TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Mõnede SQL-andmebaasisüsteemide võimekusest SQLi keelelise liiasuse silumisel

Magistritöö

Üliõpilane: Mattias Männil
Üliõpilaskood: 100227IAPMM
Juhendaja: Erki Eessaar

Tallinn

2014

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

Männil M. (2014) Mõnede SQL-andmebaasisüsteemide võimekusest SQLi keelelise liiasuse silumisel. Magistritöö, Tallinna Tehnikaülikool.

Käesolev magistritöö uurib SQL andmebaasikeele liiasuse mõjusid mõnede moodsate andmebaasisüsteemide töökiirusele. Mõjude hindamiseks viiakse läbi laiendatud versioon Fabian Pascali 1988. aasta eksperimendist, ühesuguse tulemuse kuid erineva süntaksiga SQL päringute kohta ning analüüsitakse nende päringute täitmisplaane. Uuritavateks andmebaasisüsteemideks on PostgreSQL ja Oracle. Töö järeldustena tõdetakse, et SQL keele liiasus on endiselt potentsiaalne probleemide allikas, kuna andmebaasisüsteemide optimeerimismoodulid (optimeerijad) ei suuda kõiki sama tulemusega päringuid samale täitmisplaanile taandada. Siiski on antud küsimuses aastate jooksul toimunud selge areng paremuse poole.

Magistritöö on koostatud eesti keeles ning koosneb 74 leheküljest. Töö sisaldab 30 joonist ja 10 tabelit.

Abstract

Männil M. (2014) On Capability of Some SQL Database Management Systems to Smooth Language Redundancy in SQL. Master's Thesis, Tallinn University of Technology.

Current Master's Thesis investigates the effects of the language redundancy of SQL to the performance of some of the modern database management systems (DBMSs). We perform an expanded version of Fabian Pascal's 1988 experiment about SQL queries that produce the equal results but have different syntax. We analyse the execution plans of those queries. DBMSs that we use for the investigation are PostgreSQL and Oracle. The thesis concludes that SQL language redundancy is still a potential source of problems because optimization modules of DBMSs (optimizers) are unable to optimise all queries with equal result to identical execution plan. There has been a clear progress on this issue over the years.

Master's Thesis is written in Estonian and there are 74 pages. Thesis includes 30 figures and 10 tables.

Lühendite ja mõistete sõnastik

alampäring – sub-query

andmebaasisüsteem – Database Management System (DBMS)

CTE – Common Table Expression

DCL – Data Control Language

DDL – Data Definition Language

DML – Data Manipulation Language

liiasus – redundancy

lihtoperatsioon – primitive operation

primaarvõti – primary key

päringu optimeerimine – query optimization

relatsiooniline andmebaas – relational database

räsitabel – hash table

räsiühendamine – hash-join

SQL – Structured Query Language

täitmisplaan – execution plan, query plan

Jooniste nimekiri

Joonis 1 – SQL keele kattuvus SQL standardi ja kommertssüsteemide vahel 15
Joonis 2 – täitmisplaani näide koos tegevustega.. 18
Joonis 3 – täitmisplaani näites aja ja mahuhinnangud 18
Joonis 4 – andmetabelid Pascali eksperimendis 19
Joonis 5 – Pascali eksperimendi tulemused kui indeksid olid ainult võtmeveergudel 20
Joonis 6 – eksperimendis kasutatavad andmebaasitabelid.. 27
Joonis 7 – originaalpäringus kasutatavad veerud.. 29
Joonis 8 – laiendatud päringus kasutatavad veerud 31
Joonis 9 – originaalpäringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseerimata tabelis 33
Joonis 10 – originaalpäringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseeritud tabelis 34
Joonis 11 – originaalpäringu täitmisplaan kõigil juhtudel IN2, ANY2, SOME2 korral PostgreSQL andmebaasisüsteemis.. 35
Joonis 12 – originaalpäringu täitmisplaan PostgreSQL põhigrupi päringutes kui tabelis on 9 900 rida ilma kasutaja poolt loodud indeksiteta 35
Joonis 13 – originaalpäringu täitmisplaan PostgreSQL põhigrupi päringutes kui tabelis on 20 miljonit rida koos kasutaja poolt loodud indeksitega 35
Joonis 14 – originaalpäringu täitmisplaan enamikes Oracle 11g päringutes 20 miljoni reaga kasutaja poolt indekseeritud tabelil.. 36
Joonis 15 – originaalpäringu täitmisplaan FULL JOIN korral Oracle 11g päringutes 20 miljoni reaga tabelis 37
Joonis 16 – originaalpäringu täitmisplaan FULL JOIN korral Oracle 12c päringutes 20 miljoni reaga kasutaja poolt indekseeritud tabelis.. 37
Joonis 17 – laiendatud päringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseerimata tabelis 39

Joonis 18 – laiendatud päringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseeritud tabelis 40
Joonis 19 – laiendatud päring IN2, ANY2, SOME2 päringutes PostgreSQL 9.1 ja 9.3 andmebaasis 20 miljoni reaga kasutaja poolt indekseerimata tabelis 42
Joonis 20 – laiendatud päring COUNT päringutes PostgreSQL 9.1 ja 9.3 andmebaasis 20 miljoni reaga kasutaja poolt indekseerimata tabelis 43
Joonis 21 – laiendatud päring IN1, ANY1, EXISTS, SOME päringutes PostgreSQL 9.3 andmebaasis 20 miljoni reaga kasutaja poolt indekseeritud tabelis........ 44
Joonis 22 – laiendatud päring JOIN jne. päringutes Oracle 12c andmebaasis 20 miljoni reaga kasutaja poolt indekseeritud tabelis 45
Joonis 23 – laiendatud päring FULL JOIN päringus Oracle 11g andmebaasis kasutaja poolt indekseerimata tabelites 46
Joonis 24 – laiendatud päring FULL JOIN päringus Oracle 12c andmebaasis kasutaja poolt indekseerimata tabelites 47
Joonis 25 – lisatest Oracle 11g NATURAL JOIN vea kontrollimiseks 48
Joonis 26 – laiendatud päring NATURAL JOIN Oracle 11g andmebaasis........ 48
Joonis 27 – mediaanpalga päringud 20 miljoni kirjega kasutaja poolt indekseeritud ja indekseerimata tabelites 49
Joonis 28 – MEDIAN1 täitmisplaan........ 50
Joonis 29 – MEDIAN2 täitmisplaan........ 51
Joonis 30 – MEDIAN3 täitmisplaan........ 51

Tabelite nimekiri

Tabel 1 – erinevad täitmisplaanid originaalpäringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis ... 32
Tabel 2 – päringute kestvused sekundites originaalpäringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis ... 32
Tabel 3 – erinevad täitmisplaanid originaalpäringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis ... 33
Tabel 4 – päringute kestvused sekundites originaalpäringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis ... 33
Tabel 5 – erinevad täitmisplaanid laiendatud päringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis ... 39
Tabel 6 – päringute kestvused sekundites laiendatud päringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis ... 39
Tabel 7 – erinevad täitmisplaanid laiendatud päringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis ... 40
Tabel 8 – päringute kestvused sekundites laiendatud päringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis ... 40
Tabel 9 – erinevad täitmisplaanid PostgreSQL mediaani päringutes koos ja ilma kasutaja poolt loodud indeksiteta ... 49
Tabel 10 – mediaani päringute kiirus sekundites PostgreSQLis ja Oracles ... 49

Sisukord

Annotatsioon **3**
Abstract **4**
Lühendite ja mõistete sõnastik **5**
Jooniste nimekiri **6**
Tabelite nimekiri **8**
Sisukord **9**
Sissejuhatus **11**
1. SQL keel **13**
1.1 Ajalugu 13
1.2 Ülevaade 14
1.3 Probleemid ja kriitika 15
2. SQL päringute täitmisplaanid **17**
3. Pascali eksperiment **19**
3.1 Pascali eksperiment 19
3.2 Pascali eksperimendi tulemused 20
3.3 Riina Kivi korduseksperiment 21
3.4 Muud analoogsed eksperimendid 22
4. Eksperiment **23**
4.1 Eksperimendi eesmärgid 23
4.2 Kasutatavad andmebaasid 25
4.4 Testandmebaasid ja testandmed 27
4.5 Päringud 29
5. Tulemused **32**
5.1 Pascali originaalpäring 32
5.2 Laiendatud päring 39
5.3 Mediaanpalga päring 49
6. Tulemuste analüüs ja järeldused **52**

Kokkuvõte **54**
Summary **56**
Kasutatud kirjandus **58**
Lisad **60**
Lisa 1 - Pascali originaalpäring 60
Lisa 2 - Andmebaasitabelite loomise laused 62
Lisa 3 - Testandmete sisestamise laused 64
Lisa 4 - Laiendatud päring 67
Lisa 5 - Mediaanpäring 71
Lisa 6 - Oracle 11g vea kordamise testandmed ja päring 73

Sissejuhatus

Andmebaasisüsteemid e andmebaasihaldurid (DBMS – Database Management System) on kasutusel praktiliselt kõigis suuremates asutustes ja ettevõtetes, kuna need võimaldavad hoiustada, struktureerida ja kasutada väga suuri andmehulki paremini kui alternatiivsed meetodid. Andmemahtude kasvamisel ja nende keerukuse tõusmisel kasvab paratamatult nende käitlemiseks kuluv aeg ja muu ressurss. Seetõttu on vajalik, et andmeid käideldaks võimalikult optimaalselt, kuna vastasel juhul võivad suurte andmemahtude korral tekkida probleemid andmebaasisüsteemi operatsioonide töökiiruses, mis mõjutavad lõpuks negatiivselt konkreetse organisatsiooni tööprotsesside toimimist. Iga andmebaasisüsteem võimaldab kasutada ühte või mitut andmebaasikeelt. Väga levinud andmebaasikeeleks on tänapäeval SQL keel, seega on vajalik teada antud keele potentsiaalseid ohukohti.

Fabian Pascal viis 1988. aastal läbi eksperimendi hindamaks SQL keele liiasusest (*redundancy*) tekkivaid probleeme. SQL keele „liiasuse“ all peetakse silmas asjaolu, et väga paljude andmekäitluse (andmete otsimine või muutmise) ülesannete lahenduse kirjutamiseks saab kasutada mitmeid eri viise, ilma et saavutatav tulemus muutuks. Pascali eksperimendis võrreldi kujult erineva kuid ekvivalentse tulemusega SQL päringute (SELECT lausete) kiirust viie erineva andmebaasisüsteemi korral. Tulemused näitasid, et päringute kiirus oli tugevalt sõltuv nende süntaksist ning varieerus ka eri andmebaasisüsteemide vahel.

Magistritöö eesmärk on uurida Pascali tõstatatud probleemi aktuaalsust mõnede moodsate andmebaasisüsteemide korral. Töös vaadeldavaid andmebaasisüsteeme (PostgreSQL ja Oracle) kasutatakse TTÜs andmebaaside õpetamisel ning kuuluvad maailma kõige populaarsemate andmebaasisüsteemide hulka. Antud töös on kavas laiendada uuritavate päringute hulka ning minna erinevuste uurimisel rohkem sügavuti. Erinevalt varasemast kus fookus oli põhiliselt töökiiruse võrdlemisel, on nüüd eesmärgiks vaadelda ja võrrelda konkreetsete päringute täitmisplaane, saamaks täpsemat ülevaadet probleemide olemusest.

Käesolev magistritöö võiks huvi pakkuda mahukate SQL-andmebaasidega igapäevaselt töötavatele isikutele, eriti just neile, kes kasutavad Oracle või PostgreSQL andmebaase. Samuti võib magistritöö osutuda kasulikuks SQL keelt ja selle teostusi (*implementation*) käsitlevate uurimuste jaoks ning ka SQLi tutvustavates kursustes SQLi keelelise liiasuse illustreerimiseks.

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

Töö esimene peatükk annab ülevaate SQL keelest, selle ajaloost, ülesehitusest ja tähelepanuväärsematest probleemidest.

Teine peatükk annab lühiülevaate andmebaasipäringute täitmisplaanidest.

Kolmas peatükk tutvustab Fabian Pascali 1988. aastal tehtud eksperimenti ja selle tulemusi ning antud eksperimendi hilisemaid kordamisi.

Neljas peatükk tutvustab antud töös tehtavat eksperimenti ja selle eesmärke, sealhulgas kasutatavaid andmebaasisüsteeme, testandmeid ja päringuid.

Viies peatükk sisaldab tehtud eksperimendi tulemusi, tuues esile tõsisemad kiiruslikud erinevused kasutatud päringutes, ning võrreldes täitmisplaanide erinevusi.

Kuues peatükk analüüsib kokkuvõtlikult eksperimendi tulemusi.

1. SQL keel

Andmebaasihalduses on väga tähtis, et oleks üks terviklik keel, milles kasutajad saavad lihtsalt ja arusaadavalt väljendada oma soove andmebaasisüsteemile. Erinevate andmebaasikeelte seas on kõige laiemalt kasutusel SQL ehk Structured Query Language. (Wang ja Tan 2006 p 64)

1.1 Ajalugu

Relatsiooniliste andmebaaside teooria pakkus Dr. Edgar Codd laiale avalikusele välja 1970. aasta artiklis "A Relational Model for Data for Large Shared Data Banks." Coddi teooria põhines rangetel matemaatilistel printsiipidel. Lihtsalt öeldes on relatsioon suhe kahe või rohkema elementide hulga liikmete vahel. (Fernandez 2009 p 5)

1974. aastal avaldasid Don Chamberlin ja Raymond Boyce artikli keelest "SEQUEL" (Structured English Query Language), mida arendati IBM San Jose Research Laboratorys, liideseks relatsioonilise süsteemi prototüübile System R samanimelises projektis. 1977. aastaks oli IBMi defineeritud ja realiseeritud järgmine versioon sellest keelest, SEQUEL/2. 1970ndate lõpus ilmnes, et "Sequel" on eksisteeriv kaubamärk ning keel nimetati ümber SQLiks. (Halpin ja Morgan 2010 p 556)

System R projekti kogemuste põhjal ehitas IBM oma esimese relatsioonilise andmebaasisüsteemi SQL/DS, mis paisati müüki 1981. Kaks aastat hiljem järgnes sellele suurema mõjuga toode DB2. Kuna infot SQL keele kohta oli 1970. aastatel uurimustes palju levitatud, olid ka teised firmad asunud arendama oma süsteeme. Relational Software Inc., millest sai hiljem Oracle Corporation, lausa edestas IBMi tarkvaraturule sisenemisega, avaldades oma kommertstoote juba 1979. aastal. Esimene American National Standards Institute (ANSI) standard SQL kohta valmis 1986 ja sai nimeks SQL-86. (Ibid.)

Jättes kõrvale ühe väikese erandi, sai SQL relatsiooniliselt täielikuks 1992 aastal (SQL:1992), kui keelde lisati CREATE ASSERTION lause üldiste kitsenduste loomiseks. Praeguseks on standardi viimane versioon SQL:2011. (Darwen 2012 p 21)

1.2 Ülevaade

SQLi kirjeldatakse kui andmebaasikeelt või mõnikord ka kui andmete alamkeelt (*data sublanguage*), kuna rakendused on kirjutatud mõnes üldotstarbelises programmeerimiskeeles, mis kasutab konkreetseid andmebaasikeele käske. (Ibid. p 17)

SQL keelt saab jagada mitmeks alamkeeleks:

Data Manipulation Language (DML) – DML on SQL keele alamosa, mida kasutatakse andmete pärimiseks, kui ka nende lisamiseks, uuendamiseks ja kustutamiseks. DML sisaldab selliseid lauseid nagu: (Wang ja Tan 2006 p 64-65)

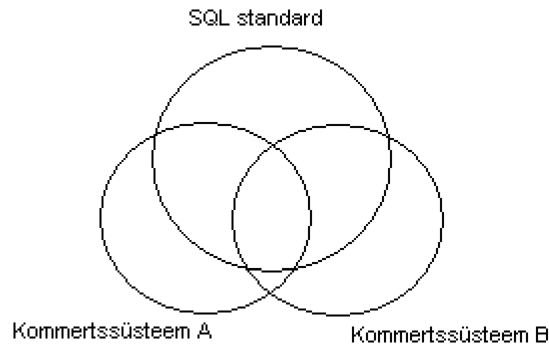
- SELECT – päringuga saada soovitava andmehulga kirjeldamine
- INSERT – tabelisse rea lisamine
- UPDATE – tabeli reas andmete muutmine
- DELETE – tabeli rea kustutamine
- MERGE – kombineerib andmete lisamise, muutmise ja kustutamise
- START TRANSACTION – andmebaasi transaktsiooni alguspunkt
- COMMIT – andmemuudatuse püsivaks muutmine
- ROLLBACK – viimase COMMIT või ROLLBACK operatsiooni tagasi keeramine

Mõnikord jaotatakse viimaseid kahte SQL lauset ka eraldi alamosaks Transaction Control Language (TCL). (Gupta ja Krishna 2013 p 56)

Data Definition Language (DDL) – DDL võimaldab kasutajale defineerida uusi tabeleid ja teisi andmebaasi elemente, neid kustutada ning mõnikord ka nende struktuuri või käitumist muuta. DDL kasutatakse andmebaasi struktuuri defineerimiseks, mille järel kasutajad saavad hakata kasutama DMLi, et tegeleda konkreetsete andmetega. DDL peamised lausete tüübid on CREATE ja DROP, mida kasutatakse vastavalt andmebaasi elementide loomiseks ja kustutamiseks. (Wang ja Tan 2006 p 64-65)

Data Control Language (DCL) – DCLi kasutatakse haldamaks kasutajate andmetele ligipääsu õiguseid. DCL laused on GRANT ja REVOKE, mida kasutatakse vastavalt kasutajale õiguste andmiseks ja ära võtmiseks. (Ibid.)

