KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

INFORMACIJOS SISTEMŲ KATEDRA

Andrius Šunauskas

Programų sistemos greitaveikos priemonių išskyrimas ir tyrimas

Magistro darbas

Darbo vadovas

prof. dr. Rimantas Butleris

KAUNAS

2009

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

INFORMACIJOS SISTEMŲ KATEDRA

Andrius Šunauskas

Programų sistemos greitaveikos priemonių išskyrimas ir tyrimas

Magistro darbas

Recenzentas

doc. dr. Eimutis Karčiauskas

2009-05-

Vadovas

prof. dr. Rimantas Butleris

2009-05-

Atliko

IFM-3/2 gr. stud.

Andrius Šunauskas

2009-05-

KAUNAS

2009

SANTRAUKA

Šio magistrinio darbo tikslas išanalizuoti spartos didinimo būdus, apžvelgti įvairius gerinimo metodus ir pasiūlyti metodiką, kuri leistų greitai ir efektyviai užtikrinti programų sistemų spartą. Pagrindinis uždavinys yra suformuluoti programų sistemos greitaveikos užtikrinimo metodiką ir atlikti metodikos eksperimentinį tyrimą bei pagrįsti jos efektyvumą.

Darbe nagrinėjami programos veiklos logikos, kodo, SQL užklausų ir techninės sistemos dalies optimizavimo būdai. Ypač didelis dėmesys skiriamas didelių duomenų bazių ir sistemų, dirbančių su jomis, greitaveikos problemoms.

Eksperimentinė darbo dalis atlikta su viešosios valstybinės įstaigos (Kauno teritorinės ligonių kasos) informacine sistema. Eksperimento metu, operuojant didelės apimties duomenimis, buvo atliktas pilnavertis greitaveikos užtikrinimo metodikos tyrimas.

Raktažodžiai: programų sistemos greitaveika, programinės įrangos sparta, greitaveikos priemonės

SUMMARY

Software system performance tools segregation and research

The aim of the work is to analyze the performance improvement techniques, an overview of the various methods to improve system performance and propose a methodology that would enable of crating fast and efficient software systems. The challenge is to create a high efficiency methodology to ensure software quality and performance.

Work topics are: business logic optimization, code tuning, SQL queries tuning and the technical system part optimization techniques. Main focus is on software operating with large databases performance problems.

The experiment was done with government organization (Kauno teritorine ligoniu kasa) information system. During the experiment the methodology efficiency research was done.

Keywords: software performance, software performance tuning, software performance engineering

TURINYS

LENTELIŲ SĄRAŠAS

PAVEIKSLĖLIŲ SĄRAŠAS

1. ĮVADAS

Informacija – neatsiejama šių dienų įmonių dalis. Informacijos srautai reikalauja didelių kaštų jiems tvarkyti ir apdoroti. Todėl reta įmonė gali funkcionuoti be programų sistemos, skirtos palengvinti šias funkcijas. Nuolatos tenka didinti vidinius IT padalinius, kad būtų užtikrinamos darbo funkcijos.

Kad šito būtų išvengta, įmonės įdiegia naujas arba atnaujina esamas sistemas. Šis darbas nėra lengvas, nes reikalauja laiko ir lėšų. Be to, ne visada naujoji sistema patenkina lūkesčius, todėl dažnai ją vėl tenka tobulinti, taisyti klaidas bei plėsti pridedant papildomą funkcionalumą. Tokioms sistemoms reikia ir nuolatinės priežiūros. Visų pirma, reikia vartotojus supažindinti su sistema, apmokyti su ja dirbti. Neužtenka ją įdiegti ir paleisti. Antra, turi būti užtikrinta techninė priežiūra ir su ja susiję dalykai. Tai suteikia perspektyvą kūrėjui, kad sukūrus, įdiegus ir pasirūpinus informacinės sistemos priežiūra būtų galima tikėtis finansinių pajamų šiandien ir rytoj.

Pagrindiniai tikslai, kurių siekiama kuriant programas, yra nauji rezultatai. Programa turi duoti geresnių rezultatų, nei jos pirmtakė, arba sistema turi tinkamai automatizuoti gyvenimišką (verslo) procesą. Į rezultatus galime įtraukti ir tokias vartotojui aktualias sistemos savybes kaip funkcionalumas ir paprastumas naudoti. Taip pat galime įtraukti savybes, būtinas ne visose sistemose, bet labai vertingas ir brangiai kainuojančias –greitį ir saugumą. Greičio kriterijus dažnai keliamas didelėse sistemose arba ten, kur vyksta realaus laiko procesai. Saugumas, bent minimalus, privalomas daugelyje programų. Šių dviejų kriterijų tobulinimas dažnai užtrunka daug laiko ir pareikalauja nemažai lėšų.

Vienas garsus finansininkas yra pasakęs: „Sekundės yra labai brangios“. Šis požiūris ypač aktualus šiandieniniame pasaulyje. Jei per tą sekundę perskaičiuojamos didelės pinigų sumos – sekundės praradimas gali atnešti didelių nuostolių. Adekvačiai galime kalbėti apie žalą, kurią patiria įmonės dėl jų naudojamų sistemų spartos.

Programos nuolat tobulėja. Kartu didėja ir operuojamų duomenų kiekiai. Bet kuri didelė programa naudoja duomenų bazę duomenims saugoti. Duomenų kiekiai gali svyruoti nuo 100 iki 100 milijonų įrašų ir daugiau. Atvejai, kai duomenų bazės įrašų skaičius neviršija milijono, yra neįdomūs, nes netgi silpnos sistemos su tokiais duomenų kiekiais gerai ir greitai susitvarko. Mūsų tikslas išnagrinėti spartos gerinimo būdus, kai duomenų bazėje saugomų įrašų skaičius viršija 100.000.000 eilučių.

Esant tokiems duomenų kiekiams ir jiems toliau augant, programų sistemos operacijų laikas didėja jau nebe tiesiškai, o eksponentiškai [1]. Taip mažėja didelių sistemų vienas svarbiausių rodiklių – greitis. Galutiniam vartotojui yra svarbu, kiek laiko jis turės laukt, kad gautų rezultatus.

Šio magistrinio darbo tikslas išanalizuoti spartos didinimo būdus, apžvelgti įvairius gerinimo metodus ir pasiūlyti metodiką, kuri leistų greitai ir efektyviai užtikrinti programų sistemų spartą. Pagrindinis uždavinys yra suformuluoti programų sistemos greitaveikos užtikrinimo metodiką ir atlikti metodikos eksperimentinį tyrimą bei pagrįsti jos efektyvumą.

Darbo uždaviniai:

1) išnagrinėti programų sistemos spartos optimizavimo būdus;
2) pasiūlyti programų sistemos greitaveikos užtikrinimo metodiką;
3) atlikti siūlomos metodikos eksperimentinį tyrimą;
4) išanalizavus rezultatus, įvertinti metodikos efektyvumą.

2. PROGRAMŲ SISTEMOS GREITAVEIKOS PRIEMONIŲ ANALIZĖ

Kompiuterijoje optimizavimo procesas[1] apibūdinimas kaip sistemos modifikavimas, kad kai kurios jos dalys (ar aspektai) veiktų efektyviau arba naudotų mažiau resursų. Pavyzdžiui, kompiuterinė programa gali būti optimizuota taip, kad ji užsikrautų itin sparčiai, arba naudotų mažiau darbinės atminties ar kitų resursų, arba naudotų mažiau energijos. Optimizuojama sistema gali būti viena kompiuterinė programa, programų kolekcija arba ištisa sistema, tokia kaip internetas.

Atrodo, kad optimizavimas yra tarsi optimalaus sinonimas, bet sukurti tikrai optimalią sistemą yra labai sunku ar netgi neįmanoma. Optimizuota sistema paprastai optimali gali būti tik tam tikru atžvilgiu: galima sutrumpinti užduočių vykdymo laiką, bet tada programa naudos daugiau atminties. Programoje, kur atminties kiekis yra kritinis dalykas, tikriausiai reikėtų rinktis lėtesnį skaičiavimo algoritmą, kuris naudotų mažiau resursų. Paprastai nėra bendro arba viskam tinkančio optimizavimo sprendimo, kuris veiktų pakankamai gerai, todėl kiekvienu atveju jis turi būti pritaikomas vis kitaip.

Dažnai programų kūrėjai yra priversti daryti mainus (kai ką pagerinti, kai ką pabloginti), kad optimizuotų svarbiausius aspektus: greitį, resursų naudojimą, našumą. Pastangos, reikalingos sukurti optimalią programinę įrangą (kurios nebegalima tobulinti), dažnai būna už logikos ribų – pasiekti rezultatai neatperka įdėtų pastangų, todėl dažniausiai optimizavimo procesas yra stabdomas nepasiekus optimalaus rezultato. Ir tai yra gerai, nes didžiausi programos patobulinimai yra atliekami optimizavimo pradžioje[2].

