostamisele.

Järgnevalt esitatakse mõningad teema põhimõisted (vt Joonis 7):



Joonis 7. Mõistekaart – SQLi vaadetega seotud probleemid

2.5 Sarnased eksperimendid

Ma ei ole leidnud teadusartikleid (uuringuid), mis uuriksid ja võrdleksid baastabelite ja vaadete põhjal tehtud päringute korral paljude erinevate päringute täitmisplaane. Siiski leidub väiksemaid eksperimente (fokuseerituid ühele probleemile), mille tulemustest raporteeritakse

veebipäevikutes või teadusartiklites. Järgnevalt viitan nendele eksperimentidele ja toon välja nende põhilised tulemused.

PostgreSQL.org veebilehel on olemas kirjavahetus [17], kus David Rowley väidab, et vaadete ja baastabelite põhjal tehtud päringute täitmisplaanid on erinevad. Päringus kasutatakse *WINDOW* funktsiooni *LAG() OVER()*. Probleem seisneb selles, et vaate põhjal tehtud päringu korral ei kasuta süsteem indeksit, mida kasutatakse baastabelite põhjal tehtud päringu korral. Vastuses väidetakse, et põhjuseks on vaate alampäringus *WINDOW* funktsiooni kasutamine.

Yan ja Larson [18] teadusartiklis uuritakse päringuid, kus sisalduvad tabelite ühendamine (*join*) ja grupeerimine (*GROUP BY*). Teadusartiklis uuritakse, millisel juhul viib andmebaasisüsteem päringu täitmisel esialgu läbi ühendamisoperatsiooni ja seejärel grupeerimisoperatsiooni ning vastupidi. Autorid [18] teevad erinevaid päringuid baastabelite ja ka vaadete põhjal.

Antud töös kirjeldan eksperimenti, kus vaate alampäringus ühendatakse baastabel grupeerimist kasutava vaate alampäringuga. Võrdluseks tehakse mitu päringut baastabelite põhjal. Andmebaasisüsteem võib kasutada vaadete põhjal tehtud päringute täitmiseks erinevaid meetodeid:

- Päringu ja vaate alampäringu põhjal luuakse uus päring. Seda nimetab Allison [19] vaate mestimiseks (*view merging*). Konkreetse näite korral toimuks grupeerimine viimasena.

- Vaate põhjal luuakse ajutine tabel ning tehakse päring kasutades loodud ajutist tabelit. Sellisel juhul toimub grupeerimine enne baastabelite ühendamist.

Testpäringutes kasutab Allison [19] vaadete asemel alampäringuid (mida nimetatakse ka *inline views*). Ning võrdlemiseks teeb ta päringuid baastabelite põhjal. Allison [19] seletab hästi teisendusi, mida Oracle Database andmebaasisüsteemi optimeerija teeb vaadete põhjal kirjutatud päringute korral. Nende teisenduste näiteks on vaate mestimine (*View Merging*). Arvatavasti hakkavad antud töös kirjeldatavad vaated Oracle Database andmebaasisüsteemis käituma nii nagu seda protsessi seletab Allison [19]. Seega vaadete ja otse baastabelite põhjal tehtavate päringute täitmisplaanid ei hakka erinema juhul, kui optimeerija on suutnud läbi viia vaate mestimise (*view merging*).

Allison [19] väidab, et Oracle Database andmebaasisüsteem suudab mestida päringuid erinevat tüüpi vaadetega.

- Lihtsad vaated (*Simple view*). Selliste vaadete alampäringud võivad sisaldada andmete küsimist ühest tabelist või andmete küsimist mitmest tabelist koos tabelite ühendamisega (*join*). Lisaks andmete küsimisele ühest või mitmest tabelist on vaates ka projektsiooni operatsioon, mis määrab vaatesse kuuluvad veerud. Öeldakse, et sellise vaate alampäring on *select-project-join* päring. Selliste vaadete korral toimub mestimine automaatselt.

- Välisühendamist (*outer join*) kasutav päring, kus väliseks tabeliks saab olla vaade, mille alampäringus pole kasutatud tabelite ühendamist;

- Keerukad vaated (*Complex view*). Selliste vaadete alampäring sisaldab *DISTINCT* klauslit või grupeerimist (*GROUP BY*) [20]. Selliste vaadete korral tuleb Oracle Database andmebaasisüsteemis sisse lülitada *Complex View Merging* funktsionaalsus.

See tähendab, et kui lauses viidatakse ühele või mitmele eelnevalt nimetatud vaatele, siis kokkuvõttes jõuab andmebaasisüsteem samasuguse täimisplaanini kui baastabelite põhjal tehtud päringu korral. Teiste sõnadega „tasandatakse“ andmebaasisüsteemi poolt vaatest tulenevad alampäringud. Muudel juhtudel täidetakse vaate lause osa eraldi, mitte mestides põhipäringuga.

Oracle Database andmebaasisüsteemi (versioon 12c) dokumentatsioonis [14] mainitakse, et kui vaade viitab teistele vaadetele (mitmetasemelised viitamised), siis see võib põhjustada mitteoptimaalse ja ressursimahuka päringu täitmise. Seega väidetakse, et see on ebasoovitav viis vaadete kasutamiseks.

Oracle Database (versioon 12c) andmebaasisüsteemi dokumentatsioonis [21] kirjutatakse, et optimeerija saab vaated ja nende põhjal tehtud päringud mestida siis, kui vaated ei sisalda *UNION, UNION ALL, INTERSECT, MINUS* operaatori poole pöördumist, *CONNECT BY* klauslit, *DISTINCT* klauslit, *SELECT* klauslis kokkuvõttefunktsioone *(Avg, Max, Min, Sum).* Oracle Database (versioon 9i) andmebaasisüsteemis dokumentatsioonis [22], lisaks ülalmainitud nimekirjale, et vaated ei tohi viidata pseudoveerule *ROWNUM.* PostgreSQL andmebaasisüsteemi kohta väidetakse, et kui vaade sisaldab kokkuvõttefunktsioonide poole pöördumisi, sorteerimist (*ORDER BY* klauslit) või ridade hulga piiramist (*LIMIT* klausel), siis see võib muuta andmebaasisüsteemile vaate ja selle põhjal tehtava päringu mestimise raskemaks ja tekitada probleeme [23].

Minu hinnangul ei ole sellel teemal piisavalt uurimistöid tehtud. Käesolevas töös käsitletavatest andmebaasisüsteemidest on rohkem käsitletud Oracle Database süsteemi, kuid PostgreSQLi peaaegu üldse mitte. Antud tööga üritan hõlmata võimalikult palju päringuid, mis võiksid takistada optimeerijat võimalikult hea täitmisplaani valimisel. Antud osas olid kirjeldatud kõige olulisemad leitud eksperimendid ja väited vaadete kasutamise mõjust päringu täitmisplaanile. Seega käesoleva töö eksperimentidest suurem osa hõlmab leitud materjalidest käsitletud probleeme. Lisaks on võrdlemiseks võetud ka teised minu arvates huvitavad vaadete põhjal tehtud päringuid.

3. Eksperimendi kirjeldus

Eksperimendi eesmärgiks on uurida kas andmebaasisüsteemid täitavad loogiliselt samaväärseid päringuid otse baastabelite põhjal ja vaadete põhjal samasuguse kiirusega ja samasuguse täitmisplaaniga või mitte ning kui ei, siis millest need erinevused tulevad. Seda uuritakse erinevate andmebaasisüsteemide korral. Uurimisküsimusele vastuse saamiseks kavandatakse iga uuritava andmebaasisüsteemi jaoks baastabelid, vaated, indeksid ja lisatakse testandmed. Seejärel uuritakse, millise täitmisplaani valib andmebaasisüsteem, kui tehakse päring otse baastabelite põhjal ja kui tehakse päring vaadete põhjal. Antud eksperimendis kõigepealt tehakse üks päring katseks ja alles seejärel hakatakse mõõtma täitmiseks kulunud aega. Esimesena käivitatakse vaate põhjal tehtud päring ja seejärel baastabeli põhjal tehtud päring.

Rääkides vaadete kasutamisest soovitab Burns [9], et andmebaasi kontseptuaalse skeemi denormaliseerimise asemel tuleks denormaliseerida andmebaasi väliseid skeeme moodustavaid vaateid (ehk vaate alampäringus ühendada mitu baastabelit). See võimaldaks varjata andmebaasi skeemi keerukust ning lihtsustada rakenduse arendamist ilma andmete terviklikkust ohverdamata [9, p. 115]. Oracle Database andmebaasisüsteemi dokumentatsioonis „Oracle Database Performance Tuning Guide“ [14] väidetakse, et vaated võivad põhjustada mitteoptimaalset ja ressursimahukat päringute täitmist. Selle põhjuseks on, et vaate kasutamine ei lase mingil põhjusel andmebaasisüsteemil leida head täitmisplaani. Ebasoovitav viis vaadete kasutamiseks on kui vaate alampäring viitab mingil viisil ühele või mitmele teisele vaatele. Sellised vaated raskendavad optimaalse täitmisplaani loomist [14].

3.1 Eksperimendi seadistamine

Eksperimendi jaoks kasutatakse kahte andmebaasisüsteemi ning mõlema puhul kahte versiooni (töö tegemise hetkel viimane versioon ja üle-eelmine versioon) – Oracle Database 12c Enterprise Edition Release 12.1.0.1.0, Oracle Database 11g Enterprise Edition Release 11.1.0.6.0, PostgreSQL 9.3 ja PostgreSQL 9.1. Mainitud andmebaasisüsteemide valiku põhjusteks on:

- andmebaasisüsteemide kättesaadavus ülikooli serveritel;
- andmebaasisüsteemide käsitlemine ülikooli aines „Andmebaasid II“, mida õpetab antud magistritöö juhendaja;

- Oracle Database andmebaasisüsteemi kasutamine töökohal;
- Andmebaasisüsteemide populaarsus (2015. aasta jaanuari seisuga on Oracle Database andmebaasisüsteem esimesel ja PostgreSQL andmebaasisüsteem populaarsuselt neljandal kohal) [24].

Töös kasutatakse sama andmebaasisüsteemi kahte erinevat versiooni, et oleks võimalik hinnata, kas andmebaasisüsteemi optimeerimismoodulis on päringute täitmisplaanide koostamise osas toimunud mingi areng või mitte.

Eksperimendi jaoks valitud Silverstoni raamatust osa universaalsest andmemudelist „Health care incidents, episodes, and visits" – „Tervishoiuteenuse intsidendid, episoodid ja visiidid" (vt Joonis 8) [25, p. 149], mille põhjal on loodud baastabelid, indeksid ja vaated ning teostatud eksperiment. Kuna töö eesmärgiks on uurida andmebaasisüsteemide päringute täitmisplaanide erinevusi baastabelite ja vaadete kasutamisel päringutes, siis sellest seisukohast lähtuvalt pole sisulist vahet millist andmemudelit eksperimendis kasutatakse. Seega antud andmemudel on valitud, kuna e-tervise teema on aktuaalne ja huvitav.

3.1.1 Tehnilised andmed

Eksperimendi läbiviimiseks kasutatakse ülikooli servereid.

- *hektor8.ttu.ee,* kus on Oracle Database 11g ja PostgreSQL 9.1.4. Serveri tehnilised andmed on järgmised: füüsiline 4-tuumaline masin, Intel(R) Xeon(R) CPU X3210 @ 2.13GHz, 450 GB HDD, 6 GB RAM, CentOS 5.7.
- *apex.ttu.ee,* kus on Oracle Database 12c ja PostgreSQL 9.3.0. Serveri tehnilised andmed on järgmised: virtuaalmasin QEMU Virtual CPU version, 811 GB HDD, 40 GB RAM, 15 virtuaalset CPU'd, CentOS 6.4.

3.1.1.1 Oracle Database andmebaasisüsteem

Nagu mainisin, siis selles eksperimendis kasutatakse kahte Oracle Database andmebaasisüsteemi:

- Oracle Database 12c Enterprise Edition Release 12.1.0.1.0;
- Oracle Database 11g Enterprise Edition Release 11.1.0.6.0.

„Enterprise Edition on Oracle Database andmebaasisüsteemi väljaannetest arendajatele kõige rohkem võimalusi pakkuv.“ [26]. Andmebaasisüsteemi versiooni saab näiteks teada järgmise SQL päringu abil:

```
SELECT *
FROM v$version
WHERE banner LIKE 'Oracle%';
```

Vaadeldavas töös kasutatakse andmebaasisüsteemiga suhtlemiseks graafilist kasutajaliidest pakkuvat programmi Oracle SQL Developer versiooniga 4.0.2.15. Samuti kasutatakse interaktiivset terminaliprogrammi SQL*Plus. Andmebaasisüsteemidega suhtlemise vahendid ei oma tegelikult sisulist tähendust ja võib kasutada ka muid vahendeid. Andmebaasisüsteemiga Oracle Database 11.1 ühenduse loomiseks kasutatakse ülikooli poolt pakutava *hostname* „hektor8.ttu.ee“, Oracle Database 12.1 puhul – „apex.ttu.ee“ ning mõlemal juhul *SID* (*system identifier*, andmebaasisüsteemi unikaalne identifikaator) on ORCL. Oracle andmebaasid on serverites juba loodud. Andmebaasi nimeks on „orcl“, mida saab teada SQL päringuga:

```
SELECT sys_context('userenv', 'db_name') FROM Dual;
```

Oracle andmebaas oli serveris juba eelnevalt loodud.

3.1.1.2 PostgreSQL andmebaasisüsteem

PostgreSQL andmebaasisüsteem on avatud lähtekoodiga ja tasuta pakutav andmebaasisüsteem, mis võimalustelt ei jää alla parimatele kommertssüsteemidele. Vaadeldavas eksperimendis kasutatakse kahte PostgreSQL andmebaasisüsteemi versiooni:

- "PostgreSQL 9.3.0 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3), 64-bit";
- "PostgreSQL 9.1.4 on x86_64-unknown-linux-gnu, compiled by gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-51), 64-bit".

Andmebaasisüsteemi versiooni saab teada SQL päringu abil:

```
SELECT version();
```

Andmebaasisüsteemiga suhtlemiseks kasutatakse käesolevas töös programmi pgAdmin III versiooniga 1.18.1. Andmebaasisüsteemiga PostgreSQL 9.1.4 ühenduse loomiseks kasutatakse hostname pordiga „hektor8.ttu.ee:5432“ ning PostgreSQL 9.3.0 puhul – „apex.ttu.ee:7301“.

Eksperimendis luuakse andmebaasi nimega „*experiment_views*“ ja kasutatakse automaatselt loodud skeemi *public*. Andmebaasi loomiseks saab serveris kasutada programmi *createdb*.

```
createdb -l et_EE.utf8 -T template0 experiment_views
```

3.1.1.3 Täitmisplaanide nägemiseks vajalikud käsud

Oracle Database andmebaasisüsteemis kasutatakse täitmisplaanide vaatamiseks SQL*Plus interaktiivset terminaliprogrammi. Selle käivitamiseks logitakse serverisse kasutades SSH protokolli. Järgmine käsk shelli promptis käivitab SQL*Plusi:

```
$ORACLE_HOME/bin/sqlplus
```

Edaspidi käivitatakse SQL*Plusis järgmised laused:

```
SET SERVEROUTPUT OFF
SET LINESIZE 300
ALTER SESSION SET STATISTICS_LEVEL = ALL;
```

Selleks, et vaadata täitmisplaani tuleb käivitada *SET SERVEROUTPUT OFF*, vastasel juhul, ilmub veateade. Selleks, et iga täitmisplaani rida mahuks ilusti ühele reale, suurendatakse rea pikkust: *SET LINESIZE* (mittekohustuslik). Kogu täitmisplaani statistika nägemiseks tuleb käivitada *SET STATISTICS_LEVEL = ALL* (vastasel juhul on nähtavad ainult täitmisplaani veerud: *id, operation, name* ja *e-rows*).

Täitmisplaanide vaatamiseks tuleb käivitada vajalik *SELECT* lause ning peale seda käivitada järgmine lause:

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR ( NULL,NULL,'ALLSTATS LAST +COST' )
);
```

PostgreSQL andmebaasisüsteemis päringute käivitamiseks avatakse pgAdmin III ja valitakse loodud andmebaas *„experiment_view“*. Täitmisplaanide vaatamiseks kasutatakse *EXPLAIN ANALYZE* lauset, mis sisaldab ka analüüsitavat *SELECT* lauset. *EXPLAIN* annab korralduse

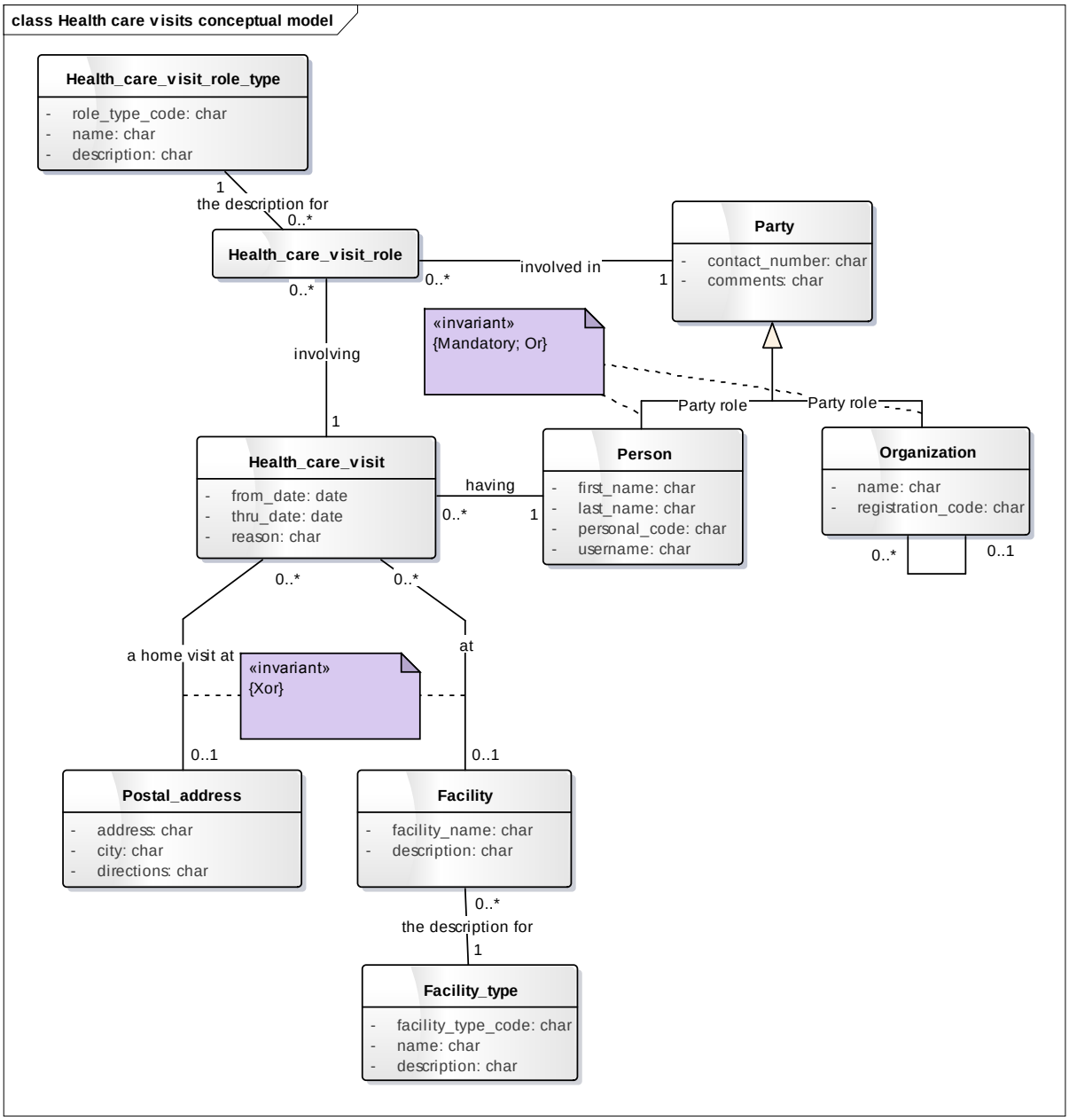
näidata täitmisplaani. *ANALYZE* lisab, et lause tuleb ka tegelikult täita, mis võimaldab süsteemil näidata üksikute operatsioonide ja terve lause täitmiseks kulunud aega.