Kommertssüsteemid kasutavad mittestandardset süntaksit teatud SQL standardis kirjeldatud funktsionaalsuste realiseerimiseks ja lisaks sisaldavad ka oma laiendusi, mida standardis ei ole. Seda olukorda illustreerib joonis 1. (Halpin ja Morgan 2010 p 557)



Joonis 1 – SQL keele kattuvus SQL standardi ja kommertssüsteemide vahel

1.3 Probleemid ja kriitika

SQL keele kohta on aastate jooksul väga palju kriitikat tehtud. Vaatame järgnevalt mõningaid tähelepanuväärsemaid probleeme.

Duplikaadid - SQL standard lubab tabelitel sisaldada duplikaatridu ja ei nõua enamasti, et duplikaadid tuleks päringu tulemusest eemaldada, rikkudes sellega Coddi poolt paika pandud printsiipe. Relatsiooniliste andmebaaside spetsialist Chris Date demonstreeris kuidas 12 ühte ja sama ülesannet lahendanud SQL lauset tagastasid duplikaatsete ridade tõttu 9 erinevat tulemust. Sellel põhjusel ei saa ka optimeerimismoodul (optimeerija) sageli ümber kirjutada päringuid kiirematele alternatiivkujudele, kuna see võib mõjutada päringu tulemuses olevate duplikaatide arvu. (Fernandez 2009 p 33)

Liiasus – Keele liiasus ei ole relatsiooniliste printsiipide rikkumine, kuid tekitab optimaalsete DML lausete koostamisel tõsiseid probleeme. Tavaliselt eksisteerib sama ülesande lahendamiseks palju erinevaid viise ning päringute optimeerija ei pruugi alati leida parimat

täitmisplaani, olgugi et kõik sõnastused tagastavad sama tulemuse. (Ibid.) Liiasuse üheks põhjuseks on SQL keele pakutav võimalus alampäringute kasutamiseks. (Pascal 2013)

NULL – SQL sisaldab NULL markerit, mis tähendab teadmata väärtust. Sellega kaasnevat kolmevalentset loogikat on kritiseeritud kui ebaintuitiivset. (Fernandez 2009 p 36) NULLi ei saa lugeda otseselt väärtuseks, kuna kahe NULL väärtuse võrdluse tulemus on teadmata. Chris Date ja Hugh Darwen rõhutavad relatsioonilise andmebaasikeele põhimõtteid kirjeldavas Kolmandas Manifestis (The Third Manifesto), et andmebaasisüsteemid ei tohiks NULLide kasutamise võimalust pakkuda. (Eessaar ja Saal 2012)

Komplitseeritus – SQL on paljude väga erinevate funktsionaalsuste ühend ja tavalise rakenduse arendaja jaoks liiga keerukas. SQLi semantika mõistmine, katmaks kõiki kombinatsioone alampäringute, NULLide, trigerite, abstraktsete andmetüüpide funktsionaalsuste jms. kohta on äärmiselt problemaatiline ja SQLi õpetamine piirdub üldiselt keele tuumaga, jättes lisavõimalused elukogemuse hooleks. (Chaudhuri ja Weikum 2000)

2. SQL päringute täitmisplaanid

SQL andmekäitluskeele lause väljendab, mida kasutaja soovib, kuid ei ütle andmebaasisüsteemile kuidas soovitud tulemuseni jõuda. Näiteks, kui pärida andmebaasist töötajaid, kellel on mingi kindel palk, siis kõigepealt süsteem sõelub lause. Kui päringus süntaksivigu ja semantilisi vigu (näiteks viidatakse olematule tabelile) ei esinenud, siis asub süsteem otsustama vastuse leidmiseks parimat viisi. Andmebaasisüsteem võrdleb erinevate variantide maksumusust (hinda). Näiteks valib andmebaasisüsteem, kas lugeda kogu töötajate tabelit või kasutada soovitud andmete leidmiseks indeksit. Protseduuri, mis kirjeldab kuidas lauset füüsiliselt läbi viia, kutsutakse täitmisplaaniks (*execution plan*). Andmebaasisüsteem üritab leida parima täitmisplaani (näiteks kogu lause täitmise kiiruse mõttes). (Fraiteur 2005)

Täitmisplaan koosneb lihtoperatsioonidest, näiteks tabeli täislugemine, indeksi kasutamine, räsiühendamine jne. Andmete otsimise operatsioonid väljastavad tulemuse (ridade hulga). Operatsioonil võib sisendeid olla üks, nagu tabeli täislugemisel, või ka kaks nagu näiteks kahe tabeli räsiühendamisel. Iga lihtoperatsiooni väljund võib olla mõne teise lihtoperatsiooni sisendiks. Optimaalsete täitmisplaanide arvutamisega tegeleb andmebaasisüsteemi optimeerimismoodul (optimeerija). (Ibid.)

Täitmisplaanide lugemine on levinud ülesanne andmebaaside administraatoritele, SQL arendajatele ja süsteemide jõudluse ekspertidele, kuna need annavad detailset infot lause täitmise kiiruslike omaduste kohta. Kuigi täitmisplaane kuvatakse sageli tabelikujul siis reaalselt on nad puu kujulised. (Colgan 2008)

Järgnevalt vaatleme PostgreSQL koostatud täitmisplaani lihtsale päringule, mis küsib andmeid kõigist tabelite *personnel* ja *department* veergudest (vt Joonis 2). Tabelid ühendatakse üle võtmeveergude.

```
Hash Join  (cost=3.01..302.64 rows=50 width=628)
  Hash Cond: (personnel.depid = department.depid)
  ->  Seq Scan on personnel  (cost=0.00..262.00 rows=9900 width=100)
  ->  Hash  (cost=3.00..3.00 rows=1 width=528)
        ->  Seq Scan on department  (cost=0.00..3.00 rows=1 width=528)
```

Joonis 2 – täitmisplaani näide koos tegevustega

Vaadates joonist 2, siis punasega ja sinisega on tähistatud konkreetsed lihtoperatsioonid ja nende järjekord. Antud juhul on tegu neljasammulise täitmisplaaniga, kus alguses käiakse järjest läbi tabel *department* (Seq Scan) ja luuakse selle põhjal räsitabel (*hash table*). Seejärel käiakse järjest läbi tabel *personnel*. Lõpuks tehakse üle vastavate võtmeveergude räsiühendamine (*hash join*). Samas on antud täitmisplaanist võimalik veel informatsiooni saada.

```
Hash Join  (cost=3.01..302.64 rows=50 width=628)
  Hash Cond: (personnel.depid = department.depid)
  ->  Seq Scan on personnel  (cost=0.00..262.00 rows=9900 width=100)
  ->  Hash  (cost=3.00..3.00 rows=1 width=528)
        ->  Seq Scan on department  (cost=0.00..3.00 rows=1 width=528)
```

cost rows width

Joonis 3 – täitmisplaani näites aja ja mahuhinnangud

Vaadates joonist 3, siis värviliselt on tähistatud konkreetse operatsiooni aja ja mahuhinnangud, mille optimeerija võttis aluseks täitmisplaani koostamisel:

- *cost* – siin sisaldub kaks väärtust, esimene on käivitamise maksumus, mis antud juhul hinnatakse nulliks ning teine on kogu sammu maksumus. Maksumuse ühikuks on plokkide lugemise arv. (PostgreSQL 9.3.4 2013)
- rows – operatsiooniga tagastatavate ridade arv, antud juhul 9 900.
- width – rea laius.

Täitmisplaanid on tänapäeval vastavate volitustega kasutajale nähtavad praktiliselt kõigis peamistest SQL-andmebaasisüsteemides, kuid nende konkreetne ülesehitus varieerub. Lisaks eksisteerivad ka erinevad rakendused, mis võimaldavad näha täitmisplaane graafilisel kujul. Täitmisplaanide puukujulise ülesehituse tõttu muudab graafiline esitus keerukate päringute puhul plaanide mõistmise lihtsamaks. Täitmisplaane näeb sageli SQL lause ette konkreetse klausli lisamisega, nagu EXPLAIN PostgreSQLis või EXPLAIN PLAN FOR Oracles.

3. Pascali eksperiment

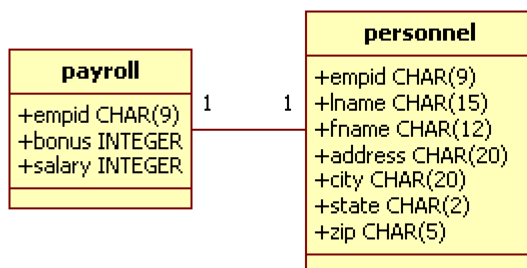
SQL keele liiasus tähendab, et praktiliselt kõiki probleeme saab lahendada paljudel eri viisidel. Fabian Pascal tegi 1988. aastal SQL-andmebaasisüsteemide töökiiruse eksperimendi, uurimaks kas SQL keele liiasus põhjustab ühesuguse tulemuse andvate kuid erineva süntaksiga lausete töökiiruses erinevusi, ning kas sellised erinevused varieeruvad andmebaasisüsteemide vahel. Erisuste esinemise korral tõstatus ka küsimus, kas kasutajale kõige lihtsamini arusaadavad, ning seega kõige tõenäolisemalt kasutatavad variandid, toimivad paremini kui vähemintuitiivsed. (Pascal 1988)

3.1 Pascali eksperiment

Pascal testis järgmiseid süsteeme:

1. Informix v. 2.10.02
2. Relational Technology Inc. Ingres v. 5.0/0.2a
3. Oracle v. 5.1A
4. Gupta Technologies SQLBase 3.4.1
5. XDB Systems XDB 2.10d

Testis kasutatud tabeliteks olid kaks üks-ühele seoses 9 900 reaga tabelit: *personnel* ja *payroll* nagu on näha joonisel 4.



Joonis 4 – andmetabelid Pascali eksperimendis

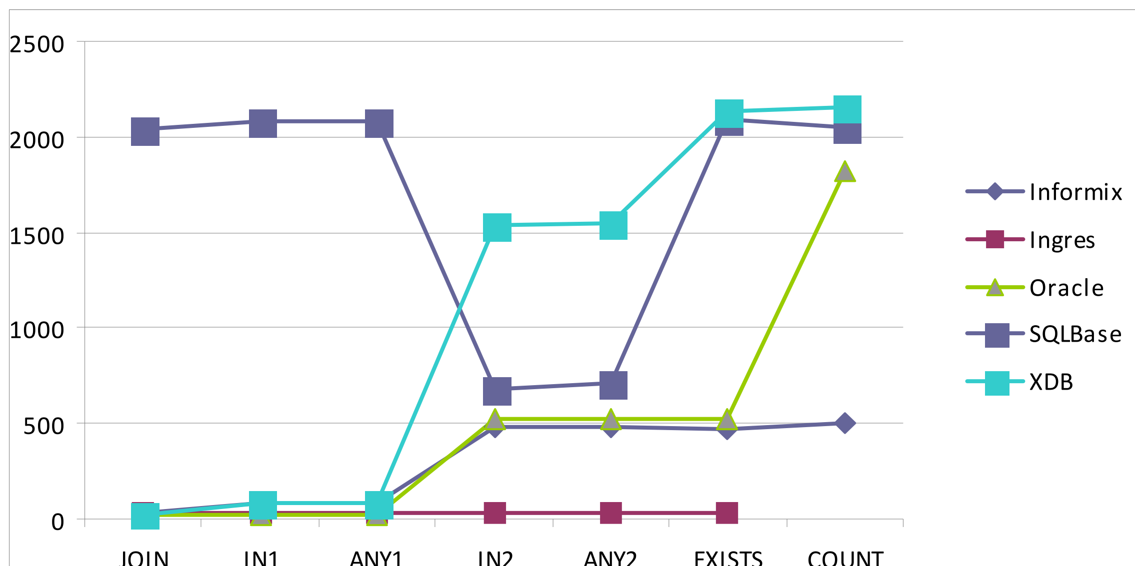
Küsimuseks, millele erinevate päringutega vastust otsiti oli: “milliste töötajate palk on 199 170?” Sellele küsimusele vastuse leidmiseks kasutas Pascal seitset erinevat päringut (SELECT lauset),

mida tähistatakse järgnevate lühikeste nimedega: JOIN, IN1, ANY1, IN2, ANY2, EXISTS, COUNT. Need päringud on näha Lisas 1.

Pascali testis käivitati päringud kolmes erinevas andmebaasi kontekstis. Esimeses olid indeksid ainult võtmeveergudel *empid*. Teisel juhul lisati ka indeks veerule *payroll.salary*. Kolmandal juhul keerati SQL lausetes ringi WHERE lause tingimuste järjekord, ehk `WHERE salary = 199170 AND payroll.empid = personnel.empid` asemel kirjutati `WHERE payroll.empid = personnel.empid AND salary = 199170` testimaks, kas tingimuste järjekord mõjutab optimeerijaid. (Ibid.)

3.2 Pascali eksperimendi tulemused

Pascali eksperimendis esinesid ühe erandiga kõigis andmebaasisüsteemides sõltuvalt päringu sõnastusest ulatuslikud erinevused (vt joonis 5) lausete täitmise kiiruses See demonstreerib probleeme, mida põhjustab SQL keele liiasus. Joonisel 5 on x teljel lausete identifikaatorid ja y teljel on päringute kestvused sekundites.



Joonis 5 – Pascali eksperimendi tulemused kui indeksid olid ainult võtmeveergudel

Üldiselt näitasid parimaid tulemusi kõige intuitiivsemad variandid ehk JOIN, IN1 ja ANY1. Kõigis süsteemides peale Ingresi olid vahed kiireima ja aeglaseima päringu vahel kümnetes kordades. Indeksite lisamine kiirendas kõigis süsteemides kõige tõenäolisemalt kasutatavate lausete täitmist, kuid paljude keerukamate lausete puhul jäi täitmise aeg samaks ning paari

SQLBase ja XDB päringu korral tõid kaasa lausa märgatava kiiruse languse. WHERE lause tingimuste järjekorra muutmine päringute optimeerijaid ei seganud ning tulemused jäid samaks. (Ibid.)

Ainus andmebaasisüsteem, mis Pascali päringus järjekindlalt samu tulemusi andis, oli Ingres. Ingresi eristas teistest toodetest selles kasutatav andmebaasikeel – QUEL. Kuna QUEL ei võimaldanud alampäringuid, siis ei kannatanud see ka liiasuse käes. Kõik Ingresi sisestatud SQL päringud teisendati samaks QUEL päringuks, mis tagas ka ühtlase käitumise. (Pascal 2013)

Pascal tõdes oma tulemuste kokkuvõttes, et probleemide põhjus oli mitte üldiselt relatsiooniliste andmebaasisüsteemide kasutamises, vaid konkreetselt SQL keele liiasuses, mis raskendab andmebaasisüsteemide tootjate tööd optimeerimislahenduste leidmisel. Samas näitasid Ingresi ühetaolised tulemused, et vastav optimeerimine pole oma olemuselt võimatu ülesanne. Positiivse külje pealt tõi ta esile, et kõige paremini toimisid kõige intuitiivsemad päringud, mida kasutaja oleks kõige tõenäolisemalt antud probleemi lahendamiseks kasutanud. (Pascal 1988)

Pascali eksperimendi ajal puudus enamikel PC andmebaasisüsteemidel funktsionaalsus, mis oleks võimaldanud näha ja analüüsida konkreetsete päringute täitmisplaane ning põhjalikumat probleemide analüüsi. (Ibid.)

3.3 Riina Kivi korduseksperiment

Tallinna Tehnikaülikooli magistrant Riina Kivi kordas ja laiendas Pascali testi 2008. aastal. R. Kivi testis järgmisi süsteeme:

1. Oracle 10g
2. MS SQL 2005
3. MS Access v. 11
4. LucidDB 0.7.4

Kivi kasutas sama struktuuriga tabeleid mis Pascal, ning viis eksperimendi läbi nii sama andmekogusega 9 900 realises andmebaasis, kui ka suuremas 989 954 realiste tabelitega andmebaasis. Lisaks seitsmele Pascali kasutatud ülesande lahendusele lisas Kivi ka kolm uut lahendust: INNER JOIN, SOME1 ja SOME2, mis on ka Lisas 1. (Kivi 2008)

Korduseksperimendis ei leidnud Kivi arvestavaid erinevusi päringute täitmisekiirustes andmebaasisüsteemides Oracle, MS SQL ja MS Access. Ainult veerupõhises andmebaasisüsteemis LucidDB esines lahenduse sõnastusest tulenevalt tõsisemaid erinevusi, ja isegi WHERE tingimuste järjekorrast põhjustatud kiiruse muutusi. (Ibid.) Samas tuleb arvestada et Kivi töö keskendus töökiiruste võrdlustele ja päringute täitmisplaane käsitleti väga lühidalt.

3.4 Muud analoogsed eksperimendid

Iggy Fernandez NoCOUGist (Northern California Oracle Users Group) kordas Pascali testi 2006. aastal teadmata Oracle versioonil (eeldatavasti 10g) ja täheldas tulemustes arvestatavaid erinevusi. JOIN lahendus osutus kordades efektiivsemaks kui muud variandid. (Fernandez 2006) 2011. aastal tegi Fernandez analoogse testi Oracle v. 11g Release 2 versioonil, kus ta kasutas osasid Pascali päringuid kui ka mõningaid muid variante. Antud testis pööras ta rohkem tähelepanu täitmisplaanidele, täheldades et päringute optimeerija otsustes esines erinevusi sõltuvalt päringute sõnastusest. (Fernandez 2011)

4. Eksperiment

4.1 Eksperimendi eesmärgid

Antud töös on eesmärgiks varasemaid eksperimente (Pascal 1988 ja Kivi 2008) laiendada, suurendades testandmebaasi nii mahuliselt kui tabelite osas ja lisades uusi võrreldavaid lauseid. Võrdluses on kavas varasemast sügavamalt käsitleda andmebaasisüsteemide kasutatavaid täitmisplaane, saamaks täpselt aru, millest tulenevad kiiruslikud erinevused lausete täitmisel. Kuna töös proovitakse ühesuguseid päringuid, sarnaste andmete põhjal ühes ja samas serverarvutis, siis annab see võimaluse töö kitsas vaates teha tähelepanekuid erinevate andmebaasisüsteemide osas (PostgreSQL ja Oracle). Kuna töös eksperimenteeritakse sama andmebaasisüsteemi erinevate versioonidega, siis annab see võimaluse töö kitsas vaates teha tähelepanekuid andmebaasisüsteemi optimeerimismooduli (optimeerija) evolutsiooni osas.