Optimizavimas gali vykti keliais lygiais (remiamasi literatūros šaltiniu [12]):

- Optimizavimas projekto lygiu (angl. Design level)
- Išeities kodo optimizavimas (angl. Source code level)
- Optimizavimas kompiliavimo metu (angl. Compile level)
- Optimizavimas mašininio kodo lygiu (angl. Assembly level)
- Optimizavimas vykdymo metu (angl. Run time)

---

[1] Daugiau informacijos galima rasti http://en.wikipedia.org/wiki/Optimization_(computer_science)

[2] Galioja Pareto taisyklė: 20 % pastangų duoda 80 % rezultatų, o likę 80 % pastangų – tik 20 % rezultatų. http://en.wikipedia.org/wiki/Pareto_principle

2.1. Programos veiklos logikos optimizavimo būdai

2.1.1. Optimizavimas projekto lygiu

Aukščiausio lygio optimizavimas leidžia pasiekti geriausia esamų resursų panaudojimą. Svarbiausia šiame etape pasirinkti efektyvius algoritmus. Kad jie būtų efektyvūs, reikia parašyti geros kokybės kodą (žiūrėti 2.2 skyrių „Programos kodo optimizavimo būdai“). Teoriškai efektyvus algoritmas realizacijos metu gali prarasti visas savo gerąsias savybes.

Architektūros pasirinkimas daugiausia lemia kuriamos programinės įrangos spartą. Kai kurių architektūrinių šablonų nelankstumas turi įtakos, kad atliekami įvairūs pagerinimai projekte nesukelia norimo spartos pagerėjimo efekto. Vis dėlto, pasinaudojant sudėtingus šablonus, algoritmus ar tam atvejui tinkančias kitas priemones (žiūrėti 2.1.2 skyrių „Kompromisinis optimizavimas“) tai įmanoma. Optimizuota programa gali būti sunkiai suprantama mažiau patyrusiam programuotojui ir turėti daugiau klaidų nei neoptimizuota versija.

2.1.2. Kompromisinis optimizavimas

Optimizavimas paprastai susitelkia į vieno ar kelių programų sistemos parametrų pagerinimą:

- vykdymo laiko
- atminties naudojimo
- disko vietos naudojimo
- ryšio taupymo
- energijos taupymo

Paprastai toks pagerinimas reikalauja kompromisinio sprendimo (angl. trade-off): vienų parametrų pagerinimas kitų sąskaita. Pavyzdžiui, spartinančios atmintinės dydžio[3] (angl. cache) padidinimas pagerina internetinių sistemų spartą, bet padidina naudojamos darbinės atminties kiekį. Tokie mainai daro įtaką programos kodo aiškumui bei glaustumui (žiūrėti 2.2 skyrių „Programos kodo optimizavimo būdai“).

Kartais pasitaiko atvejų, kai programuotojas, atliekantis optimizavimą, turi nuspręsti, kam skirti didesnį prioritetą. Pagerindamas kai kurių atliekamų užduočių spartą dažniausiai sumažina kitų operacijų efektyvumą. Priimami sprendimai gali būti nebūtinai susiję su techniniai dalykais. Pavyzdžiui, konkurentams paskelbus jų kuriamos programos našumo testo rezultatus (angl. benchmark result) jūsų kompanija turi parodyti, kad jų sukurta programa yra nė kiek ne silpnesnė, kitaip galima nesitikėti komercinės programos sėkmės. Deja, kartais tokie optimizuojantys pakeitimai padaro naudojimąsi programa labai nemalonų paprastam vartotojui. Tokie pakeitimai kartais juokais pavadinami antioptimizuojančiais (angl. pessimizations).

2.1.3. Automatinis ir rankinis optimizavimas

Pirma išskirkime, kas yra automatinis, o kas rankinis optimizavimas. Optimizuojantį kompiliatorių galime laikyti automatiniu įrankiu, kuris kompiliavimo ir bibliotekų kūrimo metu atlieka tam tikrus kodo spartos patobulinimus, o įprastą rankinį optimizavimą atlieka programuotojai.

Visos sistemos optimizavimas dažniausiai atliekamas pačių programuotojų, nes tai yra per daug sudėtinga automatiniams optimizatoriams. Žmogaus įsikišimas gali labiau paveikti visą sistemos spartą, tačiau žmogiški resursai dažnai yra riboti ir, nors jie yra efektyvesni, žymiai brangesni už automatinius įrankius.

Spartos analizuotojas (angl. profiler) tai įrankis, programuotojų naudojamas surasti programos kodo vietas, kurios yra probleminės. Dažniausiai tos vietos vadinamos kamščiais arba programos silpnosiomis vietomis (angl. bottlenecks). Tai daugiau bėdų sukeliančios vietos. Kartais programuotojai pasikliauna savo intuicija arba mano, kad tiksliai žino, kur tos

[3] Omenyje turima WEB puslapių spartinančioji atmintis. Daugiau informacijos http://en.wikipedia.org/wiki/Web_cache

silpnosios vietos yra, bet jie dažnai klysta. Netinkamos kodo vietos optimizavimas mažai pagerina, o kratais pablogina bendrą situaciją.

Suradus kamščius (žiūrėti skyrių 2.2.6 „Kamščių pašalinimas“) reikia pergalvoti problemines vietas, ypač naudojamus algoritmus. Kartais užtenka pasirinkti kitą algoritmą ir kamščio problema dingsta.

Toliau reikia perrašyti arba pakeisti blogąjį kodą. Tai atsiperka, nes čia galioja prieš tai minėta Pareto taisyklė[4]. 90/10 taisyklė[5] (modifikuota Pareto taisyklė) skelbia, kad 90% viso laiko yra praleidžiama prie 10% viso kodo, ir tik 10% laiko yra skiriama likusiai 90% kodo daliai. Taigi optimizuojant net mažą programos dalį, galima pasiekti stulbinantį efektą bendrai sistemos spartai.

Rankinis optimizavimas turi ir keletą trūkumų. Jei procesas nėra pakankamai dokumentuojamas, iškyla pavojus, kad ateityje gali niekas nežinoti, kam tai buvo padaryta, ir pašalinti kodą kaip nereikalingą. Taip pat atsirandantis papildomas kodas dažniausiai būna neaiškus ir nukreiptas tik į tam tikrų problemų sprendimą.

Dauguma automatinių optimizavimo įrankių yra įtraukti į kompiliatorius. Jie puikiai tinka, kai norima pritaikyti programą lokaliems poreikiams: konkrečioms platformoms arba tam tikriems procesoriams. Optimizuojantys kompiliatoriai sugeba esamą kodą efektyviai konvertuoti į procesoriui suprantamas instrukcijas. Taip yra išnaudojamas šiuolaikinių procesorių potencialas.

2.1.4. Lygiagrečių procesų naudojimas programų sistemoje

Kaip teigia skyriaus pavadinimas, lygiagretūs procesai naudojami paspartinti sistemos bendrą spartą išlygiagretinant sistemos operacijas laike, t.y. pritaikant lygiagretiems skaičiavimams [4]. Tik kyla klausimas, kiek tai geriau už paprastą ir nuoseklų operacijų vykdymą, ir kiek lėšų reikia, norint išskaidyti skaičiavimus keliais procesais. Norėdami suskaičiuoti teorinį duomenų gavybos pagerėjimą, galime pasinaudoti Amdahlo dėsniu[6].

---

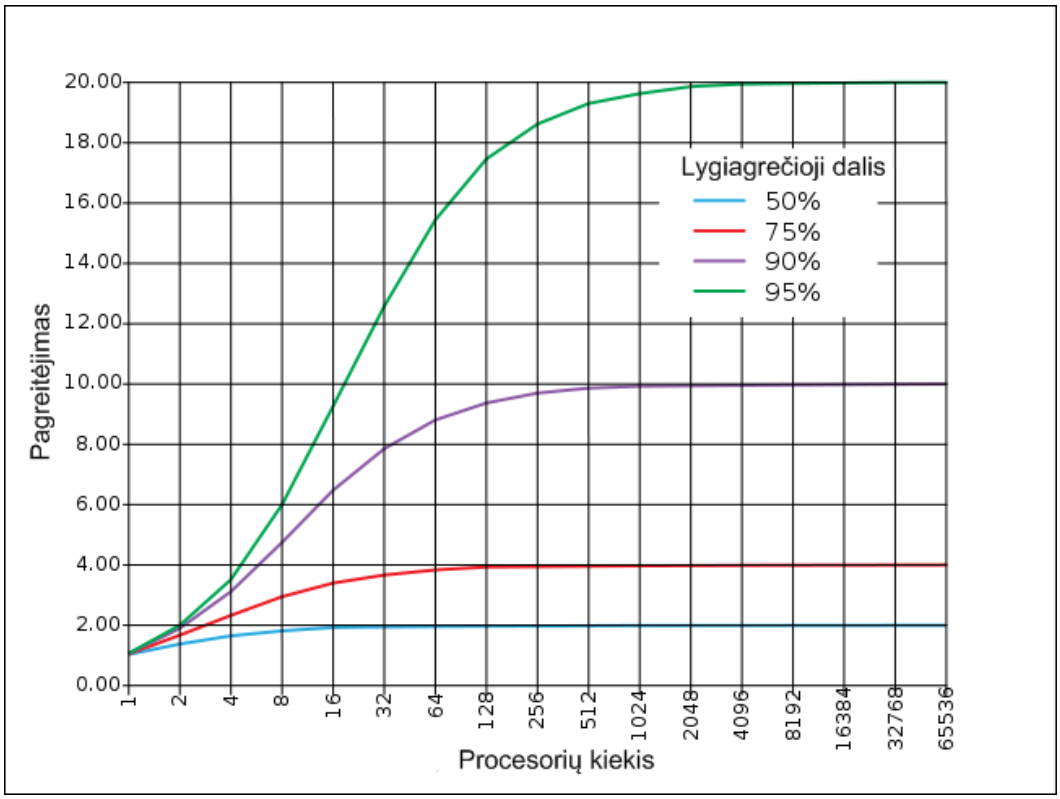
[4] Pareto taisyklė http://en.wikipedia.org/wiki/Pareto_principle