```
EXPLAIN ANALYZE
SELECT * FROM … ;
```

3.2 Eksperimendi andmebaasi kavandamine

3.2.1 Analüüsi mudel

Järgnevalt esitatakse tervishoiuteenuse visiitide kontseptuaalne andmemudel (vt Joonis 8), mis on loodud Silverston [25, p. 149] universaalse andmemudeli osa põhjal.



Joonis 8. Kontseptuaalne mudel

Osapoolteks (*PARTY*) võib olla organisatsioon (*ORGANIZATION*), mis näiteks saadab oma töötaja tervisekontrollile, ja isik (*PERSON*), kes võib olla kas tervishoiuteenuse osutaja või

patsient. Patsiendid (*PERSON)* teevad tervishoiuteenuse osutajate juurde visiite või tehakse nende juurde visiite (*HEALTH CARE VISIT*). Igal patsient teeb null või rohkem visiiti. Iga visiidiga on seotud null või rohkem erinevat osapoolt erinevates rollides (nt arst või visiidile suunaja)(olemitüüp *HEALTH CARE VISIT ROLE*). Igal organisatsioonil on null või rohkem alluvat organisatsiooni ning maksimaalselt üks organisatsioon, millele see organisatsioon allub (saab teada *parent_id* abil (vt Tabel 1 *„ORGANIZATION“* all)).

Osapoolte rollide tüübid on kirjeldatud olemitüübiga *HEALTH CARE VISIT ROLE TYPE*. Iga osapool, mis on seotud visiidiga (*HEALTH CARE VISIT*), omab selle visiidi juures rolli, mida kirjeldab olemitüüp *HEALTH CARE VISIT ROLE*. Selle olemitüübi alusel ei registreerita visiidi juures patsiendi rolli. Patsient on tõstetud „sulgude ette“ ja seotud otse visiidiga (*patient_id* (vt Tabel 1 *„HEALTH_CARE_VISIT“* all)). *HEALTH CARE VISIT ROLE* abil registreeritakse kõik ülejäänud rollid.

Iga visiit (*HEALTH CARE VISIT*) teostatakse kas mingil patsiendi poolt öeldud aadressil (*POSTAL ADDRESS*) või mingis asutuses (*FACILITY*), mis on tüüpi (*FACILITY TYPE*) – haigla, kliinik, ambulatoorse kirurgia keskus või muu tervishoiuteenuse asutus.

Mudelite loomise vahendina kasutasin Sparx Systems Enterprise Architect 11, kuna see on populaarne ja mugav vahend andmete modelleerimiseks ning ülikoolis on saadaval litsents.

3.2.2 Disaini mudel

Järgnevalt on toodud Oracle Database ja PostgreSQL andmebaasi veergude andmetüübid ja näiteväärtused, kitsendused ja mõlema andmebaasi diagrammid. Baastabeleid on üheksa (vt Tabel 1): *Health_care_visit, Party, Person, Organization, Health_care_visit_Role_type, Health_care_visit_Role, Postal_address, Facility, Facility_type.*

Tabel 1. Baastabelite veerud

Veeru nimi	**Oracle Database andmetüüp**	**PostgreSQL andmetüüp**	**Näiteväärtus**	**Kohus-tuslik**
HEALTH_CARE_VISIT				
Health_care_visit_id	NUMBER(10)	integer	1	Jah
Patient_id	NUMBER(10)	integer	17	Jah
Facility_id	NUMBER(10)	integer	23	
Postal_address_id	NUMBER(10)	integer	1	

Veeru nimi	Oracle Database andmetüüp	PostgreSQL andmetüüp	Näiteväärtus	Kohus-tuslik
From_date	DATE	date	12.10.2014	Jah
Thru_date	DATE	date	12.10.2014	
Reason	VARCHAR2(255)	varchar(255)	Kõhuvalu	
Visit_fee	NUMBER(8,2)	decimal(8,2)	213.55	
PARTY				
Party_id	NUMBER(10)	integer	17	Jah
Contact_number	VARCHAR2(20)	varchar(20)	9-(300)857-7983	Jah
Comments	VARCHAR2(255)	varchar(255)	Kommentaar	
PERSON				
Party_id	NUMBER(10)	integer	17	Jah
First_name	VARCHAR2(50)	varchar(50)	Taavi	Jah
Last_name	VARCHAR2(50)	varchar(50)	Mitt	Jah
Personal_code	CHAR(11)	char(11)	37705202755	Jah
Birth_date	DATE	date	20.05.1977	Jah
Document_title	VARCHAR2(20)	varchar(20)	ID card	Jah
Document_number	VARCHAR2(20)	varchar(20)	AA12345678	Jah
Heigth	NUMBER(5,2)	decimal(5,2)	1.65	
Weight	NUMBER(5,2)	decimal(5,2)	54.55	
Username	VARCHAR2(20)	varchar(20)	e55kaitam	Jah
ORGANIZATION				
Party_id	NUMBER(10)	integer	55	Jah
Name	VARCHAR2(100)	varchar(100)	Kaid OÜ	Jah
Registration_code	CHAR(8)	char(8)	12345678	Jah
Parent_id	NUMBER(10)	integer	1	
HEALTH_CARE_VISIT_ROLE_TYPE				
Role_type_code	CHAR(20)	char(20)	EMPLOYER02	Jah
Name	VARCHAR2(50)	varchar(50)	Sender	Jah
Description	VARCHAR2(255)	varchar(255)	Kommentaar	
HEALTH_CARE_VISIT_ROLE				
Health_care_visit_id	NUMBER(10)	integer	1	Jah
Party_id	NUMBER(10)	integer	55	Jah
Role_type_code	CHAR(20)	char(20)	EMPLOYER02	Jah

Veeru nimi	Oracle Database andmetüüp	PostgreSQL andmetüüp	Näiteväärtus	Kohus-tuslik
FACILITY_TYPE				
Role_type_code	CHAR(20)	char(20)	HOS01	Jah
Name	VARCHAR2(50)	varchar(50)	Hospital	Jah
Description	VARCHAR2(255)	varchar(255)	Health care institution	
FACILITY				
Facility_id	NUMBER(10)	integer	23	Jah
Facility_name	VARCHAR2(100)	varchar(100)	Mustamäe haigla	Jah
Facility_type_code	VARCHAR2(255)	varchar(255)	HOS01	Jah
Description	VARCHAR2(255)	varchar(255)	Regionaalhaigla Mustamäe korpus	
POSTAL_ADDRESS				
Postal_address_id	NUMBER(10)	integer	1	Jah
Address	VARCHAR2(255)	varchar(255)	Raja 5a - 1	Jah
City	VARCHAR2(50)	varchar(50)	Tallinn	Jah
Directions	VARCHAR2(255)	varchar(255)	Akadeemia tee	

Peaaegu igas baastabelis on PostgreSQL andmebaasis identifikaatori andmetüübiks *integer*, aga Oracle Database andmebaasis *NUMBER(10,0)*. Ning kui PostgreSQL andmebaasisüsteemis on kasutusel *decimal* tüüp, siis Oracle Database süsteemis tuleb kasutada *NUMBER* tüüpi.

Baastabelitele on loodud primaarvõtme, välisvõtme, unikaalsuse (*UNIQUE*) kitsendused ning välisvõtmetele lisatavad indeksid (vt Tabel 2). Baastabeli *HEALTH_CARE_VISIT_ROLE* primaarvõti moodustub veergude kombinatsioonist – tegemist on liitvõtmega.

Primaarvõtmeks mitte valitud kandidaatvõtmeid nimetatakse alternatiivvõtmeteks. Alternatiivvõtmete jõustamiseks luuakse unikaalsuse kitsendused. Märgime, et unikaalsuse kitsendust *UQ_Visit_facil_addr_pat_fromd* ei jõusta alternatiivvõtit, kuna igas reas on kas *facility_id* või *postal_address_id* veergu väärtus määramata. Samas võtme puhul kehtib nõue, et selle väärtus peab olema igas reas määratud (ei tohi sisaldada *NULL*e).

Viidete terviklikkuse tagamiseks deklareeritakse tabelites välisvõtmed. Välisvõtmete deklareerimisel on võimalik määrata kompenseerivaid tegevusi, mida andmebaasisüsteem viib läbi viidete terviklikkuse vigadest hoidumiseks. Mõlemas andmebaasisüsteemis kasutati

ORGANIZATION, *PERSON* ja *HEALTH_CARE_VISIT_ROLE* tabelites välivõtmete deklareerimisel määrangut *ON DELETE CASCADE*, mis tähendab sõltuvate ridade kaskaadset kustutamist.

PostgreSQL ja Oracle indekseerivad automaatselt primaarvõtmesse ja unikaalsuse kitsendusse kuuluvad veerud. Need andmebaasisüsteemid ei indekseeri automaatselt välisvõtmetesse kuuluvaid veerge. Välisvõtmed on mõistlik siiski indekseerida, et kiirendada ühendamisoperatsioonide läbiviimist ja vähendada lukustamist. Lisaks indekseerin ka eelnevalt nimetatud kategooriatesse mitte kuuluvaid veerge, mida hakatakse kasutama eksperimendi päringute tingimustes. Kuna mitmes päringus toimub otsing sünni aasta järgi, siis luuakse ka üks funktsioonil põhinev indeks, mis põhineb avaldisel: *EXTRACT (YEAR FROM birth_date)*.

Tabel 2. Baastabelite kitsendused ja indeksid

Tabeli nimi	**Kitsenduse/indeksi nimetus (eesliited: PK – primaarvõti FK – välisvõti UQ – unikaalsus IXFK – välisvõtmele loodud indeks AK – alternatiivvõti idx – täiendav indeks, mida ei looda välisvõtmele)**	**Veeru nimi**
HEALTH_CARE_VISIT	PK_Health_care_visit	*health_care_visit_id*
	FK_Health_care_visit_Facility	*facility_id*
	FK_Health_care_visit_Postal_address	*postal_address_id*
	FK_Health_care_visit_Person	*patient_id*
	IXFK_Health_care_visit_Facility	*facility_id*
	IXFK_Health_care_visit_Postal_address	*postal_address_id*
	idx_Health_care_visit_FromDate	*from_date*
	UQ_Visit_facil_addr_pat_fromd	*patient_id*
		facility_id
		postal_address_id
		from_date

Tabeli nimi	**Kitsenduse/indeksi nimetus (eesliited:** **PK – primaarvõti** **FK – välisvõti** **UQ – unikaalsus** **IXFK – välisvõtmele loodud indeks** **AK – alternatiivvõti** **idx – täiendav indeks, mida ei looda välisvõtmele)**	**Veeru nimi**
PARTY	PK_Party	*party_id*
PATIENT	PK_Person	*party_id*
	FK_Person_Party	*party_id*
	idx_Person_birth_date	*EXTRACT (YEAR FROM birth_date)*
	AK_Person_username	*username*
	AK_Person_document	*document_title*
		document_number
	AK_Patient_personal_code	*personal_code*
ORGANIZATION	PK_Organization	*party_id*
	FK_Organization_Party	*party_id*
	FK_Organization_Organization	*parent_id*
	IXFK_Organization_Organization	*parent_id*
	AK_Organization_name	*name*
	AK_Organization_regcode	*registration_code*
HEALTH_CARE_VISIT _ ROLE_TYPE	PK_Role_type	*role_type_code*
	AK_Health_care_visit_rol_name	*name*
HEALTH_CARE_ VISIT_ROLE	PK_Visit_Role_type_Party	*health_care_visit_id*
		party_id
		role_type_code
	FK_Role_Health_care_visit	*health_care_visit_id*
	FK_Role_Role_type	*role_type_code*
	FK_Role_Party	*party_id*
	IXFK_Role_Role_type	*role_type_code*
	IXFK_Role_Party	*party_id*

Tabeli nimi	**Kitsenduse/indeksi nimetus (eesliited:** **PK – primaarvõti** **FK – välisvõti** **UQ – unikaalsus** **IXFK – välisvõtmele loodud indeks** **AK – alternatiivvõti** **idx – täiendav indeks, mida ei looda välisvõtmele)**	**Veeru nimi**
POSTAL_ADDRESS	PK_Postal_address	*postal_address_id*
FACILITY_TYPE	PK_Facility_type	*facility_type_code*
	AK_Facility_type_name	*name*
FACILITY	PK_Facility	*facility_id*
	FK_Facility_Facility_type	*facility_type_code*
	IXFK_Facility_Facility_type	*facility_type_code*
	AK_Facility_facility_name	*facility_name*

Andmebaasi on lisatud ka täiendavad ärireegli kitsendused – *CHECK* kitsendused – *chk_Visit_Facility_Postal_addr* ja *chk_Patient_personal_code*.

CHECK kitsendus *chk_Visit_Facility_Postal_addr* luuakse baastabeli *HEALTH_CARE_VISIT* veergudele *facility_id* ja *postal_address_id*, kontrollimaks, et igas reas oleks määratud kas *facility_id* või *postal_address_id*, aga mitte mõlemad korraga või olema mõlemad määramata.

Baastabeli *PERSON* veerule *personal_code* on loodud *CHECK* kitsendus - *chk_Patient_personal_code*, mis kontrollib isikukoodi (*personal_code*) vastavust teatud reeglitele. Eeldatakse, et isikukood on Eesti isikukood ja sellest tulenevalt esimene number, mis määrab sugu, on vahemikus 3-6, sünni kuu esimene number – 0-1 ja sünni päeva esimene number – 0-3. Nende kitsenduste avaldistes kasutatakse regulaaravaldisi.

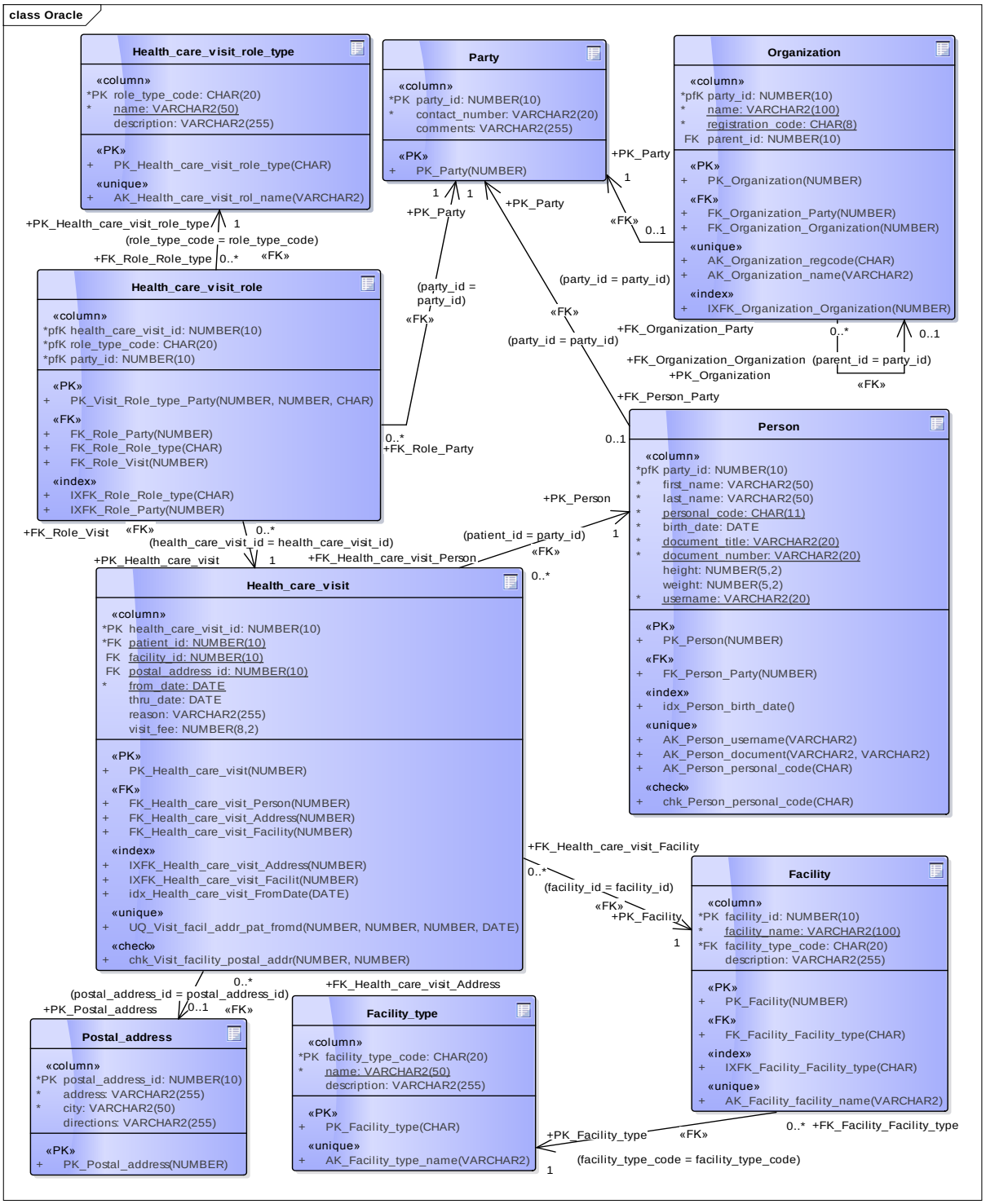
Soovitav on üldotstarbelise testandmete generaatori poolt loodud sünteetiliste testandmete kasutamise korral lisada *UNIQUE* ja *CHECK* kitsendused peale testandmete tabelitesse lisamist ja kohandamist, kuna väljastatavad andmed võivad olla kitsendustega vastuolus. Juhul kui andmebaasisüsteem võimaldab kitsenduste sisse- ja väljalülitamist, siis on teine võimalus luua kitsendused tabelite loomisel kuid lülitada kitsendused andmete laadimise ajaks välja. Andmete tabelitesse laadimise kiirendamiseks (eriti kui andmeid on palju) on mõistlik ka kasutaja-defineeritud indeksid lisada tabelitele alles peale seda, kui testandmed on tabelitesse laaditud.

Kitsenduste deklareerimine on muuhulgas oluline, kuna annab programmidele (sealhulgas andmebaasisüsteemile enesele) ja inimkasutajatele infot andmebaasis olevate andmete tähenduse kohta. Andmebaasisüsteem saab seda infot näiteks rakendada päringute täitmisplaanide koostamisel (nt selleks, et viia läbi tabeli elimineerimise teisendus [21] [27]). Lühidalt võib öelda, et tänu kitsenduste olemasolule võib andmebaasisüsteem mõningaid päringuid lihtsustada, mis tingib kokkuvõttes lihtsama täitmisplaani loomise ja lause kiirema täitmise. Käesolevas töös ei uurita kitsenduste deklareerimise mõju päringu täitmisplaanide koostamisele, kuid olen seisukohal, et kitsenduste leidmine ja jõustamine on igasuguse andmebaasi kavandamise ja realiseerimise juures oluline samm.

Kogu tabelite loomiseks kasutatud koodi leiab huviline jaotistest „Lisa 1. Realisatsioon Oracle Database andmebaasisüsteemis“ ja „Lisa 2. Realisatsioon PostgreSQL andmebaasisüsteemis“.

3.2.2.1 Andmebaasi diagramm Oracle Database näitel

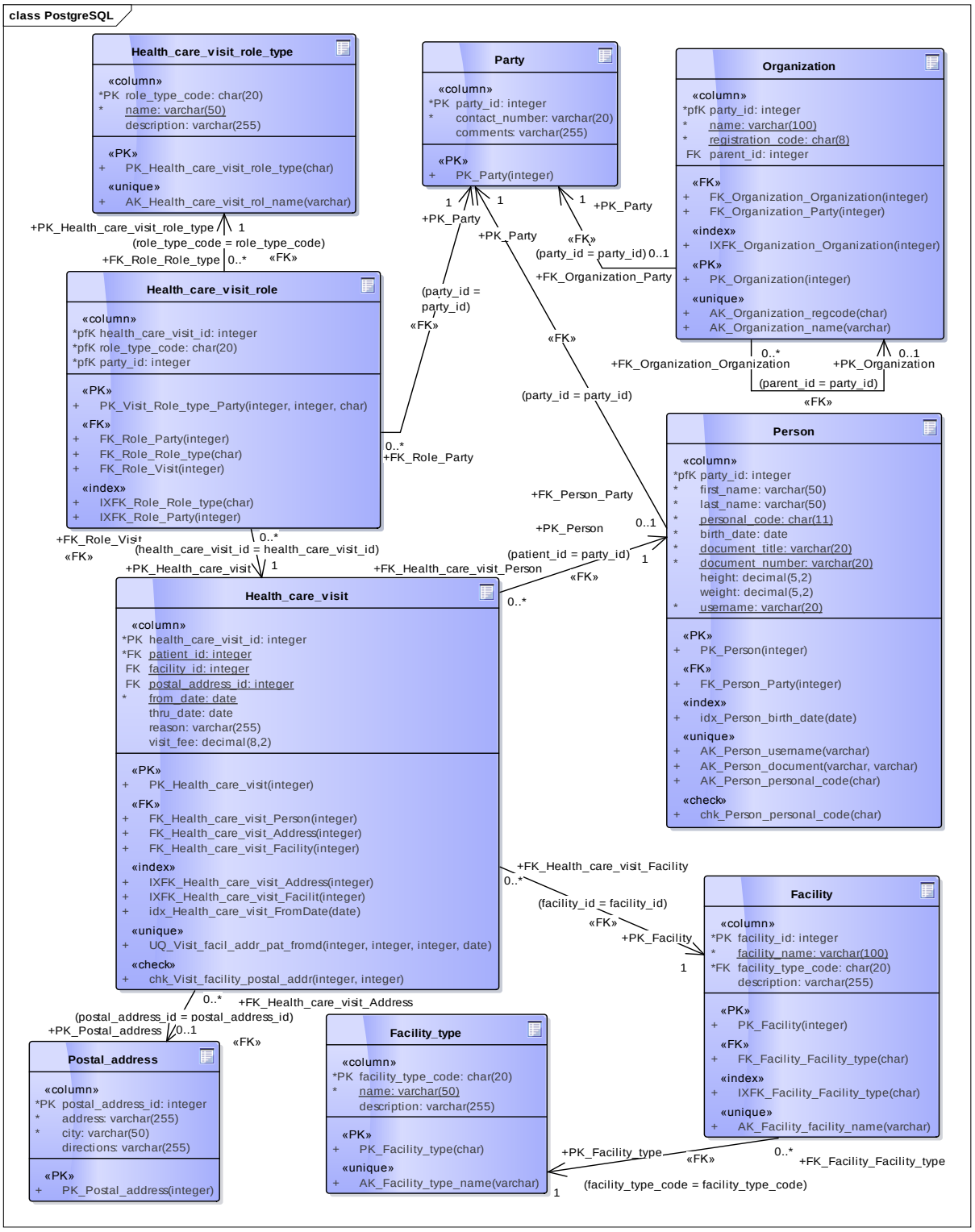
Järgnevalt esitatakse eksperimendi jaoks tehtud Oracle Database andmebaasi disaini kirjeldav diagramm (vt Joonis 9):



Joonis 9. Oracle Database andmebaasi diagramm

3.2.2.2 Andmebaasi diagramm PostgreSQL näitel

Järgnevalt esitatakse eksperimendi jaoks tehtud PostgreSQL andmebaasi disaini kirjeldav diagramm (vt Joonis 10):



Joonis 10. PostgreSQL andmebaasi diagramm

3.3 Testandmete genereerimine

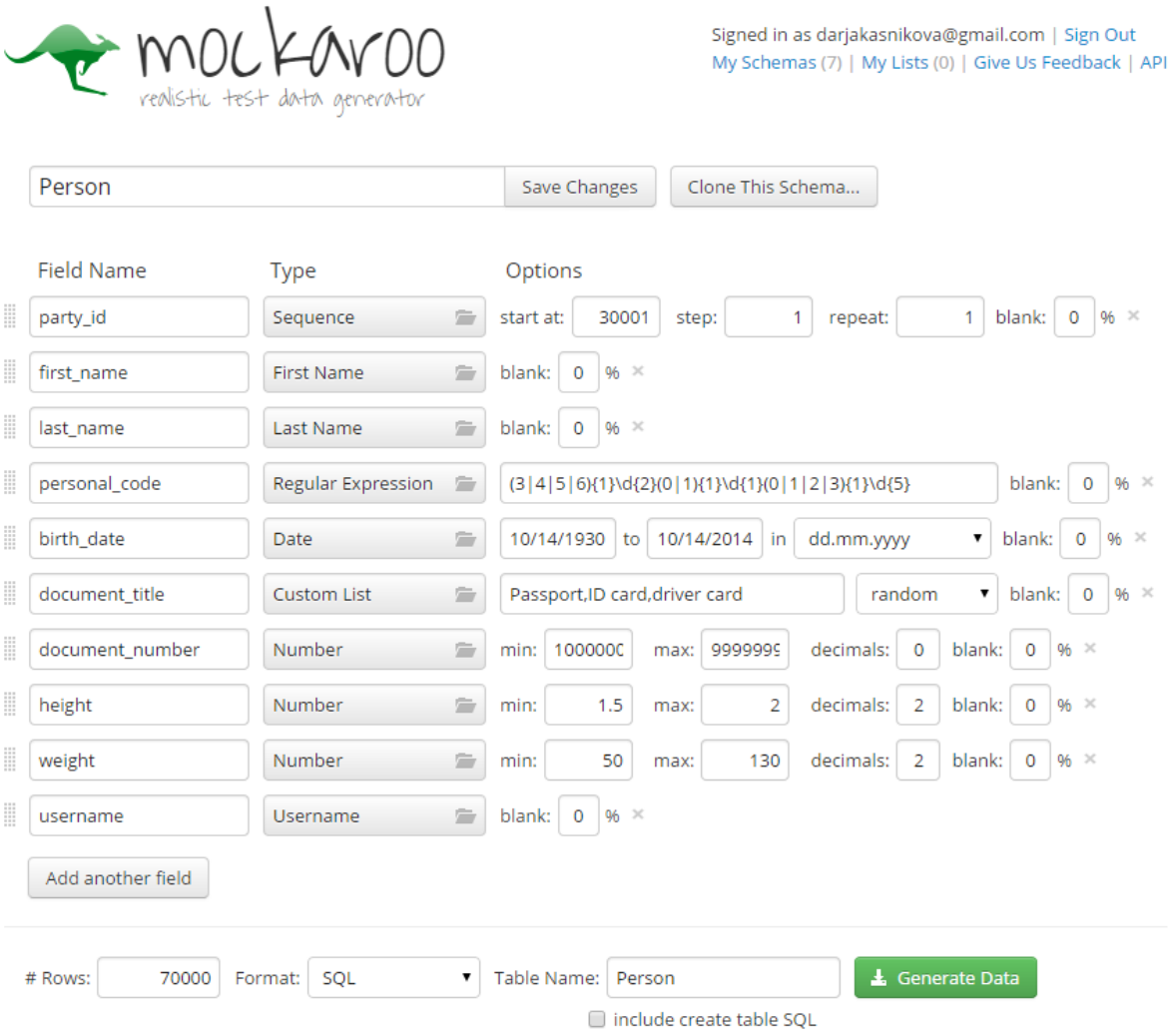
Eksperimendi teostamiseks genereeritakse samasugused testandmed mõlema andmebaasisüsteemi jaoks. Esialgu lisatakse testandmed Oracle Database andmebaasi. Sinna lisatud testandmeid muudetakse (näiteks, asendatakse mitteunikaalsed väärtused unikaalsete väärtustega). Seejärel eksporditakse muudetud andmete põhjal genereeritud *INSERT* laused failidesse, misjärel laaditakse kõik testandmed PostgreSQL andmebaasi (käivitades Oracle Database andmebaasist eksporditud *INSERT* laused). Kõige lõpuks luuakse tabelitele puuduvad kitsendused ja indeksid. Testandmed on loodud kasutades Mockaroo [28] testandmete generaatorit. Selline generaator on valitud mitmel põhjustel – tasuta vahend; võimaldab lisada reaalsetele andmetele sarnaseid andmeid ja kasutada regulaaravaldisi; ühe korraga saab genereerida 100 000 rida; lihtne kasutada. Lisaks, võimaldab Mockaroo luua mitu baastabelit ja salvestada oma kasutajanime all.

Kuidas toimus testandmete koostamine? Esialgu määrati kindlaks klassifikaatorite väärtused – rollide ja asutuste tüübid (*HEALTH_CARE_VISIT_ROLE_TYPE, FACILITY_TYPE*).

```
INSERT INTO Facility_type (facility_type_code, name, description)
VALUES ('HOS01', 'Hospital', 'Health care institution');

INSERT INTO Facility_type (facility_type_code, name, description)
VALUES ('MEDOFFICE03', 'Medical Office', 'Medical services');

INSERT INTO Facility_type (facility_type_code, name, description)
VALUES ('CLI06', 'Clinic', NULL);

INSERT INTO Facility_type (facility_type_code, name, description)
VALUES ('ASC001', 'Ambulatory Surgery Center', 'Outpatient surgery center');

INSERT INTO Facility_type (facility_type_code, name, description)
VALUES ('HF09', 'Other Healthcare Facility', 'Other');

INSERT INTO Health_care_visit_role_type (role_type_code, name, description)
VALUES ('EMPLOYER02', 'Sender', 'Organization that has employee, who may be covered
for insurance as part of their employment');

INSERT INTO Health_care_visit_role_type (role_type_code, name, description)
VALUES ('PAYOR02', 'Payor', 'Organization that plays for the claims');

INSERT INTO Health_care_visit_role_type (role_type_code, name, description)
VALUES ('DOC03', 'Doctor', NULL);

INSERT INTO Health_care_visit_role_type (role_type_code, name, description)
VALUES ('NURSE01', 'Nurse', NULL);

INSERT INTO Health_care_visit_role_type (role_type_code, name, description)
VALUES ('PHYSTHER', 'Physical therapists', 'Physitherapy professionals');

INSERT INTO Health_care_visit_role_type (role_type_code, name, description)
```

```
VALUES ('OTHER02', 'Interested Party', 'Interested Party Who Needs Notification');
```

Seejärel genereeriti andmed Mockaroo testandmete generaatori abil. Mockaroo generaator võimaldab kasutada ka regulaaravaldisi, mis on vajalik *CHECK* kitsenduse *„personal_code“* (isikukood) jaoks (vt Joonis 11).



Joonis 11. Mockaroo testandmete genereerimine *PERSON* baastabeli näitel

Kuna Mockaroo testandmete generaator käsitleb iga tabelit eraldi ning teeb eraldi SQL faili iga baastabeli kohta, siis ei eksisteeri võimalust genereerida välisvõtme väärtuseid. Kasutan välisvõtme väärtuste genereerimisel täisarvude genereerimist vajalikus vahemikus (näiteks 1 kuni 100 000). Vahemiku suurema otsa väärtuse määrab seotud primaarses tabelis olevate ridade arv.

Järgnevalt esitatakse testandmete hulk iga eksperimendi andmebaasis kasutatava baastabeli kohta:

- Baastabel *HEALTH_CARE_VISIT_ROLE_TYPE* – 6 rida;
- Baastabel *FACILITY_TYPE* – 5 rida;
- Baastabel *PARTY* [29] – 100 000 rida. Põhiliseks ridade hulgaks on valitud 100 000 rida;
- Baastabel *PERSON* [30] (vt Joonis 11) – 70 000 rida. Kuna põhiobjektideks on patsient ja tervishoiuteenuse osutaja (näiteks arst), siis suurem hulk ridu on baastabelis *PERSON* (võrreldes baastabeliga *ORGANIZATION*);
- Baastabel *ORGANIZATION* [31] – 30 000 rida;
- Baastabel *POSTAL_ADDRESS* [32] – 100 000 rida;
- Baastabel *FACILITY* [33] – 50 000 rida;
- Baastabel *HEALTH_CARE_VISIT* [34] – 1 000 000 rida (igal patsiendil keskmiselt 14 visiiti);
- Baastabel *HEALTH_CARE_VISIT_ROLE* [35] [36] – 1 300 000 rida. Iga visiidiga on seotud arst, kes patsiendi üle vaatab (1 000 000 rida). Lisaks on iga organisatsioon võib olla seotud ühe või mitme visiidiga, näiteks patsiendi visiidile saatja rollis (300 000 rida). Patsiendiks olemise rolli selles tabelis ei registreerita – patsient on seotud otse tabeliga *HEALTH_CARE_VISIT*.

Mockaroo testandmete generaator lubab ühe korraga genereerida 100 000 rida. Kui ridu on rohkem, siis generaatoris muudetakse identifikaatori juures ainult *„start at“* väljas olevat väärtust. Kui on genereeritud esimest 100 000 rida (sellisel juhul *„start at“* on võrdne 1.-ga), siis järgmise 100 000 rea genereerimiseks tuleb *„start at“* väärtuseks määrata 100001 (see kehtib sellisel juhul kui Mockaroo testandmete generaatoris on identifikaatori tüübiks *„Sequence“*).

Eksperimendis kasutatava testandmete generaatori üheks negatiivseks küljeks on andmete dubleerimine. Näiteks võimaldab Mockaroo kasutada olemasolevat organisatsiooni nimede listi, mis sobib *ORGANIZATION* baastabelis veergu *„name“* väärtuste genereerimiseks. Kahjuks saadakse väljundina mitteunikaalseid nimesid. Seega sellise olukorra vältimiseks mõtlesin Oracle Database andmebaasisüsteemi jaoks välja järgmise lahenduse.

1. Kui kitsendused on juba lisatud, siis esialgu tuleb unikaalsuse kitsendus välja lülitada. Selleks saab kasutada lauset:

```
ALTER table_name DISABLE CONTRAINT constraint_name;
```

2. Lisada testandmed.
3. Käivitada järgnevat protseduuri, võib olla mitu korda. Protseduuri väljakutsel on argumentideks baastabeli, veeru nimi ning kolmas parameeter *Pvalue_type*, mille väärtus ütleb, millist klauslit kasutada. Protseduuri käivitamisel eemaldatakse korduvaid väärtusi. Seega korduval käivitamisel peab korduvate väärtuste hulk vähenema ja muutuma lõpuks tühjaks. Järgnevalt esitatakse skript, mis otsib baastabelis korduvaid väärtuseid ning kas lisab korduvate väärtuste juurde juhusliku täisarvu või asendab juhuslikult genereeritud täisarvuga või arvuga, mis vastab regulaaravaldisele *„REGEXP_LIKE(personal_code,'^([3-6]{1}[[:digit:]]{2}[0-1]{1}[[:digit:]]{1}[0-3]{1}[[:digit:]]{5})$')“*. Protseduuril on olemas kolmas parameeter *Pvalue_type*, mille väärtus ütleb, millist klauslit kasutada:

```
create or replace PROCEDURE ReplaceDuplicateValues (
    Ptable_name IN varchar2,
    Pcolumn_name IN varchar2,
    Pvalue_type IN number )
IS
    Vcheck PLS_INTEGER := 0;
    Vset_value VARCHAR2(255);
BEGIN
    SELECT Count(*) INTO Vcheck
    FROM user_tab_columns
    WHERE table_name = Ptable_name
      AND column_name = Pcolumn_name;

    IF Vcheck != 1 THEN DBMS_OUTPUT.put_line('Vale tabeli või veeru nimi'); RETURN;
    END IF;

    /* Väärtuse juurde lisatakse juhuslik täisarv vahemikus 1 ja 9999 */
    IF Pvalue_type = 1 THEN
      Vset_value := ' ' || Pcolumn_name || ' || Round(dbms_random.value(1,9)) ';

    /* Genereeritakse uued juhuslikud numbrid, mis vastavad regulaaravaldisele -
       (isikukoodi/personal_code jaoks) REGEXP_LIKE(personal_code,'^([3-
       6]{1}[[:digit:]]{2}[0-1]{1}[[:digit:]]{1}[0-3]{1}[[:digit:]]{5})$') */
    ELSIF Pvalue_type = 2 THEN
      Vset_value :=
         'Round(dbms_random.value(3,6))||round(dbms_random.value(10,99))||
         round(dbms_random.value(0,1))||round(dbms_random.value(0,9))||
         round(dbms_random.value(0,3))||round(dbms_random.value(0,9))||
         round(dbms_random.value(1111,9999))';

    /* Genereeritakse uued juhuslikud kaheksakohalised täisarvud */
```

```
    ELSIF Pvalue_type = 3 THEN
      Vset_value := 'Round(dbms_random.value(10000000,99999999))';
    ELSE RETURN;
    END IF;

    EXECUTE IMMEDIATE
    'UPDATE ' || Ptable_name || '
    SET ' || Pcolumn_name || ' = ' || Vset_value || '
    WHERE ' || Pcolumn_name || ' IN (
      SELECT ' || Pcolumn_name || '
      FROM ' || Ptable_name || '
      GROUP BY ' || Pcolumn_name || '
      HAVING Count(*) > 1 ) ';
    COMMIT;
    DBMS_OUTPUT.put_line('OK');
END;
```

4. Viimaseks tegevuseks on unikaalsuse kitsenduse kontroll uuesti sisse lülitada.