Põhiküsimuseks on kuidas kirjutamise kujult erinevad, kuid sisuliselt sama vastusega päringute käitumine erineb erinevates andmebaasisüsteemides. Lisaks samasisuliste päringute toimimise varieerumise uurimisele ühe andmebaasisüsteemi korral, on kavas uurida siinjuures veel nelja aspekti.

1. Erinevused käitumises andmete mahu kasvades. Siinkohal on võetud üheks võrdluse punktiks 9 900 reaga andmebaas, mis mahult vastab Fabian Pascali poolt 1988. aasta eksperimendis kasutatule ning teiseks sama andmebaas, kus ridade arvuks suurimates tabelites on 20 miljonit.

2. Erinevused kommertsandmebaasisüsteemi Oracle ja avatud lähtekoodiga ja tasuta PostgreSQL vahel. Mõlemad süsteemid on maailmas praegusel ajahetkel laialt kasutusel. (DB-Engines Ranking 2014) Tuleb arvestada siin peetakse silmas võrdlust kui edukalt suudavad antud andmebaasisüsteemid erineva sõnastusega samasisulisi päringuid optimeerida ja tegu ei ole PostgreSQL ja Oracle üldise töökiiruse võrdlusega.

3. Erinevused sama andmebaasisüsteemi erinevate versioonide vahel. Võrdluseks kasutatakse PostgreSQL 9.1 ja PostgreSQL 9.3 ning Oracle 11g Enterprise Edition Release 1 ja Oracle 12c Enterprise Edition Release 1. Oracle Enterprise Edition on erinevatest Oracle andmebaasisüsteemide väljaannetest kõige võimekam ja võimalusterohkem.

4. Erinevused sõltuvalt veergude indekseerituse astmest. Võrreldakse päringuid oludes, kus kasutaja loodud indekseid ei ole, olukorraga kus veergudele, kust toimub andmete otsing, on samuti lisatud indeksid. Esimesel juhul saab süsteem kasutada ainult automaatselt primaarvõtmetele loodavaid indekseid, teisel juhul on valik suurem

PostgreSQL ja Oracle valiti eksperimendis uuritavateks andmebaasisüsteemideks kuna tegemist on populaarsete süsteemidega (DB-Engines Ranking 2014) (2014. aasta mai seisuga on Oracle ja PostgreSQL selles pingereas vastavalt esimesel ja neljandal kohal), need olid kättesaadavad Tallinna Tehnikaülikooli serveritel ja autor on nendega varasemalt kokku puutunud.

4.2 Kasutatavad andmebaasid

Eksperimendis kasutatakse andmebaasisüsteeme PostgreSQL ja Oracle.

4.2.1 PostgreSQL

PostgreSQL on laialt levinud avatud lähtekoodiga ja tasuta pakutav objekt-relatsiooniline andmebaasisüsteem. PostgreSQL toetab paljusid tänapäevaseid funktsionaalsusi nagu keerukate päringute tegemise võimalus, deklaratiivsed kitsendused, trigerid, uuendatavad vaated, transaktsioonide terviklikkus ja multiversioon-konkurentsjuhtimist. Lisaks võimaldab PostgreSQL mitmesugust kasutaja poolset laiendamist, näiteks uute andmetüüpide või agregaatfunktsioonide defineerimist, PL/pgSQL või mõne muu keele abil kasutaja-defineeritud funktsioonide loomine ja toetab enamikke ISO/IEC 9075:2011 ehk SQL:2011 standardi nõudeid. (PostgreSQL 9.3.4 2013)

1970-ndate lõpus hakati Berkeley ülikoolis arendama relatsioonilist andmebaasisüsteemi Ingres, mis oli PostgreSQLi eelkäiaks. 1986. aastal lisas Michael Stonebrakeri juhitav meeskond Berkeleys Ingresile objektorienteeritud omadusi ning uus versioon sai tuntuks kui POSTGRES. (Douglas 2003, p 1) 1994. aastal lisasid Andrew Yu ja Jolly Chen Postgresile SQL keele toe varasema PostQUEL asemel. Antud versioon sai tuntuks kui Postgres95, 1996 muudeti nimi PostgreSQL-iks. Mitteametlikult kasutatakse endiselt laialdaselt nime Postgres. (PostgreSQL 9.3.4 2013)

4.2.2 Oracle

Oracle on laialdaselt kasutatav kommertsandmebaasisüsteem mida arendab Oracle Corporation. Oracle toetab enamike samu funktsionaalsusi mis PostgreSQL kuid pakub loomulikult ka hulgaliselt erinevat funktsionaalsust. (Introduction to Oracle… 2009)

1977. aastal asutasid Larry Ellison, Bob Miner ja Ed Oates firma Software Development Laboratories, millest hiljem sai Oracle Corporation. Aastal 1979 tuli välja Oracle V2, mis oli esimene kaubanduslikult saadaval olev SQL-põhine relatsiooniline andmebaasisüsteem (RDBMS). Oracle on mõningatel andmetel hetkel maailma kõige populaarsem andmebaasisüsteem. (DB-Engines Ranking 2014)

4.3 Tehnilised andmed

Test viiakse läbi neljas andmebaasisüsteemis: PostgreSQL 9.1.4 ja PostgreSQL 9.3.0 ning Oracle 11g Enterprise Edition Release 1 ja Oracle 12c Enterprise Edition Release 1. PostgreSQLis päringute käivitamiseks, nende täitmiseks kulunud aja mõõtmiseks ja täitmisplaanide vaatamiseks kasutatakse graafilist kasutajaliidest pgAdmin. Oracle andmebaasis kasutatakse selleks Oracle Application Expressi, graafilisel kujul esitatud ja seega paremini loetavate täitmisplaanide vaatamiseks kasutatakse ka Oracle SQL Developerit.

Testide läbiviimiseks kasutatakse Tallinna Tehnikaülikooli servereid hektor8.ttu.ee (edaspidi *hektor8*) ja apex.ttu.ee (edaspidi *apex*):

hektor8 – siin tehakse katseid PostgreSQL 9.1.4 ja Oracle 11g andmebaasisüsteemides loodud andmebaaside põhjal. Tehnilised andmed: füüsiline server, 6 GB RAM, 500 GB SATA HDD, Intel(R) Xeon(R) CPU X3210 @ 2.13GHz , 4-core, CentOS 5.7.

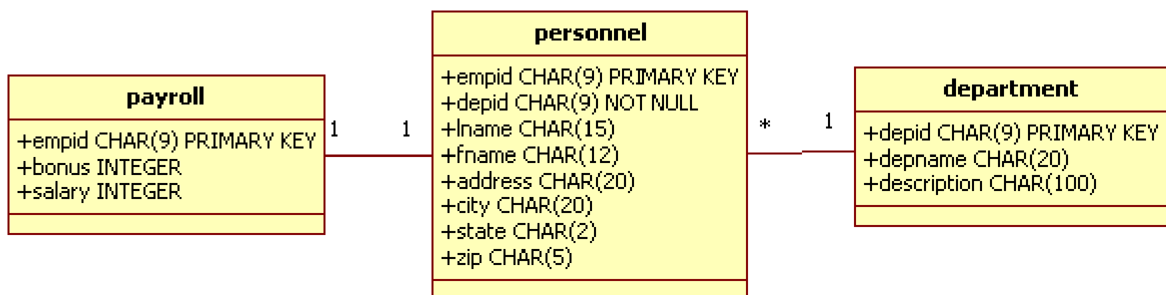
apex – siin tehakse katseid PostgreSQL 9.3.0 ja Oracle 12c andmebaasisüsteemides loodud andmebaaside põhjal. Tehnilised andmed: virtuaalmasin Dell R820 serveris, virtualiseerimise tarkvaraks KVM, 15 CPU-d, 40 GB RAM, 840 GB HDD (10K SATA), CentOS 6.

4.4 Testandmebaasid ja testandmed

Säilitamaks järjepidevust Pascali 1988. aasta eksperimendiga on algsete tabelite struktuur jäetud sisuliselt samaks. Lisaks algsele kahele tabelile *personnel* ja *payroll,* mis olid üks-ühele seoses, on nüüd lisatud ka tabel *department,* mis on tabeliga *personnel* seoses üks-mitmele (iga töötaja töötab ühes osakonnas; igas osakonnas töötab null või rohkem töötajat). Tabelisse *department* lisatakse kõigi testide ajal 50 rida andmeid.

Tabelis *department* on kolm tekstitüüpi veergu:
depid CHAR(9) mis on ka primaarvõtmeks ja seega kohustuslik
depname CHAR(20)
description CHAR(100)

Veerg *depid* on lisatud ka tabelisse *personnel* koos NOT NULL kitsendusega, võimaldamaks ühendada selles olevad andmed tabelis *department* olevate andmetega. Eksperimendis kasutatavad tabelid on näha joonisel 6.



Joonis 6 – eksperimendis kasutatavad andmebaasitabelid

Primaarvõtme deklaratsiooni alusel loovad PostgreSQL ja Oracle automaatselt primaarvõtme veergudele indeksi. Testis on plaanis käivitada samu päringuid nii andmebaasis, kus kasutaja loodud indekseid ei ole kui ka baasis, kus on loodud indeks veerule *salary* ja liitindeks üle veergude *salary* ja *bonus*:

`payroll_salary_index` – indeks veerul *payroll.salary*

`payroll_multi_index` – liitindeks mis ühendab veerge *payroll.salary* ja *payroll.bonus*

Tabelite ja indeksite loomise SQL laused on ära toodu Lisas 2.

Testandmete loomiseks kasutatakse suvaväärtuste generaatoreid. Testandmete genereerimise valemite loomisel oli peamiseks põhimõtteks, et iga uuritav päring peab tagastama vähemalt mõned väärtused, kuid samas peab varieeruvus olema piisav, et laiendatud päring ei tagastaks kõiki 50 tabeli *department* rida. Eelneva tagamiseks vähendati 9 900 rea loomise SQL lausetes veerus *salary* olevate andmete varieeruvust, võrreldes lausetega, millega loodi 20 miljonit rida.

payroll.salary 9 900 rida:
199170 + (R * 1000 - 100000)
, kus R on täisarvuline juhuarv vahemikus 0 kuni 199 (otspunktid kaasa arvatud).
Tulemuseks on juhuarvud vahemikus 99170 – 298170 (otspunktid kaasa arvatud) sammuga 1000, ehk siis 200 võimalikku erinevat väärtust.

payroll.salary 20 miljoni rida:
(R + 100000) *10
, kus R on täisarvuline juhuarv vahemikus 0 kuni 19 999 (otspunktid kaasa arvatud).
Tulemuseks on juhuarvud vahemikus 100 000 – 299 990 (otspunktid kaasa arvatud) sammuga 10, ehk siis 20 000 erinevat võimalikku väärtust.

payroll.bonus on juhuarvud 0 kuni 9 000 (otspunktid kaasa arvatud) sammuga 1000, ehk siis 10 võimalikku erinevat väärtust.

Võtmeveerud *empid* ja *depid* täideti andmete sisestamisel järjest kasvavate täisarvudega. Muud tekstilised veerud, mis päringutes otseselt kasutust ei leia, täideti juhutekstiga. Iga andmebaasisüsteemi jaoks genereeriti testandmed eraldi, kasutades samaväärseid algoritme. Testandmete loomise SQL laused on Lisas 3. Eksperimendi käigus ei sisestatud NULLe ühtegi välja.

Tulenevalt töö autori veast võeti statistika uuendamine kasutuse alles poole eksperimendi pealt. Seda viga tasuks analoogsete eksperimentide korral silmas pidada ja vältida. Seetõttu tehti 20 miljoni reaga tabelites tagantjärele kordusmõõtmisi, veendumaks, et täpsem statistika probleemsemate päringute optimeerimist ei mõjuta. Kontroll näitas, et täpsem statistika antud eksperimendis probleemseid täitmisplaane ei parandanud. Statistika uuendamiseks kasutati peale testandmete genereerimist Oracles DBMS_STATS.GATHER_TABLE_STATS protseduuri ja PostgreSQLis VACUUM ANALYZE lauset.

4.5 Päringud

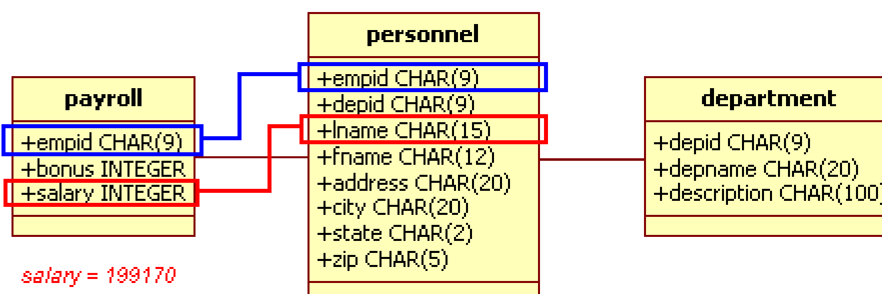
Eksperimendi SQL lausetega otsitakse vastuseid kolmele küsimusele.

Millistel töötajatel on palk 199170? (edaspidi “originaalpäring”) – antud päringut kasutas Pascal oma 1988. aasta testis, kus ta kasutas selle testimiseks seitsmel eri moel kirjutatud päringuid: JOIN, IN1, ANY1, IN2, ANY2, EXISTS ja COUNT. Riina Kivi lisas 2008. aasta testis veel kolm päringut: SOME1, SOME2 ja INNER JOIN. Nüüdses eksperimendis sai lisatud veel kaks variatsiooni: FULL JOIN ja NATURAL JOIN. Näide originaalpäringust NATURAL JOIN:

```
SELECT lname
FROM personnel NATURAL JOIN payroll
WHERE salary = 199170;
```

Päringu kohta võib öelda kriitikana, et mitmel erineval inimesel võib olla samasugune perenimi, kuid päringu tulemuses ei ole sellised inimesed üksteisest eristatavad. Kuna SQL ei eemalda üldjuhul päringu tulemusest korduvaid ridu, siis võib selle päringu tulemuses olla korduvaid ridu (kuna mitmel inimesel võib olla sama perenimi). Kasutasime selliseid päringuid järjepidevuse hoidmiseks Pascali 1988. aasta uuringuga.

Kõik originaalpäringu variandid on Lisas 1. Seega kokku tuli selle küsimuse kohta 12 eri viisil kirjutatud päringut. Antud päring kasutab tabeleid *personnel* ja *payroll*, mis on ühendatavad üle veeru *empid*. Sisuliselt küsitakse, millised *personnel.lname* väärtused vastavad *payroll.salary = 199170* väärtusele. Päringus kasutatavad veerud on näha joonisel 7.



Joonis 7 – originaalpäringus kasutatavad veerud

Milliste osakondades töötavad isikud, kellel on väikseim isiklik palk ja samaaegselt boonus 6000? (edaspidi "laiendatud päring") – antud päring on sisuliselt eelneva edasiarendus, tõstes päringu keerukust. Seda on samamoodi 12 eri variatsiooni, mis küll kindlasti ei ole lõplik nimekiri võimalikke variante, aga antud eksperimendi eesmärgi saavutamiseks peaks olema piisav. Järgnevalt paar laiendatud päringu varianti.

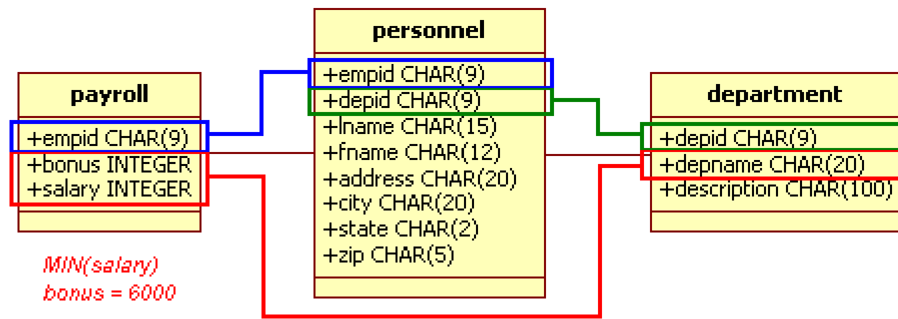
JOIN:

```
SELECT depname
FROM department, personnel, payroll
WHERE personnel.empid = payroll.empid
AND personnel.depid = department.depid
AND salary = (
     SELECT MIN(salary)
     FROM payroll)
AND bonus = 6000
GROUP BY department.depid;
```

EXISTS:

```
SELECT depname
FROM department
WHERE EXISTS (
     SELECT *
     FROM personnel
     WHERE department.depid = personnel.depid
     AND EXISTS (
          SELECT *
          FROM payroll
          WHERE personnel.empid = payroll.empid
          AND salary = (
               SELECT MIN(salary)
               FROM payroll)
          AND bonus = 6000));
```

Kõik laiendatud päringu variandid on Lisas 4. Antud päring kasutab tabeleid *personnel*, *payroll* ja *department*. Kahe esimese tabeli seos jääb samaks ning *personnel* ja *department* on ühendatavad üle veeru *depid*. Sisuliselt küsitakse, millised *department.depname* väärtused vastavad samaaegselt *payroll.bonus = 6000* ja *payroll.salary* miinimumväärtusele. Päringus kasutatavad veerud on näha joonisel 8.