[5] 90/10 taisyklė http://www.indopedia.org/90/10_law.html

[6] Apibrėžimas paimtas iš http://en.wikipedia.org/wiki/Amdahl%27s_law

Sakykime, mes galime pagerinti modulio veikimo greitį ir 70% jo perrašome naudodami lygiagrečiuosius procesus. Taip pat mes nuo šiol naudosime 4 procesorius vietoj vieno šiam moduliui vykdyti. Pasižymime: a – nuoseklių skaičiavimų dalis, o (1-a) skaičiavimų dalis, kuri naudoja lygiagrečiuosius procesus. Tada maksimalus teorinis pagerėjimas, kurį galime pasiekti naudodami P procesorių pagal Amdahlo dėsnį būtų: $\frac{1}{\mathrm{a} + \frac{(1-a)}{P}}$

Skaičiuodami pagal pavyzdį gautume: $\frac{1}{0{,}3 + \frac{(1-0.3)}{4}} = 2{,}105$



1 pav. *Programų teorinis pagreitėjimas pagal Amdahlo dėsnį*

Programos pagreitėjimas yra ribojamas nuosekliosios programos dalies. Pavyzdžiui, jei 50% programos yra nuosekli, tai teorinis galimas pagerėjimas naudojant lygiagrečiuosius procesus būtų iki 200% dabartinės spartos naudojant N procesorių, kai N labai didelis (žiūrint 1 pav.).

Taigi keturis kartus padidindami procesoriaus galią mes tik dvigubai paspartintume visą modulio veikimą (nuo 100% iki 210,5%). Mes iš tikrųjų negautume tenkinančios grąžos. Jei

bandytume dar kartą dvigubinti procesorių kiekį (nuo 4 iki 8), mes gautume: $\frac{1}{0,3+\frac{(1-0.3)}{8}} = 2,581$

Dabar mūsų gautas pagerėjimas yra dar menkesnis ir tesiekia apie ~23%. Skiriant dar daugiau procesorių uždaviniui spręsti gauname vis menkesnį spartos pagerėjimą. Kadangi daugiau procesorių reiškia daugiau atminties sunaudojimo ir daugiau I/O judėjimo, tad mes galime netgi nepajusti to pagerėjimo. Tuo labiau, kad daugiau techninės įrangos gali sukelti ir daugiau techninių problemų ar lūžimų.

Kaip matome, lygiagrečiųjų procesų naudojimas duomenų gavyboje suteikia teorinį spartos pagerėjimas. Bet įvertinus ir kitus faktorius galime teigti, kad daugiau pinigų, išleistų techninei įrangai, reiškia mažą grąžą arba netgi dar blogesnę spartą, nei turėjote su vienu procesoriumi.

2.1.5. Kada reikėtų vykdyti optimizavimą

Optimizavimas nėra visada būtina priemonė kuriant programinę įrangą. Ir tai nebūtinai suteikia visapusiškos naudos:

- kodo skaitomumas sumažėja;
- atsiranda papildomas kodas, kuris skirtas tik optimizavimo uždaviniams realizuoti;
- gali būti komplikuotas palaikomumas ir testavimas;
- kadangi spartos derinimas dažniausiai atliekamas paskutinėse programinės įrangos kūrimo stadijose, tai tampa papildomais ir nenumatytais kūrimo kaštais.

Kartais yra naudojama sąvoką pirmalaikis optimizavimas, kuri apibrėžia situaciją, kai projektuotojai, atsižvelgdami į galimas spartos problemas ateityje, modifikuoja projektą norėdami išvengti galimų trūkumų. Tai gali paveikti architektūrą ir kodą, pavyzdžiui: gali suprastėti kai kurios projekto savybės arba kodas tapti „nešvarus“ ar netgi „neteisingas“. Taip atsitinka dėl to, kad programuotojas vietoj to, kad rašytų gražų, paprastą ir teisingą kodą, turi ji perdirbti, jog jis būtų optimalus. Tai apsunkina kodo supratimą, kas ypač aktualu pradinėse kūrimo stadijose.

Optimizavimą galima atlikti ir po projektavimo bei kodo rašymo etapų. Tada kai jau yra parašytas programinis kodas (arba dalis kodo), galima atlikti našumo testus ir pasižiūrėti, kurios dalys turi trūkumų. Šiame etape turint gražų ir švarų kodą yra nesunku jį optimizuoti. Būtent dabar naudojant profiliavimą (angl. profiling) galima pastebėti spartos problemas, kurios nebuvo apgalvotos pirminio optimizavimo metu.

Žinoma, gerai yra nusistatyti spartos kriterijus jau nuo projektavimo etapo. Kaip rodo praktika, dažniausiai pirma viskas yra suprojektuojama ir paleidžiamas prototipas, o po to taisomi išryškėję greitaveikos trūkumai.

2.2. Programos kodo optimizavimo būdai

2.2.1. Išeities kodo optimizavimas

Paprasčiausias optimizavimas yra atliekamas kodo lygmenyje. Lengviausia tai padaryti iš pat pradžių nerašant blogos kokybės kodo, kuris akivaizdžiai galėtų lėtinti sistemą. Tai daugiausia priklauso nuo programuotojo patirties ir įgūdžių. Jaunesniems programuotojams tai padaryti yra sunku.

Internetinių puslapių kūrime web puslapių kodo optimizavimas turėtų būti atliekamas prieš aplikacijos įdiegimą. Į tai įeina naudojami JavaScript, CSS ir grafikos failai. Optimizavimo metu jie yra suspaudžiami ir tik po to įkeliami į serverį.

Nors ne visada pavyksta išvengti klaidų kode ir iškart gauti gerą spartą, tačiau darbą, kurio nepadaro programuotojai, gali pabaigti daryti kompiliatoriai (žiūrėti skyrių 2.2.2 „Optimizavimas kompiliavimo metu“).

Didžiausias darbas, kurį reikia padaryti šiame lygyje, tai pašalinti kamščius programoje (skaityti skyrių 2.2.6 „Kamščių pašalinimas“).

2.2.2. Optimizavimas kompiliavimo metu

Šiuolaikiniai kompiliatoriai turi įvairių galimybių kodui kompiliuoti. Optimizuojančio kompiliatoriaus naudojimas užtikrina, kad vykdomoji programa yra optimizuota bent tiek, kiek kompiliatorius tai numato. Įmanoma netgi kompiliatoriui nurodyti į ką orientuotis, pavyzdžiui:

- greitį;

- užimamą vietą;
- ir kitką.

Pasirinkus parametrus optimizuoti, dažniausiai kiti parametrai suprastėja, pavyzdžiui: padidinus greitį, padidėja ir užimama disko vieta. Tokie kompromisai yra neišvengiami (apie kompromisus galima pasiskaityti 2.1.2 skyrelyje)

2.2.3. Optimizavimas mašininio kodo lygiu

Efektyviausias būdas optimizuoti programas yra mašininio kodo lygyje. Ši mašininė kalba yra naudojama dažniausiai, kai reikia pasiekti rezultatus konkrečioje aplinkoje ir esant tam tikrai techninei įrangai. Programuotojas gali visiškai išnaudoti kalbą ir parašyti efektyvias mašinines instrukcijas procesoriui. Dažniausiai tam yra naudojama Asemblerio kalba[7].

Šiaip laikais esant sudėtingiems ir kelių branduolių procesoriams sudėtinga parašyti mašininį kodą efektyvesnį negu generuoja patys kompiliatoriai. Kompiliatoriai turi daug konfigūravimo galimybių ir netgi leidžia pasirinkti kokiai operacinės sistemos versijai (omenyje turima 32 ir 64 OS) suoptimizuoti kodą. Tuo labiau, kad kompiliatorius gali optimizuoti instrukcijas ir kelių branduolių procesoriams.

Deja, tai pats brangiausias optimizavimo būdas. Pirma, reikalingi brangūs specialistai, išmanantys mašinines kalbas. Šiuolaikiniai programuotojai dažniausiai programuoja tik aukšto lygio kalbomis. Antra, šiais laikais yra daug įvairių platformų, daugybė rūšių procesorių, todėl neįmanoma rankiniu būdu optimizuoti tiek variantų. Ir programuotojai, ir kompiliatoriai ne visada pasinaudoja naujausiomis ir efektyviausiomis instrukcijomis, pritaikytomis išnaudoti naujausių procesorių galimybes. Trečia, dažniausiai norima, kad programinė įranga veiktų daugelyje sistemų, todėl optimizavimas konkrečiai platformai netenka prasmės.