```
ALTER  table_name  ENABLE CONTRAINT constraint_name;
```

Käesolevas eksperimendis tuleb sellisel viisil muuta andmeid nii unikaalsuse kitsendusega kui CHECK kitsendusega veergudes (kitsenduste sisse- ja väljalülitamise lauseid kõigi asjassepuutuvate kitsenduste korral vaadake palun Lisa 4. Kitsenduste sisse- ja väljalülitamise laused).

Selle protseduuri abil muudeti:

- *PERSON* baastabelis *personal_code* väärtuseid, genereerides regulaaravaldisele vastava isikukoodi;
- *PERSON* baastabelis *document_number* ja *ORGANIZATION* baastabelis *registration_code* väärtuseid, genereerides juhuslikke kaheksakohalisi täisarve;
- *FACILITY* baastabelis *facility_name* ja *ORGANIZATION* baastabelis *name* väärtuseid, lisades olemasolevate väärtuse juurde juhusliku täisarvu.

Eelviimane vajalik muudatus tehakse ära *HEALTH_CARE_VISIT* baastabelis, kuna iga visiidiga peab olema seotud kas ainult mingi aadress (*POSTAL_ADDRESS*) või ainult mingi tervishoiuasutus (*FACILITY*). Seega 500 000 reas kustutatakse *facility_id* väärtus ja 500 000 reas *postal_address_id* väärtus:

```
UPDATE Health_care_visit
SET facility_id = NULL
WHERE health_care_visit_id IN (
    SELECT *
    FROM (
        SELECT hcv.health_care_visit_id
        FROM Health_care_visit hcv
        WHERE hcv.facility_id IS NOT NULL
        ORDER BY dbms_random.random )
    WHERE ROWNUM <= 500000 );
```

```
UPDATE Health_care_visit
SET postal_address_id = NULL
WHERE facility_id IS NOT NULL;
```

Viimaseks muudatuseks on tabelis *ORGANIZATION* hierarhia loomine. Selleks tuleb käivitada skript Lisa 5. Hierarhia loomiseks kasutatav protseduur.

Andmete muutmise lõpetamise järel tuleb sisse lülitada kõik väljalülitatud kitsendused (vt Lisa 4. Kitsenduste sisse- ja väljalülitamise laused).

Kuna samasugused testandmed peavad olema mõlemas andmebaasis, siis eksporditakse Oracle SQL Developer'i abil Oracle Database andmebaasi testandmed failidesse. Andsin korralduse muuta eksportimisel andmetes kuupäeva formaati, komakoha eraldajat, grupi eraldajat, et testandmete lisamisel PostgreSQL andmebaasi ei tekiks probleeme.

3.4 Testpäringud ja vaated

Eksperimendi ettevalmistamise väga oluliseks aspektiks on teha põhjendatud päringute valik. Päringud tehakse baastabelite ja vaadete põhjal, mis annab võimaluse uurida erinevate päringute täitmisplaane ja päringute täitmise kiirus.

Erinevates allikates väidetakse, et kokkuvõttefunktsioonide, ühendi leidmise (*UNION*), lõike leidmise (*INTERSECT*), rekursiivse päringu (*CONNECT BY*), *ROWNUM* pseudoveeru ja korduvate ridade eemaldamise (*DISTINCT*) kasutamine vaadete alampäringutes võib põhjustada mitteoptimaalse täitmisplaani valimise [22]. Seega, valitakse eksperimenti sellised päringud. Kuid ka lihtsamad päringud vajavad üle kontrollimist. Seega valisin päringute loomise põhiallikaks Oracle Database dokumentatsiooni peatüki „Optimizer Operations" [22], kuna dokumentatsioon sisaldab palju erinevaid päringuid ja nende täitmisplaane. Ma ei leidnud sarnast põhjalikku dokumentatsiooni Oracle Database andmebaasisüsteemi versioonide 11g või

12c jaoks. Seega otsustasin kasutada leitud Oracle Database andmebaasisüsteemi versiooni 9i dokumentatsiooni peatükki.

Järgnevalt esitatakse valiku põhjused, vaadete loomise laused, päringud vaadete põhjal ja päringud otse baastabelite põhjal (juhul kui erineb vaate alampäringust). Igale eksperimendile antakse unikaalne nimi, et hiljem oleks parem koondtulemusi analüüsida. Süsteemi sisemise taseme operatsioonid, mida eksperimentide juures kirjeldatakse on kasutusel Oracle Database andmebaasisüsteemis ning võivad PostgreSQLis erineda. Kuna käesolevas töös ei uurita vaadete kaudu andmete muutmist, siis ei kasutata vaadete loomisel *WITH CHECK OPTION* kitsendust.

A. Selles kategoorias esitatakse vaated, mis luuakse Oracle Database dokumentatsiooni peatüki „Optimizer Operations“ [22] päringute põhjal.

A1 eksperiment nimega „IN“. Järgnevas vaates on defineeritud päring, mis sisaldab endas *IN predikaati*. Tavaliselt sisaldub sellise päringu täitmisplaanis *IN-List ITERATOR* operatsioon ning sama operatsiooni kasutatakse ka *ANY predikaadiga* päringu korral. Kuna mitte üheski allikas ei soovitata loobuda *IN* ja *ANY predikaatide* kasutamisest vaadetes, siis võib esitada väite, et *IN-List ITERATOR* operatsioon ei põhjusta mitteoptimaalse plaani kasutamist vaadete põhjal tehtud päringutes. Seda väidet kontrollitakse järgneva vaate põhjal. Ühtlasi võimaldab see vaade kontrollida, kuidas mõjutab täitmisplaani valimist *INNER JOIN* operaatori kasutamine.

Leitakse tervishoiuteenuse visiitide ja nendega seotud asutuste täielik informatsioon, kus asutuse identifikaator on 11668 või 4700:

```
CREATE VIEW vw_Visit_filter_by_facility_id (
    facility_id, facility_name, facility_type_code, description,
    health_care_visit_id, patient_id, postal_address_id,
    from_date, thru_date, reason, visit_fee )
AS
SELECT fa.facility_id, fa.facility_name, fa.facility_type_code, fa.description,
    hcv.health_care_visit_id, hcv.patient_id, hcv.postal_address_id,
    hcv.from_date, hcv.thru_date, hcv.reason, hcv.visit_fee
FROM Health_care_visit hcv
INNER JOIN Facility fa ON hcv.facility_id = fa.facility_id
WHERE hcv.facility_id IN ( 11668, 4700 );

SELECT * FROM vw_Visit_filter_by_facility_id;
```

Samas viiakse läbi lisaeksperiment PostgreSQL 9.3 andmebaasisüsteemis tehes päringuid turvabarjääriga vaadete põhjal (vaate loomisel on määratud *WITH (security_barrier)*). PostgreSQL dokumentatsioonis väidetakse, et päringud turvabarjääriga vaadete põhjal võivad põhjustada mitteoptimaalse täitmisplaani valimise [37]. Sellist tüüpi vaadete näiteks on järgmine vaade (kõik järgnevad vaated luuakse samamoodi):

```
CREATE VIEW svw_Visit_filter_by_facility_id (
    facility_id, facility_name, facility_type_code, description,
    health_care_visit_id, patient_id, postal_address_id,
    from_date, thru_date, reason, visit_fee )
WITH (security_barrier)
AS
SELECT fa.facility_id, fa.facility_name, fa.facility_type_code, fa.description,
    hcv.health_care_visit_id, hcv.patient_id, hcv.postal_address_id,
    hcv.from_date, hcv.thru_date, hcv.reason, hcv.visit_fee
FROM Health_care_visit hcv
INNER JOIN Facility fa ON hcv.facility_id = fa.facility_id
WHERE hcv.facility_id IN ( 11668, 4700 );

SELECT * FROM svw_Visit_filter_by_facility_id;
```

A2 eksperiment nimega „LEFT OUTER“. Kuna iga visiit on seotud kas aadressiga või asutusega, aga mitte mõlema korraga, siis järgmine vaade sisaldab kolme baastabeli ühendamist kasutades vasakpoolse välisühendamise (*LEFT OUTER JOIN*) operaatorit. Vaate alampäringus *LEFT OUTER JOIN* operaatori kasutamine ei peaks põhjustama mitteoptimaalse täitmisplaani valimist. Seda kontrollitakse järgneva vaate põhjal.

Leitakse kogu informatsioon tervishoiuteenuse visiitide kohta, sealjuures ka teenuse osutamise asukohtade kohta. Otsitakse visiite, mis on toimunud ühel kindlal kuupäeval. Samuti piiratakse tulemust asukohtade ja visiiditasu alusel.

```
CREATE VIEW vw_Visit_at_facility_or_addr (
    health_care_visit_id, patient_id, from_date, thru_date,
    reason, visit_fee,
    facility_id, facility_name, facility_type_code, description,
    postal_address_id, address, city, directions )
AS
SELECT hcv.health_care_visit_id, hcv.patient_id, hcv.from_date, hcv.thru_date,
    hcv.reason, hcv.visit_fee,
    hcv.facility_id, fa.facility_name, fa.facility_type_code, fa.description,
    hcv.postal_address_id, pa.address, pa.city, pa.directions
FROM Health_care_visit hcv
LEFT OUTER JOIN Facility fa ON fa.facility_id = hcv.facility_id
LEFT OUTER JOIN Postal_address pa ON hcv.postal_address_id = pa.postal_address_id
WHERE hcv.from_date = TO_DATE( '23.04.2014', 'DD.MM.YYYY' )
    AND ( pa.city = 'Oslo' OR fa.facility_type_code = 'HOS01' )
    AND hcv.visit_fee > 1000;
```

```
SELECT * FROM vw_Visit_at_facility_or_addr;
```

B. Selles kategoorias on peamiselt esitatud vaated, mille alampäringu täitmisplaan sisaldab sorteerimisoperatsioone. Loomulikult on vaja ridu sorteerida kui päringus on ilmutatud kujul korraldus ridu sorteerida. Andmebaasisüsteem võib (aga ei pruugi) kasutada sorteerimist korduvate ridade eemaldamisel, grupeerimisel, ühendi ja lõike leidmisel [22] (PostgreSQL andmebaasisüsteem võib käituda teistmoodi). Päringu tulemusest korduvate ridade eemaldamist kasutatakse ka eksperimendis E1.

   Eksperimendid B1 ja B2 sisaldavad ühendi ja lõike operatsiooni läbiviimist, eksperiment B3 – kokkuvõttefunktsioonide (*Sum*, *Count*) kasutamist koos ridade grupeerimisega, B4 – korduvate ridade eemaldamist ja viimane B5 – ridade sorteerimist. Vaadeldavad eksperimendid valiti, kuna tihti kohtab väiteid, et Oracle Database andmebaasi vaadete loomisel tuleks vaadete alampäringutes selliseid operatsioone vältida [38] [19].

B1 eksperiment nimega „UNION“. Vaade *vw_Party_name_type* sisaldab ühendi leidmist (*UNION* operatsiooni). Vaate alampäring leiab kõikide isikute ja organisatsioonide ehk osapoolte nimed ning määrab osapoole tüübiks vastavalt „isik“ või „organisatsioon“:

```
CREATE VIEW vw_Party_name_type (
    name, party_type )
AS
SELECT first_name || ' ' || last_name AS name, 'person' AS party_type
FROM Person
UNION
SELECT name, 'organization' AS party_type
FROM Organization;

SELECT * FROM vw_Party_name_type;
```

B1.1 eksperiment nimega „UNION ALL“. Proovitakse ka eraldi ühendi leidmist *UNION ALL* operatsiooni abil saama päringu peal.

```
CREATE VIEW vw_Party_name_type_all (
    name, party_type )
AS
SELECT first_name || ' ' || last_name AS name, 'person' AS party_type
FROM Person
UNION ALL
SELECT name, 'organization' AS party_type
FROM Organization;

SELECT * FROM vw_Party_name_type_all;
```

UNION ja *UNION ALL* operatsioonide vahe on selles, et esimese korral eemaldatakse päringu tulemusest korduvad read ja teisel juhul ei eemaldata. Antud juhul peaksid mõlemad päringud andma ühesuguse tulemuse, kuid huvi pakub see, kas on erinevusi nende põhjal tehtud päringute täitmisplaanides.

B2 eksperiment nimega „INTERSECT". Vaade *vw_Equal_names* sisaldab lõike leidmist (*INTERSECT* operatsiooni). Leitakse korduvad nimed (eesnimi ja perekonnanimi) patsientide ja tervishoiuteenuse osutajate seas:

```
CREATE VIEW vw_Equal_names (
    first_name, last_name )
AS
SELECT d.first_name, d.last_name
FROM Person d
INNER JOIN Health_care_visit_role vr ON d.party_id = vr.party_id
INTERSECT
SELECT p.first_name, p.last_name
FROM Person p
INNER JOIN Health_care_visit hcv ON p.party_id = hcv.patient_id;

SELECT * FROM vw_Equal_names;
```

B3 eksperiment nimega „COUNT". Vaade *vw_Number_of_patients_who_man* sisaldab kokkuvõttefunktsiooni *Count()* poole pöördumist, et leida ridade arv. Isikukoodi alusel leitakse meessoost isikute koguarv:

```
CREATE VIEW vw_Number_of_patients_who_man ( quantity )
AS
SELECT Count(personal_code) AS quantity
FROM Person
WHERE personal_code LIKE '3%'
    OR personal_code LIKE '5%';

SELECT * FROM vw_Number_of_patients_who_man;
```

B3.1 eksperiment nimega „GROUP BY“. Järgmises vaates *vw_Patient_suminfo* kasutatakse kokkuvõttefunktsiooni *Count(), Sum()* ja *Max()* ning *GROUP BY* klauslit. Leitakse iga patsiendi (isiku, kes on teinud vähemalt ühe visiidi) poolt visiiditasudele kulutatud rahasumma, visiitide kogus ja viimase visiidi kuupäev:

```
CREATE VIEW vw_Patient_visitinfo (
    patient_id, visit_fee_sum, visit_count, last_visit_date )
AS
SELECT hcv.patient_id,
    Sum(hcv.visit_fee) AS visit_fee_sum, Count(*) AS visit_count,
    Max(hcv.from_date) AS last_visit_date
FROM Health_care_visit hcv
GROUP BY hcv.patient_id;

SELECT * FROM vw_Patient_visitinfo;
```

B3.2 eksperiment nimega „GROUP JOIN“. Eksperimendis *„GROUP BY“* vaadeldakse grupeerimist. Selle vaate alampäringus üritatakse ühendada vaade *vw_Patient_info* (kus on kasutatud grupeerimist) ühe baastabeliga. Sellist tüüpi vaate valimise põhjuseks on teadusartikkel, kus uuritakse, mis juhtudel grupeerimine toimub enne ühendamist ja vastupidi [18]. Lisaks algses vaates juba esitatud informatsioonile esitatakse uues vaates patsiendi nimi, perekonnanimi, ja sünniaeg. See eksperiment aitab ka kindlaks teha, kas vaate loomine vaate põhjal segab optimeerijal parima täitmisplaani valimist või mitte. Otse baastabelite põhjal tehakse kaks loogiliselt samaväärset päringut, et kontrollida kas päringute täitmisplaan on sama. Esimesel juhul kasutatakse *FROM* klauslis alampäringut (*inline view*), teises päringus tasandatakse päring ära, st eemaldatakse *FROM* klauslis olev alampäring.

```
CREATE VIEW vw_Patient_suminfo (
    patient_id, first_name, last_name, birth_date,
    visit_fee_sum, visit_count, last_visit_date )
AS
SELECT hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.visit_count, hp.last_visit_date
FROM vw_Patient_visitinfo hp INNER JOIN Person pe ON hp.patient_id = pe.party_id;

SELECT * FROM vw_Patient_suminfo;

SELECT hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.visit_count, hp.last_visit_date
FROM (
    SELECT hcv.patient_id,
        Sum(hcv.visit_fee) AS visit_fee_sum, Count(*) AS visit_count,
        Max(hcv.from_date) AS last_visit_date
    FROM Health_care_visit hcv
    GROUP BY hcv.patient_id ) hp
INNER JOIN Person pe ON hp.patient_id = pe.party_id;
```

```
SELECT hcv.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    Sum(hcv.visit_fee) AS visit_fee_sum, Count(*) AS visit_count,
    Max(hcv.from_date) AS last_visit_date
FROM Health_care_visit hcv INNER JOIN Person pe ON hcv.patient_id = pe.party_id
GROUP BY hcv.patient_id, pe.first_name, pe.last_name, pe.birth_date;
```

B3.3 eksperiment nimega „GROUP JOIN FILTER“. Täiendatakse eelmist eksperimenti lisades päringusse *WHERE* klauslisse tingimuse, kus kasutatakse funktsiooni *EXTRACT (YEAR FROM ...)* ning mittekorreleeruvat alampäringut. Selle tingimusega otsitakse patsiente, kelle sünniaasta võrdub maksimaalse olemasolevaga sünniaastaga kõikide patsientide seas:

```
CREATE VIEW vw_Patient_suminfo_max_bdt (
    patient_id, first_name, last_name, birth_date,
    visit_fee_sum, visit_count, last_visit_date )
AS
SELECT hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.visit_count, hp.last_visit_date
FROM vw_Patient_visitinfo hp INNER JOIN Person pe ON hp.patient_id = pe.party_id
WHERE ( EXTRACT ( YEAR FROM pe.birth_date ) ) = (
    SELECT Max( EXTRACT ( YEAR FROM birth_date ) )
    FROM Person );

SELECT * FROM vw_Patient_suminfo_max_bdt;

SELECT hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.visit_count, hp.last_visit_date
FROM (
    SELECT hcv.patient_id,
        Sum(hcv.visit_fee) AS visit_fee_sum, Count(*) AS visit_count,
        Max(hcv.from_date) AS last_visit_date
    FROM Health_care_visit hcv
    GROUP BY hcv.patient_id ) hp
INNER JOIN Person pe ON hp.patient_id = pe.party_id
WHERE ( EXTRACT ( YEAR FROM pe.birth_date ) ) = (
    SELECT Max( EXTRACT ( YEAR FROM birth_date ) )
    FROM Person );

SELECT hcv.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    Sum(hcv.visit_fee) AS visit_fee_sum, Count(*) AS visit_count,
    Max(hcv.from_date) AS last_visit_date
FROM Health_care_visit hcv INNER JOIN Person pe ON hcv.patient_id = pe.party_id
WHERE ( EXTRACT ( YEAR FROM pe.birth_date ) ) = (
    SELECT Max( EXTRACT ( YEAR FROM birth_date ) )
    FROM Person )
GROUP BY hcv.patient_id, pe.first_name, pe.last_name, pe.birth_date;
```

B4 eksperiment nimega „DISTINCT“. Järgmine vaade sisaldab nii kokkuvõttefunktsiooni *Count()* poole pöördumist kui ka *DISTINCT* klausli abil korduvate ridade eemaldamist. Leitakse, kui mitu erinevat osapoolt on tervishoiu visiitidega seoses mingit rolli mänginud (v.a patsiendi roll, mida selles tabelis ei registreerita):

```
CREATE VIEW vw_Party_quantity_distinct_1 ( quantity )
AS
SELECT Count( DISTINCT vr.party_id ) AS quantity
FROM Health_care_visit_role vr;

SELECT * FROM vw_Party_quantity_distinct_1;
```

B4.1 eksperiment nimega „DISTINCT 2“. Vaade *vw_Party_quantity_distinct_2* on samaväärne eelmisele vaatele *vw_Party_quantity_distinct_1* ehk väljastab samasuguse tulemuse. Vaate *vw_Party_quantity_distinct_1* loomine tekitas idee luua samaväärse päringu, mis sisaldab *FROM* klauslis alampäringut, kus on kasutatud *DISTINCT* klauslit. Selleks loodi järgmine vaade:

```
CREATE VIEW vw_Party_quantity_distinct_2 ( quantity )
AS
SELECT Count(party_id) AS quantity
FROM (
    SELECT DISTINCT vr.party_id
    FROM Health_care_visit_role vr) pp;

SELECT * FROM vw_Party_quantity_distinct_2;
```

B5 eksperiment nimega „ORDER BY“. Selle vaate abil uuritakse *ORDER BY* klausli mõju täitmisplaani koostamisele. Esimeses osas leitakse visiitide andmed koos mõnede patsiendi isikuandmetega ja sorteeritakse tulemus patsiendi arvulise identifikaatori alusel kasvavalt. Teises osas (*B5.1*) lisatakse päringusse *WHERE* klausel, et leida ainult ühel kindlal kuupäeval tehtud visiidid.

```
CREATE VIEW vw_Visit_Patient_asc (
    health_care_visit_id, from_date, thru_date,
    patient_id, first_name, last_name, personal_code)
AS
SELECT hcv.health_care_visit_id, hcv.from_date, hcv.thru_date,
    hcv.patient_id, pe.first_name, pe.last_name, pe.personal_code
FROM Health_care_visit hcv, Person pe
WHERE hcv.patient_id = pe.party_id
ORDER BY hcv.patient_id;

SELECT * FROM vw_Visit_Patient_asc;
```

Tegin sellise otsuse kuna Heikens [23] väidab, et vaadetes (tehtud teise vaate põhjal) võib probleemiks olla *ORDER BY* klausli kasutamine. Samuti mainib Eessaar, et SQL standardi vanem versioon (SQL:1999) seab koguni piirangu: „Alampäring ei tohi sisaldada *ORDER BY* klauslit“ [1]. Seega teises osas kasutatakse vaadet kui alampäringut ning selle abil saab võrrelda millised täitmisplaanid lõpuks erinevates andmebaasisüsteemides saadakse.

B5.1 eksperiment nimega „ORDER BY FILTER“. See on eelmise eksperimendi teine osa, kus olemasolevatele päringutele (vaate põhjal ja baastabelite põhjal) lisatakse juurde *WHERE* klausel. Otse baastabelite põhjal tehtud päringud pole rangelt võttes samaväärsed. Päring, kus *ORDER BY* klausel on alampäringus, ei pea tegelikult lõpptulemuses ridade järjekorda säilitama. Praktikas ilmselt ridade järjekord säilib.