Joonis 8 – laiendatud päringus kasutatavad veerud

Mis on mediaanpalk? – päritakse välja *payroll.salary* mediaanväärtust. Oracles on selle jaoks sisseehitatud (süsteemi-defineeritud) funktsioon MEDIAN(). Samas PostgreSQLis ei ole analoogset sisseehitatud funktsiooni ja lahendus tuleb kasutajal endal luua. Antud eksperimendis proovitakse kolme erinevat PostgreSQL mediaani leidmise päringut. Päringute loomisel võeti aluseks erinevad Internetist leitud variandid (Brandstetter 2011, DeBarros 2010 ja Rasheed 2010), mida kohandati vastavaks konkreetse eksperimendi vajadustele. Kahtlemata pole tegu kõigi võimalike variantidega ning arvestatava tõenäosusega leidub ka kiiremaid variante, aga samas teatava esmase ülevaate kuidas kasutaja valikud mõjutava sellise keerukama päringu efektiivsust peaks olema võimalik saada. Järgnevalt näide mediaani päringust MEDIAN3:

```
SELECT CASE WHEN c%2 = 0 AND c > 1
THEN CAST((salary[1]+salary[2]) AS FLOAT)/2
ELSE salary[1] END
     FROM (
          SELECT ARRAY(
               SELECT salary
               FROM payroll
               WHERE  salary IS NOT NULL
               ORDER BY salary
               OFFSET (c-1)/2
               LIMIT 2)
          AS salary, c
          FROM (
               SELECT count(*) AS c
               FROM payroll
               WHERE  salary IS NOT NULL)
          AS count
      OFFSET 0)
AS midrows;
```

Kõik mediaani päringud on Lisas 5.

5. Tulemused

5.1 Pascali originaalpäring

Pascali originaalpäring “millistel töötajatel on palk 199170”, jooksis 9 900 reaga andmebaasides äärmiselt kiiresti, suurimaks ajakuluks tuli 0,12 sekundit. Niivõrd väikeste väärtuste juures pole võimalik ajaliste tulemuste võrdlusest mingeid tõsiseltvõetavaid järeldusi teha ning seetõttu selle katse tulemusi eraldi välja ei tooda. Samas 20 miljoni reaga tabelites hakkavad ilmnema selged erisused. Kõigepealt vaatleme ilma indeksiteta varianti.

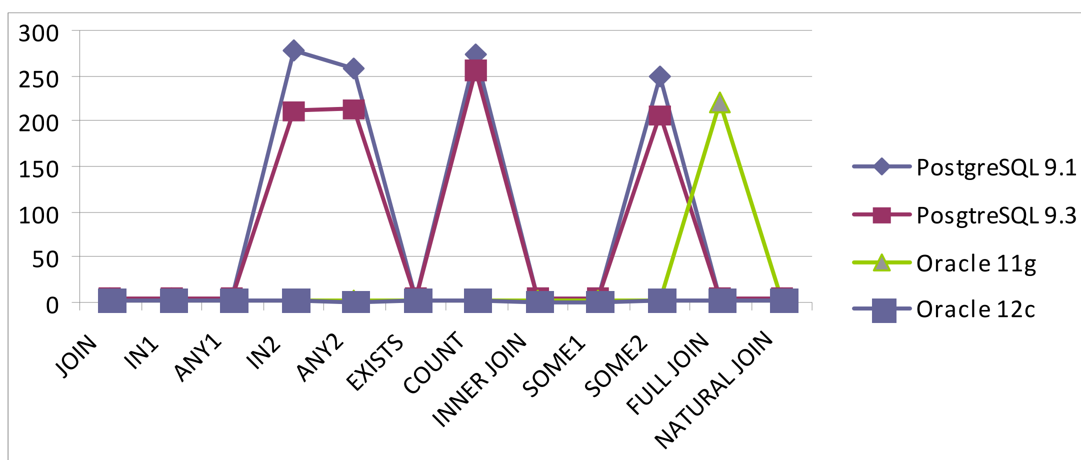
Tabelis 1 on iseloomustatud lausete täitmiseks kasutatud täitmisplaane. Iga arv tähendab erinevat täitmisplaani, sama süsteemi kasutatakse ka järgmistes täitmisplaanide tabelites. Kui mitmes väljas on ühesugune arv, siis järelikult kasutati kõigil nendel juhtudel andmebaasisüsteemide poolt sama täitmisplaani. PostgreSQLi mõlemad versioonid kasutavad viite erinevat täitmisplaani, kuid samas üleminek 9.1 versioonilt 9.3 versioonile omas suhteliselt väikest mõju. Oracle 11g kasutas ainult kahte ja Oracle 12c kolme erinevat täitmisplaani. Tabelis 2 ja joonisel 9 on näha päringute kiirused. PostgreSQLis osutusid aeglasteks IN2, ANY2, SOME2 ja COUNT. Oracle 11g osutus aeglaseks FULL JOIN. Muude täitmisplaanide vahel olid ajalised erinevused praktiliselt olematud.

Tabel 1 – erinevad täitmisplaanid originaalpäringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	1	2	2	3	3	2	4	1	2	3	5	1
Posgtres 9.3	1	1	1	3	3	1	4	1	1	3	5	1
Oracle 11g	6	6	6	6	6	6	6	6	6	6	7	6
Oracle 12c	8	8	8	8	8	9	9	8	8	8	10	8

Tabel 2 – päringute kestvused sekundites originaalpäringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	3,67	3,69	3,61	278,72	258,64	3,61	273,61	3,88	3,66	248,42	3,54	3,53
Posgtres 9.3	3,84	3,93	4,41	211,17	213,67	4,73	254,91	4,01	4,01	207,91	4,60	3,92
Oracle 11g	1,69	1,70	1,71	1,73	1,73	1,95	1,73	1,71	1,87	1,81	219,77	1,90
Oracle 12c	1,15	1,16	1,22	1,34	1,03	1,37	1,23	1,08	1,05	1,21	1,29	1,13



Joonis 9 – originaalpäringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseerimata tabelis, x-teljel on päringute identifikaatorid ja y-teljel nende kestvus sekundites

Tabeli 2 puhul tuleb öelda, et kiiruseid sama andmebaasisüsteemi erinevate versioonide vahel ei saa rangelt võttes võrrelda kuna need olid erinevatel serveritel ja ka andmeväärtused tabelites olid erinevad. Siiski on huvitav märkida, et kuigi PostgreSQL 9.3 on paremate tehniliste omadustega serveril kui PostgreSQL 9.1 on ilma kasutaja loodud indeksiteta tabelite korral mitmel juhul töökiirus isegi langenud. Oracle puhul on olukord vastupidine ning uuem versioon paremal riistvaral annab ka veidi parema töökiiruse.

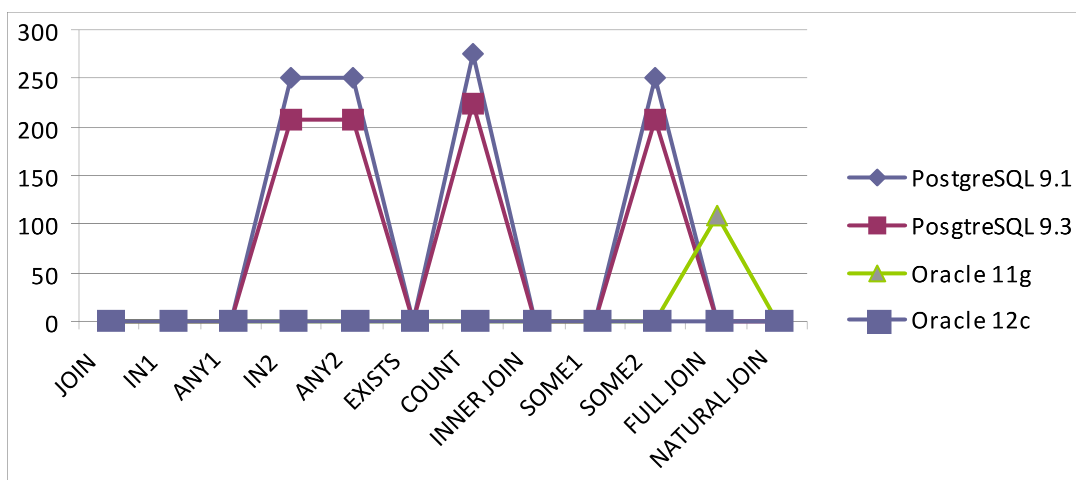
Järgnevalt vaatame sama päringut indekseeritud tabelites.

Tabel 3 – erinevad täitmisplaanid originaalpäringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	1	2	2	3	3	2	4	1	2	3	5	1
Posgtres 9.3	1	1	1	3	3	1	4	1	1	3	5	1
Oracle 11g	6	6	6	6	6	7	7	6	6	6	8	6
Oracle 12c	9	9	9	9	9	10	10	9	9	9	11	9

Tabel 4 – päringute kestvused sekundites originaalpäringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	0,13	0,13	0,13	251,15	251,30	0,15	275,09	0,13	0,13	250,61	0,14	0,14
Posgtres 9.3	0,10	0,10	0,10	207,79	207,83	0,14	224,81	0,15	0,09	208,13	0,14	0,09
Oracle 11g	0,16	0,16	0,17	0,16	0,16	0,14	0,15	0,16	0,17	0,10	108,87	0,16
Oracle 12c	0,12	0,11	0,12	0,13	0,12	0,12	0,13	0,13	0,12	0,12	0,12	0,12



Joonis 10 – originaalpäringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseeritud tabelis

Tabelis 3 on näha kasutatud täitmisplaanid. PostgreSQLi päringute optimeerija toimis täpselt analoogselt varasemaga kasutades jälle viit erinevat täitmisplaani. Oracle mõlemad versioonid kasutasid kolme erinevat täitmisplaani. PostgreSQL ja Oracle täitmisplaanide seas leidus suhteliselt sarnaseid variante, kuid otseselt identseid täitmisplaane ei ilmnenud. Tabelis 4 ja joonisel 10 on näha päringute kiirused. Probleemsed päringud jäid samaks, kuid kui PostgreSQL probleemsete päringute kiiruses ei olnud arvestatavaid muudatusi siis Orace 11g FULL JOIN muutus arvestatavalt kiiremaks. Kõik hästi optimeeritud päringud võitsid märgatavalt indeksite olemasolust. Lisaks on tabelitest 2 ja 4 näha et Oracle andmebaasid ilma indeksiteta tabelites näitasid paremaid tulemusi kui PostgreSQL andmebaasid, aga indeksite lisamise järel kiirused võrdsustusid.

5.1.1 PostgreSQL

PostrgeSQLi süsteemides võib antud päringud jagada kaheks grupiks.

1. Aeglane grupp: IN2, ANY2, SOME2 + COUNT
2. Põhigrupp kus on kõik ülejäänud päringud

Aeglase grupi päringud kasutavad kogu aeg samu täitmisplaane, sõltumata tabeli mahust või kasutaja lisatud indeksite olemasolust. Järgnevalt on toodud sellise plaani näidis (vt joonis 11):

```
Seq Scan on personnel  (cost=0.00..136899123.63 rows=10000004 width=16)
        Filter: (SubPlan 1)
        SubPlan 1
            ->  Index Scan using payroll_pkey on payroll  (cost=0.00..13.63
            rows=1 width=4)
                  Index Cond: (personnel.empid = empid)
```

Joonis 11 – originaalpäringu täitmisplaan kõigil juhtudel IN2, ANY2, SOME2 korral PostgreSQL andmebaasisüsteemis

Eeltoodu plaani vaadates on probleemiks ilmselt andmebaasisüsteemi optimeerimismooduli (optimeerija) võimetus leida alternatiivi alapäringu *SubPlan 1* eraldi käivitamisele kõigi ridade korral, millest tulenevalt andmekoguste kasvades kasvab järjest ka päringule kuluv aeg. Seetõttu osutus ka indeks veerul *salary* kasutuks, kuna seda päringu täitmiseks ei kasutatud. Samas võrdluses põhigrupi täitmisplaanid näitavad selgelt arengut muutudes nii indeksite lisamisel kui ka andmehulga suurenemisel. Järgnevalt kaks näidet.

```
Hash Join  (cost=188.36..487.98 rows=49 width=16)
        Hash Cond: (personnel.empid = payroll.empid)
        ->  Seq Scan on personnel  (cost=0.00..262.00 rows=9900 width=26)
        ->  Hash  (cost=187.75..187.75 rows=49 width=10)
                ->  Seq Scan on payroll  (cost=0.00..187.75 rows=49 width=10)
                        Filter: (salary = 199170)
```

Joonis 12 – originaalpäringu täitmisplaan PostgreSQL põhigrupi päringutes kui tabelis on 9 900 rida ilma kasutaja poolt loodud indeksiteta

Joonisel 12 on suhteliselt lihtne täitmisplaan. Käiakse läbi *payroll* vastavalt tingimusele ning lisatakse kirjed räsitabelisse (*hash*) mis seejärel räsiühendatakse vastavate *personnel* tabeli ridadega.

```
Nested Loop  (cost=22.12..17211.56 rows=990 width=16)
        ->  Bitmap Heap Scan on payroll  (cost=22.12..3721.86 rows=990
        width=10)
                Recheck Cond: (salary = 199170)
                ->  Bitmap Index Scan on payroll_multi_index
                (cost=0.00..21.87 rows=990 width=0)
                        Index Cond: (salary = 199170)
        ->  Index Scan using personnel_pkey on personnel  (cost=0.00..13.61
        rows=1 width=26)
             Index Cond: (empid = payroll.empid)
```

Joonis 13 – originaalpäringu täitmisplaan PostgreSQL põhigrupi päringutes kui tabelis on 20 miljonit rida koos kasutaja poolt loodud indeksitega

Joonisel 13 on näha kuidas täitmisplaan kasutab *salary* veerul operatsiooni *Bitmap Heap Scan – Recheck Cond – Bitmap Index Scan*. Sisuliselt tavalises indeksite skaneerimises andmebaasisüsteem peale indeksist rea viite saamist kohe loeb antud rida, aga *Bitmap Scan* korral alguses kogutakse kokku ja sorteeritakse kõik ridade viited kasutades mälus olevat "bitmap" andmestruktuuri, ja seejärel läbitakse read vastavalt nende füüsilisele asukohale. (Winand 2011)

5.1.2 Oracle

Oracle süsteemis oli optimeerija originaalpäringus edukam kasutaja ebatraditsiooniliste otsuste silumisel ning ainsaks päringuks, mida ei suudetud teistega samaväärseks muuta, oli FULL JOIN. 9 900 reaga tabelite korral ei võetud indeksit veerul *salary* üldse kasutusele. Samas 20 miljoni reaga tabelis need olid muidugi kasutusel nagu on näha järgnevas täitmisplaanis.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		989	2966
NESTED LOOPS			
NESTED LOOPS		989	2966
TABLE ACCESS (BY INDEX ROWID)	PAYROLL	989	987
INDEX (RANGE SCAN)	PAYROLL_SALARY_INDEX	989	5
Access Predicates			
SALARY=199170			
INDEX (UNIQUE SCAN)	PERSONNEL_PK	1	1
Access Predicates			
PERSONNEL.EMPID=PAYROLL.EMPID			
TABLE ACCESS (BY INDEX ROWID)	PERSONNEL	1	2

Joonis 14 – originaalpäringu täitmisplaan enamikes Oracle 11g päringutes 20 miljoni reaga kasutaja poolt indekseeritud tabelil

Nagu jooniselt 14 on näha, siis andmebaas ühelegi tabelile täisskaneeringut ei teinud, vaid kasutas ära indekseid, sealhulgas seda mis lisati kasutaja poolt veerule *salary*. Lisaks tasub antud täitmisplaanis märkimist topelt NESTED LOOPS. Siinkohal on tegu Oracle 11 versioonist alates kasutusele võetud meetodiga, kus Oracle mitte ei ühenda kahte tabelit otse, vaid alguses sisemises tsüklis ühendab tabeli *personnel* tabeli *payroll* indeksiga ning välises tsüklis ühendab selle põhjal ülejäänud *payroll* tabeli. (Van Dyke 2012)

Oracle 11g puhul osutus selgelt probleemiks FULL JOIN operatsioon, mida kasutav lause jäi üle saja korra aeglasemaks kui muud Oracle päringud. Samas erinevalt PostgreSQLi probleemsetest päringutest, suutis Oracle FULL JOIN indeksite lisamisel neid kasutada ning päringu kestvust umbes poole võrra lühendada.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		20000000	159050
VIEW	VW_FOJ_0	20000000	159050
Filter Predicates			
SALARY=199170			
HASH JOIN (FULL OUTER)		20000000	159050
Access Predicates			
PERSONNEL.EMPID=PAYROLL.EMPID			
TABLE ACCESS (FULL)	PAYROLL	20000000	17305
TABLE ACCESS (FULL)	PERSONNEL	20000000	79948

Joonis 15 – originaalpäringu täitmisplaan FULL JOIN korral Oracle 11g päringutes 20 miljoni reaga tabelis