2.2.4. Optimizavimas vykdymo metu

Pasinaudoję esamo laiko kompiliatoriais (angl. „just in time compilers“) ir Asemblerio kalba, programuotojai gali atlikti optimizavimą programos vykdymo metu. Šie dinaminiai

[7] Informacija apie Asemblerio kalbą http://lt.wikipedia.org/wiki/Asemblerio_kalba

kompiliatoriai lenkia statinių kompiliatorių sugebėjimus. Jų pavadinimas pasako, kad jie gali dinamiškai derinti parametrus priklausomai nuo programoje vykdomų operacijų – įėjimo reikšmių ar kitų faktorių. Tai suteikia daugiau įvairovės ir laisvės, nes galima matyti, kaip realiu laiku kinta programos pajėgumai pakeitus vienus ar kitus parametrus.

2.2.5. Nuo platformos priklausomas ir nepriklausomas optimizavimas

Kodo optimizavimo būdai taip pat gali būti skirstomi į priklausomus nuo platformos ir nepriklausomus. Pastarieji veikia daugelyje pagrindinių platformų. Priklausomumą galima panaudoti kuriant ypač gerus sprendimus. Žinoma, tai, kas sukuriama konkrečioje aplinkoje konkrečiai mašinai arba netgi konkrečiam procesoriui, vargiai pavyksta panaudoti kitur. Tenka kurti naują programos versiją naujais sistemai (apie tai rašoma 2.2.3 skyrelyje „Optimizavimas mašininio kodo lygiu“).

Pavyzdžiui, kompiliavimo metu (bendru atveju) stengiamasi sumažinti bendrą instrukcijų kiekį ir kelią, reikalingą programai ar operacijai įvykdyti, taip pat sumažinti bendrą naudojamos atminties kiekį. Nuo platformos priklausomo kompiliavimo metu papildomai naudojamos tokios technologijos: instrukcijų planavimas (angl. scheduling), gijos instrukcijų lygiu, gijos duomenų lygiu, spartinančiosios atminties optimizavimas. Tai papildomi optimizavimo būdai, kurie gali skirtis net tos pačios architektūros procesoriuose.

2.2.6. Kamščių pašalinimas

Duomenų kamštis (angl. bottle neck[8]) inžinerijoje yra sistemos dalis, kuri smarkiai riboja sistemos spartą. Tokia sistemos dalimi gali būti ir viena kodo eilutė ir serverių masyvas. Viskas priklauso nuo to, apie kokio dydžio sistemą yra kalbama. Pavadinimas „butelio kakliukas“ suteiktas dėl to, kad realybėje butelio kakliukas riboja skysčio tekėjimą, kaip inžinerijoje jis riboja sistemos galimybes.

Tokią sistemos ribojančią vietą galima pavadinti ir kritiniu keliu, kurio pralaidumas yra pats mažiausias sistemoje. Tokių kritinių sistemos vietų galima išvengti jau projektavimo etape (žiūrėti skyrių 2.1.1 „Optimizavimas projekto lygyje“). Jei visgi išvengti nepavyksta,

[8] Apibrėžimas paimtas iš http://en.wikipedia.org/wiki/Bottleneck_(engineering)

svarbu mokėti nustatyti, kur kamščiai susidaro, ir mokėti juos pašalinti. Kaip ir minėta prieš tai, kamščiu gali būti procesorius, komunikavimo linija, ryšys ar bet kuri kodo vieta.

Norint nustatyti kritines vietas (karštuosius taškus) neužtenka pasikliauti vien programuotojo intuicija ar tikėjimu. Geriausia naudoti analizavimo įrankius, kurie gali nustatyti kurios kodo vietos vykdymas užtrunka daugiausiai laiko ar sunaudoja daugiausiai resursų.

Čia taip pat galioja Pareto principas, kuris teigia, jog 20% kodo patobulinimas gali duoti 80% galimų rezultatų. Informatikoje Pareto principas gali būti pritaikytas resursams optimizuoti: 80% resursų sunaudoja 20% sistemos operacijų. Kartais panaudojama modifikuota Pareto taisyklė 90/10, kuri teigia, kad 90% operacijų vykdymo laiko yra sugaištama vykdant 10% kodo (apie šias taisykles daugiau kalbėta 2.1.4 skyrelyje „Automatinis ir rankinis optimizavimas“). Ta dešimtoji dalis yra kritinė sistemos vieta.

Kamščių problemas gali sukelti ir naudojami algoritmai. Sudėtingi algoritmai ir sudėtingos duomenų struktūros gali gerai veikti, kai turime daug elementų, ir prastai veikti su mažai elementų, nes sudėtingų mechanizmų vykdymas gali būti žymiai ilgesnis, nei laikas vienam elementui apdoroti. Todėl kartais paprasti algoritmai su mažais duomenų kiekiais yra žymiai efektyvesni.

Kamščių problemą galima bandyti spręsti kai kuriais atvejais padidinus darbinės atminties kiekį ir tikintis, jog programos veiks sparčiau. Tai veikia tada, kai išties atminties kiekį galima išnaudoti (kaip ir kelių branduolių procesorius), pavyzdžiui, norint faile surasti simbolių seką galima elgtis dvejopai:

1) skaityti failą po vieną eilutę į atmintį ir joje ieškoti reikiamos simbolių sekos. Suradus stabdyti algoritmą ir grąžinti surastą eilutę,
2) nusiskaityti visą failo turinį į atmintį ir skaityti atmintyje esantį failą (po eilutę ar visą iškart) ieškant simbolių sekos. Suradus stabdyti algoritmą ir grąžinti surastą eilutę.

Pirmu atveju, ilgiausiai užtruks nuskaityti iš failo operacijas, nes jų gali būti atliekama labai daug. Operacijos atmintyje yra atliekamos žymiai sparčiau, todėl simbolių paieška abiem atvejais užtruks panašiai. Jei failas turės labai daug eilučių, antru atveju mums gali pritrūkti darbinės atminties. Jei jos turime pakankamai, simbolių paieška antruoju atveju veiktų sparčiau, nes viskas talpinama darbinėje atmintyje.

Tokių sistemoje apdorojamų failų gali būti tūkstančiai. Panaudojus neefektyvų algoritmą šioje vietoje galima prarasti minutes. Visada patariama apgalvoti ir atsižvelgti į esamą situaciją: duomenų kiekius ir turimus resursus.

2.2.7. SQL užklausų optimizavimas

Viena iš galimų spartos problemų šiuolaikinėse sistemose yra neoptimizuotos užklausos [10]. Užklausoms optimizuoti (angl. SQL statements tuning [11]) yra naudojamos tam tikros metodikos. Šiuo metu pasaulyje yra standartinės ir truputį nuo standarto nutolusios SQL užklausos. Tad įvairios metodikos gali būti taikomos visoms standartinėms užklausoms ir kai kurioms nestandartinėms SQL variacijoms.

Šiame skyriuje apžvelgsime šiuos punktus:

- SQL rašymas
- SQL standartai
- Indeksų naudojimas
- Oracle užklausų optimizavimas
- FROM ir WHERE sąlygų patobulinimai
- Sub-užklausos (angl. sub-selects) ir lentelių sujungimai (angl. joins)
- SQL trasavimas (ang. SQL trace)
- Bendri patarimai užklausoms optimizuoti

SQL rašymas

Nors dauguma toliau aprašomų principų tinka visoms SQL užklausoms ir jas apdorojančioms DBVS, bet pateikiamuose pavyzdžiuose naudosime Oracle tipo užklausas.

Nuo ko pradedame rašyti SQL užklausą?

- Primas žingsnis: kokios informacijos mums reikia – pasirenkami stulpeliai
- Antras žingsnis: kur ji yra – pasirenkamos lentelės
- Trečias žingsnis: rašoma užklausa

Select columns

From tables

Where ... (joins, filters, subqueries)

Užklausa jau parašyta. Ar tai viskas? Tai dar ne pabaiga. Rezultatai gauti įvykdžius užklausą, bet kokia gautų rezultatų kaina? Yra daug būdų gauti tuos pačius rezultatus, bet yra tik vienas greičiausias kelias tai padaryti. Galima sakyti yra 1000 galimų patobulinimų vienai vienintelei užklausai.

Dažnai neefektyvios užklausos labai smarkiai sumažina bendrą sistemos spartą. Nepatogumą jaučia ir sistemos vartotojas, priverstas laukti rezultatų. Geriausias sprendimas yra, kai programuotojai ir duomenų bazių administratoriai kartu peržiūri užklausas ir derina pačią aplikaciją su DBVS.

Kad derinimas vyktų teisingai, iš pradžių būtina atsakyti į šiuos į klausimus:

- Kiek ilgai yra per ilgai?
- Ar naudojamas optimalus išrinkimo kelias?
- Kaip dažnai užklausa bus vykdoma?
- Kada ji bus vykdoma?