```
CREATE VIEW vw_Visit_Patient_dt (
    health_care_visit_id, from_date, thru_date,
    patient_id, first_name, last_name, personal_code)
AS
SELECT health_care_visit_id, from_date, thru_date,
    patient_id, first_name, last_name, personal_code
FROM vw_Visit_Patient_asc
WHERE from_date = TO_DATE('23.04.2013','DD.MM.YYYY');

SELECT * FROM vw_Visit_Patient_dt;

SELECT health_care_visit_id, from_date, thru_date,
    patient_id, first_name, last_name, personal_code
FROM (
    SELECT hcv.health_care_visit_id, hcv.from_date, hcv.thru_date,
        hcv.patient_id, pe.first_name, pe.last_name, pe.personal_code
    FROM Health_care_visit hcv, Person pe
    WHERE hcv.patient_id = pe.party_id
    ORDER BY hcv.patient_id ) pp
WHERE pp.from_date = TO_DATE('23.04.2013','DD.MM.YYYY');

SELECT hcv.health_care_visit_id, hcv.from_date, hcv.thru_date,
    hcv.patient_id, pe.first_name, pe.last_name, pe.personal_code
FROM Health_care_visit hcv, Person pe
WHERE hcv.patient_id = pe.party_id
  AND from_date = TO_DATE('23.04.2013','DD.MM.YYYY')
ORDER BY hcv.patient_id;
```

C. Selles kategoorias, sarnaselt A kategooriale, esitatakse sellised vaated, mille kasutamine ei tohiks tekitada optimeerijale segadust ja mis on võetud Oracle Database dokumentatsioonist [22].

C1 eksperiment nimega „NONCORR IN“. Vaade *vw_Visit_in_hospital* sisaldab mittekorreleeruvat alampäringut ja *IN predikaati.* Mittekorreleeruv alampäring pole põhipäringuga läbi põimunud ja seda saab käivitada eraldiseisva üksusena. Leitakse kõik tervishoiuteenuse visiidid, mis on tehtud haiglasse (*Hospital, facility type code = 'HOS01'*):

```
CREATE VIEW vw_Visit_in_hospital(
    health_care_visit_id, from_date, thru_date, patient_id,
    facility_id, postal_address_id, reason, visit_fee)
AS
SELECT health_care_visit_id, from_date, thru_date, patient_id,
    facility_id, postal_address_id, reason, visit_fee
```

```
FROM Health_care_visit
WHERE facility_id IN (
    SELECT facility_id
    FROM Facility
    WHERE facility_type_code = 'HOS01');

SELECT * FROM vw_Visit_in_hospital;
```

C2 eksperiment nimega „NONCORR NOTIN“. Vaade *vw_Organization_not_party* sisaldab mittekorreleeruva alampäringut ja *NOT IN predikaati*. Sellega leitakse kõik organisatsioonid, mis ei olnud tervishoiuteenuse visiitides osapoolteks:

```
CREATE VIEW vw_Organization_not_party (
    party_id, name, registration_code)
AS
SELECT party_id, name, registration_code
FROM Organization
WHERE party_id NOT IN (
    SELECT party_id
    FROM Health_care_visit_role
    WHERE party_id IS NOT NULL);

SELECT * FROM vw_Organization_not_party;
```

C3 eksperiment nimega „CORR“. Järgmises vaates kasutatakse korreleeruvat alampäringut, kus alampäring ja põhipäring on omavahel läbipõimunud. Leitakse kõik isikud, kelle kaal on suurem kui sama pikkusega isikute kaal keskmiselt.

```
CREATE VIEW vw_Person_weight (
    party_id, first_name, last_name, personal_code,
    birth_date, height, weight, username)
AS
SELECT p.party_id, p.first_name, p.last_name, p.personal_code,
    p.birth_date, p.height, p.weight, p.username
FROM Person p
WHERE p.weight > (
    SELECT Avg(pe.weight) AS avg_weight
    FROM Person pe
    WHERE pe.height = p.height );

SELECT * FROM vw_Person_weight;
```

D. Antud punktis vaadeldakse *WINDOW* funktsioone *LAG() OVER()* ja *ROW_NUMBER()*, mille kasutamise toetavad nii Oracle Database kui ka PostgreSQL andmebaasisüsteemid. Lisaks vaadatakse ridade hulga piiramist päringu tulemuses *FETCH FIRST n ROWS ONLY* klausli abil, mida samuti toetavad mõlemad andmebaasisüsteemid. Samuti käsitletakse Oracle Database andmebaasisüsteemi-spetsiifilist pseudoveergu *ROWNUM*, mis samuti aitab piirata

väljastava ridade hulga. Valiku põhjuseks sai veebipäeviku postitus [39] *ROW_NUMBER()* ja *ROWNUM* kohta, kus päringute *FROM* klauslites kasutati alampäringuid (*inline views*) ning päringu täitmisplaanis on näha, et optimeerija töötleb alampäringuid eraldi (täitmisplaanis kasutatud *VIEW* operatsioon).

D1 eksperiment nimega „ROWNUMBER“. Vaade *vw_Party_rownumber_by_username* on võimalik luua nii Oracle Database kui ka PostgreSQL andmebaasisüsteemis. Vaate abil leitakse iga isiku identifikaator, eesnimi ja perekonnanimi, kasutajanimi ja järjekorranumber kui sorteerida read kasutajanime alusel kasvavalt:

```
CREATE VIEW vw_Party_rownumber_by_username (
    party_id, first_name, last_name, username, nr)
AS
SELECT party_id, first_name, last_name, username,
    ROW_NUMBER() over (ORDER BY username) AS nr
FROM Person;

SELECT * FROM vw_Party_rownumber_by_username;
```

D2 eksperiment nimega „FETCH“. Järgneva vaate saab luua ainult Oracle Database andmebaasisüsteemis kuna PostgreSQL andmebaasisüsteem ei toeta pseudoveergu *ROWNUM*. *FETCH FIRST n ROWS ONLY* klausel, mis piirab väljastava ridade arvu, on kasutatav mõlemas andmebaasisüsteemis. Vaate alampäring on sarnane eelmisele, kuid tulemuses on 999 esimest isikut (esimesed nimekirjas, mis tekib kasutajanime järgi kasvavalt sorteerimise tulemusel).

```
CREATE VIEW vw_Party_fetch_999_username (
    party_id, first_name, last_name, username, nr)
AS
SELECT party_id, first_name, last_name, username, ROWNUM AS nr
FROM (
    SELECT party_id, first_name, last_name, username
    FROM Person
    ORDER BY username ) pp
FETCH FIRST 999 ROWS ONLY;

SELECT * FROM vw_Party_fetch_999_username;
```

PostgreSQL andmebaasisüsteemis on võimalik rea numbrit väljastada, kui luua arvujada generaator ning vaate alampäringus selle poole pöörduda (iga kord tuleb enne päringu käivitamist arvujada generaator *setval* funktsiooni abil *„nullida“* (määrata, et järgmine väljastatav väärtus on 1)).

```
CREATE SEQUENCE nr MINVALUE 0 START WITH 1;
```

```
CREATE VIEW vw_Party_fetch_999_username (
    party_id, first_name, last_name, username, nr)
AS
SELECT party_id, first_name, last_name, username, nextval('nr') AS nr
FROM (
    SELECT party_id, first_name, last_name, username
    FROM Person
    ORDER BY username ) pp
FETCH FIRST 999 ROWS ONLY;

SELECT SETVAL('nr',0);

SELECT * FROM vw_Party_fetch_999_username;
```

D2.1 eksperiment nimega „ROWNUM“. Kuna *FETCH FIRST n ROWS ONLY* töötab Oracle Database andmebaasisüsteemis alates versioonist 12c Release 1, siis varasemate versioonide puhul saab kasutada järgmist vaate alampäringut:

```
CREATE VIEW vw_Party_rownum_999_username (
    party_id, first_name, last_name, username, nr)
AS
SELECT party_id, first_name, last_name, username, ROWNUM AS nr
FROM (
    SELECT party_id, first_name, last_name, username
    FROM Person
    ORDER BY username ) pp
WHERE ROWNUM <= 999;

SELECT * FROM vw_Party_rownum_999_username;
```

D2.2 eksperiment nimega „ROWNUMBER 999“. Järgmine vaate alampäring, mis sisaldab aknafunktsiooni (*WINDOW*) *ROW_NUMBER() OVER ()* poole pöördumist, töötab mõlemas käsitletavas andmebaasisüsteemis:

```
CREATE VIEW vw_Party_rownumbe_999_username (
    party_id, first_name, last_name, username, nr)
AS
SELECT party_id, first_name, last_name, username, nr
FROM (
  SELECT party_id, first_name, last_name, username,
      ROW_NUMBER() OVER (ORDER BY username) AS nr
  FROM Person ) pp
WHERE nr <= 999;

SELECT * FROM vw_Party_rownumbe_999_username;
```

D3 eksperiment nimega „LAG“. David Rowley meilivahetuses [17], mis on saadaval PostgreSQL veebilehel, tõstatati probleem seoses aknafunktsiooniga *LAG() OVER()*. Kui seda kasutada vaates, siis vaate põhjal tehtud päringu täitmisplaanis ei kasutata andmebaasis loodud

indeksit. Samas seda indeksit kasutatakse baastabeli põhjal otse tehtud päringu korral. Vastuses oletatakse, et seda tingib aknafunktsiooni kasutamine vaate alampäringus. Sellisel juhul ei suuda PostgreSQL mestida vaate põhjal tehtud päringut vaate alampäringuga. Kirjavahetuse kohaselt esines see probleem ka sellel hetkel kõige viimases PostgreSQL versioonis – PostgreSQL 9.3.

Seega järgnevalt luuakse sarnane vaade. Selle abil leitakse iga tervishoiuteenuse visiidi kohta järgmise informatsioon: visiidi identifikaator, patsiendi identifikaator, visiidi algus- ja lõpukuupäev, visiiditasu ning patsiendi eelmise visiidi tasu. Kui see on patsiendi esimene visiit, siis eelmise visiidi tasu asendatakse *COALESCE* funktsiooni abil väärtusega „0.00“:

```
CREATE VIEW vw_Visit_fee_previous (
    health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
prev_visit_fee )
AS
SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
    COALESCE (
    LAG(visit_fee) OVER (
        PARTITION BY patient_id ORDER BY from_date), 0.00) AS prev_visit_fee
FROM Health_care_visit;

SELECT * FROM vw_Visit_fee_previous;
```

D3.1 eksperiment nimega „LAG FILTER“. Järgmisena sammuna lisatakse päringule *WHERE* klausel (nagu ka David Rowley päringus [17]), et leida andmed ainult ühe kindla patsiendi kohta:

```
CREATE VIEW vw_Visit_fee_prev_filter (
    health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
prev_visit_fee )
AS
SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
prev_visit_fee
FROM vw_Visit_fee_previous
WHERE patient_id = 67329;

SELECT * FROM vw_Visit_fee_prev_filter;

SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
prev_visit_fee
FROM (
    SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
        COALESCE (
        LAG(visit_fee) OVER (
            PARTITION BY patient_id ORDER BY from_date), 0.00) AS prev_visit_fee
    FROM Health_care_visit ) pp
WHERE pp.patient_id = 67329;

SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
```

```
  COALESCE (
  LAG(visit_fee) OVER (
    PARTITION BY patient_id ORDER BY from_date), 0.00) AS prev_visit_fee
FROM Health_care_visit
WHERE patient_id = 67329;
```

E. Kõige problemaatilisemaks osaks on vaadete loomine, mitte baastabelite, vaid teiste vaadete põhjal, mis võivad ka omakorda olla loodud vaadete põhjal. Sellised vaated raskendavad optimaalse täitmisplaani leidmist ja seda probleemi mainitakse paljudes allikates. Oracle Database dokumentatsioonis [14] rõhutakse, et kõige halvem vaate kasutamise viis on siis, kui vaade viitab teistele vaadetele ja kui vaated ühendatakse päringus. Larry Burns [9, p. 175] soovitab vältida vaadete alampäringutes vaadetele viitamist, eriti rohkem kui kahel või kolmel tasemel. Järgnevad vaated ja päringud puudutavad seda problemaatikat.

E1 eksperiment nimega „VIEWofVIEW“ või „OUTER JOIN“. Kui vaate alampäringus kasutatakse välisühendamist (*OUTER JOIN*) mitme teise vaate ühendamiseks, siis optimeerija võib valida mitteoptimaalse täitmisplaani. Ühes Oracle Database dokumentatsioonis [40] tuuakse välja ka näidispäringu täitmisplaan *(„Outer join to a multitable view“)* ja selgitatakse, et optimeerija ei suuda baastabeli välisühendamisel mitme tabeli ühendamise tulemusena loodud vaatega baastabelit ja vaadet mestida. Seetõttu täidetakse vaate lause osa eraldi, baastabeli põhjal tehtud lause osa eraldi ning tulemus ühendatakse.

Selleks, et leida milline täitmisplaan pakutakse välja kahe vaate välisühendamise korral, luuakse kaks vaated – *vw_Patients_visits* ja *vw_Employers*. Vaates *vw_Patients_visits* ühendatakse tervishoiuteenuse visiidid patsientidega.

```
CREATE VIEW vw_Patients_visits (
    party_id, first_name, last_name, personal_code,
    health_care_visit_id, from_date, thru_date, reason, visit_fee,
    facility_id, postal_address_id  )
AS
SELECT pe.party_id, pe.first_name, pe.last_name, pe.personal_code,
    v.health_care_visit_id, v.from_date, v.thru_date, v.reason, v.visit_fee,
    v.facility_id, v.postal_address_id
FROM Person pe INNER JOIN Health_care_visit v ON v.patient_id = pe.party_id;
```

Vaates *vw_Employers* otsitakse kõiki organisatsioone, mis saatsid oma töötaja tervisekontrolli.

```
CREATE VIEW vw_Employers (
    party_id, name, registration_code, health_care_visit_id,
    role_type_code )
AS
SELECT vr.party_id, org.name, org.registration_code, vr.health_care_visit_id,
    vr.role_type_code
FROM Organization org INNER JOIN Health_care_visit_role vr
ON vr.party_id = org.party_id
WHERE vr.role_type_code = 'EMPLOYER02';
```

Vaates *vw_Employee_Employer* leitakse patsiente ja nende tööandjaid (juhul, kui info on olemas). Selle vaate alampäringus leitakse vasakpoolse välisühendamise (*LEFT OUTER JOIN*) kõik read vaatest *vw_Patients_visits* ning ühendatakse tööandjatega (vaade *vw_Employers*). Tulemuses on näidatud ka sellised patsiendid, keda ei saadetud tervisekontrollile tööandja poolt:

```
CREATE VIEW vw_Employee_Employer (
    employee, personal_code, employer, registration_code )
AS
SELECT DISTINCT ( v_pe.first_name || ' ' || v_pe.last_name ) AS employee,
    v_pe.personal_code, org_vr.name AS employer, org_vr.registration_code
FROM vw_Patients_visits v_pe LEFT OUTER JOIN  vw_Employers org_vr
    ON org_vr.health_care_visit_id = v_pe.health_care_visit_id;

SELECT * FROM vw_Employee_Employer;
```

Samasuguse tulemuse andvad päringud baastabelite põhjal:

```
SELECT DISTINCT ( v_pe.first_name || ' ' || v_pe.last_name ) AS employee,
    v_pe.personal_code, org_vr.name AS employer, org_vr.registration_code
FROM
    (SELECT pe.party_id, pe.first_name, pe.last_name, pe.personal_code,
        v.health_care_visit_id, v.from_date, v.thru_date, v.reason, v.visit_fee
    FROM Person pe
    INNER JOIN Health_care_visit v ON v.patient_id = pe.party_id ) v_pe
LEFT OUTER JOIN (
    SELECT vr.party_id, org.name, org.registration_code, vr.health_care_visit_id,
        vr.role_type_code
    FROM Organization org
    INNER JOIN Health_care_visit_role vr ON vr.party_id = org.party_id
    WHERE vr.role_type_code = 'EMPLOYER02' ) org_vr
ON org_vr.health_care_visit_id = v_pe.health_care_visit_id;
```

```
SELECT DISTINCT ( pe.first_name || ' ' || pe.last_name ) AS employee,
    pe.personal_code, org.name AS employer, org.registration_code
FROM  Health_care_visit v INNER JOIN Person pe  ON v.patient_id = pe.party_id
  LEFT OUTER JOIN (
      SELECT vr.party_id, vr.health_care_visit_id, vr.role_type_code
      FROM Health_care_visit_role vr
      WHERE vr.role_type_code = 'EMPLOYER02') vr
    ON vr.health_care_visit_id = v.health_care_visit_id
  LEFT OUTER JOIN Organization org ON vr.party_id = org.party_id;
```

E2 eksperiment nimega „VIEWofVIEW 2“ või „OUTER JOIN 2“. Vaates *vw_Visits_facil_and_addr* välisühendatakse *vw_Patients_visits* baastabelitega *FACILITY* ja *POSTAL_ADDRESS*. Selle päringuga saab ära proovida, kas baastabelite välisühendamine vaatega, kus on ühendatud kaks baastabelid (vt *E1*), võib tekitada optimeerijale segadust optimaalse täitmisplaani genereerimisel või mitte.

```
CREATE VIEW vw_Visits_facil_and_addr (
    health_care_visit_id, patient_id, first_name, last_name,
    from_date, thru_date, reason, visit_fee,
    facility_id, facility_name, facility_type_code, description,
    postal_address_id, address, city, directions )
AS
SELECT hcv.health_care_visit_id, hcv.party_id, hcv.first_name, hcv.last_name,
    hcv.from_date, hcv.thru_date, hcv.reason, hcv.visit_fee,
    hcv.facility_id, fa.facility_name, fa.facility_type_code, fa.description,
    hcv.postal_address_id, pa.address, pa.city, pa.directions
FROM vw_Patients_visits hcv
LEFT OUTER JOIN Facility fa ON fa.facility_id = hcv.facility_id
LEFT OUTER JOIN Postal_address pa ON pa.postal_address_id= hcv.postal_address_id
WHERE hcv.from_date
               BETWEEN TO_DATE( '01.01.2013', 'DD.MM.YYYY' )
               AND TO_DATE( '01.01.2014', 'DD.MM.YYYY' );

SELECT * FROM vw_Visits_facil_and_addr;
```

Samasuguse tulemuse andvad päringud baastabelite põhjal:

```
SELECT hcv.health_care_visit_id, hcv.party_id, hcv.first_name, hcv.last_name,
    hcv.from_date, hcv.thru_date, hcv.reason, hcv.visit_fee,
    hcv.facility_id, fa.facility_name, fa.facility_type_code, fa.description,
    hcv.postal_address_id, pa.address, pa.city, pa.directions
FROM ( SELECT pe.party_id, pe.first_name, pe.last_name, pe.personal_code,
        v.health_care_visit_id, v.from_date, v.thru_date, v.reason,
        v.visit_fee, v.facility_id, v.postal_address_id
    FROM Person pe
    INNER JOIN Health_care_visit v ON v.patient_id = pe.party_id ) hcv
LEFT OUTER JOIN Facility fa ON fa.facility_id = hcv.facility_id
LEFT OUTER JOIN Postal_address pa ON pa.postal_address_id= hcv.postal_address_id
WHERE hcv.from_date
        BETWEEN TO_DATE( '01.01.2013', 'DD.MM.YYYY' )
        AND TO_DATE( '01.01.2014', 'DD.MM.YYYY' );

SELECT hcv.health_care_visit_id, pe.party_id, pe.first_name, pe.last_name,
```

```
    hcv.from_date, hcv.thru_date, hcv.reason, hcv.visit_fee,
    hcv.facility_id, fa.facility_name, fa.facility_type_code, fa.description,
    hcv.postal_address_id, pa.address, pa.city, pa.directions
FROM Person pe INNER JOIN Health_care_visit hcv ON hcv.patient_id = pe.party_id
LEFT OUTER JOIN Facility fa ON fa.facility_id = hcv.facility_id
LEFT OUTER JOIN Postal_address pa ON pa.postal_address_id= hcv.postal_address_id
WHERE hcv.from_date
        BETWEEN TO_DATE( '01.01.2013', 'DD.MM.YYYY' )
        AND TO_DATE( '01.01.2014', 'DD.MM.YYYY' );
```

F. Leidub ka selliseid päringuid, mida on keelatud kasutada vaate alampäringuna. Üheks selliseks on päring, mis tagastab pesastatud kursori (*nested cursor*) [41]. Oracle Database dokumentatsioonis [41] on kirjutatud, et kursori kasutamisele on piiranguid ning üheks neist on, et pesastatud kursorit ei tohi kasutada vaadetes. Selle kontrollimiseks üritatakse luua järgmise vaate:

```
CREATE VIEW vw_Cursor_test (
    contact_number, last_name_party_id)
AS
SELECT CURSOR(
    SELECT p.contact_number
    FROM Party p
    WHERE p.party_id = pp.party_id ),
    pp.last_name, pp.party_id
FROM Person pp;
```

Veateateks saadakse Oracle Database andmebaasisüsteemis:

```
SQL Error: ORA-22902: CURSOR expression not allowed
*Cause:    CURSOR on a subquery is allowed only in the top-level SELECT list of a
query.
```

PostgreSQL andmebaasisüsteemis saab kursoreid luua *DECLARE* lausega, kuid ei ole võimalik kursorit defineerida *SELECT* lauses.

G. Oracle Database dokumentatsioonis (vanemas, 2003. a) [38] räägitakse, et ka rekursiivne päring kasutades *CONNECT BY* takistab vaadete mestimist. Selle eksperimendi jaoks lõin baastabelis *Organization* veeru *parent_id*. Seega igal organisatsioonil on null või rohkem alluvat organisatsiooni ning null kuni üks organisatsiooni, millele see organisatsioon allub.

G1 eksperiment nimega „RECURSIVE“. Järgnevalt esitatakse esimene vaade Oracle Database andmebaasisüsteemi jaoks ja teine vaade PostgreSQL andmebaasisüsteemi jaoks.

```
CREATE VIEW vw_Organization_hierarchy ( level_id, party_id, parent_id )
AS
SELECT LEVEL AS level_id, party_id, parent_id
FROM Organization
CONNECT BY PRIOR party_id = parent_id
START WITH parent_id IS NULL;

CREATE VIEW vw_Organization_hierarchy (level, party_id, parent_id)
AS
WITH RECURSIVE hierarchy( level, party_id, parent_id ) AS (
    SELECT 1 as level, party_id, parent_id
    FROM Organization
    WHERE parent_id IS NULL
    UNION ALL
    SELECT hierarchy.level + 1, org.party_id, org.parent_id
    FROM Organization org JOIN hierarchy ON hierarchy.party_id = org.parent_id )
SELECT level, party_id, parent_id
FROM hierarchy;