Vaadates joonist 15, skaneerib andmebaasisüsteem mõlemat tabelit tervenisti, teostab räsiühendamise ja loob selle põhjal vaate *"SALARY" = 199170*. Kahe 20 miljoni realise tabeli ühendamine on ilmselgelt äärmiselt ebaefektiivne lähenemine. Oracle 12c on juba veidi teistsugune lähenemine, mis originaalpäringu puhul oli ajaliselt praktiliselt võrdne ülejäänud tabelites kasutatava lähenemisega.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		989	2966
HASH JOIN (OUTER)		989	2966
Access Predicates			
PERSONNEL.EMPID(+)=PAYROLL.EMPID			
NESTED LOOPS (OUTER)		989	2966
STATISTICS COLLECTOR			
TABLE ACCESS (BY INDEX ROWID BATCHED)	PAYROLL	989	987
INDEX (RANGE SCAN)	PAYROLL_SALARY_INDEX	989	5
Access Predicates			
PAYROLL.SALARY=199170			
TABLE ACCESS (BY INDEX ROWID)	PERSONNEL	1	2
INDEX (UNIQUE SCAN)	PERSONNEL_PK	1	1
Access Predicates			
PERSONNEL.EMPID(+)=PAYROLL.EMPID			
TABLE ACCESS (FULL)	PERSONNEL	1	2

Joonis 16 – originaalpäringu täitmisplaan FULL JOIN korral Oracle 12c päringutes 20 miljoni reaga kasutaja poolt indekseeritud tabelis

Joonisel 16 on näha selgelt paremat ühendamise strateegiat, mis kasutab ära ka indekseid ja enne ühendamist otsib välja tingimustele vastavad tabeli *payroll* read. Lisaks tasub Oracle 12c päringu puhul pöörata tähelepanu ka reale STATISTICS COLLECTOR. Nimelt on antud Oracle versioonist alates päringute optimeerijal võime täitmisplaani lause täitmise käigus muuta. Ei saa küll tervet plaani poole kasutamise pealt ümber vahetada, aga teatud lokaalseid otsuseid saab kohandada vastavalt päringu täitmise käigus kogutud statistikale. Täitmisplaani koostades tekitab optimeerija nii põhiplaani, kui ka alternatiivvariandid, mis algavad statistika kogumise punktist. Selline strateegia tähendab, et kui optimeerija algsed eeldused osutusid täiesti valeks, siis on statistika kogumise punkti järel võimalik minna üle alternatiivplaanile. (Colgan 2013)

5.2 Laiendatud päring

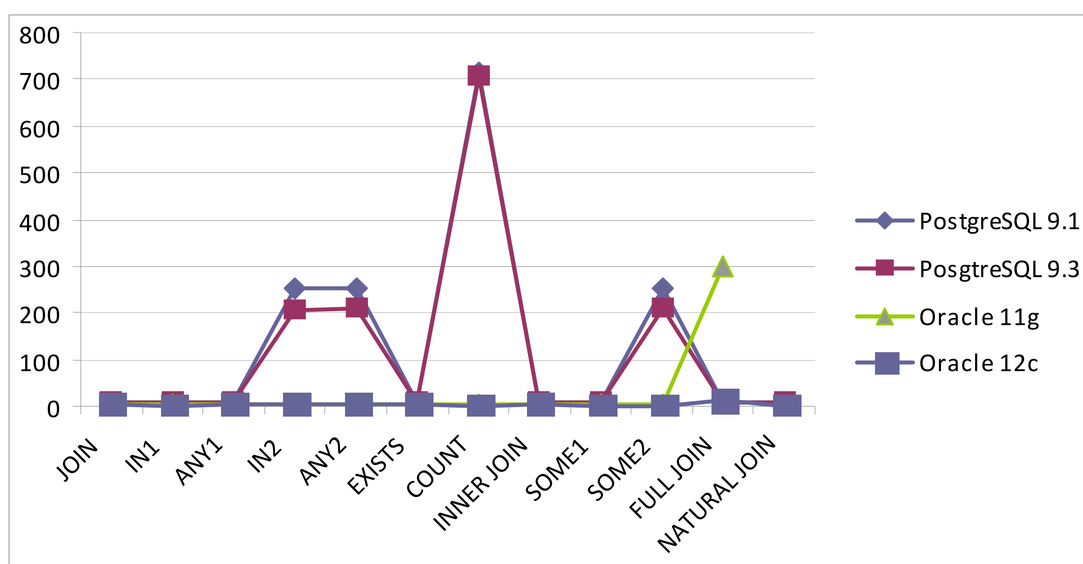
Laiendatud päring “milliste osakondade töötajatel on väikseim palk ja boonus 6000?” järgis ajaliselt suuremalt jaolt sama joont mis originaalpäring. 9 900 reaga tabeli tulemused osutusid ajaliselt jällegi mitte kõige informatiivsemaks, kuna aeglaseim päring võttis kõigest 0,3 sekundit. 20 miljoni reaga tulid aga jälle esile tõsisemad erisused kõigis süsteemides peale Oracle 12c. Lisaks kiiruse probleemidele ilmnes Oracle 11g puhul NATURAL JOIN korral ka erijuht, millest kirjutatakse täpsemalt Oracle alajaotuses.

Tabel 5 – erinevad täitmisplaanid laiendatud päringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	1	2	2	3	3	2	4	1	2	3	5	1
Posgtres 9.3	6	7	7	3	3	7	4	6	7	3	5	6
Oracle 11g	8	9	9	9	9	9	8	8	9	9	10	NA
Oracle 12c	11	12	12	12	12	12	13	11	12	12	14	11

Tabel 6 – päringute kestvused sekundites laiendatud päringus 20 miljoni reaga kasutaja poolt indekseerimata tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	7,85	8,29	7,53	251,84	250,42	7,65	715,71	7,68	7,82	251,54	7,72	7,99
Posgtres 9.3	8,08	8,18	7,98	206,89	209,94	7,70	707,14	9,00	7,70	209,98	7,54	7,78
Oracle 11g	3,42	4,08	3,88	3,80	3,97	3,93	3,58	3,60	4,12	3,94	299,82	NA
Oracle 12c	2,16	2,05	2,31	2,40	2,59	2,21	2,03	2,15	2,12	1,99	11,65	2,01



Joonis 17 – laiendatud päringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseerimata tabelis

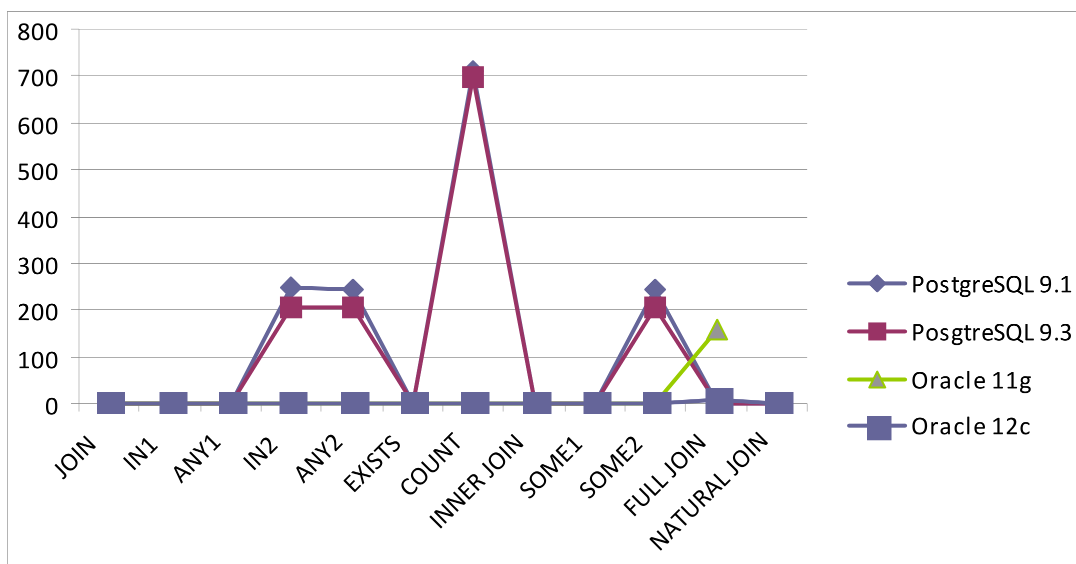
Tabelist 5 on näha, et PostgreSQL andmebaasid kasutavad laiendatud päringus endiselt viite erinevat täitmisplaani, kuid erinevalt originaalpäringust, esineb nüüd rohkem versioonist tulenevaid erinevusi. Oracle mõlemad versioonid kasutavad nelja erinevat täitmisplaani. Vaadates tabelis 6 ja joonisel 17 olevaid tulemusi päringute kestvuse kohta, on näha, et probleemsed päringud jäid samaks mis originaalpäringus, kuid esile on tõusnud COUNT, mis nüüd eristub selgelt kõige aeglasema päringuna.

Tabel 7 – erinevad täitmisplaanid laiendatud päringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	1	2	2	3	3	2	4	1	2	3	5	1
Posgtres 9.3	6	7	7	8	8	7	9	6	7	8	10	6
Oracle 11g	11	12	12	12	12	12	11	11	12	12	13	NA
Oracle 12c	14	15	15	15	15	15	16	14	15	15	17	14

Tabel 8 – päringute kestvused sekundites laiendatud päringus 20 miljoni reaga kasutaja poolt indekseeritud tabelis

	JOIN	IN1	ANY1	IN2	ANY2	EXISTS	COUNT	INNER	SOME1	SOME2	FULL J	NAT. J
Postgres 9.1	0,01	0,02	0,01	247,22	245,07	0,10	711,61	0,10	0,01	245,55	0,01	0,10
Posgtres 9.3	0,01	0,01	0,02	204,20	207,10	0,02	696,40	0,02	0,01	206,70	0,02	0,01
Oracle 11g	0,11	0,33	0,44	0,44	0,49	0,26	0,12	0,11	0,29	0,29	157,18	NA
Oracle 12c	0,04	0,05	0,05	0,07	0,06	0,08	0,08	0,03	0,07	0,07	8,05	0,04



Joonis 18 – laiendatud päringu kestvus sekundites 20 miljoni reaga kasutaja poolt indekseeritud tabelis

Tabelist 7 on näha, et kasutaja poolt indekseeritud tabelite korral, ei muutunud ühe andmebaasisüsteemi lõikes laiendatud päringu korral kasutatud erinevate täitmisplaanide hulk. Samas tuleb nüüd PostgreSQLi korral paremini esile erinevus versioonide 9.1 ja 9.3 vahel, kuna enam polnud ühtegi täitmisplaani, mis oleks mõlemas versioonis olnud täpselt samasugune. Tabelis 8 ja joonisel 18 päringute kiiruseid vaadates on näha täpselt analoogsed muudatused nagu toimusid originaalpäringus. Kõik hästi optimeeritud päringud ja Oracle FULL JOIN läksid kiiremaks, samas kui PostgreSQLi aeglased päringud tulemusi ei parandanud.

5.2.1 PostgreSQL

PostgreSQLis võib laiendatud päringu täitmisplaanide järgi jagada viieks eri grupiks.

1. JOIN, INNER JOIN, NATURAL JOIN
2. FULL JOIN
3. IN1, ANY1, EXISTS, SOME
4. IN2, ANY2, SOME2
5. COUNT

Ajaliselt on kolm esimest gruppi sisuliselt sama kiired, aga neljas ja viies andmehulga kasvades märgatavalt aeglasemad (vt tabelid 7 ja 8). Siin esimese kolme grupi omavahelisi erinevusi põhjalikult kirjeldama ei hakka, vaid keskendume erinevustele, mis toovad kaasa selgeid vahesid päringu kiiruses. Vaatame esiteks neljandat gruppi.

```
Join Filter: (department.depid = personnel.depid)
->  Seq Scan on department  (cost=0.00..3.00 rows=1 width=124)
->  Materialize  (cost=0.00..210599717.98 rows=10000004 width=10)
        ->  Seq Scan on personnel  (cost=0.00..210500888.96
        rows=10000004 width=10)
                Filter: (SubPlan 3)
                SubPlan 3
                   ->  Index Scan using payroll_pkey on payroll
             (cost=3.68..17.31 rows=1 width=4)
                        Index Cond: (personnel.empid = empid)
                        Filter: (salary = $1)
                        InitPlan 2 (returns $1)
                         ->  Result  (cost=3.67..3.68 rows=1 width=0)
                               InitPlan 1 (returns $0)
                                 ->  Limit  (cost=0.00..3.67 rows=1
                   width=4)
                                      ->  Index Scan using
                               payroll_salary_index on payroll
                               (cost=0.00..73376735.89 rows=20000000
                               width=4)
                                              Index Cond: (salary IS
                                            NOT NULL)
```

Joonis 19 – laiendatud päring IN2, ANY2, SOME2 päringutes PostgreSQL 9.1 ja 9.3 andmebaasis 20 miljoni reaga kasutaja poolt indekseerimata tabelis

Joonisel 19 olevast täitmisplaanist on näha, et päringul on kolm alamosa, mida kasutatakse tabeli *payroll* läbimiseks. Iseenesest mõlemad *InitPlan*-id siinkohal tegelikult probleemiks ei ole kuna seda peab andmebaasisüsteem ainult ühe korra käivitama, probleemiks on ilmselt *SubPlan 3,* mida tuleb päringu käigus korduvalt käivitada. Seda järeldust toetab ka viienda grupi ehk COUNT täitmisplaani vaatamine samadel tingimustel.

```
HashAggregate  (cost=3.00..3.01 rows=1 width=124)
  ->  Seq Scan on department  (cost=0.00..3.00 rows=1 width=124)
        Filter: (0 < (SubPlan 3))
        SubPlan 3
            ->  Aggregate  (cost=7548074789837.71..7548074789837.72
        rows=1 width=0)
                ->  Seq Scan on personnel
                (cost=0.00..7548074789497.58 rows=136054 width=0)
                      Filter: ((department.depid = depid) AND (0 <
                   (SubPlan 2)))
                      SubPlan 2
                          ->  Aggregate  (cost=377403.55..377403.56
                        rows=1 width=0)
                                InitPlan 1 (returns $0)
                                   ->  Aggregate
                                (cost=377389.90..377389.91 rows=1
                                width=4)
                                         ->  Seq Scan on payroll
                         (cost=0.00..327389.72 rows=20000072 width=4)
                                ->  Index Scan using payroll_pkey on
                                payroll  (cost=0.00..13.63 rows=1
                                width=0)
                                      Index Cond: (personnel.empid =
                                empid)
                                       Filter: ((salary = $0) AND (bonus
                                       = 6000))
```

Joonis 20 – laiendatud päring COUNT päringutes PostgreSQL 9.1 ja 9.3 andmebaasis 20 miljoni reaga kasutaja poolt indekseerimata tabelis

Joonisel 20 COUNT kasutab lausa kahte *SubPlan* tsüklit nii *payroll* kui ka *personnel* tabelite peal. Sellega on hästi seletatav antud päringu absoluutselt halvim ajaline tulemus PostgreSQL andmebaasides. Kusjuures kui korra vaadata näiteks päringuid EXIST ja COUNT kõrvuti, siis ligemale sajakordne ajaline vahe tuleb sisse puhtalt sellest et päringu optimeerija ei suuda taibata kahe järgneva klausli samaväärsust:

1. “ `WHERE EXISTS (SELECT * ` “
2. “ `WHERE 0 < (SELECT COUNT(*) ` “

Tsüklite põhjustatavat ajalist kaotust ei aita kuidagi märgatavalt parandada ka indeksite lisamine. Probleemsed päringud üldsegi ignoreerivad liitindeksit, mis ühendab veerge *salary* ja *bonus* ning kasutavad ainult veerul *salary* olevat indeksit. Vaatame järgnevalt ka ühte efektiivset täitmisplaani.

```
Hash Semi Join  (cost=1237.53..1249.84 rows=49 width=84)
   Hash Cond: (department.depid = personnel.depid)
   InitPlan 2 (returns $1)
     ->  Result  (cost=2.95..2.96 rows=1 width=0)
           InitPlan 1 (returns $0)
             ->  Limit  (cost=0.44..2.95 rows=1 width=4)
                   ->  Index Only Scan using payroll_salary_index on
                   payroll payroll_1  (cost=0.44..50347527.06
                   rows=20000000 width=4)
                         Index Cond: (salary IS NOT NULL)
   ->  Seq Scan on department  (cost=0.00..11.40 rows=140 width=124)
   ->  Hash  (cost=1233.34..1233.34 rows=98 width=10)
        ->  Nested Loop  (cost=5.99..1233.34 rows=98 width=10)
              ->  Bitmap Heap Scan on payroll  (cost=5.44..392.25
              rows=98 width=10)
                      Recheck Cond: ((salary = $1) AND (bonus = 6000))
                      ->  Bitmap Index Scan on payroll_multi_index
                      (cost=0.00..5.42 rows=98 width=0)
                            Index Cond: ((salary = $1) AND (bonus =
                            6000))
              ->  Index Scan using personnel_pkey on personnel
              (cost=0.55..8.57 rows=1 width=20)
                    Index Cond: (empid = payroll.empid)
```


Joonis 21 – laiendatud päring IN1, ANY1, EXISTS, SOME päringutes PostgreSQL 9.3 andmebaasis 20 miljoni reaga kasutaja poolt indekseeritud tabelis

Joonisel 21 on näha märgatavalt efektiivsemat lähenemist kus välditakse liigseid tsükleid. Optimeerija kasutab *InitPlan* osa miinimumi leidmiseks ning seejärel indekseid ära kasutades leiab tingimustele vastavad tabeli *payroll* veerud ning viib seejärel läbi vajalikud ühendamised. Indeksite lisamine omas arvestatavat lisaväärtust kuna seni 7–8 sekundit kestnud päringuid hakati seejärel täitma millisekundites.

Lisaks saab antud plaanis olevast miinimumi arvutusest esile tuua ka konkreetse PostgreSQL 9.2 lisandunud uue funktsiooni *Index Only Scan*, mida 9.1 päringutes veel ei olnud. Nagu nimigi ütleb, tähendab see et ridu endid üldse ei vaadata, ning piirdutakse ainult nendele viitavate indeksite skaneerimisega. (Paquier 2012)

5.2.2 Oracle

Oracle päringud võib täitmisplaanide järgi sisuliselt jagada järgmisteks gruppideks.