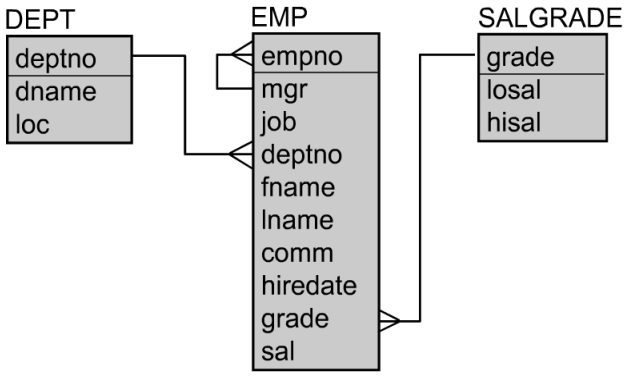
SQL standartai

Labai svarbu optimizuojant peržiūrėti užklausų standartiškumą. Tai yra svarbu, nes standartizuotos užklausos yra lengviau skaitomos, taip pat lengvesnis jų palaikymas ateityje (angl. maintainability). Standartinės užklausos yra vykdomos greičiau. Nors Microsoft ir Oracle kompanijos palaiko standartines užklausos, bet yra susikūrusios ir modifikuotas užklausų kalbas. Jei tik galima, reikia naudoti gryną SQL kalbą.

Pavyzdys. Kurios iš šių užklausų yra tokios pat?

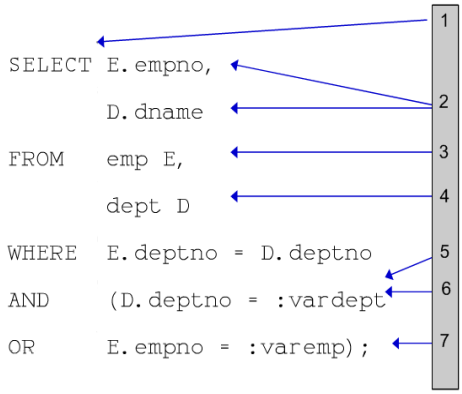
- *SELECT LNAME FROM EMP WHERE EMPNO = 12;*
- *SELECT lname FROM emp WHERE empno = 12;*
- *SELECT lname FROM emp WHERE empno = :id;*
- *SELECT lname FROM emp*
  WHERE empno = 12;

Teisingas atsakymas – visos užklausos yra skirtingos. Tarpai, didžiosios ir mažosios raidės, kintamieji ir konstantos – štai kas tarpusavyje skiria šias užklausas. Jei šios užklausos būtų užrašytos standartiškai, praktiškai jos visos būtų vienodos.



2 pav. *Duomenų lentelės ir sąryšiai tarp jų toliau naudojamuose pavyzdžiuose*

Standartinės užklausos pavyzdys:



3 pav. *Standartinės SQL užklausos pavyzdys.*

- 1 – raktiniai žodžiai lygiuojami pagal kairę ir rašomi didžiosiomis raidėmis
- 2 – stulpeliai rašomi iš naujos eilutės
- 3 – naudojami suprantami sutrumpinimai
- 4 – tarpelis tarp žodžių turi būtų tik 1, ne daugiau
- 5 – naudojami kintamieji
- 6 – raktažodžiai AND/OR rašomi iš naujos eilutės
- 7 – jokių tuščių tarpų prieš / po skliaustų

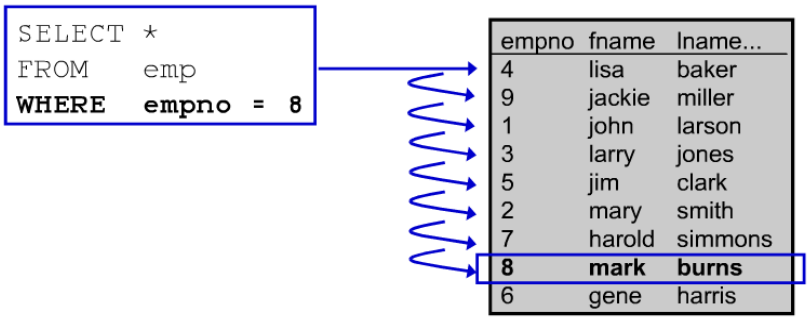
Šios kelios pavaizduotos taisyklės yra logiškos ir mažai skiraisi nuo įprastų teksto rašybos taisyklių. Ar sunku jų laikytis?

Indeksų naudojimas

Indeksai yra naudojami paspartinti eilučių išrinkimą iš duomenų lentelių. Indeksą sudaro indeksacijos reikšmė (raktas) ir nuoroda į eilutę duomenų lentelėje. Indeksų kiekvienai lentelei galima sukurti ne vieną, o kelis, ir jie nebūtinai bus unikalūs.

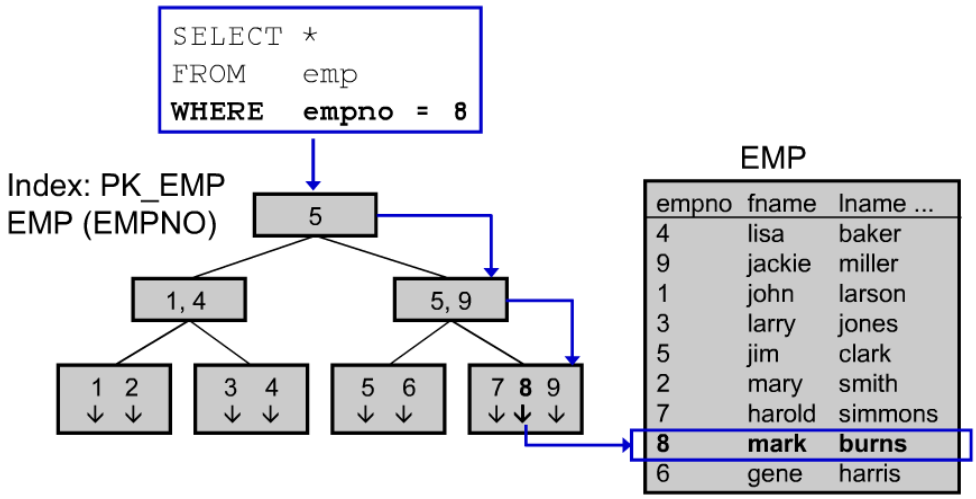
Pasirinkti, kuriuos stulpelius reikia indeksuoti, galima taip: jei stulpelis rašomas po WHERE raktažodžio, jis kandidatas būti suindeksuotas; jei stulpelis yra indeksuotas, vykdant užklausą pagal indeksą galima išrinkti reikiamas eilutes ir nebereikia eiti per visas lentelės eilutes. Deja, jei užklausa parašyta blogai, nors stulpelis ir yra indeksuotas, duomenų bazė skenuos visas duomenų lentelės eilutes. Todėl indeksų sukūrimas dar nėra spartos problemos sprendimas. Žinant, kurie stulpeliai yra indeksuoti, galima rašyti efektyvesnes užklausas.

Žemiau pateiktame paveikslėlyje matyti, kaip vykdoma užklausa, kai duomenų lentelės EMP stulpelis EMPNO neturi indekso:



4 pav. *Užklausos vykdymas, kur indeksai nenaudojami*

Matome, kad einama per visas eilutes, kol surandama užklausos kriterijus atitinkanti eilutė. Viskas vyktų žymiai greičiau, jei stulpelis EMPNO būtų indeksuotas ir būtų vykdoma tokia pati užklausą kaip prieš tai:



5 pav. *Užklausos vykdymas, kur indeksai naudojami*

Matome, kad šiuo atveju užklausos vykdymas paspartėja apie 50%.

Indeksavimu negalima piktnaudžiauti, t.y. indeksuoti kiekviena stulpelį lentelėje. Tai sukeltų spartos sumažėjimą, kai būtų vykdomos įterpimo, atnaujinimo ir trynimo užklausos. Praktika rodo, kad mažoms lentoms nereikia indeksų, nes jos gali būti sparčiai skenuojamos ir taip. Taip pat užklausos, grąžinančios daug eilučių (>5%) iš lentelės gali būti vykdomos greičiau, jei indeksai yra nenaudojami.

Pažvelgus į kitą pavyzdį (6 pav.), kuris parodo, kaip rašyti užklausą, naudojant arba nenaudojant indeksų, matyti, kad indeksai yra sukurti stulpeliams EMPNO ir DEPTNO (lentelių schema 1 pav.).

```
SELECT *
FROM   emp
WHERE  deptno = 10;
```

Indeksas nenaudojamas

```
SELECT *
FROM   emp
WHERE  empno > 0
AND    deptno = 10;
```

Indeksas naudojamas

6 pav. *Pavyzdys kaip priversti užklausą naudoti indeksaciją*

Taip yra todėl, kad stulpelis EMPNO yra pagrindinis indekse, ir antrojoje užklausoje (6 pav.) jis yra kviečiamas, o pirmoje užklausoje ne. Panašus pavyzdys kaip užklausoje nenaudoti indeksų pateikiamas žemiau:

```
SELECT *
FROM   dept
WHERE  deptno != 0;
...    deptno NOT = 0;
...    deptno IS NOT NULL;

SELECT *
FROM   dept
WHERE  deptno > 0;
```

Indeksas nenaudojamas

Indeksas naudojamas

7 pav. *Pavyzdys kaip priversti užklausą nenaudoti indekso*

Raktažodis NOT pašalina indeksuota stulpelį DEPTNO ir indeksas nėra naudojamas.