SELECT * FROM vw_Organization_hierarchy;
```

4. Eksperimendi tulemused

Selles peatükis tutvustatakse eksperimendi tulemusi tuues välja nii koondtulemused kui ka analüüsides mõningaid huvipakkuvaid päringuid. Töö maht ei luba enamike päringute täitmist detailselt kirjeldada kuid loodan, et kirjeldatud päringute täitmisega tutvumine annab lugejale rohkem infot optimeerija otsuste kaalutlustest ning raskustest, millega see peab igapäevaselt silmitsi seisma.

4.1 Koondtulemused

Selles jaotises esitatakse eksperimentide koondtulemused. Meenutan, et töö eesmärkideks on uurida, kuidas mõjutab vaadete kasutamine päringute täitmisplaane, st kas otse baastabelite põhjal tehtud päringule ja loogiliselt samaväärsele vaate põhjal tehtud päringule koostatakse ühesugused täitmisplaan või mitte ning kuidas erinevad nende päringute täitmise kiirused. Eksperimendi tulemused pannakse kirja koondtabelitesse. Selles jaotises esitatud koondtabeli ridadeks on andmebaasisüsteemi versioonid koos infoga selle kohta kas päring on vaate või baastabelite põhjal ja veergudeks eksperimendid. Tabeli lahtrites on täisarvud. Kahes või rohkemas lahtris on ühesugune täisarv kui täitmisplaanid olid ühesugused. Arvud on suvalised arvud, mis ei ole seotud päringu maksumuse, keerukuse või täitmiseks kulunud ajaga. Kui ühes lahtris on kaks täisarvu, siis see tähendab, et täitmisplaan on võrreldatavatest täitmisplaanidest erinev, kuid omab sarnasust mõne teise täitmisplaaniga. Näiteks "13 / 11" korral tähendab 13, et see täitmisplaan ei ole identne mingi teise täitmisplaaniga, kuid on millegi poolest osaliselt sarnane 11 täitmisplaaniga (kuid mitte täiesti identne).

Koondtabelites kasutatavad lühendid:

- *„VW“ (views)* tähendab vaate (või vaadete) põhjal tehtud päringute täitmisplaane;
- *„BT“ (base tables)* tähendab otse baastabeli (või baastabelite) põhjal tehtud päringute täitmisplaane;
- *„BT 2“* tähendab samuti baastabeli (või baastabelite) põhjal tehtud päringute täitmisplaane kusjuures päring on lahti kirjutatud (ära tasandatud) alternatiivne päring *„BT“*-le;
- *„SVW“* – tähendab PostgreSQL turvabarjääriga vaate (või vaadete) põhjal tehtud päringute täitmisplaane (selliseid vaateid on võimalik defineerida alates PostgreSQL

andmebaasisüsteemi versioonist 9.2). Selliste vaadete põhjal tehtud päringute täitmisplaane võrreldakse ainult *„VW“* täitmisplaanidega (ainult PostgreSQL versioonis 9.3).

Sinise taustavärviga märgitud täitmisplaanidel on ühesugune erinevus. Nendes on vaadetele tehtud päringute täitmisplaanides kasutatud lisaks *„VIEW“* operatsiooni (või PostgreSQL andmebaasisüsteemis *„SubQuery“* operaatorit).

Helesinise taustavärviga märgitud täitmisplaanid tähendavad sama, mis sinisega tähistatud. Kuid lisaks erinevad sellised täitmisplaanid uuema andmebaasisüsteemi versiooni täitmisplaanidest täielikult või osaliselt (st optimeerimismooduli töös on toimunud suured muudatused). Erinevused ei ole põhjustatud vaadete kasutamises.

Punase värviga tähistatud täitmisplaanid erinevad teistel põhjustel. Need päringud on sarnased selle poolest, et kasutavad vaadet, mis on tehtud teiste vaadete põhjal.

Tabel 3 esitab Oracle Database andmebaasisüsteemi erinevates versioonides saadud tulemused.

Tabel 3. Täitmisplaanide erinevused Oracle Database andmebaasisüsteemi erinevates versioonides

		IN	LEFT OUTER	UNION	UNION ALL	INTERSECT	COUNT	GROUP BY	GROUP JOIN	GROUP JOIN FILTER	DISTINCT	DISTINCT 2	ORDER BY	ORDER BY FILTER	NONCORR IN	NONCORR NOTIN	CORR	ROWNUMBER	FETCH	ROWNUM	ROWNUMBER 999	LAG	LAG FILTER	VIEWofVIEW	VIEWofVIEW 2	RECURSIVE
Oracle 12c	***VW***	1	3	5	7	9	11	15	17	20 /19	22 /11	26 /11	28	30	32	33	17	34	38	39	41	34	43 /34	45	51	53
	BT	1	3	6	8	10	12	15	17	20 /19	23 /12	27 /12	28	31	32	33	17	35	38	40	41	35	43 /34	46	51	54
	BT 2								19	20 /19				31									44 /35	47	51	
Oracle 11g	***VW***	2	4	5	7	9	13 /11	16	18	21 /18	24	26 /11	29	30	29	33	17	36		39	42	34	43 /34	48	52 /51	55
	BT	2	4	6	8	10	14 /12	16	18	21 /18	25	27 /12	29	31	29	33	17	37		40	42	35	43 /34	49	52 /51	56
	BT 2								18	21 /18				31									44 /35	50	52 /51	

Oracle Database andmebaasisüsteemis versiooniga 12c (Release 1) on optimeerija töös toimunud suured muudatused võrreldes Oracle Database 11.1 andmebaasisüsteemiga. Enamus

täitmisplaane erineb vaadeldavate versioonide vahel. Kuna erinevus ei tulene vaadete kasutamisest, siis nende täitmisplaanide erinevusi täpsemalt ei käsitleta.

Eelneva tabeli (vt Tabel 3) põhjal arvutatud koondtulemused (vt Tabel 4).

Tabel 4. Oracle Database koondandmete põhjal leitud arvulised näitajad

Erinevate eksperimentide arv (Oracle 12c / Oracle 11g)	***25 / 24***
Oracle Database 12.1 põhjal tehtud eksperimentide arv (ja protsent eksperimentide koguarvust), millal vaate põhjal ja otse baastabeli põhjal andsid tulemuseks *erineva* täitmisplaani (erinevust arvestati ka siis, kui see tekkis otse baastabelite põhjal tehtud alternatiivse päringu korral)	***14 (56%)***
Oracle Database 11.1 põhjal tehtud eksperimentide arv (ja protsent eksperimentide koguarvust), millal vaate põhjal ja otse baastabeli põhjal andsid tulemuseks *erineva* täitmisplaani (erinevust arvestati ka siis, kui see tekkis otse baastabelite põhjal tehtud alternatiivse päringu korral)	***13 (54%)***
Eksperimentide arv (ja protsent eksperimentide koguarvust), millal Oracle Database 12.1 ja Oracle Database 11.1 vaadete põhjal tehtud päringud andsid tulemuseks *erineva* täitmisplaani	***15 (63%)***

Tabel 5 esitab PostgreSQL andmebaasisüsteemi erinevates versioonides saadud tulemused. Rohelisega märgitud täitmisplaanid erinevad PostgreSQL 9.3 korral *„VW“* täitmisplaanidest.

Tabel 5. Täitmisplaanide erinevused PostgreSQL andmebaasisüsteemi erinevates versioonides

		IN	LEFT OUTER	UNION	UNION ALL	INTERSECT	COUNT	GROUP BY	GROUP JOIN	GROUP JOIN FILTER	DISTINCT	DISTINCT 2	ORDER BY	ORDER BY FILTER	NONCORR IN	NONCORR NOTIN	CORR	ROWNUMBER	FETCH	ROWNUMBER 999	LAG	LAG FILTER	VIEWofVIEW	VIEWofVIEW 2	RECURSIVE
PostgreSQL 9.3	***SVW***	1	1	2	3 /2	4	6	7	8	10	6	12	13	27 /13	15	16	17 /16	18	19	20	21	22	28	29	26
	VW	1	1	2	3 /2	4	6	7	8	10	6	12	13	14	15	16	17 /16	18	19	20	21	22	24	25	26
	BT	1	1	2	3 /2	5	6	7	8	10	6	12	13	14	15	16	17 /16	18	19	20	21	22	24	25	26
	BT 2								9 /7	11				14								23 /18	24	25	
PostgreSQL 9.1	***VW***	1	1	2	3 /2	4	6	7	8	10	6	12	13	14	15	16	17 /16	18	19	20	21	22	24	25	26
	BT	1	1	2	3 /2	5	6	7	8	10	6	12	13	14	15	16	17 /16	18	19	20	21	22	24	25	26
	BT 2								9 /7	11				14								23 /18	24	25	

Eelneva tabeli (vt Tabel 5) põhjal arvutatud koondtulemused (vt Tabel 6).

Tabel 6. PostgreSQL koondandmete põhjal leitud arvulised näitajad

Erinevate eksperimentide arv	***24***
PostgreSQL 9.3 põhjal tehtud eksperimentide arv (ja protsent eksperimentide koguarvust), millal vaate põhjal ja otse baastabeli põhjal andsid tulemuseks *erineva* täitmisplaani (erinevust arvestati ka siis, kui see tekkis otse baastabelite põhjal tehtud alternatiivse päringu korral)	***4 (17%)***
PostgreSQL 9.1 põhjal tehtud eksperimentide arv (ja protsent eksperimentide koguarvust), millal vaate põhjal ja otse baastabeli põhjal andsid tulemuseks *erineva* täitmisplaani (erinevust arvestati ka siis, kui see tekkis otse baastabelite põhjal tehtud alternatiivse päringu korral)	***4 (17%)***

PostgreSQL 9.3 põhjal tehtud eksperimentide arv (ja protsent eksperimentide koguarvust), millal turvabarjääriga ja turvabarjäärita vaadete põhjal tehtud päringutel olid *erinevad* täitmisplaanid	***3 (13%)***
Eksperimentide arv (ja protsent eksperimentide koguarvust), millal PostgreSQL 9.3 ja PostgreSQL 9.1 vaadete põhjal tehtud päringud andsid tulemuseks *erineva* täitmisplaani	***0 (0%)***

4.2 Oracle Database andmebaasisüsteemi täitmisplaanide erinevusi

Selles töös kirjeldatavad täitmisplaanid on andmebaasisüsteemi poolt koostatud füüsilised täitmisplaanid. Täitmisplaan sisaldab süsteemi sisemise-taseme operatsioone ja nende teostamise järjekorda. Andmebaasisüsteem peab päringu tulemuse saavutamise läbi viima täitmisplaanis kirjeldatud operatsioone ja tegema seda plaaniga ettenähtud järjekorras. Täitmisplaan loetakse alt üles, paremalt vasakule. Allpool/Paremal esitatud operatsiooni tulemus on sisendiks ülalpool/vasakul olevale operatsioonile. Iga operatsiooni juures esitatakse hinnanguline (*E-Rows*) ja tegelik ridade arv (*A-Rows*), hinnanguline andmete hulk baitides (*E-Bytes*), hinnang operatsiooni maksumusele (*Cost*), operatsiooni käivitamise arv (*Starts*), toimunud lugemisi muutmälust (*Buffers*, tuntud ka kui *Consistent gets*) ja kettalt loetud andmeplokkide arv (*Reads*, tuntud ka kui *Physical Reads*). Puu juur ehk rida identifikaatoriga 0 esitab hinnangu lausele kui tervikule [42].

Järgnevalt kirjeldan lühidalt vaate põhjal tehtud päringu täitmisplaani (*B3.2 eksperiment nimega „GROUP JOIN“* – päring vaatele) (vt Joonis 12). Selle vaate alampäringus ühendatakse kokku vaade (*vw_Patient_visitinfo*) ja baastabel (*Person*).

```
CREATE VIEW vw_Patient_suminfo (
    patient_id, first_name, last_name, birth_date,
    visit_fee_sum, visit_count, last_visit_date )
AS
SELECT hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.visit_count, hp.last_visit_date
FROM vw_Patient_visitinfo hp INNER JOIN Person pe ON hp.patient_id = pe.party_id;
```

```
-------------------------------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                 | Starts | Cost (%CPU)| A-Rows |   A-Time   | Buffers | Used-Mem |
-------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                      |      1 |  6148 (100)|  70000 |00:00:01.72 |   12912 |          |
|*  1 |  HASH JOIN           |                      |      1 |  6148   (1)|  70000 |00:00:01.72 |   12912 | 5557K (0)|
|   2 |   TABLE ACCESS FULL  | PERSON               |      1 |   239   (1)|  70000 |00:00:00.02 |     822 |          |
|   3 |   VIEW               | VW_PATIENT_VISITINFO |      1 |  5575   (1)|  70000 |00:00:01.41 |   12090 |          |
|   4 |    HASH GROUP BY     |                      |      1 |  5575   (1)|  70000 |00:00:01.33 |   12090 | 4963K (0)|
|   5 |     TABLE ACCESS FULL| HEALTH_CARE_VISIT    |      1 |  3298   (1)|   1000K|00:00:00.30 |   12090 |          |
-------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("HP"."PATIENT_ID"="PE"."PARTY_ID")
```

Joonis 12. Vaate põhjal tehtud päringu täitmisplaan *GROUP JOIN*

Operatsioon *TABLE ACCESS* näitab, et päringu täitmiseks loetakse tabeli plokke. *VIEW* operatsioon näitab, et süsteem täidab vaate põhjal tehtud päringu ja siis tagastab tulemuse järgmisele operatsioonile sisendiks. Esialgu loetakse kõiki tabeli *Health_care_visit* kasutuses olevaid plokke (kuni kõrgveemärgini) ja vaadatakse selle käigus läbi kõik tabeli read (*TABLE ACCESS FULL*). Järgmisena viiakse läbi grupeerimisoperatsioon.

Tabelite poole pöördumise järjekord täitmisplaanis näitab milline tabelitest on sisemine ja milline on väline (*build table, probe table*) [43]. Tabelite ühendamiseks kasutatakse antud juhul algoritmi *hash join*. Sisemiseks tabeliks on *Person,* mille põhjal luuakse räsitabel, kuna selles tabelis on vähem ridu (ja seega on suurem tõenäosus, et see räsitabel mahub täielikult mällu). Väliseks tabeliks on *Vaade VW_Patient_Visitinfo*.

Võrdluseks esitan alternatiivse päringu (*B3.2 eksperiment nimega „GROUP JOIN“* – teine päring, mis tehtud baastabelite põhjal), milles pole kasutatud viitamist vaatele ega alampäringuid (ja *GROUP BY* klausel on tavapäraselt päringu lõpus). Kuid lõpptulemusena koostas Oracle Database andmebaasisüsteem eelmisega (vt Joonis 12) *sarnase* täitmisplaani (vt Joonis 13).

```
SELECT hcv.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    Sum(hcv.visit_fee) AS visit_fee_sum, Count(*) AS visit_count,
    Max(hcv.from_date) AS last_visit_date
FROM Health_care_visit hcv INNER JOIN Person pe ON hcv.patient_id = pe.party_id
GROUP BY hcv.patient_id, pe.first_name, pe.last_name, pe.birth_date;
```

```
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation            | Name              | Starts | Cost (%CPU)| A-Rows |   A-Time   | Buffers | Used-Mem |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                   |      1 |  7248 (100)|  70000 |00:00:01.85 |   12912 |          |
|   1 |  HASH GROUP BY       |                   |      1 |  7248   (1)|  70000 |00:00:01.85 |   12912 | 9493K (0)|
|*  2 |   HASH JOIN          |                   |      1 |  6121   (1)|  70000 |00:00:01.64 |   12912 | 5544K (0)|
|   3 |    TABLE ACCESS FULL | PERSON            |      1 |   239   (1)|  70000 |00:00:00.02 |     822 |          |
|   4 |    VIEW              | VW_GBC_5          |      1 |  5575   (1)|  70000 |00:00:01.42 |   12090 |          |
|   5 |     HASH GROUP BY    |                   |      1 |  5575   (1)|  70000 |00:00:01.37 |   12090 | 4963K (0)|
|   6 |      TABLE ACCESS FULL| HEALTH_CARE_VISIT |      1 |  3298   (1)|   1000K|00:00:00.33 |   12090 |          |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("ITEM_1"="PE"."PARTY_ID")
```

Joonis 13. Baastabelite põhjal tehtud päringu täimisplaan *GROUP JOIN*

Erinevuseks on ainult see, et peale ühendamist (*HASH JOIN*) viiakse veelkord läbi grupeerimine (*HASH GROUP BY*). Selline andmebaasisüsteemi käitumine oleks põhjendatud, kui peale ühendamist ridade arv (*A-Rows*) suureneks. Sellisel juhul tuleks tõesti grupeerimine uuesti läbi viia (st peale ühendamist tekkinud korduvaid ridu ära grupeerida). Antud juhul ridade arv ei muutu, sest iga visiit on seotud täpselt ühe patsiendiga.

Selline käitumine, kus grupeerimine viidi läbi enne ühendamist, on Oracle Database andmebaasisüsteemis *„Group-By Placement“* teisenduse ilming. See on päringu teisendus, mis võimaldab optimeerijal vähendada järgnevalt ühendatavate ridade arvu [44]. Sooviksin hoiatada, et *„Group-By Placement“* kasutamine võib Oracle 11g korral põhjustada ebakorrektseid päringu tulemusi [45]. Mina siiski ei märganud sellist probleemi. Võimalike ohtude vältimiseks on võimalik *„Group-By Placement“* kasutamine andmebaasisüsteemis välja lülitada (sellisel korral on täitmisplaan nagu järgmisel joonisel (vt Joonis 14)):

```
ALTER SESSION SET "_optimizer_group_by_placement" = FALSE;
```

Andmebaasi struktuur on vastavalt kavandatud ja kitsendused jõustatud, kuid kahjuks ei oska optimeerija seda infot täielikult ära kasutada ja teeb sama operatsiooni kaks korda, mis kokkuvõttes muudab lause täitmise aeglasemaks. Kõrvalepõikena on see näide selle kohta, kuidas kitsenduste abil antakse andmebaasisüsteemile teada olulist informatsiooni, mida see saaks päringute optimeerimiseks (ja kokkuvõttes efektiivsemaks täitmiseks) tarvitada.

Võrdluseks sunnin optimeerijat koostama täitmisplaani, kus grupeerimisoperatsioon tehakse viimasena ehk esialgu tehakse alampäringute (*inline view*) tulemuste ühendamine. Selle saavutamiseks kasutan *MERGE* vihjet. Oracle Database andmebaasisüsteemis võib päringu

kirjutaja anda optimeerijale vihje, kuidas võiks paremini päringut täita (tegelikult sundida optimeerijat mingeid valikuid tegema).

```
SELECT /*+MERGE(hp)*/ hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.last_visit_date, hp.visit_count
FROM (
    SELECT hcv.patient_id, Sum(hcv.visit_fee) AS visit_fee_sum,
        Count(*) AS visit_count, Max(hcv.from_date) AS last_visit_date
    FROM Health_care_visit hcv
    GROUP BY hcv.patient_id ) hp
INNER JOIN Person pe ON hp.patient_id = pe.party_id;
```

Sellisel juhul on täitmisplaan järgmine ning selle alusel toimub lause täitmine 3,52 sekundi võrra aeglasemalt (võrreldes päringuga baastabelite põhjal) (vt Joonis 14). Ühtlasi võib seda ka võtta kui hoiatavat näidet selle kohta, kuidas inimkasutaja vahelesegamine andmebaasisüsteemi töösse vihjete kasutamise kaudu võib päringute täitmise kiirust vähendada.

```
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation           | Name               | Starts | Cost (%CPU)| A-Rows |   A-Time   | Buffers | Used-Mem |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                    |      1 | 18974 (100)|  70000 |00:00:05.37 |   12912 |          |
|   1 |  HASH GROUP BY      |                    |      1 | 18974   (1)|  70000 |00:00:05.37 |   12912 |   26M (0)|
|*  2 |   HASH JOIN         |                    |      1 |  5134   (1)|   1000K|00:00:03.46 |   12912 |   13M (0)|
|   3 |    TABLE ACCESS FULL| PERSON             |      1 |   239   (1)|  70000 |00:00:00.07 |     822 |          |
|   4 |    TABLE ACCESS FULL| HEALTH_CARE_VISIT  |      1 |  3298   (1)|   1000K|00:00:00.84 |   12090 |          |
-----------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("HCV"."PATIENT_ID"="PE"."PARTY_ID")
```

Joonis 14. *GROUP JOIN* päringu täitmisplaan vihje kasutamise korral

Erinevate täitmisplaanide alusel lause täitmist iseloomustab järgnev koondtabel (vt Tabel 7).

Tabel 7. Täitmisplaanide erinevuste koondtabel eksperimendi *GROUP JOIN* korral

	Päring vaate põhjal	**Päring baastabelite põhjal**	**Vihje kasutamine**
Täitmise kiirus (A-Time) sekundites	1,72	1,85	5,37
Esimesena tehakse	Grupeerimine	Grupeerimine	Tabelite ühendamine
Viimasena tehakse	Tabelite ühendamine	Tabelite ühendamine	Grupeerimine

Põhjuseks, miks esimene ja teine päring kiiremini täideti oli see, et tänu varasemale grupeerimisele vähenes ridade arv, mida andmebaasisüsteem pidi kokku ühendama. Esimese ja

teise päringu täitmisel ühendati 70 000 rida 70 000 reaga, kolmanda päringu täitmisel 1 000 000 rida 70 000 reaga. See vastab üldisele päringute optimeerimise strateegiale, mille kohaselt tuleks kõigepealt viia läbi ridade hulka piiravad operatsioonid, et vähendada ühendamist vajavate ridade hulka.

Teine erinevus vaadete ning baastabelite põhjal tehtud päringute täitmisplaanide vahel seisneb selles, et vaadetele tehtud päringute korral kasutatakse täitmisplaanides *VIEW* operatsiooni.

Optimeerija kirjutab lahti vaate põhjal tehtud päringu, asendades viite vaatele vaate alampäringuga ja teostab vaate mestimist (*view merging*). Kui optimeerija ei suuda vaadet ja selle põhjal tehtud päringut mestida, siis optimeerija töötleb vaate alampäringut eraldi ning sellisel juhul ilmub täitmisplaani *VIEW* operatsioon. Järgmises täitmisplaanis töödeldakse vaate alampäringut eraldi (vt Joonis 15).

```
CREATE VIEW vw_Number_of_patients_who_man ( quantity )
AS
SELECT Count(personal_code) AS quantity
FROM Person
WHERE personal_code LIKE '3%'
    OR personal_code LIKE '5%';

SELECT * FROM vw_Number_of_patients_who_man;
```

```
-----------------------------------------------------------------------------------------------------------------------------
| Id  | Operation             | Name                        | Starts | Cost (%CPU)| A-Rows |   A-Time   | Buffers | Reads  |
-----------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                             |      1 |    61 (100)|      1 |00:00:00.06 |     223 |     30 |
|   1 |  VIEW                 | VW_NUMBER_OF_PATIENTS_WHO_MAN |      1 |    61   (2)|      1 |00:00:00.06 |     223 |     30 |
|   2 |   SORT AGGREGATE      |                             |      1 |            |      1 |00:00:00.06 |     223 |     30 |
|*  3 |    INDEX FAST FULL SCAN| AK_PERSON_PERSONAL_CODE     |      1 |    61   (2)|  26228 |00:00:00.05 |     223 |     30 |
-----------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(("PERSONAL_CODE" LIKE '5%' OR "PERSONAL_CODE" LIKE '3%'))
```

Joonis 15. Vaate põhjal tehtud päringu täitmisplaan *COUNT*

Antud täitmisplaan võimaldab saada päringule vastuse ainult indeksi plokke lugedes. Esialgu loetakse tabeli *Person* indeksit *AK_Person_Personal_code*, mis on loodud veerule *personal_code* (isikukood). *AK_Person_Personal_code* on unikaalsuse kitsendus ja unikaalsuse kitsenduse poolt hõlmatud veergudele luuakse paljudes andmebaasisüsteemides (sh Oracle ja PostgreSQL) automaatselt indeks.

SORT AGGREGATE operatsioon täitmisplaanis viitab kokkuvõttefunktsiooni (*Count(*)*) alusel ridade hulga põhjal ühe väärtuse leidmisele.

Identse, baastabelite põhjal tehtud päringu täitmisplaan on sarnane, kuid ei sisalda *VIEW* operatsiooni (vt Joonis 16). Siiski, sellel korral vaate kasutamine/mittekasutamine ei mõjuta päringu täitmise kiirust.

```
SELECT Count(personal_code) AS quantity
FROM Person
WHERE personal_code LIKE '3%'
    OR personal_code LIKE '5%';
```

```
-------------------------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                   | Starts | Cost (%CPU)| A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                        |      1 |    61 (100)|      1 |00:00:00.05 |     223 |
|   1 |  SORT AGGREGATE      |                        |      1 |            |      1 |00:00:00.05 |     223 |
|*  2 |   INDEX FAST FULL SCAN| AK_PERSON_PERSONAL_CODE |      1 |    61   (2)|  26228 |00:00:00.04 |     223 |
-------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(("PERSONAL_CODE" LIKE '5%' OR "PERSONAL_CODE" LIKE '3%'))
```

Joonis 16. Baastabeli põhjal tehtud päringu täitmisplaan *COUNT*

Järgnevalt nimetatakse eksperimendid, mille korral Oracle ei suutnud vaate alampäringut ja vaate põhjal tehtud päringut mestida (*view merging*) ning täitis vaate alampäringu eraldi (sellest annab teada *VIEW* operatsioon täitmisplaanis).

- *UNION*
- *UNION ALL*
- *INTERSECT*
- *RECURSIVE*
- *ROWNUMBER*
- *COUNT*
- *LAG*
- *LAG FILTER*
- *DISTINCT*
- *DISTINCT 2*
- *VIEWofVIEW (DISTINCT)*
- *ORDER BY FILTER*
- *FETCH*

Nimekirjas on enamuses sellised eksperimendid, kus vaadete alampäringud sisaldavad *DISTINCT* klauslit, ühendi või lõike leidmist, sorteerimist, kokkuvõtte- või aknafunktsiooni,

või rekursiivset päringut. See tulemus on üldiselt kooskõlas [38] esitatud infoga selle kohta, milliste operatsioonide korral ei suuda Oracle vaadet mestida. *ORDER BY FILTER* ja *FETCH* kohta [38] infot ei andnud. Kuid nende päringute korral kasutatakse sorteerimist. Tundub, et just see on põhjus, miks vaate mestimist ei toimu.

Kui selliste vaadete põhjal tehakse päring, siis on võimalikud järgmised päringu täitmise stsenaariumid.

- Kui (põhi-) päring vaatele sisaldab *WHERE* klauslit, siis optimeerija kasutab seda filtrit vaate alampäringus (*predicate pushdown*). Seda tehakse AINULT siis kui filtri kasutamisest sõltub ridade arv, mida tuleb vaate alampäringus töödelda [40].
- Kui päringus tehakse vaate ja baastabeli ühendamist, siis on võimalik et:
    - ühendamistingimus surutakse vaatesse nii, et vaate alampäringu täitmine toimub korreleeruva alampäringuna, mis täidetakse üks kord iga põhipäringu rea kohta. Sellist optimeerimise meetodit nimetatakse *join predicate pushdown* [46] (ühendamistingimuse vaatesse surumine)
    - vaate alampäring täidetakse eraldi ja siis ühendatakse baastabeliga.

Kui vaate (mille puhul ei toimu vaate ja selle põhjal tehtud päringu mestimist) põhjal tehtud päringus kasutatakse *WHERE* klauslit, siis optimeerija võiks seda rakendada vaate alampäringus.

Tabel 8 esitatakse kokkuvõte eksperimendist, mis proovis kindlaks teha, kas andmebaasisüsteem oskab päringu tingimust vaate alampäringusse suruda (*predicate pushdown*) või mitte. Eksperimendi läbiviimiseks lisati vaate põhjal tehtud päringusse WHERE klausel (näiteks, *„WHERE health_care_visit_id = 1“*). Võrdluseks on toodud ka PostgreSQL andmebaasisüsteemis tehtud katsetused. + näitab, et surumine on toimunud, -, et ei toimu. Nagu tabelist näha, siis Oracle Database andmebaasisüsteem on selles osas võimekam.

Tabel 8. Filtri *(WHERE klausel)* kasutamine vaate alampäringu sees

Päringu nimetus	Kas vaate põhjal tehtud päringu tingimus *(WHERE klausel)* on surutud vaate alampäringusse?			
	Oracle Database 12c	Oracle Database 11g	PostgreSQL 9.3	PostgreSQL 9.1
IN	+	+	+	+
LEFT OUTER	+	+	+	+
UNION	+	+	-	-
UNION ALL	+	+	-	-
INTERSECT	+	+	+	+
GROUP BY	+	+	+	+
NONCORR IN	+	+	+	+
NONCORR NOTIN	+	+	-	-
CORR	+	+	-	-
VIEWofVIEW (DISTINCT)	+	+	+	+
VIEWofVIEW 2	+	+	+	+
ORDER BY FILTER	+	+	+	+
LAG FILTER	+	+	-	-
GROUP JOIN FILTER	+	+	+	+

Leidub erinevaid põhjuseid, miks optimeerija võib loobuda päringu ja vaate mestimisest või predikaatide vaatesse surumisest. On võimalik, et tulemuseks olev täitmisplaan on sellisel juhul väiksema maksumusega või ei luba andmebaasisüsteemi realisatsiooni piirangud mestimist läbi viia [21].

Järgnevalt esitatakse *VIEWofVIEW* eksperimendi koondtulemused (vt Tabel 9).