1. JOIN, INNER JOIN, COUNT (O11), NATURAL JOIN (O12)
2. IN1, ANY1, IN2, ANY2, EXISTS, SOME1, SOME2
3. COUNT(O12)
4. FULL JOIN

Esimesed kolm gruppi on seejuures kiiruselt praktiliselt samaväärsed mõlemas testitud Oracle versioonis (vt tabelid 7 ja 8). COUNT, mille optimeerimisega PostgreSQLil niipalju probleeme tekkis, optimeeriti Oracle 11g versioonis analoogselt JOIN ja INNER JOIN päringutega ja 12c versioonis unikaalselt kuid väga sarnaselt IN, ANY jne päringutega.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		90	276
HASH (GROUP BY)		90	276
HASH JOIN		90	276
Access Predicates			
PERSONNEL.DEPID=DEPARTMENT.DEPID			
HASH JOIN		90	273
Access Predicates			
PERSONNEL.EMPID=PAYROLL.EMPID			
NESTED LOOPS			
NESTED LOOPS		90	273
STATISTICS COLLECTOR			
TABLE ACCESS (BY INDEX ROWID BATCHED)	PAYROLL	90	93
INDEX (RANGE SCAN)	PAYROLL_MULTI_INDEX	90	3
Access Predicates			
AND			
SALARY= (SELECT MIN(SALARY) FROM PAYROLL PAYROLL)			
BONUS=6000			
SORT (AGGREGATE)		1	
INDEX (FULL SCAN (MIN/MAX))	PAYROLL_MULTI_INDEX	1	3
INDEX (UNIQUE SCAN)	PERSONNEL_PK	1	1
Access Predicates			
PERSONNEL.EMPID=PAYROLL.EMPID			
TABLE ACCESS (BY INDEX ROWID)	PERSONNEL	1	2
TABLE ACCESS (FULL)	PERSONNEL	1	2
TABLE ACCESS (FULL)	DEPARTMENT	50	3

Joonis 22 – laiendatud päring JOIN jne. päringutes Oracle 12c andmebaasis 20 miljoni reaga kasutaja poolt indekseeritud tabelis

Vaadates joonist 22, siis on näha kuidas kõigepealt käiakse indekseid kasutades läbi tabelid *personnel* ja *payroll*, mis ühendatakse. Seejärel toimub ühendamine tabeliga *department*. Siia konkreetset plaani ära tooma ei hakka, aga IN, ANY jne täitmisplaan on suhteliselt sarnane, lihtsalt peale *personnel* ja *payroll* tabelite ühendamist luuakse nende põhjal vaade (*view*),

misjärel toimub ühendamine tabeliga *department*. Kuna täheldatavat kiiruse vahet kummalgi variandil ei ole, siis ei saa ka öelda, et üks oleks konkreetselt halvem või parem kui teine.

Vaatleme nüüd aga probleemsemaid juhte alustades Oracle 11g FULL JOIN-iga (vt joonis 23).

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1768	196394
HASH (GROUP BY)		1768	196394
VIEW	VW_FOJ_0	20000000	178041
Filter Predicates			
AND			
BONUS=6000			
SALARY= (SELECT MIN(SALARY) FROM PAYROLL PAYROLL)			
HASH JOIN (FULL OUTER)		20000000	178041
Access Predicates			
PERSONNEL.EMPID=PAYROLL.EMPID			
TABLE ACCESS (FULL)	PAYROLL	20000000	17305
VIEW	VW_FOJ_1	20000000	80028
HASH JOIN (FULL OUTER)		20000000	80028
Access Predicates			
DEPARTMENT.DEPID=PERSONNEL.DEPID			
TABLE ACCESS (FULL)	DEPARTMENT	50	3
TABLE ACCESS (FULL)	PERSONNEL	20000000	79928
SORT (AGGREGATE)		1	
TABLE ACCESS (FULL)	PAYROLL	20000000	17305

Joonis 23 – laiendatud päring FULL JOIN päringus Oracle 11g andmebaasis kasutaja poolt indekseerimata tabelites

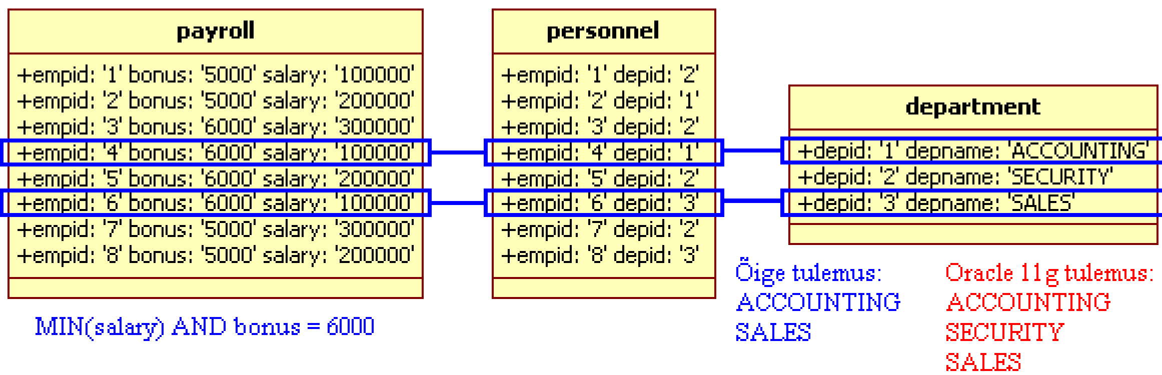
Probleem ja aeglus on siinkohal praktiliselt sama, mis ilmnes originaalpäringus. Andmebaasisüsteem teeb esimesena tabelite täieliku ühendamise, mis 20 miljoni realiste *personnel* ja *payroll* tabelite juures võtab väga kaua aega. Väike *department* tabel oma 50 reaga siinjuures omab suhteliselt väikest rolli. Indekseerimine aitab küll saavutada märgatava päringu kiirenemise, kuid vahe hästi optimeeritud päringutega jääb endiselt umbes sajakordseks. 12c versioonis on FULL JOIN endiselt ebaefektiivsem, aga siiski edasiminek on arvestatav, nüüd on vahe teiste päringute täitmise kiirusega vähem kui kümnekordne.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		90	113231
HASH (GROUP BY)		90	113231
HASH JOIN (OUTER)		90	96024
Access Predicates			
PERSONNEL.EMPID(+)=PAYROLL.EMPID			
TABLE ACCESS (FULL)	PAYROLL	90	17231
Filter Predicates			
AND			
PAYROLL.BONUS=6000			
PAYROLL.SALARY= (SELECT MIN(SALARY) FROM PAYROLL PAYROLL)			
SORT (AGGREGATE)		1	
TABLE ACCESS (FULL)	PAYROLL	20000000	17206
VIEW	VW_FOJ_0	20000000	78744
HASH JOIN (FULL OUTER)		20000000	78744
Access Predicates			
DEPARTMENT.DEPID=PERSONNEL.DEPID			
TABLE ACCESS (FULL)	DEPARTMENT	50	3
TABLE ACCESS (FULL)	PERSONNEL	20000000	78691

Joonis 24 – laiendatud päring FULL JOIN päringus Oracle 12c andmebaasis kasutaja poolt indekseerimata tabelites

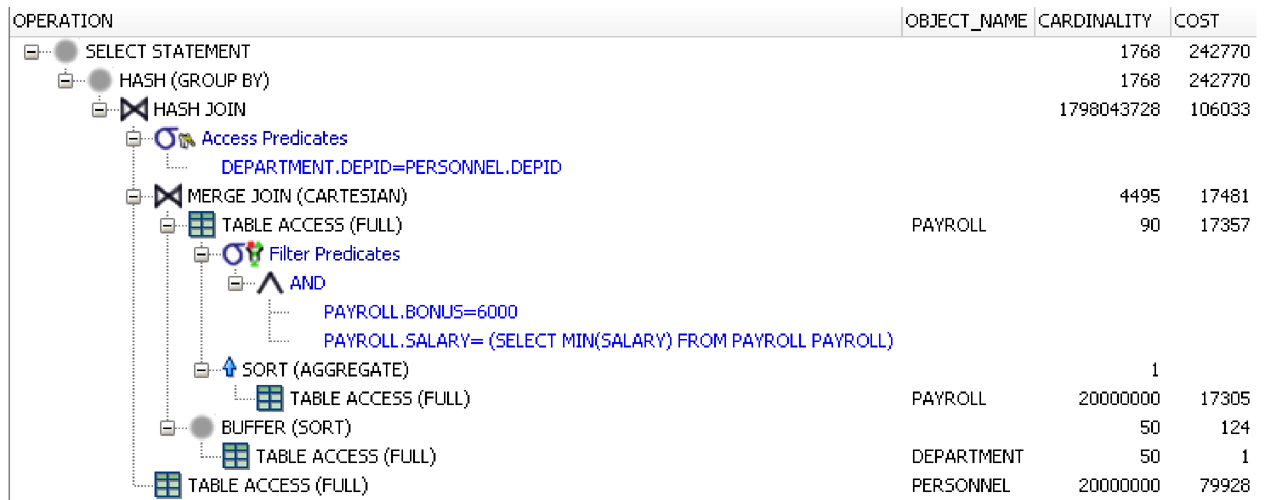
Lühidalt, vaadates joonist 24, siis Oracle 12c alustab sellest, et ühendab tervenisti *personnel* ja *department,* aga tabelis *payroll* teeb vajaliku otsingu ning seega väldib vanemas versioonis tekkinud kahe 20 miljoni reaga tabeli ühendamist.

Laiendatud päringu teema lõpetuseks vaatame NATURAL JOIN päringut Oracle 11g versioonis. NATURAL JOIN suhtes soovitatakse üldiselt ka normaalolukorras ettevaatliku lähenemist, kuna tabeli struktuuri muutmisel või lihtsalt veergude ümbernimetamine võib muuta antud päringu toimimist. (Gorman jt. 2014 p 185-186) Oracle 11 versioonis ilmnes aga otsene tarkvaraviga kus antud päring andis järjekindlalt teistsuguseid vastuseid kui kõik ülejäänud päringud. Samas Oracle 12 versioonis ja ka PostgreSQLi mõlemas versioonis NATURAL JOIN samadel tingimustel toimis. Kontrollimaks, et viga pole kuidagi andmetega seotud, sai tehtud väike lisatest, kus mõlema Oracle versiooni andmebaasi sisestati käsitsi väike kogus identseid testandmeid. Tabelitesse *personnel* ja *payroll* sisestati kaheksa rida ja tabelisse *department* kolm rida. Nende testandmete sisestamise laused ja kasutatud päring on ära toodud Lisas 6. Antud lisatest kinnitas kahtlust, et Oracle versioonid käituvad identsete andmete ja identse sisendiga erinevalt kuna versioon 12 tagastas kaks rida nagu oli õige aga samas versiooni 11 tagastas kolm rida. Lisatesti tabelid on näha joonisel 25.



Joonis 25 – lisatest Oracle 11g NATURAL JOIN vea kontrollimiseks

Interneti otsinguga õnnestus leida viide Oracle veale 5031632, mis puudutavat NATURAL JOINi ning võis seega olla probleemi põhjuseks. (Hasler 2008)



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1768	242770
HASH (GROUP BY)		1768	242770
HASH JOIN		1798043728	106033
Access Predicates			
DEPARTMENT.DEPID=PERSONNEL.DEPID			
MERGE JOIN (CARTESIAN)		4495	17481
TABLE ACCESS (FULL)	PAYROLL	90	17357
Filter Predicates			
AND			
PAYROLL.BONUS=6000			
PAYROLL.SALARY= (SELECT MIN(SALARY) FROM PAYROLL PAYROLL)			
SORT (AGGREGATE)		1	
TABLE ACCESS (FULL)	PAYROLL	20000000	17305
BUFFER (SORT)		50	124
TABLE ACCESS (FULL)	DEPARTMENT	50	1
TABLE ACCESS (FULL)	PERSONNEL	20000000	79928

Joonis 26 – laiendatud päring NATURAL JOIN Oracle 11g andmebaasis

Joonisel 26 on täitmisplaanist näha kaks JOIN operatsiooni. Üks on tavaline räsiühendamine üle ühise välja *depid*. Samas teine on CARTESIAN JOIN, ehk siis ebaõnnestus ühise veeru *empid* leidmine ning andmebaas tegi JOIN operatsiooni kus kõik ühe tabeli read liideti kõigi teise tabeli ridadega. See tähendas et kui tabelis *payroll* leiti kasvõi üksainus päringu tingimustele vastav rida, siis liitmise järel oli see rida seotud kõigi 50 tabelis *department* oleva reaga ning päringu vastuseks tuli alati kogu *department.depname* nimekiri.

5.3 Mediaanpalga päring

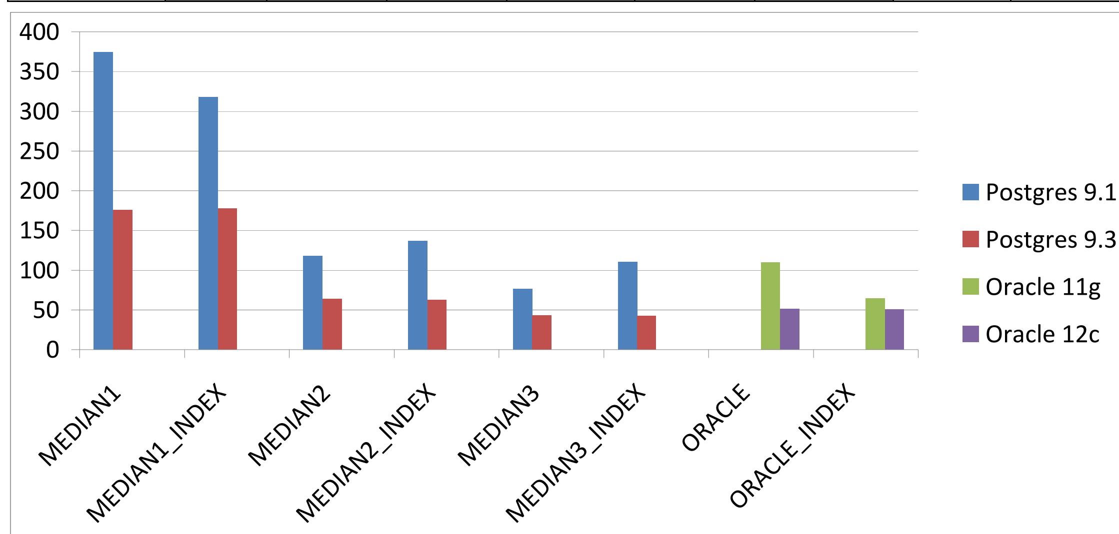
Viimaseks küsimuseks, millele antud eksperimendis vastuseid päriti, oli mediaanpalga leidmine. Oracles on sisse ehitatud MEDIAN() funktsioon, mis lahendab selle küsimuse ilma probleemideta. Samas PostgreSQLis taolist funktsiooni sisse ehitatud ei ole ja iga kasutaja peab ise leidma endale sobivaima lahenduse. Variantideks on kas kasutaja poolt agregaatfunktsiooni loomine (Aggregate Median 2014) või tavalise päringu kirjutamine, viimast varianti vaatleme antud eksperimendis lähemalt. Antud testis kasutati PostgreSQLis kolme erinevat mediaani leidmise päringut. Seejuures tuleb arvestada, et tegemist pole täieliku nimekirjaga viisidest kuidas antud küsimust saab lahendada, ega ka tingimata mitte absoluutselt parimate võimalike viisidega. Aga teatava esialgse ülevaate probleemi olemusest peaks antud tulemuste põhjal olema võimalik saada.

Tabel 9 – erinevad täitmisplaanid PostgreSQL mediaani päringutes koos ja ilma kasutaja poolt loodud indeksiteta

	M1	M1_INDX	M2	M2_INDX	M3	M3_INDX
Postgres 9.1	1	1	2	2	3	3
Posgtres 9.3	1	1	2	2	4	4

Tabel 10 – mediaani päringute kiirus sekundites PostgreSQLis ja Oracles

	M1	M1_INDX	M2	M2_INDX	M3	M3_INDX	OR	OR_INDX
Postgres 9.1	374,02	317,91	118,1	136,94	76,55	110,4		
Posgtres 9.3	175,84	177,55	63,43	62,76	42,85	42,45		
Oracle 11g							109,48	62,47
Oracle 12c							51,29	50,68



Joonis 27 – mediaanpalga päringud 20 miljoni kirjega kasutaja poolt indekseeritud ja indekseerimata tabelites, x-teljel on päringute identifikaatorid ja y-teljel nende kestvus sekundites

Tabelist 9 on näha et kõigil mediaani päringutel olid erinevad täitmisplaanid, aga eri PostgreSQL versioonide vahel püsisid need üldiselt samasugused, ainult MEDIAN3 korral esines väike erinevus. Tabelis 10 ja joonisel 27 on näha päringute kiirused. Tulemused olid väga selged, MEDIAN1 oli vaieldamatult aeglaseim päring, MEDIAN2 oli üsnagi kiire ning MEDIAN3 sellest veel parem. Tabelite indekseerituse kohta mingeid üheseid järeldusi teha ei saa. PostgreSQL 9.1 see tegi MEDIAN1 kiiremaks ja samas teisi aeglasemaks. Oracle 11g süsteemis tõstis indekseerimine samuti MEDIAN() funktsiooni kiirust. Samas uuemates versioonides PostgreSQL 9.3 ja Oracle 12c indekseerimine mingit arvestatavat muutust antud päringute töökiiruses kaasa ei toonud.

```
Aggregate  (cost=1429128.82..1429128.83 rows=1 width=4)
  ->  Subquery Scan on x  (cost=1294447.35..1429010.68 rows=47256 width=4)
        Filter: ((x.rowasc = x.rowdesc) OR (x.rowasc = (x.rowdesc - 1)) OR (x.rowasc =
        (x.rowdesc + 1)))
         ->  WindowAgg  (cost=1294447.35..1357771.27 rows=3166196 width=14)
               ->  Sort  (cost=1294447.35..1302362.84 rows=3166196 width=14)
                     Sort Key: payroll.salary, payroll.empid
                     ->  WindowAgg  (cost=726934.19..790258.11 rows=3166196 width=14)
                           ->  Sort  (cost=726934.19..734849.68 rows=3166196 width=14)
                                 Sort Key: payroll.salary, payroll.empid
                                 ->  Seq Scan on payroll  (cost=0.00..222744.96
                                 rows=3166196 width=14)
                                       Filter: (salary IS NOT NULL)
```

Joonis 28 – MEDIAN1 täitmisplaan

MEDIAN1, mille täitmisplaan on joonisel 28, kasutab suhteliselt lihtsat lähenemist. Kasutatakse sisseehitatud *row_number()* funktsiooni, mis täitmisplaanis on näha *WindowAgg* all, et moodustada kasvav ja kahanev nimekiri tulemustest ning siis leida nende keskmiste väärtuste põhjal mediaan. Kahe nimekirja tekitamine pole kahtlemata andmemahtude kasvades kõige efektiivsem variant ning see nähtub ka tulemustest. Antud päringu täitmisplaan ei muutunud ka vähimalgi määral andmehulga või andmebaasisüsteemi verisooni muutudes.