Oracle užklausų optimizavimas

Oracle turi galimybę pakeisti užklausos vykdymo tvarką. Pažvelgus į pavyzdžius žemiau, pirmasis pavyzdyje matyti (8 pav.), kaip kuo greičiau grąžinamos pirmosios rezultatų eilutės (ne visi rezultatai). Antrasis pavyzdys (9 pav.) rodo, kaip priversti naudoti konkretų indeksą užklausos vykdymo metu.

```
SELECT /*+ FIRST_ROWS */ empno
FROM   emp E
       dept D,
WHERE  E.deptno = D.deptno;
```

8 pav. *FIRST_ROWS naudojimo pavyzdys*

```
SELECT /*+ INDEX (E idx_hiredate) */ empno
FROM   emp E
WHERE  E.hiredate > TO_DATE('01-JAN-2000');
```

9 pav. *Kaip priversti naudoti konkretų indeksą užklausos vykdymo metu*

FROM ir WHERE sąlygų patobulinimai

Taip pat galima tobulinti užklausos FROM ir WHERE dalis. Paveikslėliuose 10 ir 11 pavaizduota, kad FROM sakinyje paskutinė lentelė turi būti valdančioji (angl. driving table). Tai lentelė, kuri Oracle DBVS sujungiant keletą lentelių, yra apdorojama pirmiausia.

```
SELECT *
FROM   dept D,  -- 10 rows
       emp E    -- 1,000 rows
WHERE  E.deptno = D.deptno;
```

10 pav.*EMP yra valdančioji lentelė*

```
SELECT *
FROM   emp E,  -- 1,000 rows
       dept D -- 10 rows
WHERE  E.deptno = D.deptno;
```

11 pav. *DEPT yra valdančioji lentelė*

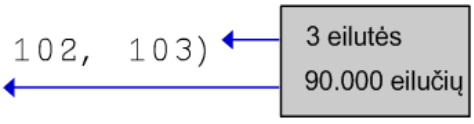
Kai užklausa sujungia duomenis iš 3 ar daugiau lentelių, valdančioji lentelė turėtų būti ta, kuri turi daugiausia bendrų stulpelių (ryšių) su kitomis lentelėmis. Pavyzdys 12 paveikslėlyje:

```
SELECT *
FROM   dept D,
       salgrade S,
       emp E
WHERE  E.deptno = D.deptno
AND    E.grade = S.grade;
```

12 pav. *EMP lentelė siejasi su dviem kitomis lentelėmis, todėl šiuo atžvilgiu ji yra valdančioji.*

Tinkamai naudojant WHERE sakinį galima atmesti daug netinkamų eilučių ir taip sutrumpinti užklausos vykdymo laiką:

```
SELECT *
FROM   emp E
WHERE  E.empno IN (101, 102, 103)
AND    E.deptno > 10;
```



13 pav. *Iš pradžių atmetame visas eilutes, kur stulpelis empno nelygus 101, 102 ar 103. Taip atmetama daugiausia eilučių. Jei darytume atvirkščiai, reiktų eiti per 90000 eilučių ir tikrinti, ar empno lygus vienam iš skaičių.*

Rašant užklausas taip pat reiktų atkreipti dėmesį į AND ir sub užklausas. Jų vykdymas netinkamoje vietoje labai smarkiai padidina laiko sąnaudas, reikalingas užklausai įvykdyti. Pavyzdį žiūrėti 14 pav.

```
SELECT *                                          CPU = 156 sec
FROM   emp E
WHERE  E.sal > 50000
AND    25 > (SELECT COUNT(*)
             FROM   emp M
             WHERE  M.mgr = E.empno)
```

☞

```
SELECT *                                          CPU = 10 sec
FROM   emp E
WHERE  25 > (SELECT COUNT(*)
             FROM   emp M
             WHERE  M.mgr = E.empno)
AND    E.sal > 50000
```

14 pav. *Kai yra naudojama AND ir sub užklausa, sub užklausą reikia vykdyti pirmiau*

Atvirkščia metodika yra taikoma OR ir sub užklausoms:

```
SELECT *                                          CPU = 100 sec
FROM   emp E
WHERE  25 > (SELECT COUNT(*)
             FROM   emp M
             WHERE  M.mgr = E.empno)
OR     E.sal > 50000
```

```
SELECT *                                          CPU = 30 sec
FROM   emp E
WHERE  E.sal > 50000
OR     25 > (SELECT COUNT(*)
             FROM   emp M
             WHERE  M.mgr = E.empno)
```

15 pav. *Kai yra naudojama OR ir sub užklausa, sub užklausą reikia vykdyti paskiausiai*

Naudojant WHERE sąlygą pirmiausia reikia atlikti filtravimą, o po to sujungimą (angl. join):

```
SELECT *
FROM   emp E,
       dept D
WHERE  (E.empno = 123
OR     D.deptno > 10)
AND    E.deptno = D.deptno;
```

16 pav. *Pirma vykdomas filtravimas, o po to sujungimas*

SQL trasavimas

Svarbiausias dalykas optimizuojat užklausas, tiksliai nustatyti, kiek laiko trunka užklausos vykdymas, taip pat sužinoti, kokiais etapais ir kaip ji yra vykdoma.

Trasavimas atliekamas tokiais žingsniais:

1. Įjungiamas trasavimas

```
ALTER SESSION SET SQL_TRACE TRUE;
```

17 pav.*Trasavimo įjungimas*

2. Įvykdoma užklausa:

```
SELECT E.*
FROM   emp E,
       dept D
WHERE  D.dname = 'SALES'
AND    D.deptno = E.deptno;
```

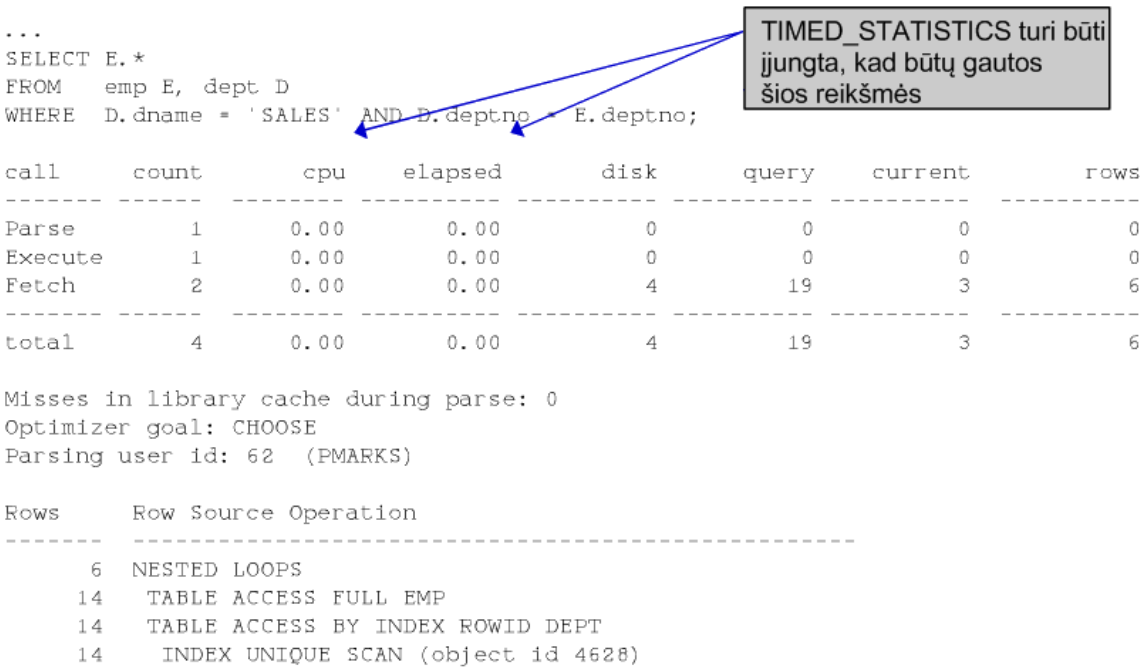
18 pav.*Vykdoma užklausa*

3. Trasavimas išjungiamas:

```
ALTER SESSION SET SQL_TRACE FALSE;
```

19 pav.*Trasavimo išjungimas*

4. Sukuriamas rezultatų failas duomenų bazės serveryje. Jo vaizdas:



20 pav.*Trasavimo rezultatai*

Iš trasavimo rezultatų matyti vykdyta užklausa ir jos statistika. Matyti, kokios komandos buvo vykdomos ir kiek resursų jos sunaudojo. Pagal vykdomų komandų kiekį ir tipą galima spręsti, ar užklausa yra parašyta tinkamai.