```
CREATE VIEW vw_Employee_Employer (
    employee, personal_code, employer, registration_code )
AS
SELECT DISTINCT ( v_pe.first_name || ' ' || v_pe.last_name ) AS employee,
    v_pe.personal_code, org_vr.name AS employer, org_vr.registration_code
FROM vw_Patients_visits v_pe LEFT OUTER JOIN  vw_Employers org_vr
    ON org_vr.health_care_visit_id = v_pe.health_care_visit_id;
```

Tabel 9. Täitmisplaanide erinevuste koondtabel eksperimendi *VIEWofVIEW* korral

	Oracle Database 12c		**Oracle Database 11g**	
	Päring vaate põhjal	**Päring baastabelite põhjal (ilma alampäringuteta)**	**Päring vaate põhjal**	**Päring baastabelite põhjal (ilma alampäringuteta)**
Täitmise kiirus (A-Time) sekundites	6,61	7,71	9,22	9,65
1. ühendamise tulemuse ridade arv (A-Rows)	100 000	1 004 000	100 000	1 004 000
2. ühendamise tulemuse ridade arv (A-Rows)	1 000 000	1 004 000	1 004 000	1 004 000
3. ühendamise tulemuse ridade arv (A-Rows)	1 004 000	1 004 000	1 004 000	1 004 000
Tabelite ühendamise algoritm	*Hash join*		*Merge join*	
Kuidas leitakse tabelist Health_care_visit vajalikke andmeid sisaldavad plokid	Tabeli täielik läbiskaneerimine (*TABLE FULL ACCESS*)		Indeksipuu lehtede läbiskaneerimine (*INDEX FULL SCAN*) ja sealt leitud info alusel tabeli lugemine (TABLE ACCESS BY INDEX ROWID)	

Vaate alampäringus ühendatakse kaks vaadet. Iga viidatud vaate alampäringus ühendatakse kaks baastabelit. Esimeses baastabelite põhjal tehtavas päringus asendatakse viited vaadetele vaadete alampäringutega (*inline view*). See tähendab, et baastabelite põhjal tehtud päringus tehakse kahe alampäringu tulemuse ühendamine. Kummaski alampäringus ühendatakse omakorda kaks tabelit. Selle päringu täitmisplaan ja otse vaate põhjal tehtud päringu täitmisplaan on sarnased ning sellel põhjusel pole neid antud jaotises välja toodud. Erinevuseks on ainult *VIEW* operatsiooni kasutamine vaate põhjal tehtud päringus.

Alternatiivne baastabelite põhjal tehtud päring on selline kus tehakse nelja baastabeli ühendamine ilma alampäringute kasutamiseta. Sellele päringule viidatakse tabelis (vt Tabel 9). Selle päringu täitmisplaan erineb teistest. Optimeerija otsustas teha esialgu kahe kõige suurema baastabeli ühendamise ja tulemuseks saadud tabeli ühendada iga järgmise baastabeliga. Seega iga ühendamise tulemusena saadakse 1 004 000 rida ja seetõttu suurenes päringu täitmiseks

kulunud aeg 1,1 sekundi võrra. Jällegi näeme, et mida rohkem ridu peab andmebaasisüsteem kokku ühendama, seda aeglasemaks muutub päringu täitmine. Paraku ei osanud optimeerija antud juhul kõige paremat ühendamise järjekorda valida. Ühendamise järjekorda on Oracles võimalik mõjutada kasutades päringus *„ORDERED“* vihjet [47]. Sellisel juhul ühendatakse tabelid selles järjekorras nagu need on päringu *FROM* klauslis esitatud. Sellisel viisil on võimalik saavutada samasugune täitmisplaan nagu praegu tekkis vaadete kasutamise korral.

Kahe andmebaasisüsteemi versiooni vaheliseks erinevuseks on:

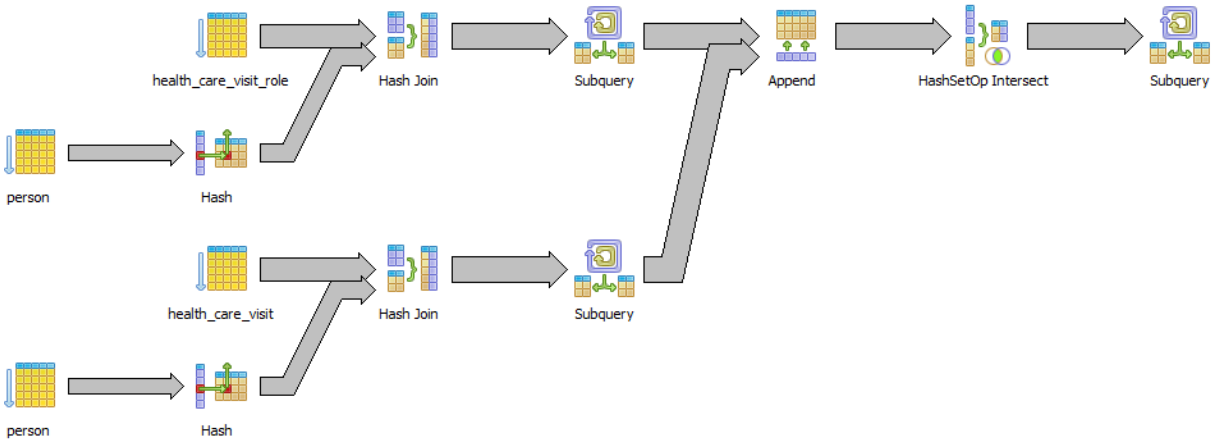
- Erinevate tabelite ühendamise algoritmite kasutamine. Oracle Database 12c andmebaasisüsteem kasutab *hash join* algoritmi, Oracle Database 11g – *merge join* algoritmi (selline tabelite ühendamise algoritm kasutab sorteerimist);
- Oracle Database 12c andmebaasisüsteemi puhul loetakse kõiki tabeli *Health_care_visit* plokke (kuni kõrgveemärgini) ja vaadatakse läbi kõik tabeli read (*TABLE ACCESS FULL*). Oracle Database 11g andmebaasisüsteemi korral toimub kõigi indeksi plokkide lugemine.

Lisaks testisin Oracle Database 12c andmebaasisüsteemis ka päringuid vaadete põhjal, millele on deklareeritud primaarvõtme, välisvõtme või unikaalsuse kitsendus. Kuna päringute täitmisplaanid jäid samaks, siis ei tooda neid võrdlemiseks välja. Saab väita, et täitmisplaani genereerimisel ei suuda Oracle andmebaasisüsteem nende kitsenduste poolt antavat infot nende vaadete põhjal tehtavate päringute täitmisplaanide koostamisel ära kasutada.

4.3 PostgreSQL andmebaasisüsteemi täitmisplaanide erinevusi

Järgnevalt võrreldakse mõningate PostgreSQL andmebaasisüsteemi päringute (füüsilisi) täitmisplaane. Täitmisplaani iga operatsiooni juures esitatakse hinnanguline ja tegelik ridade arv (*rows*), hinnanguline rea laius (*width*), hinnang operatsiooni maksumusele (*cost*), operatsiooni käivitamise arv (*loops*), päringu tegelik täitmise kiirus (*actual time*). Puu juur esitab hinnangu lausele kui tervikule.

```
CREATE VIEW vw_Equal_names (
    first_name, last_name )
AS
SELECT d.first_name, d.last_name
FROM Person d
INNER JOIN Health_care_visit_role vr ON d.party_id = vr.party_id
INTERSECT
SELECT p.first_name, p.last_name
FROM Person p
INNER JOIN Health_care_visit hcv ON p.party_id = hcv.patient_id;
```



Joonis 17. Vaate *INTERSECT* põhjal tehtud päringu täitmisplaani graafiline esitus

Joonisel (vt Joonis 17) esitatakse PostgreSQL 9.3 andmebaasisüsteemis tehtud päringu täitmisplaan (*B2 eksperiment nimega „INTERSECT“*). *INTERSECT* operaatori abil leitakse hulgateoreetiline vahe. Vaate alampäringus olevad alampäringud täidetakse eraldi. Mõlema alampäringu täitmisel kasutatakse tabelite ühendamiseks räsiväärtuste leidmisel põhinevat *hash join* algoritmi. Sisemiseks tabeliks on *Person*, mille põhjal luuakse räsitabel (*hash table*), kuna selles tabelis on vähem ridu. Väliseks tabeliks on *Health_care_visit*, ja teise alampäringu puhul – *Health_care_visit_Role*. Loetakse kõiki tabeli *Person* plokke (kuni kõrgveemärgini) ja vaadatakse läbi kõik tabeli read (*Seq Scan*). Samamoodi ka teiste tabelite puhul.

Subquery Scan ja *Append* operaatorite kasutamine on *INTERSECT* ja *UNION* operaatorite korral tavapärane [48].

Vaate ja baastabelite põhjal tehtud päringud erinevad viimase *Subquery Scan* operaatori kasutamise poolest. Siiski, sellel korral vaade ei mõjuta päringu täitmise kiirust (vt Tabel 10). *Subquery Scan* operaator töötleb alampäringut eraldi ning on sarnane Oracle Database andmebaasisüsteemi täitmisplaanides kasutatavale *VIEW* operatsioonile.

Tabel 10. *INTERSECT* päringute täitmise kiirus

	PostgreSQL 9.3		PostgreSQL 9.1	
	Vaate põhjal tehtud päring	**Baastabelite põhjal tehtud päring**	**Vaate põhjal tehtud päring**	**Baastabelite põhjal tehtud päring**
Täitmise kiirus (actual time) sekundites	8,53	7,89	5,51	5,64

Järgnevalt (vt Tabel 11) esitatakse koondtabel *GROUP JOIN* eksperimendi päringute täitmisplaanide erinevustest (*B3.2 eksperiment nimega „GROUP JOIN“*) PostgreSQL 9.3 andmebaasisüsteemis. Võrdlemiseks on võetud päring vaatele, ning kuna baastabelitele tehtud päringu täitmisplaan on sama, siis on võrdluseks võetud teine ehk alternatiivne päring baastabelitele, kus alampäringuid ei kasutatud.

Tabel 11. *GROUP JOIN* päringute täitmisplaanide erinevused

	Päring vaatele	**Päring baastabelitele (ilma alampäringuta)**
Täitmise kiirus (actual time) sekundites	3,40	11,11
Tagastatav ridade arv (rows)	70 000	70 000
Esimesena tehakse	Grupeerimine	Tabelite ühendamine
Viimasena tehakse	Tabelite ühendamine	Grupeerimine
Grupeeritavate ridade arv	1 000 000	1 000 000
Ühendatavate ridade arv		
1. ***baastabel***	70 000	70 000
2. ***baastabel***	70 000 (grupeeritud)	1 000 000
Ühendamise tulemus (ridade arv)	70 000	1 000 000 (grupeerimiseks)

Vaate põhjal tehtud päring täitmine võtab 7,71 sekundit vähem aega. Mõlemal juhul on grupeerimise operatsiooni sisendiks 1 000 000 rida ja tulemuseks 70 000 rida. Ühendamise tulemuseks olev ridade arv on esimesel juhul 70 000 ning teisel juhul 1 000 000. Töökiiruse erinevused tulenevad sellest, et baastabelite põhjal tehtud päringu korral ühendab süsteem palju suuremat hulka ridu.

GROUP JOIN FILTER eksperimendis on täitmisplaanide tulemused vastupidised (*B3.3 eksperiment nimega „GROUP JOIN FILTER“*). Päringus vaatele teeb andmebaasisüsteem esialgu grupeerimist (mida nõuab vaate *vw_Patient_visitinfo* alampäring) ja seejärel tabelite ühendamist. Efektiivsem oleks kui ühendamist rakendataks enne grupeerimist, et vähendada grupeeritavate ridade hulka.

```
CREATE VIEW vw_Patient_suminfo_max_bdt (
    patient_id, first_name, last_name, birth_date,
    visit_fee_sum, visit_count, last_visit_date )
AS
SELECT hp.patient_id, pe.first_name, pe.last_name, pe.birth_date,
    hp.visit_fee_sum, hp.visit_count, hp.last_visit_date
FROM vw_Patient_visitinfo hp INNER JOIN Person pe ON hp.patient_id = pe.party_id
WHERE ( EXTRACT ( YEAR FROM pe.birth_date ) ) = (
    SELECT Max( EXTRACT ( YEAR FROM birth_date ) )
    FROM Person );
```

GROUP JOIN FILTER eksperimendi tulemused esitatakse järgmises koondtabelis (vt Tabel 12).

Tabel 12. *GROUP JOIN FILTER* päringute täitmisplaanide erinevused

	Päring vaatele	**Päring baastabelitele (ilma alampäringuta)**
Kogumaksumus (cost)	171 072	28 205
Täitmise kiirus (actual time) sekundites	3,40	1,12
Tagastatav ridade arv (rows)	649	649
Esimesena tehakse	Grupeerimine	Tabelite ühendamine
Viimasena tehakse	Tabelite ühendamine	Grupeerimine
Grupeeritavate ridade arv	1 000 000	9 152
Ühendatavate ridade arv		
1. *baastabel*	649	649
2. *baastabel*	70 000 (grupeeritud)	1 000 000
Ühendamise tulemus (ridade arv)	649	9 152 (grupeerimiseks)

Töökiiruse erinevuste põhjus peitub selles, et baastabelitele tehtud päringus rakendatakse päringu täitmise varases faasi tabelite ühendamist ja sellega seoses vähendatakse (võrreldes vaate põhjal tehtud päringu täitmisplaaniga) oluliselt ridade hulka, mida on vaja kokku grupeerida.

Järgnevalt esitatakse koondtabel eksperimendi *LAG FILTER* päringute täitmisplaanide erinevustest (*D3.1 eksperiment nimega „LAG FILTER“*). Vaate põhjal ja baastabelite põhjal (kus viide vaatele on asendatud vaate alampäringuga) tehtud päringute täitmisplaanid on identsed. Erineb alternatiivse päringu täitmisplaan, kus pole kasutatud alampäringuid ega viitamist vaatele. Vaate alampäringus on viide teisele vaatele, kus kasutatakse aknafunktsiooni, ning lisatud *WHERE* klausel, mis filtreerib välja ainult ühe patsiendi. Päringus baastabelitele (ilma alampäringuta) on aknafunktsioon ja *WHERE* klausel ühes päringus.

Tabel 13. *LAG FILTER* päringute täitmisplaanide erinevused

	Päring vaatele	**Päring baastabelile (ilma alampäringuta)**
Täitmise kiirus (actual time) sekundites	4,90	0,0005
Aknafunktsiooni sisendiks olev ridade arv (rows)	1 000 000	13
Tagastatav ridade arv (rows)	13	13
Kuidas leitakse vajalikke andmeid sisaldavad plokid tabelist health_care_visit	Tabeli *health_care_visit* täielik läbiskaneerimine (*Seq Scan*)	Tabelile *health_care_visit* loodud indeksi kasutamine (*Bitmap Index Scan*)

Otse baastabelite põhjal tehtud päringu täitmine võtab 4,8995 sekundit vähem aega kui vaate põhjal täidetud päringu täitmine (vt Tabel 13). Põhjus on selles, et vaate põhjal tehtud päringu täitmiseks loetakse kõiki tabeli *Health_care_visit* plokke (kuni kõrgveemärgini) ja vaadatakse läbi kõik tabeli read (*Seq Scan*). Seejärel saadud tulemust sorteeritakse ja töödeldakse aknafunktsiooni abil (*WindowAgg*). Ning alles siis rakendatakse filtrit (*WHERE* klausel) (vt Joonis 18).

```
CREATE VIEW vw_Visit_fee_prev_filter (
    health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
prev_visit_fee )
AS
SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
prev_visit_fee
FROM vw_Visit_fee_previous
WHERE patient_id = 67329;
```

```
Subquery Scan on vw_visit_fee_previous  (cost=162856.84..192856.84 rows=15 width=54) (actual time=3871.678..5345.705 rows=13 loops=1)
  Filter: (vw_visit_fee_previous.patient_id = 67329)
  Rows Removed by Filter: 999987
  -> WindowAgg  (cost=162856.84..180356.84 rows=1000000 width=22) (actual time=2809.259..4892.421 rows=1000000 loops=1)
     -> Sort  (cost=162856.84..165356.84 rows=1000000 width=22) (actual time=2809.234..3500.183 rows=1000000 loops=1)
        Sort Key: health_care_visit.patient_id
        Sort Method: external merge  Disk: 31440kB
        -> Seq Scan on health_care_visit  (cost=0.00..22179.00 rows=1000000 width=22) (actual time=0.092..857.669 rows=1000000 loops=1)
```

Joonis 18. Vaate põhjal tehtud päringu täitmisplaan *LAG FILTER*

Kui andmebaasisüsteem suudaks kohe filtrit rakendada (ehk lause täitmise varases faasis töödeldavate ridade hulka oluliselt piirata), siis päringu täitmise kiirus oleks palju suurem. Sellisel juhul oleks päringu täitmisplaan optimaalsem nagu näitab alternatiivse päringu täitmisplaan (vt Joonis 19).

```
SELECT health_care_visit_id, patient_id, from_date, thru_date, visit_fee,
  COALESCE (
  LAG(visit_fee) OVER (
    PARTITION BY patient_id ORDER BY from_date), 0.00) AS prev_visit_fee
FROM Health_care_visit
WHERE patient_id = 67329;
```

```
WindowAgg  (cost=4.54..63.34 rows=15 width=22) (actual time=0.143..0.401 rows=13 loops=1)
  -> Bitmap Heap Scan on health_care_visit  (cost=4.54..63.15 rows=15 width=22) (actual time=0.129..0.348 rows=13 loops=1)
     Recheck Cond: (patient_id = 67329)
     -> Bitmap Index Scan on uq_visit_facil_addr_pat_fromd  (cost=0.00..4.54 rows=15 width=0) (actual time=0.112..0.112 rows=13 loops=1)
        Index Cond: (patient_id = 67329)
```

Joonis 19. Baastabeli põhjal tehtud päringu täitmisplaan *LAG FILTER*

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). Antud näites täidetakse vaate põhjal tehtud päringud 2.8 sekundit ning otse baastabeli põhjal tehtud päringut 0.0001 sekundit.

Leidub lahendus eelnevalt kirjeldatud aknafunktsioonide vaates kasutamise probleemile:

- installeerida süsteemis parandus (*patch*) [17], mille tulemusena rakendab andmebaasisüsteem filtrit (*WHERE tingimus*) varakult (*predicate pushdown*) juhul, kui filtris kasutatav veerg on sama kui *PARTITION BY* klauslis olev veerg;
- ära oodata andmebaasisüsteemi versioon PostgreSQL 9.5, kus see parandus on loodetavasti vaikimisi installeeritud [17].

4.4 Päringute täitmise kiirus

Selles jaotises esitatakse koondtabelid päringute täitmise kiiruse kohta. Kui lahtris on *„0,00“*, siis see tähendab, et täitmise kiirus oli väga väike (alla 0,01 sekundi). Sinise värviga tähistatud väljad viitavad, et antud eksperimendi korral päringute täitmise kiirus erineb rohkem kui kahe sekundi võrra. **Rasvase fondiga** on iga andmebaasisüsteemi versiooni ja eksperimendi juures esile tõstetud kõige parem (antud juhul kõige väiksem tulemus). Üldiselt võib öelda, et Oracle andmebaasisüsteemis on erinevused päringute täitmise kiiruses (vt Tabel 14) väikesed. Enamasti on otse baastabelite peal tehtud päring veidi kiirem kui vaate põhjal tehtud päring või sellega võrdne. Huvitav on märkida, et nt Oracle 12c puhul on ka neli eksperimenti, mille korral oli vaate põhjal tehtud päringu täitmine isegi veidi kiirem kui baastabelite põhjal tehtud päringu täitmine. See on üllatav, sest „tavalisi“ vaateid (erinevalt hetktõmmistest e materialiseeritud vaadetest) ei seostata andmetöötluse operatsioonide kiiruse paranemisega. Siiski, nimetatud neljast eksperimendist vaid ühes (*DISTINCT 2*) loodi vaate põhjal tehtud päringule ja otse baastabelite põhjal tehtud päringule erinev täitmisplaan. Seega pidid töökiiruse erinevusi tingima mingid muud tehnilised põhjused.

Tabel 14. Päringute täitmise kiirus Oracle Database andmebaasisüsteemi erinevates versioonides (sekundites)

	Oracle Database 12c			Oracle Database 11g		
	VW	**BT**	**BT 2**	**VW**	**BT**	**BT 2**
IN	**0,01**	**0,01**		**0,01**	**0,01**	
LEFT OUTER	0,02	**0,01**		**0,01**	**0,01**	
UNION	0,76	**0,67**		0,57	**0,22**	
UNION ALL	0,51	**0,38**		**0,11**	**0,11**	
INTERSECT	8,35	**4,99**		3,98	**3,96**	
COUNT	0,06	**0,05**		0,40	**0,03**	
GROUP BY	1,60	**1,45**		3,78	**3,43**	
GROUP JOIN	1,72	**1,64**	1,85	3,60	**3,54**	3,65
GROUP JOIN FILTER	2,11	**1,89**	1,94	3,72	**3,56**	3,58
DISTINCT	2,42	**1,72**		0,63	**0,61**	
DISTINCT 2	**1,89**	2,07		**0,65**	0,67	
ORDER BY	5,03	**4,98**		2,69	**1,69**	
ORDER BY FILTER	**0,01**	**0,01**	**0,01**	**0,01**	**0,01**	**0,01**
NONCORR IN	**2,09**	**2,09**		4,50	**4,36**	
NONCORR NOTIN	**3,24**	3,53		0,86	**0,83**	
CORR	**0,36**	0,40		0,13	**0,12**	
ROWNUMBER	0,46	**0,36**		2,97	**0,21**	
FETCH	**0,08**	**0,08**				
ROWNUM	0,12	**0,11**		**0,01**	**0,01**	
ROWNUMBER 999	0,14	**0,13**		**0,01**	**0,01**	
LAG	8,62	**6,53**		6,99	**6,20**	
LAG FILTER	**0,01**	**0,01**	**0,01**	**0,01**	**0,01**	**0,01**
VIEWofVIEW	6,61	**6,30**	7,71	9,22	**9,19**	9,65
VIEWofVIEW 2	**2,18**	2,19	2,20	1,28	**0,76**	0,97
RECURSIVE	0,25	**0,21**		0,22	**0,17**	

Kui võrrelda päringute täitmist Oracle 12.1 ja 11.1 korral, siis vaadete ja otse baastabelite põhjal tehtud päringute täitmise kiiruste suhe on mõlemas süsteemis umbes samasugune. Huvitav on tähele panna, et on palju päringuid, mis töötasid hilisema andmebaasisüsteemi versiooniga ning parema riistvaraga serveris aeglasemalt kui vanema versiooniga ja nõrgema riistvaraga serveris. Kui võrrelda otse baastabelite põhjal tehtud päringuid Oracle 12.1 ja Oracle 11.1 süsteemides, siis oli 15 korral (25-st) Oracle 11.1 tehtud päring kiirem. Selle põhjus võib peituda nii andmebaasisüsteemis kui riistvaras, kuid praegu seda erinevust seletada ei oska.

Järgnevalt esitatakse koondtabel suurematest päringute täitmise kiiruse erinevustest Oracle Database andmebaasisüsteemi versioonides (vt Tabel 15).

Tabel 15. Päringute täitmise kiiruse suuremad erinevused Oracle Database andmebaasisüsteemi erinevates versioonides (sekundites)

	Oracle Database 12c (Release 1)		
	Päring vaatele	**Päring baastabelitele**	**Vahe**
INTERSECT	8,35	**4,99**	3,36
ROWNUMBER	0,46	**0,36**	0,10
LAG	8,62	**6,53**	2,09
	Oracle Database 11g (Release 1)		
INTERSECT	3,98	**3,96**	0,02
ROWNUMBER	2,97	**0,21**	2,76
LAG	6,99	**6,20**	0,79

Kõige suuremaks erinevuseks päringu täitmise kiiruses on 3,36 sekundit eksperimendis *INTERSECT* Oracle Database andmebaasisüsteemis versiooniga 12c (Release 1). *INTERSECT* eksperimendi päringute täitmisplaanid on väga sarnased. Erinevuseks on see, et vaate põhjal tehtud päringu täitmisplaanis kasutatakse *VIEW* operatsiooni (st optimeerija ei suuda vaadet ja selle põhjal tehtud päringut mestida ning töötleb vaate alampäringut eraldi). Kõikide selles tabelis välja toodud erinevuste korral on tegemist olukorraga, kus baastabelite põhjal tehtud päring on kiirem kui vaate põhjal tehtud päring.

Järgnevalt esitatakse töökiiruste võrdlustabel PostgreSQLis. Rohelise värviga tähistatud väljad viitavad, et antud eksperimendi korral põhjustas PostgreSQL 9.3 vaatesse turvabarjääri lisamine selle vaate põhjal tehtava päringu töökiiruse tuntava langemise (vt veerg *SVW*).

Tabel 16. Päringute täitmise kiirus PostgreSQL andmebaasisüsteemi erinevates versioonides (sekundites)

	PostgreSQL 9.3				PostgreSQL 9.1		
	SVW	**VW**	**BT**	**BT 2**	**VW**	**BT**	**BT 2**
IN	**0,00**	**0,00**	**0,00**		**0,00**	**0,00**	
LEFT OUTER	**0,00**	0,02	**0,00**		**0,01**	**0,01**	
UNION	**1,29**	1,66	1,71		1,19	**1,14**	
UNION ALL	0,25	0,26	**0,24**		0,12	**0,11**	
INTERSECT	7,94	8,53	**7,89**		**5,51**	5,64	
COUNT	**0,06**	**0,06**	**0,06**		**0,04**	**0,04**	
GROUP BY	4,05	**3,45**	3,80		**3,11**	3,13	
GROUP JOIN	3,85	**3,43**	3,84	11,14	**3,33**	**3,33**	11,89
GROUP JOIN FILTER	3,43	3,42	3,76	**1,12**	3,38	3,25	**0,54**
DISTINCT	**1,89**	2,16	2,77		**2,10**	2,45	
DISTINCT 2	2,48	2,45	**1,08**		5,53	**5,38**	
ORDER BY	**4,25**	4,78	5,04		3,41	**3,40**	
ORDER BY FILTER	5,64	0,01	**0,00**	**0,00**	0,01	0,01	**0,00**
NONCORR IN	**1,08**	1,19	1,42		0,73	**0,56**	
NONCORR NOTIN	-*	-*	**661,03**		371,13	**351,02**	
CORR	1739,87	**1724,55**	1755,34		**1945,73**	2022,90	
ROWNUMBER	**0,19**	0,26	0,27		0,43	**0,14**	
FETCH	**0,00**	0,01	**0,00**		**0,00**	**0,00**	
ROWNUMBER 999	**0,26**	0,47	0,41		**0,16**	**0,16**	
LAG	4,59	4,20	**4,07**		**4,12**	4,31	
LAG FILTER	4,39	4,90	4,50	**0,00**	3,95	3,92	**0,00**
VIEWofVIEW	30,61	27,45	**25,50**	26,89	**25,73**	26,06	26,75
VIEWofVIEW 2	4,81	**1,20**	1,34	1,68	1,03	0,77	**0,74**
RECURSIVE	0,46	0,29	**0,28**		0,14	**0,12**	

** päringu täitmine jäi lõpetamata*

Ka PostgreSQLis leidus päringuid, mis olid kiiremad vaadete põhjal või kiiremad baastabelite põhjal, kuid need kiiruse erinevused on väga väikesed. Kuid samas leidus mitmeid päringuid, mille täitmisel töökiirus oli võrreldes Oraclega üldiselt väga halb ning vaadete põhjal tehtud päringu korral eriti halb. See näitab PostgreSQL optimeerimismooduli teatavat mahajäämust Oracle vastavast moodulist. Samuti vähendas vaatesse turvabarjääri lisamine kolmel juhul märgatavalt vaate põhjal tehtava päringu töökiirust. Oli ka juhus (GROUP JOIN), kus vaate kasutamine aitas andmebaasisüsteemil paremat täitmisplaani valida.

Nii nagu Oracle korral on ka PostgreSQLis korral täheldatav, et mitmete päringute korral toimub täitmine hilisema andmebaasisüsteemi versiooniga ning parema riistvaraga serveris

aeglasemalt kui vanema versiooniga ja nõrgema riistvaraga serveris. See tugevdab arvamust, et probleem peab olema uuema serveri (apex.ttu.ee) riistvaras või operatsioonisüsteemis.

Järgnevalt esitatakse koondtabel suurematest päringute täitmise kiiruse erinevustest PostgreSQL andmebaasisüsteemi versioonides (vt Tabel 17).

Tabel 17. Päringute täitmise kiiruse suuremad erinevused PostgreSQLi andmebaasisüsteemi erinevates versioonides (sekundites)

	PostgreSQL 9.3			
	Päring vaatele	**Päring baastabelitele**	**Päring baastabelitele (ilma alampäringuteta)**	**Vahe**
GROUP JOIN	**3,43**	3,84	11,14	7,30
GROUP JOIN FILTER	3,42	3,76	**1,12**	2,30
LAG FILTER	4,90	4,50	**0,00**	4,90
	PostgreSQL 9.1			
GROUP JOIN	**3,33**	3,33	11,89	8,56
GROUP JOIN FILTER	3,38	3,25	**0,54**	2,84
LAG FILTER	3,95	3,92	**0,00**	3,95

Suuremaks päringute täitmise kiiruse erinevuseks on 8,56 sekundit eksperimendis *GROUP JOIN* kirjeldatud päringutes. Mainitud päringute täitmisplaane kirjeldati eelmises jaotises.

PostgreSQL andmebaasisüsteemis on suured täitmise kiiruse erinevused korreleeruvate ja mittekorreleeruvate alampäringute korral – *NONCORR NOTIN* ja *CORR* (vt Tabel 16). Kuna nende päringute täitmisplaanid on täiesti identsed, siis ei hakata neid omavahel võrdlema. Üldiselt on näha, et päringud on väga aeglased ja põhjuseks on *SubPlan* operaatori kasutamine täitmisplaanis. Päring vaatele *NONCORR NOTIN* oli PostgreSQL 9.3 andmebaasisüsteemis nii aeglane, et jäigi täitmata. Selle pärast on koondtabelis (vt Tabel 16) antud väli on tühi.