MEDIAN2 kasutab rekursiivset lähenemist, mis täitmisplaanis kajastub kui CTE ehk *Common Table Expression*. Lühidalt on esimene päring, mis tagastab palga ja *row_number()* funktsiooniga selle reanumbri, sisendiks teisele päringule, mis tagastab kogu ridade arvu. Kogu ridade arvu tulemuse põhjal kutsub kolmas päring esimest välja. Indekseid kasutati antud täitmisplaanis ainult 9 900 realise tabeli korral. MEDIAN2 täitmisplaan on näha joonisel 29.

```
CTE Scan on rowcount  (cost=920.74..1490.28 rows=1 width=8)
  CTE realsalary
    ->  WindowAgg  (cost=0.00..697.98 rows=9900 width=4)
          ->  Index Scan using payroll_salary_index on payroll  (cost=0.00..549.48
         rows=9900 width=4)
                Index Cond: (salary IS NOT NULL)
  CTE rowcount
    ->  Aggregate  (cost=222.75..222.76 rows=1 width=0)
          ->  CTE Scan on realsalary  (cost=0.00..198.00 rows=9900 width=0)
  SubPlan 3
    ->  Aggregate  (cost=297.25..297.26 rows=1 width=4)
          ->  CTE Scan on realsalary  (cost=0.00..297.00 rows=99 width=4)
                Filter: (rn = ANY (ARRAY[(rowcount.ct / 2), ((rowcount.ct / 2) + 1)]))
  SubPlan 4
    ->  CTE Scan on realsalary  (cost=0.00..272.25 rows=50 width=4)
          Filter: (rn = ((rowcount.ct + 1) / 2))
```

Joonis 29 – MEDIAN2 täitmisplaan

MEDIAN3 kasutas suhteliselt lihtsat lähenemist, kus päriti ridade arv ja seda kasutades seejärel kaks keskmist väärtust sorteeritud *salary* nimekirjast. Ridade arvu põhjal valiti paaritu arvu korral neist üks ning paarisarvu korral arvutati nende keskmine. Kuna mingeid selgelt ära tuntavaid liigseid tegevusi ei ole, siis võib arvata et sellest plaanist edasi enam hüppelist tulemuse parandamist pole võimalik saavutada. MEDIAN3 täitmisplaan on näha joonisel 30.

```
Subquery Scan on midrows  (cost=230660.45..925920.24 rows=1 width=40)
  ->  Subquery Scan on count  (cost=230660.45..925920.22 rows=1 width=8)
        ->  Aggregate  (cost=230660.45..230660.46 rows=1 width=0)
              ->  Seq Scan on payroll  (cost=0.00..222744.96 rows=3166196 width=0)
                    Filter: (salary IS NOT NULL)
        SubPlan 1
          ->  Limit  (cost=695259.74..695259.75 rows=2 width=4)
                ->  Sort  (cost=694468.19..702383.68 rows=3166196 width=4)
                      Sort Key: payroll_1.salary
                        ->  Seq Scan on payroll payroll_1  (cost=0.00..222744.96
                        rows=3166196 width=4)
                              Filter: (salary IS NOT NULL)
```

Joonis 30 – MEDIAN3 täitmisplaan

6. Tulemuste analüüs ja järeldused

Erineva sõnastusega päringuid ei teisendatud alati täpselt samasugusteks täitmisplaanideks. Igasugused ajalised erinevused PostgreSQLis ja Oracles 9 900 realistes tabelites jäid aga millisekunditesse ning seega suhteliselt väheütlevaks, kuna sellistel juhtudel võivad hakata rolli mängima pisiasjad – näiteks kas päringut on juba varem käivitatud ning selle täitmisplaan ja päringuga leitavad andmed on vahemälus olemas või mitte. 20 miljoni realiste tabelitega andmebaas tõi aga esile tuntavaid ajalisi vahesid, mis võimaldas täpsemalt hinnata täitmisplaanides tehtavate valikute mõju päringu täitmise kiirusele.

PostgreSQLis ilmnesid selged optimeerimisprobleemid nii Pascali originaalpäringus kui ka laiendatud päringus nelja sõnastuse korral: ANY2, IN2, SOME2 ja COUNT. Kõigil juhtudel oli probleemiks tsüklid, mida tuli liigselt läbi viia ning mis ka takistasid efektiivset indeksite kasutuselevõttu. Laiendatud päringus kannatas COUNT teistest veel rohkemgi, kuna seal kasutati lausa kahte ebaefektiivset tsüklit. Ka kiirelt toimivate päringute täitmisplaanides esines väiksemaid erinevusi, mida saaks üldiselt jagada kaheks: esiteks erinevad JOIN plaanid ja teiseks IN1, ANY1 jne plaanid. Kuna kiiruses märgatavaid erinevusi ei olnud, siis võib öelda, et tavakasutaja seisukohast need ei omanud tähtsus, kuid ei saa välistada, et veelgi keerukamate kombinatsioonide korral võiksid need hakata kiirust mõjutama.

Oracle puhul kannatas kiiruseprobleemide käes ainult üks originaalpäringu ja laiendatud päringu variant: FULL JOIN. Oracle 11g versioonis oli see probleem äärmiselt tuntav, kuna andmebaasisüsteemi valitud täitmisplaan, mis sisaldas tabelite täielikku ühendamist, osutus täiesti mitteskaleeruvaks ning 20 miljoni realise tabeli korral võttis palju kordi rohkem aega kui muud variandid. Oracle 12c oli FULL JOIN endiselt täiesti teistsuguse täitmisplaaniga kui ülejäänud päringud, ja endiselt tuntavalt aeglasem, kuid vahe polnud enam nii suur. Ka kiirelt toimivate päringute seas ilmnes lähemal uurimisel täitmisplaanide seas väiksemaid erinevusi, täpselt samamoodi nagu PostgreSQLis. Üldiselt võib siiski öelda et kommertssüsteem Oracle ületab tasuta saadaval olevat ja avatud lähtekoodiga PostgreSQLi kasutaja halbade valikute silumisel, eriti kui keskenduda ainult Oracle viimasele versioonile 12c.

Eelnevas lõigus sai juba puudutatud erinevusi erinevate versioonide vahel. Üldiselt oli nii PostgreSQLi 9.1 ja 9.3 kui ka Oracle 11g ja 12c täitmisplaanide võrdluses näha, et päringute optimeerijaid arendatakse pidevalt ja lisandub ka uusi funktsionaalsusi. Tähelepanuväärsemad näited olid Oracles kahe tabeli ühendamisel ühe tervikliku tsükli asemel sisemise ja välimise tsükli kombinatsiooni kasutamine ja 12c lisandunud statistika kogumine, mis võimaldab täitmisplaane päringu täitmise käigus muuta. PostgreSQLis tuli osade täitmisplaanide võrdluses esile 9.2 versioonis lisandunud indeksite skaneerimine ilma tabelit ennast puutumata.

Vaadeldes eksperimendis koostatud täitmisplaane, siis vaadeldud ülesannete korral koostasid PostgreSQL ja Oracle sama lause korral erinevad täitmisplaanid. See näitab kuidas erinevate andmebaasisüsteemide optimeerimismoodulite loojad on langetanud erinevaid valikuid.

Oracle 11g juures väärib märkimist ka NATURAL JOIN päringus avastatud tarkvaraviga. See aitab demonstreerida veel ühte väga harva kasutatavate SQL keele konstruktsioonidega seotud ohtu. Eksootilisemate konstruktsioonide kasutamine mitte ainult ei või päringut muuta aeglasemaks, vaid halvimal juhul võib ka osutuda vigaseks ning tagastada vale vastuse. Tarkvara arendajate testimisressurss on paratamatult piiratud ning seetõttu on testimise fookus loomulikult laialdaselt kasutatavatel konstruktsioonidel/funktsioonidel.

Sisemiselt paremini optimeeritud päringud kasutasid ka kasutaja poolt lisatud indekseid. Väiksema, 9 900 realise tabeli korral, sõltus see päringu keerukusest. Pascali originaalpäringu jaoks indekseid ei kasutatud, aga laiendatud päringute täitmisplaanides olid need kasutuses. 20 miljoni realise tabeli korral kasutasid kõik kiiremad päringud lisatud indekseid ära ja saavutasid ka märgatavat tulemuste paranemist. Halvasti optimeeritud päringud üldiselt kasutaja lisatud indekseid ei kasutanud või kasutasid ainult osaliselt, näiteks laiendatud päringu korral võidi kasutada ainult indeksit veerul *salary* ja mitte mõlemat päringus kasutatavat veergu *salary* ja *bonus* ühendavat indeksit. Kuid Oracle 11g FULL JOIN näitas, et ka väga halvasti optimeeritud päringud võivad teatud juhtudel indeksitest märgatavalt võita.

Eksperimendi viimases etapis võrreldi lühidalt kolme erinevat mediaani päringut PostgreSQLis. Kõigi kolme täitmisplaanid olid täiesti erinevad ja leidsid vastuse eri kiirustega. Andmebaasisüsteemi versiooni ning andmehulga muutus mõjutas neid väga vähe. See tulemus demonstreerib kuidas päringu keerukuse tõstmisel läheb päringute optimeerijal samasuste äratundmine järjest raskemaks.

Kokkuvõte

Andmebaasisüsteemid omavad arvestatavat tähtsust enamike organisatsioonide töös. Andmemahud ja andmete keerukus on pidevalt kasvamas. Andmebaasidega suhtlemiseks kasutatakse andmebaasikeeli, millest maailmas on hetkel kõige laiemalt levinud SQL. Antud keele kohta on tehtud palju kriitikat, üheks tõstatatud probleemiks on keele liiasus, ehk siis ühesugust ülesannet saab lahendada paljude erinevate andmekäitluse lausetega.

Käesoleva magistritöö eesmärgiks oli uurida kas ja kuidas SQL keele liiasus mõjutab tänapäeva moodsate andmebaasisüsteemide tööd. Aluseks võeti Fabian Pascali poolt 1988. aastal teostatud SQL andmebaasisüsteemide töökiiruse eksperiment. Peamiseks edasiarenduseks Pascali eksperimendist oli töökiiruste võrdlemise asemel keskendumine päringute täitmisplaanide analüüsile. Lisaks kasutati suuremat testandmete hulka, laiendati testandmebaasi struktuuri ja lisati uusi päringuid. Eksperimendis kasutati laialt populaarsete andmebaasisüsteemide PostgreSQL versioone 9.1.4 ja 9.3.0 ning Oracle versioone 11g Enterprise Edition Release 1 ja 12c Enterprise Edition Release 1.

Eksperimendi tulemused tõestasid, et SQL keele liiasus on endiselt probleemide allikas ning tänapäevaste andmebaasisüsteemide optimeerimismoodulid (optimeerijad) ei suuda alati ühesuguse tulemusega päringutele luua ühesuguseid täitmisplaane. Probleemsed juhtumid kasutasid üldiselt liigseid tsükleid ning ei suutnud efektiivselt rakendada veergudele lisatud indekseid. Samas oli näha, et optimeerijate efektiivsus on aastate jooksul selgelt kasvanud ja pidevalt lisandub lausete töökiiruse parandamiseks uusi funktsionaalsusi. Lisaks tuleb arvestada, et üldiselt optimeeriti edukamalt intuitiivsema süntaksiga päringuid, mida programmeerija suurema tõenäosusega antud ülesande lahendamiseks kasutaks. Ebatüüpilise (vähem kasutatava) lause optimeerimisel näitas kommertssüsteem Oracle paremaid tulemusid kui PostgreSQL, mis tähendab, et viimase kasutamisel langeb suurem vastutus päringuid koostava inimese õlgadele.

SQL keele liiasus on endiselt reaalne probleemide allikas. Seega on antud töös teostatud eksperimenti kahtlemata võimalik märgatavalt edasi arendada ja laiendada. Siinkohal on vähemalt kaks võimalikku suunda. Esiteks saaks kasutada sama eksperimenti, kuid laiendada uurimise skoopi hõlmamaks rohkem erinevaid SQL keelt kasutavaid andmebaasisüsteeme. Teine ja ilmselt märksa laiem suund oleks aga tõsta päringutega otsitava küsimuse keerukust. Käesolevas töös kasutati päringutes kolme tabelit ja kuute veergu. Siin on selgelt võimalus tõsta päringu keerukust lisades tabeleid ja kasutatavaid veerge.

Summary

Database Management Systems (DBMS) have major role in the functioning of most modern organizations. Amount of data and its complexity is constantly growing. One has to use database languages for operating databases. Among such languages the SQL is most widely used. SQL has received a plenty of criticism. Among the raised issues is redundancy of the language, meaning that there are many different ways to write a SQL query that solves the same task.

The purpose of this Master's Thesis was to investigate, how the redundancy of SQL language affects functioning of modern database management systems. The basis of the experiment was Fabian Pascal's 1988 test about performance of the SQL DBMSs. While Pascal's experiment focused on the query performance based on time taken, in this thesis the focus was in the investigation of query execution plans. Additionally, we increased the amount of the test data, extended the database structure, and added more queries to the experiment. We used popular mainstream DBMSs PostgreSQL versions 9.1.4 and 9.3.0 and Oracle versions 11g Enterprise Edition Release 1 and Enterprise Edition 12c Release.

The experiment demonstrated that SQL language redundancy is still a source of problems. Query optimisers of modern DBMSs are not always able to create identical execution plans for queries that return equivalent results. Problematic cases generally used internally too many loops and failed to use indexes effectively. On other hand, one could see that the efficiency of optimisers has significantly increased over the years and new features to improve the performance are regularly added. In addition, one should note that more intuitive queries, that a programmer is more likely to use for solving specific questions, generally performed better. Then optimising statements with more exotic (less used) language constructs, the commercial DBMS Oracle generally showed better results than PostgreSQL, meaning that then using the latter a bigger responsibility falls on shoulders of the programmer to ensure a satisfactory performance.

Future work in the investigation of SQL redundancy could expand on results of this thesis in multiple ways. First option would be widening the scope while using the same experiment. It means repeating tests based on additional SQL DBMSs. Second and potentially far deeper direction would be increasing complexity of queries used in the experiment. In this thesis, three tables and six columns were used by the queries. Clearly, here is potential for the expansion by increasing the number of tables and number of columns and raising the complexity of queries.

Kasutatud kirjandus

1. Aggregate Median (2014), https://wiki.postgresql.org/wiki/Aggregate_Median (14.05.2014)
2. **Brandstetter E.** (2011) Is there a better way to calculate the median (not average), http://stackoverflow.com/a/8214352 (14.05.2014)
3. **Chaudhuri S., Weikum G.** (2000) Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. Proceedings of the 26th International Conference on Very Large Databases.
4. **Colgan M.** (2008) Inside the Oracle Optimizer - Removing the black magic, http://optimizermagic.blogspot.com/2008/02/displaying-and-reading-execution-plans.html (14.05.2014)
5. **Colgan M.** (2013) What's new in 12c: Adaptive joins, https://blogs.oracle.com/optimizer/entry/what_s_new_in_12c (14.05.2014)
6. **Darwen H.** (2012) SQL: A Comparative Survey. http://bookboon.com/en/sql-a-comparative-survey-ebook
7. DB-Engines Ranking (2014), http://db-engines.com/en/ranking (14.05.2014)
8. **DeBarros A.** (2010) Calculating Medians With SQL, http://www.anthonydebarros.com/2010/07/25/calculating-medians-sql/ (14.05.2014)
9. **Douglas K., Douglas S.** (2003) PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases. Sams Publishing.
10. **Van Dyke R.** (2012) Why a double Nested Loop?, http://ricramblings.blogspot.com/2012/11/why-double-nested-loop.html (14.05.2014)
11. **Eessaar E., Saal E.** (2012) Evaluation of Different Designs to Represent Missing Information in SQL Databases. Proceedings of the International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 2011).
12. **Fernandez I.** (2006) SQL Rocks!, http://www.nocoug.org/download/2006-08/SQL_Sucks_NoCOUG_Conference_Presentation.pdf (14.05.2014)
13. **Fernandez I.** (2009) Beginning Oracle Database 11g Administration: From Novice to Professional. Apress.