Bendri patarimai užklausoms optimizuoti

Kaip teigia Sajal Dam [10] ir Dan Tow [11], yra dar keletas bendrų patarimų rašant SQL užklausas. Vienas svarbiausių ir labai dažnai naudojamas patarimas yra pateiktas 21 pav. Kiti patarimai aptariami prieš tai minėtose knygose.

```
SELECT COUNT(empno)
FROM   emp;

SELECT COUNT(*)
FROM   emp;
```

~ 50 % greitesnė

21 pav.*Teisingai naudojama COUNT funkcija gali būti apie 50% greitesnė*

2.3. Techninės sistemos dalies optimizavimas

2.3.1. Sistemos spartos derinimas

Spartos derinimas (Performance Tuning [6]) yra ne kas kita kaip sistemos spartos gerinimas. Paprastai kalbant apie sistemos spartą, omeny turimos kompiuterio aplikacijos, bet taip pat galime kalbėti ir taikyti tuos pačius metodus ekonomikai, biurokratijai ir kitoms sudėtingoms sistemoms. Kam to reikia?

Sistemos spartos derinimą sudaro šie žingsniai:

1. Įvertinti problemą ir numatyti tinkamą veikimą (skaitine verte);
2. Išmatuoti sistemos spartą;
3. Identifikuoti sistemos dalį, kuri yra kritinė spartos požiūriu. Ta dalis vadinama „butelio kakliuku“;
4. Pakeisti (pataisyti) sistemos dalį, kad pašalintume „butelio kakliuką“;
5. Išmatuoti sistemos spartą po atliktų veiksmų.

Techninės įrangos sparta dažniausiai priklauso nuo to, kokie įrenginiai ir kaip yra naudojami. Bet galima išskirti šiuos teorinius metodus:

Apkrautumo lygio suvienodinimas (angl. Load balancing and computing)

Sistema dažnai susideda iš nepriklausomų komponentų, kurie sugeba apdoroti užklausas. Jeigu vienas komponentas (arba mažas jų kiekis) apdoroja visas užklausas, o kiti nieko neveikia, tai daugumos komponentų laikas yra tiesiog švaistomas veltui. Tolygus užklausų paskirstymas visiems komponentams vadinamas apkrautumo lygio suvienodinimu. Jis gali pagerinti bendrą sistemos spartą.

Paskirstytieji skaičiavimai (angl. Distributed computing)

Tai dar vienas būdas didinti sistemų našumą ir spartą. Šis būdas remiasi tuo, kad leidžia lygiagrečiai atlikti įvairius veiksmus vienu metu. Tam panaudoti gali būti keli procesoriai, vykdantys kelias operacijas vienu metu lygiagrečiai. Kai operacijos yra vykdomos tuo pačiu metu, svarbu sinchronizuoti procesų veiksmus, kad būtų gauti teisingi rezultatai.

Šiuo metu pasaulyje populiarėjantys daugiabranduoliniai procesoriai taip pat gali būti naudojami šiam metodui įgyvendinti. Kiekvieną branduolį galime traktuoti kaip atskirą procesorių, kuris turi savo dažnį. Sudėtingi tinkliniai skaičiavimai yra atliekami didelių klasterių pagalba. Deja, dažnai tokie skaičiavimai sukelia ilgesnius sistemos uždelsimus arba vėlavimus. Pavyzdžiu gali būti duomenų bazių sistemos, nes užklausų apdorojimas naudojant lygiagrečius procesus yra labai sudėtingas, brangus ir ganėtinai neefektyvus.

Savireguliacija (angl. self-tuning)

Savireguliuojanti sistema sugeba pati optimizuoti savo vidinius procesus ir parametrus. Šiuo metodu dažnai naudojasi telekomunikacijų bendrovės, kurios dinamiškai keičia savo operuojamų sistemų parametrus priklausomai nuo esamos situacijos, kad pasiektų maksimalią spartą. Tokioms sistemoms būtinas grįžtamasis ryšys, nes sistema turi pati nuspręsti geru ar blogu keliu yra tobulinama. Kompiuterių sistemos, kurios turi savireguliacijos mechanizmus:

- TCP (angl. Transfer Control Protocol)
- Microsoft SQL Server (tik naujesnės versijos)
- FFTW (angl. Fastest Fourier Transform in the West)
- ATLAS (angl. Automatically Tuned Linear Algebra Software)
- Libtune (angl. Tunables library for Linux)
- PhiPAC (angl. Self Tuning Linear Algebra Software for RISC)

Žymus Amerikos informatikos mokslininkas Jack Dongarra[9] teigia, kad tokios sistemos savo spartą gali padidinti iki 300%.

2.3.2. Duomenų bazės spartos derinimas

Duomenų bazės spartos derinimas [7] atliekamas panašiais metodais kaip ir minėta prieš tai esančiame skyriuje. Čia naudojami ir keli konkretūs metodai būtent duomenų bazėms arba DBVS.

I/O derinimas (angl. I/O tuning)

[9] Daugiau informacijos http://www.netlib.org/utk/people/JackDongarra

Tikslas yra optimizuoti rašymo ir skaitymo apkrovas. I/O operacijos laiko požiūriu yra pačios brangiausios [8]. Paprastai tai ir yra pirmasis „butelio kakliukas“ duomenų bazės veikloje.

Norint sumažinti operacijų sekundes, dažnai kuriami RAID diskų masyvai. Jie sumažina skaitymo ir rašymo trukmę. Naudojamos lentelės ir indeksai gali būti patalpinti failų sistemoje keliomis kopijomis (į atskirus diskus). Tada jų iškvietimas gali būti žymiai spartesnis: vienu metu galima gauti informaciją iš kelių failų, esančių toje sistemoje. Pavyzdžiui, vienu metu skaitant iš keleto šaltinių sparta gali siekti iki 150% lyginant su vieno disko sparta. Taip subalansuojamas sistemos I/O ir išvengiama užklausų rikiavimo į eilę.

Duomenų bazės priežiūra (angl. Database maintenance)

Duomenų bazės priežiūra sudaro atsarginių kopijų darymas, skilčių statistikos atnaujinimas ir duomenų defragmentacija. Visa tai prisideda prie našumo gerinimo.

Labai apkrautose duomenų bazėse log failas auga labai sparčiai, todėl būtina nuolat valyti senas transakcijas, kad būtų laisvos vietos naujoms. Mažesnis log failas leidžia greičiau veikti atsarginių kopijų darymo mechanizmams, todėl sutrumpėja duomenų bazės neveikimo laikas kopijos metu. Defragmentacija padidina duomenų pasiekiamumo efektyvumą, nes duomenis sudėlioja „tvarkingai“ naudojamoje atmintyje.

Sutaupomas laikas yra žymiai didesnis, nei laikas, reikalingas duomenų bazei prižiūrėti. Tai rodo, kad keletas nesudėtingų veiksmų gali turėti įtakos bendrai sistemos spartai.

2.3.3. Duomenų saugyklos kūrimas

Vienas iš būdų spręsti didelių duomenų bazių problemas yra kurti duomenų saugyklas. Bill Inmon[10] duomenų saugyklą (angl. Data warehouse [5]) aprašė tokiomis sąlygomis:

- Orientuota į esmę – duomenų bazė yra organizuotos struktūros, taigi visi elementai, susiję su realaus pasaulio įvykiu arba objektu, yra susiję;

[10] Bill Inmon laikomas duomenų saugyklos kūrėju ir bendraautoriu kuriant „Corporate Information Factory“ ir „Government Information Factory“, taip pat kaip ir kuriant naujos kartos duomenų saugyklos architektūrą (DW 2.0). Daugiau informacijos http://www.inmoncif.com/about/

- Nepastovi laike – duomenų pasikeitimai yra sekami ir išsaugomi duomenų bazėje, taigi gali būti pateikiamos ataskaitos, kurios rodo pakitimus laikui bėgant;
- Nepažeidžiama – duomenys duomenų bazėje nėra prarandami: jų niekas netrina, neužrašo ant viršaus; duomenys yra statiški ir saugomi iki tol, kol bus panaudoti;
- Integruota – duomenų bazė turi duomenų iš visų organizacijos naudojamų aplikacijų, ir bazė sistemingai papildoma naujais duomenimis.

Taip, tai panašu į duomenų bazę. Pavyzdžiui, duomenų saugyklos naudojamos surasti savaitės dieną 2005 m., per kurią buvo parduota daugiausia gaminių arba kurią savaitę sirgo daugiausia įmonės darbuotojų per 2000 – 2007 metus. Duomenis išgauti iš duomenų saugyklos galima naudojantis standartinėmis sąsajomis (per kuriuos bendraujame su duomenų bazėmis) arba duomenų gavyba. Tad kuo saugyklos geresnės?

Duomenų saugyklos buvo sukurtos apie 1985 m. dėl vis didėjančio informacijos poreikio ir poreikio išanalizuoti gaunamą informaciją. Tuo metu operuojamos sistemos nesugebėdavo atlikti šių veiksmų. Galima teigti, kad duomenų saugyklos yra optimizuotos ataskaitoms kurti ir duomenims analuoti (tam naudojama OLAP technologija). Todėl užklausos yra atliekamos sparčiau, nei įprastoje duomenų bazėje.

Taip pat duomenų saugyklos gali surinkti duomenis iš įvairiausių šaltinių, tokių kaip serveriai, darbo vietų kompiuteriai ir vidinių informacinių sistemų į vieną vietą. Šis gebėjimas kartu su draugiška vartotojui aplinka ir nepriklausomybe nuo išorės poveikių leido išpopuliarėti šio tipo sistemoms.