```
CREATE VIEW vw_Organization_not_party (
    party_id, name, registration_code)
AS
SELECT party_id, name, registration_code
FROM Organization
WHERE party_id NOT IN (
    SELECT party_id
    FROM Health_care_visit_role );
```

Selle vaate *(vw_Organization_not_party)* alampäring sisaldab *NOT IN* predikaati ja mittekorreleeruvat alampäringut. Selle vaate ülesande saab lahendada ka alampäringuga, kus

kasutatakse vasakpoolset välisühendamist. Sellise vaate põhjal tehtud päringu täitmisplaan on parem ja päringu täitmise kiirus on alla ühe sekundi (vt Joonis 20).

```
CREATE VIEW vw_Organization_not_party_new (party_id, name, registration_code)
AS
SELECT org.party_id, org.name, org.registration_code
FROM Organization org
LEFT OUTER JOIN Health_care_visit_role vr ON org.party_id = vr.party_id
WHERE vr.party_id IS NULL;

SELECT * FROM vw_Organization_not_party_new;
```

```
Merge Anti Join  (cost=0.71..14023.71 rows=1 width=24) (actual time=524.633..656.109 rows=1 loops=1)
 Merge Cond: (org.party_id = vr.party_id)
 -> Index Scan using pk_organization on organization org  (cost=0.29..1300.17 rows=30000 width=24)
                                                          (actual time=0.013..60.211 rows=30000 loops=1)
 -> Index Only Scan using ixfk_role_party on health_care_visit_role vr  (cost=0.43..33768.43 rows=1300000 width=4)
                                                                        (actual time=0.068..299.170 rows=299988 loops=1)
    Heap Fetches: 0
Total runtime: 656.285 ms
```

Joonis 20. Vaatele tehtud parandatud päringu parandatud täitmisplaan

Järgnevalt esitatakse koondtabel suurematest päringute täitmise kiiruse erinevustest PostgreSQL 9.3 andmebaasisüsteemis (vt Tabel 18) kui võrrelda päringuid turvabarjääriga (*WITH (security_barrier)*) ja turvabarjäärita vaadete põhjal.

Tabel 18. Päringute täitmise kiiruse suuremad erinevused PostgreSQL 9.3 (sekundites)

	PostgreSQL 9.3		
	Päring vaatele *(SVW)* *koos turvabarjääriga*	**Päring vaatele *(VW)***	**Vahe**
ORDER BY FILTER	5,64	**0,01**	5,63
VIEWofVIEW	30,61	**27,45**	3,16
VIEWofVIEW 2	4,19	**1,20**	2,99

Nagu näha, siis võib turvabarjäär põhjustada mitteoptimaalse täitmisplaani genereerimist, kuna andmebaasisüsteem ei tee vaate mestimist. Tabelis (vt Tabel 18) on välja toodud sellised vaated, mille põhjal päringute täitmise kiirus on tänu turvabarjääri kasutamisele suurenenud rohkem kui kahe sekundi võrra, võrreldes vaadetega, mis on defineeritud „Testpäringud ja vaated“ peatükis. Tabelis on vaated, mis on tehtud teiste vaadete põhjal ja/või sisaldavad *DISTINCT* või *ORDER BY* klausli.

Järgnevalt esitatakse koondtabelina *ORDER BY FILTER* eksperimendi vaate põhjal tehtud päringute täitmisplaanide erinevusi (vt Tabel 19).

```
CREATE VIEW svw_Visit_Patient_dt (
    health_care_visit_id, from_date, thru_date,
    patient_id, first_name, last_name, personal_code)
WITH (security_barrier)
AS
SELECT health_care_visit_id, from_date, thru_date,
    patient_id, first_name, last_name, personal_code
FROM svw_Visit_Patient_asc
WHERE from_date = TO_DATE('23.04.2013','DD.MM.YYYY');
```

Tabel 19. *ORDER BY FILTER* päringute täitmisplaanide erinevused

	Päring vaatele *(SVW)* *koos turvabarjääriga*	**Päring vaatele *(VW)***
Täitmise kiirus (actual time) sekundites	5,64	0,01
Tagastatav ridade arv (rows)	207	207
1. *samm*	Tehakse tabelite ühendamine	Rakendatakse filter (*WHERE*)
2. *samm*	Sorteeritakse ridu	Tehakse tabelite ühendamine
3. *samm*	Rakendatakse filter (*WHERE*)	Sorteeritakse ridu
Tabelite ühendamise tulemus (ridade arv)	1 000 000	207

On selgelt näha, et *turvabarjäärita* vaate korral rakendatakse kõigepealt filter (WHERE klauslis olev tingimus) ja sellega vähendatakse oluliselt ühendatavate ridade hulka, mis kokkuvõttes annab oluliselt parema töökiiruse võrreldes päringuga, mis tehti turvabarjääriga vaate põhjal. Samas võib andmebaasi ründaja seda kurjasti ära kasutada luues ja kasutades vaate põhjal tehtud päringu tingimuses funktsiooni, mis reedab ründaja eest varju jääma pidanud andmed. Seega oleme jällegi olukorras, kus tuleb paika panna prioriteedid – kas olulisem on andmete turvalisus või päringute töökiirus – ja teha nendest lähtuvalt disaini valik.

5. Järeldusi ja soovitusi

Käesolevas peatükis võtan lühidalt kokku saadud tulemused ning esitan järeldusi ja soovitusi.

Nii Oracle Database kui PostgreSQL andmebaasisüsteem võib vaate põhjal tehtud päringu täitmiseks kasutada erinevaid lähenemisi. Võimalik on päringu ja vaate mestimine aga ka vaate alampäringu eraldi töötlemine. On mitmeid põhjuseid, miks andmebaasisüsteemi optimeerimismoodul (e optimeerija) võib mitte teha vaate mestimist – selline lähenemine võib olla maksumuse poolest odavam või ei suuda andmebaasisüste seda tehniliste piirangute tõttu teha. Kui päringu täitmisplaanis eksisteerib *VIEW* operatsioon (Oracle) või *SubPlan* (PostgreSQL), siis see näitab, et (vaate) alampäring täidetakse eraldi (st mestimist ei toimu).

Eksperimendi tulemus kinnitas Oracle dokumentatsioonis olevat infot [38], et optimeerimismoodul ei tee vaate mestimist selliste vaadete korral, mis sisaldavad *DISTINCT* klauslit, ühendi või lõike leidmist, sorteerimist, kokkuvõtte- või aknafunktsiooni, või rekursiivset päringut. Selliseid päringuid täidetakse järgnevalt.

- Kui (põhi-) päring vaatele sisaldab *WHERE* klauslit, siis optimeerija kasutab seda filtrit vaate alampäringus (*predicate pushdown*). Seda tehakse ainult siis kui filtri kasutamisest sõltub ridade arv, mida tuleb vaate alampäringus töödelda.
- Kui päringus tehakse vaate ja baastabeli ühendamist, siis vaate alampäring täidetakse eraldi ja siis ühendatakse baastabeliga.

PostgreSQL andmebaasisüsteem ei suuda kasutada põhipäringu tingimust (*WHERE* klausel) vaate alampäringus kui:

- Vaate alampäringus tehakse ühendi leidmist *UNION* või *UNION ALL*;
- Vaate alampäring sisaldab aknafunktsiooni (antud töös kasutati *ROWNUMBER() OVER()* ja *LAG() OVER()*);
- Vaate alampäringu täitmisplaanis kasutatakse *SubPlan* operaatorit (vaated *NONCORR IN* ja *CORR*).

Kui vaate alampäring sisaldab agregaatfunktsiooni (ilma *GROUP BY* klauslita), *ROWNUM* pseudoveergu, aknafunktsiooni *ROWNUMBER() OVER()* või rekursiivset päringut, siis täidetakse mõlemas andmebaasisüsteemis vaate põhjal tehtud päring ja vaate alampäring eraldi, st mestimist ei toimu. Vaate alampäringu tulemusele rakendatakse filtrit (*WHERE* klausel).

Oracle Database andmebaasisüsteemi puhul võib öelda, et vaated ei takistanud optimeerijal hea täitmisplaani valimist. Eksperimendi kirjelduses mainiti tihti, et pole soovitatav luua vaateid, mille alampäringus viidatakse teisele vaatele. Räägitakse, et selliste vaadete põhjal tehtud päringute korral võib tulemuseks olla mitteoptimaalne täitmisplaan. Kuid eksperimendis kasutatud sellist tüüpi vaate korral need väited tõeks ei osutunud ning andmebaasisüsteem valis täitmisplaani, mille alusel täideti lause kiiresti.

Lisaks on Oracle üks väheseid SQL-andmebaasisüsteeme, kus saab deklareerida vaadetele samasuguseid kitsendusi nagu baastabelitele (primaarvõtmed, unikaalsus ja välisvõtmed). Kitsendused saavad anda andmebaasisüsteemile (ja ka teistele tabelite kasutajatele) täiendavat infot tabelite kohta. Sellel põhineb näiteks päringute semantiline teisendamine (asendamine lihtsama kuid loogiliselt samaväärse päringuga). Seega oli Oracle Database 12c andmebaasisüsteemis oluline katsetada päringuid vaadete põhjal, millele on deklareeritud primaarvõtme, välisvõtme või unikaalsuse kitsendus. Nagu kahtlustasin, siis Oracle andmebaasisüsteem ei suuda neid kitsendusi vaadete põhjal tehtavate päringute täitmisplaanide koostamisel ära kasutada. Seega kindlasti üks viis kuidas andmebaasisüsteeme tulevikus paremaks muuta on kitsendustes tuleneva info parem ärakasutamine.

Antud töös saadud tulemuste järgi võib väita, et PostgreSQL andmebaasisüsteemis olid kõige problemaatilisteks vaated, mis on tehtud teiste vaadete põhjal (ühes viidatud vaate alampäringus kasutati *GROUP BY* klauslit koos kokkuvõttefunktsioonidega ja teises – aknafunktsioone (*WINDOW functions*)). Kuna sellised vaated põhjustavad mitteoptimaalse ja ressursimahuka (vaate põhjal tehtud) päringu täitmisplaani genereerimise, siis järeldusena soovitan PostgreSQLis mitte defineerida selliseid vaateid.

Teine soovitus PostgreSQL andmebaasisüsteemi kohta on, et päringutes pole soovitatav kasutada *NOT IN* või võrdluse (*>; <; ja teised*) predikaati koos korreleeruvate või mittekorreleeruvate alampäringutega. Üks päring, mis tehti vaate põhjal, mis sisaldas mittekorreleeruvat alampäringu koos *NOT IN* predikaadiga jäi andmebaasisüsteemil isegi täitmata. Sellised päringud on väga aeglased ning selle asemel on võimalik päringus kasutada välisühendamist või *(NOT) EXISTS* predikaati.

Kui PostgreSQLis soovitakse kasutada vaateid turvalisuse tagamiseks (ridadele juurdepääsu piiramiseks), siis tuleb vaated luua koos turvabarjääriga (*WITH (security_barrier*). Kahjuks seab see omakorda piirangud vaate mestimise (*view merging*) läbiviimisele. Mõnikord ei ole

sellisel juhul vaadete põhjal tehtud päringute täitmisplaanid kõige optimaalsemad. Eriti puudutab vaateid, mille alampäringus viidatakse teistele vaadetele (vt *eksperiment VIEWofVIEW*). Kui selliste vaadete põhjal tehakse päringuid, siis nende vaadete alampäringuid täidetakse eraldi ja sellel põhjusel suureneb ka päringu täitmise kiirus.

Oracle Database andmebaasisüsteemi puhul soovitan kasutada uuemat versiooni – 12c (Release 1). Katsete tulemusel saab öelda, et optimeerimismooduli töös on toimunud suured muudatused (võrreldes Oracle 11.1 süsteemiga). Samas vaate põhjal tehtud päringute täitmisplaanide genereerimise printsiibid ja algoritmid pole muutunud. Kuigi Oracle 11.1 näitas mitmel juhul paremat päringute täitmise kiirust kui 12.1, siis ei saa välistada, et see on tingitud mingitest riistvara seadistamise probleemidest kasutatud serveril. Eksperimendi päringute osas pole PostgreSQL 9.3 andmebaasisüsteemi optimeerimismoodul võrreldes versiooniga 9.1 üldse muutunud.

Sain tehtud tööst kasulikke teadmisi, mida edaspidi oma igapäevatöös kasutada. Töökohal kasutan Oracle Database andmebaasisüsteemi. Katsetused näitasid, et selle korral pole vaadete mõju päringute töökiirusele negatiivne. Seega saab julgelt soovitada selles virtuaalse andmekihi realiseerimist ja selleks vaadete kasutamist. Nüüdsest hakkan keerukate (näiteks, mitme tabeli ühendamist nõudvate) päringute korral kasutama vahetulemuste leidmist vaadete abil ning nende põhjal uue vaate abil lõpptulemuse kokku panemist. Sellise lähenemise eeliseks on, et see võimaldab keerulise ülesande lahendamisele läheneda samm sammult. Selle asemel, et üritada keerulist ülesannet ühekorraga (ühe lausega) lahendada, saab mõelda alamprobleemidele, mille lahendamine, viib samm sammult lõplikule lahendusele lähemale. Selline suure probleemi väiksemateks osadeks jagamine on tegelikult ammu tuntud probleemide lahendamise strateegia – jaga ja valitse [49].

Kokkuvõte

Käesoleva töö uurimisküsimuseks oli vaadete kasutamise mõju päringute täitmisplaanide valiku tegemisele ning töökiirusele kahe andmebaasisüsteemi näitel. Töö eesmärkideks oli uurida erinevusi täitmisplaanides ja neid erinevusi põhjustavaid faktoreid juhul kui päring on tehtud vaate põhjal ja kui loogiliselt samaväärne päring on tehtud otse baastabelite põhjal. Lisaks võrreldi päringute täitmise kiiruseid.

Eksperimendi teostamiseks kavandati ja realiseeriti andmebaas (baastabelid, primaar-, välisvõtmed, unikaalsuse kitsendused, indeksid ja vaated), lisati suur hulk testandmeid, valiti päringuid ning tehti ühesuguste ülesannete lahendamiseks mõeldud päringuid otse baastabelite põhjal ja vaadete põhjal. Toodi välja erinevused päringute täitmisplaanides ja analüüsiti nende põhjuseid.

Põhiliste andmebaasisüsteemidena kasutati eksperimendis Oracle Database 12c Enterprise Edition Release 1 ja PostgreSQL 9.3. Lisaks viidi eksperiment läbi ka nende andmebaasisüsteemide varasemates versioonides Oracle Database 11.1 ja PostgreSQL 9.1 nägemaks, kuidas on andmebaasisüsteemide optimeerimismoodul vaadete käsitlemise osas arenenud. Uuringu tulemused näitasid, et kuigi Oracle andmebaasisüsteemis põhjustab vaate põhjal päringu tegemine sageli teistsuguse täitmisplaani valimise kui otse baastabelite põhjal päringuid tehes, siis sellest tulenevad töökiiruse erinevused ei ole kuigi märgatavad. PostgreSQLis koostab andmebaasisüsteem (ilma turvabarjäärita) vaate põhjal tehtud päringule ning otse baastabelil tehtud päringule suurema tõenäosusega ühesuguse täitmisplaani. Kuid probleem on selles, et võrreldes Oraclega on see täitmisplaan mõnikord lihtsalt halb, olgu päring tehtud vaate põhjal või mitte. PostgreSQLis tuleks andmete turvalisuse tõstmiseks kasutada vaadetes turvabarjääre, kuid see võib tingida vaate põhjal tehtud päringule mitteoptimaalse (ja seega aeglaselt täidetava) täitmisplaani valimise. Eksperiment näitas ka seda, et Oracle optimeerimismooduli tööpõhimõtted on kahe vaadeldud versiooni vahel palju rohkem muutunud kui vaadeldud PostgreSQL versioonide korra.

Kuigi töös läbiviidud eksperimentide hulk oli suur – 25 (koos variatsioonidega veelgi rohkem), ei saa ma väita, et proovitud on kõikvõimalikke probleemseid päringuid. See on käesoleva töö piiranguks. Kuid eksperimentide valimisel on lähtutud kirjanduses välja toodud viidetest probleemsetele päringutele ja seega julgen väita, et need tulemused annavad kätte üldise õige suuna, mida loomulikult saab edasiste töödega täpsustada.

Vaadete loomisest andmebaasis tuleneb mitmesugust kasu. Soovitan andmebaasis vaateid luua. Selle töö tulemused aitavad andmebaasi kavandajatel otsustada, kas vaadetest tulenevad võimalikud töökiiruse probleemid kaaluvad tema jaoks üle vaadetest saadava muu kasu.

Käesolevas töös käsitleti andmebaasikeelt SQL ja andmebaasisüsteeme kus saab seda keelt kasutada (SQL-andmebaasisüsteeme). Aga vaateid võimaldavad teha ka teised andmebaasisüsteemid (need süsteemid võivad põhineda mõnel teisel andmemudelil kui SQLi aluseks olev andmemudel) ning seal võivad kuid ei pruugi olla sarnased probleemid. Seega töö edasiarenduseks on uurida vaadete mõju teist tüüpi andmebaasisüsteemide korral. Teiseks edasiarendamise suunaks on uurida vaadete mõju (mitte ainult töökiirusele) SQL-andmebaasisüsteemide korral, kuid kasutades sellist andmebaasi, mis on tegelikult kasutuses (näiteks, töö juures). See võimaldab uurida veel suuremat hulka vaateid ja päringuid ning nende mõju tegelikule süsteemile.

Summary

Research question of the master's thesis is how the usage of views affects the creation of query execution plans in the example of two database management systems (DBMSs). Differences of query execution plans as well as factors that cause the differences were researched. Two types of queries were analyzed: queries that reference views and logically equivalent queries on base tables. In addition, execution times of queries were compared.

An experiment was carried out: a database was designed and implemented (including base tables, primary and foreign keys, unique constraints, indexes and views); large amount test data was added; queries were chosen; queries based on views and a logically equivalent queries on base tables were executed and measured. Finally, differences in the query execution plans and the reasons were presented.

The database management systems (DBMSs) studied in the thesis include Oracle Database 12c Enterprise Edition Release 1 and PostgreSQL 9.3. The experiment was also carried out in the older versions of the DBMSs – Oracle 11.1 and PostgreSQL 9.1. The results showed that although in Oracle making queries based on views causes often selection of different execution plans compared to logically equivalent queries based on base tables, it does not cause notable differences in the speed of execution. In PostgreSQL there is a bigger probability that logically equivalent queries based on views (that do not have security barrier specified) and base tables will have the same execution plan. However, a problem is that sometimes PostgreSQL just selects a bad execution plan compared to Oracle. It does not depend on as to whether the query is based on views or base tables. In PostgreSQL one should use the security barrier in views to improve data security but it could cause selection of not optimal (and hence slowly performed) execution plans. The experiment also showed that the principles of Oracle optimizer have changed much more between the two reviewed versions than in case of two PostgreSQL versions.

Although the number of experiments carried out in this work is high – 25 in total (even more if variations are included), the author cannot suggest that all possible problematic queries have been studied. This is a restriction of the work. However, the selection of the experiments was based on the problematic queries that are referred in the literature. Hence, I claim that the results of the research give generally right direction that certainly can be elaborated by future studies.

The use of views in databases gives various advantages. I recommend to create views in databases. The results of this work help database designers to decide whether the possible performance problems due to views are more important to them compared to the advantages that the use of views will provide.

In this thesis, Structured Query Language (SQL) and some database management systems that use this language (SQL database management systems) were studied. However, there are other database management systems that also allow creating views (these systems can be based on different data models than the underlying data model of SQL) and that may or may not have similar problems. Hence, this research could be continued by studying the influence of views to the creation of query execution plans in other types of database management systems. Second line of research is to study the effect of views (not only query execution time) in case of SQL database management systems but in case of a database that is in real use (for instance at the workplace of the researcher). It would allow us to study wider range of views and queries and their influence to a real system.

Kasutatud kirjandus

[1] E. Eessaar, „Teema 4. Keerukamad SELECT laused,“ 2014. [Võrgumaterjal]. Available: http://maurus.ttu.ee/ained/IDU0220_2014/doc/3/Teema_IDU0220_4_2014.pdf. [Kasutatud 08. november 2014].

[2] E. Eessaar, „Teema 1. Andmebaaside põhimõisteid,“ 2014. [Võrgumaterjal]. Available: http://maurus.ttu.ee/ained/IDU0220_2014/doc/3/Teema_IDU0220_1_2014_ver3.pdf. [Kasutatud 08. november 2014].

[3] E. Eessaar, „Teema 3. Relatsioonialgebra. Sissejuhatus SQL keelde,“ 2014. [Võrgumaterjal]. Available: http://maurus.ttu.ee/ained/IDU0220_2014/doc/3/Teema_IDU0220_3_2014_ver3.pdf. [Kasutatud 08. november 2014].

[4] E. Eessaar, „Teema 7. SQL andmekäitluslausete töötlemine ja optimeerimine,“ Tallinn, 2011.

[5] M.-C. Shan ja P. Lyngbaek, „View composition in a data base management system“. Patent US5276870, 04 jaanuar 1994.

[6] T. Põder, „Explain Plan For command may show you the wrong execution plan – Part 1,“ 17 november 2009. [Võrgumaterjal]. Available: http://blog.tanelpoder.com/2009/11/17/explain-plan-for-command-may-show-you-the-wrong-execution-plan-part-1/. [Kasutatud 05. detsember 2014].

[7] The PostgreSQL Global Development Group, „PostgreSQL 9.3.5 Documentation, EXPLAIN,“ 1996-2014. [Võrgumaterjal]. Available: http://www.postgresql.org/docs/9.3/static/sql-explain.html. [Kasutatud 25. november 2014].

[8] E. Eessaar, „Teema 5. Andmemuudatused SQLis. SQLi andmekirjelduskeel,“ 2014. [Võrgumaterjal]. Available:

http://maurus.ttu.ee/ained/IDU0220_2014/doc/3/Teema_IDU0220_5_2014.pdf. [Kasutatud 01. detsember 2014].

[9] L. Burns, Building the Agile Database: How to Build a Successful Application Using Agile without Sacrificing Data Management, New Jersey: Technics Publications, 2011.

[10] J. R. Groff ja P. N. Weinberg, SQL: The Complete Reference, Second Edition, United States of America: McGraw-Hill Osborne Media, 2002.

[11] S. Pandya, „Predicate Pushing," 2007, 20 aprill. [Võrgumaterjal]. Available: http://www.devx.com/tips/Tip/34429. [Kasutatud 09. jaanuar 2015].

[12] J. Lewis, Cost-Based Oracle Fundamentals, New York: Apress, 2006.

[13] C. Antognini, Troubleshooting Oracle Performance, Second Edition, Apress, 2014.

[14] Oracle Corporation, „Oracle Database Performance Tuning Guide, 12c Release 1 (12.1)," Juuli 2014. [Võrgumaterjal]. Available: http://docs.oracle.com/database/121/TGDBA/E49058-05.pdf. [Kasutatud 08. september 2014].

[15] T. Connoly ja C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, Fourth Edition, Pearson Education, 2005.

[16] A. A. Pedersen, „Oracle Constraints on Views," www.databasedesign-resource.com, [Võrgumaterjal]. Available: http://www.databasedesign-resource.com/constraints-on-views.html. [Kasutatud 06. jaanuar 2015].

[17] G. Hull ja D. Rowley, „Mailing Lists, Re: View has different query plan than select statement," PostgreSQL, 20 mai 2014. [Võrgumaterjal]. Available: http://www.postgresql.org/message-id/552030819.222866.1400627206245.JavaMail.zimbra@mccarthy.co.nz. [Kasutatud 24. november 2014].

[18] W. P. Yan ja P.-Å. Larson, „Performing Group-By before Join,“ *Proceedings of the Tenth International Conference on Data Engineering*, Washington, 1994.

[19] Allison, „Optimizer Transformations: View Merging part 1,“ Oracle' Blogs, 13 oktoober 2010. [Võrgumaterjal]. Available: https://blogs.oracle.com/optimizer/entry/optimizer_transformations_view_merging_part_1. [Kasutatud 21. november 2014].

[20] Allison, „Optimizer Transformations: View Merging part 2,“ Oracle' Blogs, 29 oktoober 2010. [Võrgumaterjal]. Available: https://blogs.oracle.com/optimizer/entry/optimizer_transformations_view_merging_part_2. [Kasutatud 01. detsember 2014].

[21] Oracle Corporation, „Database SQL Tuning Guide: Query Transformations: View Merging,“ 2014. [Võrgumaterjal]. Available: https://docs.oracle.com/database/121/TGSQL/tgsql_transform.htm#TGSQL209. [Kasutatud 13. detsember 2014].

[22] Green ja C. Dialeris, „Oracle9i Database Performance Tuning Guide and Reference, Release 2 (9.2). Optimizer Operations,“ Oracle Corporation, Oktoober 2002. [Võrgumaterjal]. Available: https://docs.oracle.com/cd/B10501_01/server.920/a96533/opt_ops.htm. [Kasutatud 31. oktoober 2014].

[23] Paz ja F. Heikens, „Are PostgreSQL VIEWS created newly each time they are queried against?,“ stackoverflow, 04 august 2010. [Võrgumaterjal]. Available: http://stackoverflow.com/questions/3402834/are-postgresql-views-created-newly-each-time-they-are-queried-against. [Kasutatud 24. november 2014].

[24] „DB-Engines Ranking,“ DB-Engines, Jaanuar 2014. [Võrgumaterjal]. Available: http://db-engines.com/en/ranking. [Kasutatud 09. jaanuar 2015].

[25] L. Silverston, The Data Model Resource Book, Volume 2, New York: John Wiley & Sons, Inc., 2001.

[26] E. Eessaar, „Füüsiline disain - andmebaasisüsteemide võrdlus,“ Tallinn, 2011.

[27] „Join Elimination,“ Oracle-Base, [Võrgumaterjal]. Available: http://oracle-base.com/articles/misc/join-elimination.php. [Kasutatud 29. detsember 2014].

[28] „Mockaroo, realistic test data generator,“ [Võrgumaterjal]. Available: http://www.mockaroo.com/. [Kasutatud 14. oktoober 2014].

[29] „Mockaroo. Table "Party",“ [Võrgumaterjal]. Available: http://www.mockaroo.com/845321c0. [Kasutatud 14. oktoober 2014].

[30] „Mockaroo. Table "Person",“ [Võrgumaterjal]. Available: http://www.mockaroo.com/ea49c520. [Kasutatud 14. november 2014].

[31] „Mockaroo. Table "Organization",“ [Võrgumaterjal]. Available: http://www.mockaroo.com/67e62c60. [Kasutatud 14. november 2014].

[32] „Mockaroo. Table "Postal address",“ [Võrgumaterjal]. Available: http://www.mockaroo.com/d3a09b20. [Kasutatud 14. oktoober 2014].

[33] „Mockaroo. Table "Facility",“ [Võrgumaterjal]. Available: http://www.mockaroo.com/55bcecf0. [Kasutatud 14. oktoober 2014].

[34] „Mockaroo. Table "Health care visit",“ [Võrgumaterjal]. Available: http://www.mockaroo.com/ea8d1a50. [Kasutatud 14. oktoober 2014].

[35] „Mockaroo. Table "Health care visit role" - Person,“ [Võrgumaterjal]. Available: http://www.mockaroo.com/2292c1e0. [Kasutatud 14. november 2014].

[36] „Mockaroo. Table "Health care visit role" - Organization,“ [Võrgumaterjal]. Available: http://www.mockaroo.com/e6dbfe70. [Kasutatud 14. november 2014].

[37] The PostgreSQL Global Development Group, „PostgreSQL 9.3.5 Documentation: Rules and Privileges,“ 1996-2015. [Võrgumaterjal]. Available: http://www.postgresql.org/docs/9.3/interactive/rules-privileges.html. [Kasutatud 04. jaanuar 2015].

[38] „Oracle SQL Tuning Guide,“ 2003. [Võrgumaterjal]. Available: http://www.orafaq.com/tuningguide/. [Kasutatud 23. november 2014].

[39] Quassnoi, „Oracle: ROW_NUMBER vs ROWNUM,“ 06 mai 2009. [Võrgumaterjal]. Available: http://explainextended.com/2009/05/06/oracle-row_number-vs-rownum/. [Kasutatud 23. november 2014].

[40] Oracle Corporation, „Oracle Database Performance Tuning Guide, 10g Release 1 (10.1), The Query Optimizer,“ Oracle Corporation, Detsember 2003. [Võrgumaterjal]. Available: https://docs.oracle.com/cd/B13789_01/server.101/b10752/optimops.htm. [Kasutatud 24. november 2014].

[41] Oracle Corporation, „Oracle Database SQL Language Reference 12c Release 1 (12.1) , CURSOR Expressions,“ Juuli 2014. [Võrgumaterjal]. Available: http://docs.oracle.com/database/121/SQLRF/expressions006.htm#SQLRF52077. [Kasutatud 23. november 2014].

[42] E. Eessaar, „Andmebaasid II (IDU0230) ja Andmebaaside programmeerimine (IDU0120) (sügis 2014). SQL andmekäitluskeele lausete täitmine Oracle näitel,“ 22 detsember 2014. [Võrgumaterjal]. Available: http://maurus.ttu.ee/ained/IDU0230_2014/doc/11/SQL_DML_keele_lausete_taitmine_Oracle_naitel_IDU0230_2014_ver2.ppt. [Kasutatud 22. detsember 2014].

[43] J. Lewis, „Execution Plans Part 3: “The Rule”,“ 25 aprill 2014. [Võrgumaterjal]. Available: http://allthingsoracle.com/execution-plans-part-3-the-rule/. [Kasutatud 04. jaanuar 2015].

[44] Oracle Corporation, „Upgrading from Oracle Database 10g to 11g: What to expect from the Optimizer,“ What to expect from the Optimizer , November 2010. [Võrgumaterjal]. Available: http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-upgrading-10g-to-11g-what-to-ex-133707.pdf. [Kasutatud 10. jaanuar 2015].

[45] Burleson Consulting, „_optimizer_group_by_placement tips,“ 23 jaanuar 2013. [Võrgumaterjal]. Available: http://www.dba-oracle.com/p_optimizer_group_by_placement.htm. [Kasutatud 10. jaanuar 2015].

[46] M. Colgan, „Optimizer Transformation: Join Predicate Pushdown,“ 03 jaanuar 2011. [Võrgumaterjal]. Available: https://blogs.oracle.com/optimizer/entry/basics_of_join_predicate_pushdown_in_oracle. [Kasutatud 11. jaanuar 2015].

[47] Burleson Consulting, „Oracle leading hint tips,“ [Võrgumaterjal]. Available: http://www.dba-oracle.com/t_leading_hint.htm. [Kasutatud 11. jaanuar 2015].

[48] K. Douglas ja S. Douglas, „PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgreSQL Databases,“ books.google.ee, 2003. [Võrgumaterjal]. Available: https://books.google.ee/books?id=gkQVL9pyFVYC&hl=ru&source=gbs_navlinks_s. [Kasutatud 25. detsember 2014].

[49] „Divide and rule,“ Wikipedia, [Võrgumaterjal]. Available: http://en.wikipedia.org/wiki/Divide_and_rule. [Kasutatud 11. jaanuar 2015].

Lisa 1. Realisatsioon Oracle Database andmebaasisüsteemis