14. **Fernandez I.** (2011) Day 4: The way you write your query matters, http://iggyfernandez.wordpress.com/2011/12/04/day-4-the-twelve-days-of-sql-there-way-you-write-your-query-matters/ (14.05.2014)
15. **Fraiteur G.** (2005) SQL Tuning Tutorial - Understanding a Database Execution Plan (1), http://www.codeproject.com/Articles/9990/SQL-Tuning-Tutorial-Understanding-a-Database-Execu (14.05.2014)
16. **Gorman T., Jørgensen I., Caffrey M., deHaan L.** (2014) Beginning Oracle SQL: for Oracle Database 12c. Apress.
17. **Gupta P. K. D., Krishna P. R.** (2013) – Database Management System Oracle SQL and PL/SQL. PHI Learning Pvt. Ltd.
18. **Halpin T., Morgan T.** (2010) Information Modeling and Relational Databases. Morgan Kaufmann.
19. **Hasler T.** (2008) Why I Like ANSI SQL Join Syntax, http://tonyhasler.files.wordpress.com/2008/09/why-i-like-ansi-sql.pdf (14.05.2014)
20. Introduction to Oracle Database (2009), http://docs.oracle.com/cd/E11882_01/server.112/e40540/intro.htm#CNCPT001 (14.05.2014)
21. **Kivi R.** (2008) SQL päringute töökiiruse analüüs. Magistritöö, Tallinna Tehnikaülikool.
22. **Pascal F.** (1988) SQL redundancy and DBMS performance. Database Programming and Design.
23. **Pascal F.** (2013) Language Redundancy and DBMS Performance: A SQL Story, http://www.dbdebunk.com/2013/02/language-redundancy-and-dbms.html (14.05.2014)
24. **Paquier M.** (2012) PostgreSQL 9.2 highlight: Index-only scans, http://michael.otacoo.com/postgresql-2/postgresql-9-2-highlight-index-only-scans/ (14.05.2014)
25. PostgreSQL 9.3.4 Documentation (2013), http://www.postgresql.org/docs/9.3/static/index.html (14.05.2014)
26. **Rasheed D.** (2010) Re: median for postgresql 8.3, http://postgresql.1045698.n5.nabble.com/median-for-postgresql-8-3-tp3267671p3269201.html (14.05.2014)
27. **Wang L., Tan K. C.** (2006) Modern Industrial Automation Software Design. John Wiley & Sons.
28. **Winand M.** (2011) PostgreSQL Execution Plan Operations, http://use-the-index-luke.com/sql/explain-plan/postgresql/operations (14.05.2014)

Lisad

Lisa 1 - Pascali originaalpäring

1. JOIN

```
SELECT lname
FROM personnel, payroll
WHERE personnel.empid = payroll.empid
AND salary = 199170;
```

2. IN1

```
SELECT lname
FROM personnel
WHERE empid IN (SELECT empid
     FROM payroll
     WHERE salary = 199170);
```

3. ANY1

```
SELECT lname
FROM personnel
WHERE empid = ANY (SELECT empid
     FROM payroll
     WHERE salary = 199170);
```

4. IN2

```
SELECT lname
FROM personnel
WHERE 199170 IN (SELECT salary
     FROM payroll
     WHERE personnel.empid = payroll.empid);
```

5. ANY2

```
SELECT lname
FROM personnel
WHERE 199170 = ANY (SELECT salary
     FROM payroll
     WHERE personnel.empid = payroll.empid);
```

6. EXISTS

```
SELECT lname
FROM personnel
WHERE EXISTS (SELECT *
     FROM payroll
     WHERE personnel.empid = payroll.empid
     AND salary = 199170);
```

7. COUNT

```
SELECT lname
FROM personnel
WHERE 0 < (SELECT COUNT(*)
     FROM payroll
     WHERE personnel.empid = payroll.empid
     AND salary = 199170);
```

8. INNER JOIN

```
SELECT lname
FROM personnel INNER JOIN payroll ON
     personnel.empid = payroll.empid
WHERE salary = 199170;
```

9. SOME1

```
SELECT lname
FROM personnel
WHERE empid = SOME (SELECT empid
     FROM payroll
     WHERE salary = 199170);
```

10. SOME2

```
SELECT lname
FROM personnel
WHERE 199170 = SOME (SELECT salary
     FROM payroll
     WHERE personnel.empid = payroll.empid);
```

11. FULL JOIN

```
SELECT lname
FROM personnel
FULL JOIN payroll ON
     personnel.empid = payroll.empid
WHERE salary = 199170;
```

12. NATURAL JOIN

```
SELECT lname
FROM personnel NATURAL JOIN payroll
WHERE salary = 199170;
```

Lisa 2 - Andmebaasitabelite loomise laused

PostgreSQL:

```
CREATE TABLE personnel
(
	empid CHAR(9) PRIMARY KEY,
	depid CHAR(9) NOT NULL,
	lname CHAR(15),
	fname CHAR(12),
	address CHAR(20),
	city CHAR(20),
	state CHAR(2),
	ZIP CHAR(5)
);

CREATE TABLE payroll
(
	empid CHAR(9) PRIMARY KEY,
	bonus INTEGER,
	salary INTEGER
);

CREATE TABLE department
(
	depid CHAR(9) PRIMARY KEY,
	depname CHAR(20),
	description CHAR(100)
);
```

Oracle:

```
CREATE TABLE personnel
(
	empid CHAR(9),
	depid CHAR(9) NOT NULL,
	lname CHAR(15),
	fname CHAR(12),
	address CHAR(20),
	city CHAR(20),
	state CHAR(2),
	ZIP CHAR(5),
	CONSTRAINT personnel_pk PRIMARY KEY (empid)
);

CREATE TABLE payroll
(
	empid CHAR(9),
	bonus INTEGER,
	salary INTEGER,
	CONSTRAINT payroll_pk PRIMARY KEY (empid)
);

CREATE TABLE department
(
	depid CHAR(9),
	depname CHAR(20),
	description CHAR(100),
	CONSTRAINT department_pk PRIMARY KEY (depid)
);
```

Indeksid:

```
CREATE INDEX payroll_salary_index ON payroll (salary);

CREATE INDEX payroll_multi_index ON payroll (salary, bonus);
```

Lisa 3 - Testandmete sisestamise laused

PostgreSQL 9 900 rida:

```
INSERT INTO personnel
SELECT
     empid,
     trunc (random() * 49),
     substr(upper(md5(random()::TEXT)), 5, 15),
     substr(upper(md5(random()::TEXT)), 5, 12),
     substr(upper(md5(random()::TEXT)), 5, 20),
     substr(upper(md5(random()::TEXT)), 5, 20),
     substr(upper(md5(random()::TEXT)), 2, 2),
     trunc(random() * 89999) + 10000
FROM generate_series(0, 9899) AS empid

INSERT INTO payroll
SELECT
     personnel.empid,
     trunc (random() * 10) * 1000,
     199170 + (trunc (random() * 200) * 1000 - 100000)
FROM personnel

INSERT INTO department
SELECT
     depid,
     substr(upper(md5(random()::TEXT)), 5, 20),
     substr(upper(md5(random()::TEXT)), 20, 100)
FROM generate_series(0, 49) AS depid
```

PostgreSQL 20 000 000 rida:

```
INSERT INTO personnel
SELECT
     empid,
     trunc (random() * 49),
     substr(upper(md5(random()::TEXT)), 5, 15),
     substr(upper(md5(random()::TEXT)), 5, 12),
     substr(upper(md5(random()::TEXT)), 5, 20),
     substr(upper(md5(random()::TEXT)), 5, 20),
     substr(upper(md5(random()::TEXT)), 2, 2),
     trunc(random() * 89999) + 10000
FROM generate_series(0, 19999999) AS empid
```

```
INSERT INTO payroll
SELECT
     personnel.empid,
     trunc (random() * 10) * 1000,
     (trunc (random() * 20000) + 10000) * 10
FROM personnel

INSERT INTO department
SELECT
     depid,
     substr(upper(md5(random()::TEXT)), 5, 20),
     substr(upper(md5(random()::TEXT)), 20, 100)
FROM generate_series(0, 49) AS depid
```

Oracle 9 900 rida:

```
INSERT INTO personnel
SELECT
     Rownum r,
     TRUNC(DBMS_RANDOM.VALUE(1, 51)),
     DBMS_RANDOM.STRING('U', 15),
     DBMS_RANDOM.STRING('U', 12),
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 2),
     TRUNC(DBMS_RANDOM.VALUE(10000, 100000))
FROM dual
CONNECT BY LEVEL <= 9900;

INSERT INTO payroll(empid, bonus, salary)
SELECT
     personnel.empid,
     TRUNC(DBMS_RANDOM.VALUE(0, 10)) * 1000,
     199170 + (TRUNC(DBMS_RANDOM.VALUE(0, 200)) * 1000  -
100000)
FROM personnel;

INSERT INTO department
SELECT
     Rownum r,
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 100)
FROM dual
CONNECT BY LEVEL <= 50;
```

Oracle 20 miljonit rida:

```
INSERT INTO personnel
SELECT
     Rownum r,
     TRUNC(DBMS_RANDOM.VALUE(1, 51)),
     DBMS_RANDOM.STRING('U', 15),
     DBMS_RANDOM.STRING('U', 12),
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 2),
     TRUNC(DBMS_RANDOM.VALUE(10000, 100000))
FROM dual
CONNECT BY LEVEL <= 20000000;

INSERT INTO payroll(empid, bonus, salary)
SELECT
     personnel.empid,
     TRUNC(DBMS_RANDOM.VALUE(0, 10)) * 1000,
     TRUNC(DBMS_RANDOM.VALUE(10000, 30000)) * 10
FROM personnel;

INSERT INTO department
SELECT
     Rownum r,
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 100)
FROM dual
CONNECT BY LEVEL <= 50;
```

personnel alternatiivvariant mitmes osas kasutamiseks, kui mälust ei peaks piisama et 20 miljonit rida korraga peale lasta. X1 tuleb asendada enne käivitamist olemasoleva ridade hulgaga tabelis ja X2 hulgaga mis konkreetne andmebaas suudab ühe korraga peale lasta:

```
INSERT INTO personnel
SELECT
     X1 + Rownum r,
     TRUNC(DBMS_RANDOM.VALUE(1, 51)),
     DBMS_RANDOM.STRING('U', 15),
     DBMS_RANDOM.STRING('U', 12),
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 20),
     DBMS_RANDOM.STRING('U', 2),
     TRUNC(DBMS_RANDOM.VALUE(10000, 100000))
FROM dual
CONNECT BY LEVEL <= X2;
```

Lisa 4 - Laiendatud päring

1. JOIN

```
SELECT depname
FROM department, personnel, payroll
WHERE personnel.empid = payroll.empid
AND personnel.depid = department.depid
AND salary = (
     SELECT MIN(salary)
     FROM payroll)
AND bonus = 6000
GROUP BY department.depid;
```

2. IN

```
SELECT depname
FROM department
WHERE depid IN (
     SELECT depid
     FROM personnel
     WHERE empid IN (
          SELECT empid
          FROM payroll
          WHERE salary = (
               SELECT MIN(salary)
               FROM payroll)
          AND bonus = 6000));
```

3. ANY

```
SELECT depname
FROM department
WHERE depid = ANY (
     SELECT depid
     FROM personnel
     WHERE empid = ANY(
          SELECT empid
          FROM payroll
          WHERE salary = (
               SELECT MIN(salary)
               FROM payroll)
          AND bonus = 6000));
```

4. IN2

```
SELECT depname
FROM department
WHERE depid IN (
     SELECT depid
     FROM personnel
     WHERE 6000 IN (
          SELECT bonus
          FROM payroll
          WHERE personnel.empid = payroll.empid
          AND salary = (
               SELECT MIN(salary)
               FROM payroll)));
```

5. ANY2

```
SELECT depname
FROM department
WHERE depid = ANY (
     SELECT depid
     FROM personnel
     WHERE 6000 = ANY (
          SELECT bonus
          FROM payroll
          WHERE personnel.empid = payroll.empid
          AND salary = (
               SELECT MIN(salary)
               FROM payroll)));
```

6. EXISTS

```
SELECT depname
FROM department
WHERE EXISTS (
     SELECT *
     FROM personnel
     WHERE department.depid = personnel.depid
     AND EXISTS (
          SELECT *
          FROM payroll
          WHERE personnel.empid = payroll.empid
          AND salary = (
               SELECT MIN(salary)
               FROM payroll)
          AND bonus = 6000));
```

7. COUNT

```
SELECT depname
FROM department
WHERE 0 < (
     SELECT COUNT(*)
     FROM personnel
     WHERE department.depid = personnel.depid
     AND 0 < (
          SELECT COUNT(*)
          FROM payroll
          WHERE personnel.empid = payroll.empid
          AND salary = (
               SELECT MIN(salary)
               FROM payroll)
          AND bonus = 6000))
GROUP BY department.depid;
```

8. INNER JOIN

```
SELECT depname
FROM department
INNER JOIN personnel
     ON department.depid = personnel.depid
INNER JOIN payroll
     ON personnel.empid = payroll.empid
WHERE salary = (
     SELECT MIN(salary)
     FROM payroll)
AND bonus = 6000
GROUP BY department.depid;
```

9. SOME

```
SELECT depname
FROM department
WHERE depid = SOME (
     SELECT depid
     FROM personnel
     WHERE empid = SOME (
          SELECT empid
          FROM payroll
          WHERE salary = (
               SELECT MIN(salary)
               FROM payroll)
          AND bonus = 6000));
```

10. SOME2

```
SELECT depname
FROM department
WHERE depid = SOME (
     SELECT depid
     FROM personnel
     WHERE 6000 = SOME (
          SELECT bonus
          FROM payroll
          WHERE personnel.empid = payroll.empid
          AND salary = (
               SELECT MIN(salary)
               FROM payroll)));
```

11. FULL JOIN

```
SELECT depname
FROM department
FULL JOIN personnel ON
     department.depid = personnel.depid
FULL JOIN payroll ON
     personnel.empid = payroll.empid
WHERE salary = (
     SELECT MIN(salary)
     FROM payroll)
AND bonus = 6000
GROUP BY department.depid;
```

12. NATURAL JOIN

```
SELECT depname
FROM department
NATURAL JOIN personnel
NATURAL JOIN payroll
WHERE salary = (
     SELECT MIN(salary)
     FROM payroll)
AND bonus = 6000
GROUP BY department.depid;
```

Lisa 5 - Mediaanpäring

1. MEDIAN1

```
SELECT CAST(AVG(x.salary) AS DECIMAL(10,2)) AS Median
FROM (
     SELECT salary, row_number() OVER (
          ORDER BY salary ASC, empid ASC) AS RowAsc,
     row_number() OVER (
          ORDER BY salary DESC, empid DESC) AS RowDesc
     FROM payroll
     WHERE  salary IS NOT NULL
) AS x
WHERE RowAsc IN (RowDesc, RowDesc - 1, RowDesc + 1);
```

2. MEDIAN2

```
WITH realsalary AS (
     SELECT salary, row_number() OVER (ORDER BY salary) AS rn
     FROM payroll
     WHERE salary IS NOT NULL
     ), rowcount AS (
          SELECT count(*) As ct
          FROM   realsalary
     )
SELECT CASE WHEN rowcount.ct%2 = 0
THEN round(
     (SELECT avg(salary)
     FROM realsalary
     WHERE realsalary.rn IN (rowcount.ct/2, (rowcount.ct/2)+1)),
3)
ELSE
     (SELECT salary
     FROM realsalary
     WHERE realsalary.rn = (rowcount.ct+1)/2)
END AS median
FROM rowcount;
```

3. MEDIAN3

```
SELECT CASE WHEN c%2 = 0 AND c > 1
THEN CAST((salary[1]+salary[2]) AS FLOAT)/2
ELSE salary[1] END
     FROM (
          SELECT ARRAY(
               SELECT salary
               FROM payroll
               WHERE  salary IS NOT NULL
               ORDER BY salary
               OFFSET (c-1)/2
               LIMIT 2)
          AS salary, c
          FROM (
               SELECT count(*) AS c
               FROM payroll
               WHERE  salary IS NOT NULL)
          AS count
      OFFSET 0)
AS midrows;
```

4. ORACLE MEDIAN

```
SELECT MEDIAN(salary)
FROM payroll;
```

Lisa 6 - Oracle 11g vea kordamise testandmed ja päring

```
INSERT ALL
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (1, 2, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (2, 1, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (3, 2, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (4, 1, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (5, 2, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (6, 3, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (7, 2, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
INTO personnel(empid, depid, lname, fname, address, city, state,
zip) VALUES (8, 3, 'smith', 'john', 'vabaduse', 'tallinn', 'EE',
'10651')
SELECT  * FROM dual;

INSERT ALL
INTO payroll(empid, bonus, salary) VALUES (1, 5000, 100000)
INTO payroll(empid, bonus, salary) VALUES (2, 5000, 200000)
INTO payroll(empid, bonus, salary) VALUES (3, 6000, 300000)
INTO payroll(empid, bonus, salary) VALUES (4, 6000, 100000)
INTO payroll(empid, bonus, salary) VALUES (5, 6000, 200000)
INTO payroll(empid, bonus, salary) VALUES (6, 6000, 100000)
INTO payroll(empid, bonus, salary) VALUES (7, 5000, 300000)
INTO payroll(empid, bonus, salary) VALUES (8, 5000, 200000)
SELECT * FROM dual;

INSERT ALL
INTO department(depid, depname, description) VALUES (1,
'ACCOUNTING', 'desc')
INTO department(depid, depname, description) VALUES (2,
'SECURITY', 'desc')
INTO department(depid, depname, description) VALUES (3, 'SALES',
'desc')
SELECT * FROM dual;
```

Päring

```
SELECT depname
FROM department
NATURAL JOIN personnel
NATURAL JOIN payroll
WHERE salary = (SELECT MIN(salary) FROM payroll)
AND bonus = 6000
GROUP BY depname;
```