```
DROP TABLE Facility CASCADE CONSTRAINTS;
DROP TABLE Facility_type CASCADE CONSTRAINTS;
DROP TABLE Health_care_visit CASCADE CONSTRAINTS;
DROP TABLE Health_care_visit_role CASCADE CONSTRAINTS;
DROP TABLE Health_care_visit_role_type CASCADE CONSTRAINTS;
DROP TABLE Organization CASCADE CONSTRAINTS;
DROP TABLE Party CASCADE CONSTRAINTS;
DROP TABLE Person CASCADE CONSTRAINTS;
DROP TABLE Postal_address CASCADE CONSTRAINTS;

CREATE TABLE Facility_type (
    facility_type_code  CHAR(20) NOT NULL,
    name                VARCHAR2(50) NOT NULL,
    description         VARCHAR2(255)
);
ALTER TABLE Facility_type ADD CONSTRAINT PK_Facility_type PRIMARY KEY
(facility_type_code);
ALTER TABLE Facility_type ADD CONSTRAINT AK_Facility_type_name UNIQUE (name);

CREATE TABLE Facility (
    facility_id         NUMBER(10) NOT NULL,
    facility_name       VARCHAR2(100) NOT NULL,
    facility_type_code  CHAR(20) NOT NULL,
    description         VARCHAR2(255)
);
ALTER TABLE Facility ADD CONSTRAINT PK_Facility PRIMARY KEY (facility_id);
ALTER TABLE Facility ADD CONSTRAINT FK_Facility_Facility_type FOREIGN KEY
(facility_type_code) REFERENCES Facility_type (facility_type_code);
CREATE INDEX IXFK_Facility_Facility_type ON Facility (facility_type_code ASC);
ALTER TABLE Facility ADD CONSTRAINT AK_Facility_facility_name UNIQUE
(facility_name);

CREATE TABLE Postal_address (
    postal_address_id  NUMBER(10) NOT NULL,
    address            VARCHAR2(255) NOT NULL,
    city               VARCHAR2(50) NOT NULL,
    directions         VARCHAR2(255)
);
ALTER TABLE Postal_address ADD CONSTRAINT PK_Postal_address PRIMARY KEY
(postal_address_id);

CREATE TABLE Health_care_visit_role_type (
    role_type_code  CHAR(20) NOT NULL,
    name            VARCHAR2(50) NOT NULL,
    description     VARCHAR2(255)
);
ALTER TABLE Health_care_visit_role_type ADD CONSTRAINT
PK_Health_care_visit_role_type PRIMARY KEY (role_type_code);
ALTER TABLE Health_care_visit_role_type ADD CONSTRAINT
AK_Health_care_visit_rol_name UNIQUE (name);

CREATE TABLE Party (
    party_id        NUMBER(10) NOT NULL,
    contact_number  VARCHAR2(20) NOT NULL,
    comments        VARCHAR2(255)
```

```
);
ALTER TABLE Party ADD CONSTRAINT PK_Party PRIMARY KEY (party_id);

CREATE TABLE Organization (
    party_id           NUMBER(10) NOT NULL,
    name               VARCHAR2(100) NOT NULL,
    registration_code  CHAR(8) NOT NULL ,
    parent_id          NUMBER(10)
);
ALTER TABLE Organization ADD CONSTRAINT PK_Organization PRIMARY KEY (party_id);
ALTER TABLE Organization ADD CONSTRAINT FK_Organization_Party FOREIGN KEY
(party_id) REFERENCES Party (party_id) ON DELETE CASCADE;
ALTER TABLE Organization ADD CONSTRAINT AK_Organization_name UNIQUE (name);
ALTER TABLE Organization ADD CONSTRAINT AK_Organization_regcode UNIQUE
(registration_code);
CREATE INDEX IXFK_Organization_Organization ON Organization (parent_id ASC);

ALTER TABLE Organization ADD CONSTRAINT FK_Organization_Organization FOREIGN KEY
(parent_id) REFERENCES Organization (party_id) ON DELETE CASCADE;

CREATE TABLE Person (
    party_id         NUMBER(10) NOT NULL,
    first_name       VARCHAR2(50) NOT NULL,
    last_name        VARCHAR2(50) NOT NULL,
    personal_code    CHAR(11) NOT NULL,
    birth_date       DATE NOT NULL,
    document_title   VARCHAR2(20) NOT NULL,
    document_number  VARCHAR2(20) NOT NULL,
    height           NUMBER(5,2),
    weight           NUMBER(5,2),
    username         VARCHAR2(20) NOT NULL
);
ALTER TABLE Person ADD CONSTRAINT PK_Person PRIMARY KEY (party_id);
ALTER TABLE Person ADD CONSTRAINT FK_Person_Party FOREIGN KEY (party_id) REFERENCES
Party (party_id) ON DELETE CASCADE;
CREATE INDEX idx_Person_birth_date ON Person (EXTRACT ( YEAR FROM birth_date ));
ALTER TABLE Person ADD CONSTRAINT AK_Person_document UNIQUE (document_title,
document_number);
ALTER TABLE Person ADD CONSTRAINT AK_Person_username UNIQUE (username);
ALTER TABLE Person ADD CONSTRAINT AK_Person_personal_code UNIQUE (personal_code);
ALTER TABLE Person ADD CONSTRAINT chk_Person_personal_code CHECK
(REGEXP_LIKE(personal_code,'^([3-6]{1}[[:digit:]]{2}[0-1]{1}[[:digit:]]{1}[0-
3]{1}[[:digit:]]{5})$'));

CREATE TABLE Health_care_visit (
    health_care_visit_id  NUMBER(10) NOT NULL,
    patient_id            NUMBER(10) NOT NULL,
    facility_id           NUMBER(10),
    postal_address_id     NUMBER(10),
    from_date             DATE NOT NULL,
    thru_date             DATE,
    reason                VARCHAR2(255),
    visit_fee             NUMBER(8,2)
);
ALTER TABLE Health_care_visit ADD CONSTRAINT PK_Health_care_visit PRIMARY KEY
(health_care_visit_id);
```

```
ALTER TABLE Health_care_visit ADD CONSTRAINT FK_Health_care_visit_Address FOREIGN
KEY (postal_address_id) REFERENCES Postal_address (postal_address_id);
ALTER TABLE Health_care_visit ADD CONSTRAINT FK_Health_care_visit_Facility FOREIGN
KEY (facility_id) REFERENCES Facility (facility_id);

ALTER TABLE Health_care_visit ADD CONSTRAINT FK_Health_care_visit_Person FOREIGN
KEY (patient_id) REFERENCES Person (party_id);

CREATE INDEX IXFK_Health_care_visit_Facilit ON Health_care_visit (facility_id ASC);
CREATE INDEX IXFK_Health_care_visit_Address ON Health_care_visit (postal_address_id
ASC);
CREATE INDEX idx_Health_care_visit_FromDate ON Health_care_visit (from_date ASC);

ALTER TABLE Health_care_visit ADD CONSTRAINT UQ_Visit_facil_addr_pat_fromd UNIQUE
(patient_id, facility_id, postal_address_id, from_date);

ALTER TABLE Health_care_visit ADD CONSTRAINT chk_Visit_facility_postal_addr CHECK
(( facility_id IS NOT NULL AND postal_address_id IS NULL ) OR ( facility_id IS NULL
AND postal_address_id IS NOT NULL ));


CREATE TABLE Health_care_visit_role
(
    health_care_visit_id  NUMBER(10) NOT NULL,
    role_type_code        CHAR(20) NOT NULL,
    party_id              NUMBER(10) NOT NULL
);
ALTER TABLE Health_care_visit_role ADD CONSTRAINT PK_Visit_Role_type_Party PRIMARY
KEY (health_care_visit_id, party_id, role_type_code);

ALTER TABLE Health_care_visit_role ADD CONSTRAINT FK_Role_Party FOREIGN KEY
(party_id) REFERENCES Party (party_id);

ALTER TABLE Health_care_visit_role ADD CONSTRAINT FK_Role_Visit FOREIGN KEY
(health_care_visit_id) REFERENCES Health_care_visit (health_care_visit_id) ON
DELETE CASCADE;

ALTER TABLE Health_care_visit_role ADD CONSTRAINT FK_Role_Role_type FOREIGN KEY
(role_type_code) REFERENCES Health_care_visit_role_type (role_type_code);

CREATE INDEX IXFK_Role_Role_type ON Health_care_visit_role (role_type_code ASC);
CREATE INDEX IXFK_Role_Party ON Health_care_visit_role (party_id ASC);
```

Lisa 2. Realisatsioon PostgreSQL andmebaasisüsteemis

```
DROP TABLE Facility CASCADE;
DROP TABLE Facility_type CASCADE;
DROP TABLE Health_care_visit CASCADE;
DROP TABLE Health_care_visit_role CASCADE;
DROP TABLE Health_care_visit_role_type CASCADE;
DROP TABLE Organization CASCADE;
DROP TABLE Party CASCADE;
DROP TABLE Person CASCADE;
DROP TABLE Postal_address CASCADE;

CREATE TABLE Facility_type (
    facility_type_code char(20) NOT NULL,
    name varchar(50) NOT NULL,
    description varchar(255)
);
ALTER TABLE Facility_type ADD CONSTRAINT PK_Facility_type PRIMARY KEY
(facility_type_code);
ALTER TABLE Facility_type ADD CONSTRAINT AK_Facility_type_name UNIQUE (name);

CREATE TABLE Facility (
    facility_id integer NOT NULL,
    facility_name varchar(100) NOT NULL,
    facility_type_code char(20) NOT NULL,
    description varchar(255)
);
ALTER TABLE Facility ADD CONSTRAINT PK_Facility PRIMARY KEY (facility_id);
ALTER TABLE Facility ADD CONSTRAINT FK_Facility_Facility_type FOREIGN KEY
(facility_type_code) REFERENCES Facility_type (facility_type_code) ON DELETE NO
ACTION ON UPDATE CASCADE;
CREATE INDEX IXFK_Facility_Facility_type ON Facility (facility_type_code);
ALTER TABLE Facility ADD CONSTRAINT AK_Facility_facility_name UNIQUE
(facility_name);

CREATE TABLE Postal_address (
    postal_address_id integer NOT NULL,
    address varchar(255) NOT NULL,
    city varchar(50) NOT NULL,
    directions varchar(255)
);
ALTER TABLE Postal_address ADD CONSTRAINT PK_Postal_address PRIMARY KEY
(postal_address_id);

CREATE TABLE Health_care_visit_role_type (
    role_type_code char(20) NOT NULL,
    name varchar(50) NOT NULL,
    description varchar(255)
);
ALTER TABLE Health_care_visit_role_type ADD CONSTRAINT
PK_Health_care_visit_role_type PRIMARY KEY (role_type_code);
ALTER TABLE Health_care_visit_role_type ADD CONSTRAINT
AK_Health_care_visit_rol_name UNIQUE (name);

CREATE TABLE Party (
    party_id integer NOT NULL,
    contact_number varchar(20) NOT NULL,
```

```
    comments varchar(255)
);
ALTER TABLE Party ADD CONSTRAINT PK_Party PRIMARY KEY (party_id);

CREATE TABLE Organization (
    party_id integer NOT NULL,
    name varchar(100) NOT NULL,
    registration_code char(8) NOT NULL,
    parent_id integer
);
ALTER TABLE Organization ADD CONSTRAINT PK_Organization PRIMARY KEY (party_id);
ALTER TABLE Organization ADD CONSTRAINT FK_Organization_Party FOREIGN KEY
(party_id) REFERENCES Party (party_id) ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE Organization ADD CONSTRAINT AK_Organization_regcode UNIQUE
(registration_code);
ALTER TABLE Organization ADD CONSTRAINT AK_Organization_name UNIQUE (name);
CREATE INDEX IXFK_Organization_Organization ON Organization (parent_id);

ALTER TABLE Organization ADD CONSTRAINT FK_Organization_Organization FOREIGN KEY
(parent_id) REFERENCES Organization (party_id) ON DELETE CASCADE ON UPDATE CASCADE;


CREATE TABLE Person (
    party_id integer NOT NULL,
    first_name varchar(50) NOT NULL,
    last_name varchar(50) NOT NULL,
    personal_code char(11) NOT NULL,
    birth_date date NOT NULL,
    document_title varchar(20) NOT NULL,
    document_number varchar(20) NOT NULL,
    height decimal(5,2),
    weight decimal(5,2),
    username varchar(20) NOT NULL
);
ALTER TABLE Person ADD CONSTRAINT PK_Person PRIMARY KEY (party_id);
ALTER TABLE Person ADD CONSTRAINT FK_Person_Party FOREIGN KEY (party_id) REFERENCES
Party (party_id) ON DELETE CASCADE ON UPDATE CASCADE;
CREATE INDEX idx_Person_birth_date ON Person (EXTRACT ( YEAR FROM birth_date ));
ALTER TABLE Person ADD CONSTRAINT AK_Person_username UNIQUE (username);
ALTER TABLE Person ADD CONSTRAINT AK_Person_document UNIQUE (document_title,
document_number);
ALTER TABLE Person ADD CONSTRAINT AK_Person_personal_code UNIQUE (personal_code);
ALTER TABLE Person ADD CONSTRAINT chk_Person_personal_code CHECK
(personal_code~'^([3-6]{1}[[:digit:]]{2}[0-1]{1}[[:digit:]]{1}[0-
3]{1}[[:digit:]]{5})$');

CREATE TABLE Health_care_visit (
    health_care_visit_id integer NOT NULL,
    patient_id integer NOT NULL,
    facility_id integer,
    postal_address_id integer,
    from_date date NOT NULL,
    thru_date date,
    reason varchar(255),
    visit_fee decimal(8,2)
);
ALTER TABLE Health_care_visit ADD CONSTRAINT PK_Health_care_visit PRIMARY KEY
(health_care_visit_id);
```

```
ALTER TABLE Health_care_visit ADD CONSTRAINT FK_Health_care_visit_Person FOREIGN
KEY (patient_id) REFERENCES Person (party_id) ON DELETE NO ACTION ON UPDATE
CASCADE;

ALTER TABLE Health_care_visit ADD CONSTRAINT FK_Health_care_visit_Address FOREIGN
KEY (postal_address_id) REFERENCES Postal_address (postal_address_id) ON DELETE NO
ACTION ON UPDATE CASCADE;

ALTER TABLE Health_care_visit ADD CONSTRAINT FK_Health_care_visit_Facility FOREIGN
KEY (facility_id) REFERENCES Facility (facility_id) ON DELETE NO ACTION ON UPDATE
CASCADE;

CREATE INDEX IXFK_Health_care_visit_Address ON Health_care_visit
(postal_address_id);
CREATE INDEX IXFK_Health_care_visit_Facilit ON Health_care_visit (facility_id);
CREATE INDEX idx_Health_care_visit_FromDate ON Health_care_visit (from_date);

ALTER TABLE Health_care_visit ADD CONSTRAINT UQ_Visit_facil_addr_pat_fromd
UNIQUE (patient_id, facility_id, postal_address_id, from_date);

ALTER TABLE Health_care_visit ADD CONSTRAINT chk_Visit_facility_postal_addr
CHECK (( facility_id IS NOT NULL AND postal_address_id IS NULL )
OR ( facility_id IS NULL AND postal_address_id IS NOT NULL ));


CREATE TABLE Health_care_visit_role (
    health_care_visit_id integer NOT NULL,
    role_type_code char(20) NOT NULL,
    party_id integer NOT NULL
);
ALTER TABLE Health_care_visit_role ADD CONSTRAINT PK_Visit_Role_type_Party
PRIMARY KEY (health_care_visit_id, party_id, role_type_code);

ALTER TABLE Health_care_visit_role ADD CONSTRAINT FK_Role_Party FOREIGN KEY
(party_id) REFERENCES Party (party_id) ON DELETE NO ACTION ON UPDATE CASCADE;

ALTER TABLE Health_care_visit_role ADD CONSTRAINT FK_Role_Role_type FOREIGN KEY
(role_type_code) REFERENCES Health_care_visit_role_type (role_type_code) ON DELETE
NO ACTION ON UPDATE CASCADE;

ALTER TABLE Health_care_visit_role ADD CONSTRAINT FK_Role_Visit FOREIGN KEY
(health_care_visit_id) REFERENCES Health_care_visit (health_care_visit_id) ON
DELETE CASCADE ON UPDATE CASCADE;

CREATE INDEX IXFK_Role_Role_type ON Health_care_visit_role (role_type_code);
CREATE INDEX IXFK_Role_Party ON Health_care_visit_role (party_id);
```

Lisa 3. Vaadete kustutamise laused

```
DROP VIEW vw_Visit_filter_by_facility_id;
DROP VIEW vw_Visit_at_facility_or_addr;
DROP VIEW vw_Party_name_type;
DROP VIEW vw_Party_name_type_all;
DROP VIEW vw_Equal_names;
DROP VIEW vw_Number_of_patients_who_man;
DROP VIEW vw_Patient_suminfo;
DROP VIEW vw_Patient_suminfo_max_bdt;
DROP VIEW vw_Patient_visitinfo;
DROP VIEW vw_Party_quantity_distinct_1;
DROP VIEW vw_Party_quantity_distinct_2;
DROP VIEW vw_Visit_Patient_dt;
DROP VIEW vw_Visit_Patient_asc;
DROP VIEW vw_Visit_in_hospital;
DROP VIEW vw_Organization_not_party;
DROP VIEW vw_Person_weight;
DROP VIEW vw_Party_rownumber_by_username;
DROP VIEW vw_Party_fetch_999_username;
DROP VIEW vw_Party_rownum_999_username;
DROP VIEW vw_Party_rownumbe_999_username;
DROP VIEW vw_Visit_fee_prev_filter;
DROP VIEW vw_Visit_fee_previous;
DROP VIEW vw_Visits_facil_and_addr;
DROP VIEW vw_Employee_Employer;
DROP VIEW vw_Employers;
DROP VIEW vw_Patients_visits;
DROP VIEW vw_Organization_hierarchy;

DROP VIEW vw_Organization_not_party_new;
```

Lisa 4. Kitsenduste sisse- ja väljalülitamise laused

```
ALTER TABLE Organization DISABLE CONSTRAINT AK_Organization_name;
ALTER TABLE Organization DISABLE CONSTRAINT AK_Organization_regcode;
ALTER TABLE Facility DISABLE CONSTRAINT AK_Facility_facility_name;
ALTER TABLE Person DISABLE CONSTRAINT AK_Person_personal_code;
ALTER TABLE Person DISABLE CONSTRAINT chk_Person_personal_code;
ALTER TABLE Person DISABLE CONSTRAINT AK_Person_document;
ALTER TABLE Health_care_visit DISABLE CONSTRAINT chk_Visit_facility_postal_addr;
ALTER TABLE Health_care_visit DISABLE CONSTRAINT FK_Health_care_visit_Address;
ALTER TABLE Health_care_visit DISABLE CONSTRAINT FK_Health_care_visit_Facility;
ALTER TABLE Health_care_visit DISABLE CONSTRAINT UQ_Visit_facil_addr_pat_fromd;
ALTER TABLE Organization DISABLE CONSTRAINT FK_Organization_Organization;

ALTER TABLE Organization ENABLE CONSTRAINT AK_Organization_name;
ALTER TABLE Organization ENABLE CONSTRAINT AK_Organization_regcode;
ALTER TABLE Facility ENABLE CONSTRAINT AK_Facility_facility_name;
ALTER TABLE Person ENABLE CONSTRAINT AK_Person_personal_code;
ALTER TABLE Person ENABLE CONSTRAINT chk_Person_personal_code;
ALTER TABLE Person ENABLE CONSTRAINT AK_Person_document;
ALTER TABLE Health_care_visit ENABLE CONSTRAINT chk_Visit_facility_postal_addr;
ALTER TABLE Health_care_visit ENABLE CONSTRAINT chk_Visit_facility_postal_addr;
ALTER TABLE Health_care_visit ENABLE CONSTRAINT FK_Health_care_visit_Address;
ALTER TABLE Health_care_visit ENABLE CONSTRAINT FK_Health_care_visit_Facility;
ALTER TABLE Health_care_visit ENABLE CONSTRAINT UQ_Visit_facil_addr_pat_fromd;
ALTER TABLE Organization ENABLE CONSTRAINT FK_Organization_Organization;
```

Lisa 5. Hierarhia loomiseks kasutatav protseduur

```
create or replace PROCEDURE CreateHierarchy
IS
     TYPE cur_typ IS REF CURSOR;
    c cur_typ;
    Vparent_0_id NUMBER(10,0);

    TYPE cur_typ_1 IS REF CURSOR;
    cc cur_typ_1;
    Vparent_1_id NUMBER(10,0);
BEGIN
    UPDATE Organization
    SET parent_id = 0
    WHERE ROWNUM <= 15000;
    COMMIT;

    OPEN c FOR
      SELECT party_id
      FROM Organization
      WHERE parent_id = 0
        AND ROWNUM <= 5000;

      LOOP
          FETCH c INTO Vparent_0_id;
          EXIT WHEN c%NOTFOUND;

          UPDATE Organization
          SET parent_id = Vparent_0_id
          WHERE parent_id IS NULL
            AND ROWNUM <= 2;
          COMMIT;
      END LOOP;
    CLOSE c;

    OPEN cc FOR
      SELECT party_id
      FROM Organization
      WHERE parent_id != 0
        AND parent_id IS NOT NULL
        AND ROWNUM <= 5000;

      LOOP
          FETCH cc INTO Vparent_1_id;
          EXIT WHEN cc%NOTFOUND;

          UPDATE Organization
          SET parent_id = Vparent_1_id
          WHERE parent_id IS NULL
            AND ROWNUM = 1;
          COMMIT;
      END LOOP;
    CLOSE cc;
END;
```