Bėgant laikui, tobulėjant techninei įrangai ir atsirandant vis daugiau vartotojo reikalavimų (pvz. greitesnis duomenų užkrovimo ciklas), duomenų saugyklos evoliucionavo keliais etapais:

- neprisijungusi veikianti duomenų bazė (Off line Operational Databases) – duomenų saugykla šioje stadijoje sukuriama nukopijuojant veikiančią duomenų bazę (jos duomenis) į neprisijungusį serverį, kurio apkrautumas neturi įtakos bendram sistemos našumui;
- neprisijungusi duomenų saugykla (Off line Data Warehouse) – duomenų saugykla šioje stadijoje yra reguliariai atnaujinama (kasdien, kas savaitę, kas

mėnesį) ir duomenys išsaugojami atitinkama struktūra (reporting-oriented) – paruošti naudoti;

- realaus laiko duomenų saugykla (Real time Data Warehouse) – saugykla atnaujinama kiekvieną kartą įvykus įvykiui ar transakcijai;
- integruota duomenų saugykla (Integrated Data Warehouse) – šioje stadijoje duomenų saugykla naudojama darbams ir transakcijoms generuoti, kurios yra perduodamos į bendrą organizacijos sistemą ir naudojamos kasdien.

Siejant duomenų saugyklas su realiu pasauliu, duomenų saugyklos yra tarsi dideli informacijos organizacijų sandėliai, kuriuose dirba savi darbuotojai – procesai, aptarnaujantys ateinančius ir išeinančius klientus. Iš duomenų saugyklos duomenys gali būti perskirstomi į mažesnius sandėlius (angl. Retail stores) arba duomenų vitrinas (angl. Data mart), iš kurių duomenis pasiims atitinkami vartotojai, atliekantys tik tam tikrus veiksmus. Taigi duomenų saugyklą galime suprasti kaip duomenų sandėlį ir tiekėją mažesniems sandėliams.

Duomenų bazių trūkumai:

- ataskaitos kūrimo procesas sulėtina atsako laiką iš visos sistemos;
- duomenų bazės architektūra nėra optimizuota informacijai ir ataskaitoms analizuoti;
- dauguma organizacijų turi daugiau nei vieną operacinę sistemą, tai neleidžia sukurti mechanizmo, apimančio ataskaitų kūrimą įmonės mastu; tai apeiti galima nebent sujungiant visų padalinių duomenis į vieną sistemą ir kuriant ataskaitas jų pagrindu;

Duomenų saugyklų privalumai:

- duomenų saugykla leidžia vartotojui pasiekti didelę duomenų įvairovę paprastai ir greitai;
- vartotojai, priimantys sprendimus, gali gauti ataskaitas, rodančias tendencijas ir kryptis, kuriomis juda rinka;

- duomenų saugykla gali būti sistemos dalis arba dirbti kaip atskira sistema (tuo ji panaši į CRM[11] sistemas).

Duomenų saugyklų trūkumai:

- ateityje bus brangus duomenų saugyklų išlaikymas.
- gali labai greitai moraliai pasenti. Kainuoja pateikti optimalią informaciją organizacijai.
- dažnai yra ryškus skirtumas tarp duomenų saugyklos ir operacinių sistemų. Kartais reikia išvystyti dubliuojantį funkcionalumą.

Nors minėti privalumai yra ir duomenų bazėse, bet jos apima žymiai platesnius dalykus. Tuo tarpu duomenų saugyklos yra sukoncentruotos ties darbu su ataskaitomis ir analizėmis ir visą savo spartą skiria būtent šiam dalykui.

2.4. Analizės išvados

Tobulėjant šiuolaikinėms technologijoms, didėjant duomenų kiekiui, darosi vis sudėtingiau juos išanalizuoti ir daryti greitus, efektyvius ir teisingus sprendimus. Duomenų bazės jau peržengė terabaitines ribas ir žmogus jau tampa nepajėgus išanalizuoti visą duomenų gausą. Tokiame milžiniškame kiekyje informacijos gali slėptis ir strategiškai svarbi ir niekinė informacija. Tokios problemos paskatino atsirasti aukštos kokybės taikomiesiems paketams, programavimo įrankiams, duomenų analizės priemonėms, kurios padeda nepasimesti informacijos gausoje. Tai padidino ir vartotojų prieinamumą prie pažangiausių technologijų, atvėrė realaus laiko ir kitas analizės galimybes.

Šios apžvalgos metu padarytos išvados:

- ❖ duomenų gavyba padeda geriau panaudoti turimus duomenis ir greičiau bei tiksliau priimti sprendimus ir taip minimizuoti klaidų tikimybę. Tai vienas iš spartos gerinimo būdu;
- ❖ praktika rodo, kad programų optimizavimas kartais sukelia visai priešingą efektą, nei buvo tikėtasi. Ne visada pavyksta darant pakeitimus bet kuriame programos

[11] Daugiau http://en.wikipedia.org/wiki/Customer_relationship_management

kūrimo etape neįnešti pokyčių į jau išvystytą programos dalį. Tai gali pabloginti programos kokybę ir sustrigdyti vystymo procesą.

- Lygiagretieji skaičiavimai suteikia teorinį spartos pagerinimą. Bet, praktiškai įvertinus faktus (pinigai išleisti techninei įrangai, reikalingi papildomi resursai, programų modulių pritaikymas lygiagretumui, skaičiavimo valdymo sudėtingumas), pasiekta sparta atrodo neefektyvi;
- Duomenų saugyklos yra sukoncentruotos ties darbu su ataskaitomis ir analizėmis, todėl visą savo spartą skiria būtent šiam dalykui. Specializuota duomenų saugyklos versija – duomenų vitrina – pasižymi greitu priėjimu prie dažnai naudojamų duomenų, todėl vartotojas mažiau užtrunka gaudamas jam reikalingą informaciją. Šis efektyvumas pasiekiamas duomenų vitriną kuriant tik tam tikram panaudojimo atvejui.
- Kombinuojant skirtingas metodikas, galima pasiekti itin efektyvių spartos rezultatų. Visada reikia vengti nesuderinamų dalykų ir saugotis „butelio kakliuko“ efekto.

Gerai, jeigu mes turime didelius kompiuterinius resursus, labai gerą programinę įrangą ir mažai klientų (mažai duomenų). Dažniau susiduriame su tokiais atvejais, kai turime tik vieną iš paminėtų dalykų. O kartais ir nė vieno. Todėl reikia stengtis tobulinti sistemas, jas optimizuoti ir pasiekti maksimalių spartos galimybių.

3. PROGRAMŲ SISTEMOS GREITAVEIKOS UŽTIKRINIMO METODIKA

3.1. Greitaveikos užtikrinimo procesas

Pirmasis žingsnis ir pirmoji klaida sprendžiant spartos problemas yra jų sprendimas ne laiku ir ne vietoje. Pradėti spartos gerinimo procesą neturint duomenų sistemoje yra beviltiška. Duomenys sistemoje sukuria realias apkrovas ir padeda išskirti problemines vietas ir parinkti atitinkamus sprendimus joms pašalinti. Be žinojimo kas programinėje įrangoje veikia blogai (kuris dažniausiai atsiranda po pirmųjų realių testavimų) sunku imtos optimizuoti sistemos dalis, nes tas gerinimas gali galutiniame rezultate nesimatyti visai.

Turint ribotus laiko resursus svarbu sutelkti dėmesį į tas sritis, kurios turi didžiausią potencialą atpirkti įdėtas pastangas ir duoti rezultatus [13]. Juk neverta investuoti ten, kur grąža nėra aiški. Norint paspartinti sistemą reikia turėti aiškų planą, žinoti ką reikia pataisyti ir kuo tai bus naudinga. Neteisingai įvertinus probleminę sritį visos dedamos pastangos neduos norimų rezultatų.

Toliau aprašyti žingsniai leidžia greitai ir tiksliai identifikuoti spartos problemas sistemoje, sutelkti pastangas būtent į tas sritis, kurios turi didžiausią potencialą tobulinimui ir įvertinti potencialių patobulinimų santykinę naudą.

3.1.1. Optimizacijos tikslų nustatymas

Pirma reikia išsiaiškinti ko norima pasiekti. Daugelis projektų neturi gerai apibrėžtų spartos reikalavimų ar tikslų. Reikalavimas „greitai kaip tik įmanoma“ skamba kvailai ir nėra naudingas. Kaip tada sužinoti kada (ar?) reikalavimai įvykdyti?

Pirmajame žingsnyje reikia tiksliai ir kiekybiškai apsibrėžti optimizavimo tikslus. Tikslus galima apibrėžti keliais būdais:

- numatyti konkrečių operacijų vykdymo laikus
- nusakyti pralaidumo matmenis
- numatyti sunaudojamų resursų apribojimus

Pavyzdžiui, ataskaitos suformavimo laikas turėtų būti mažesnis nei 2 sekundės, kai sistema naudojasi ne mažiau 1000 vartotojų.

Taip pat reiktų nepamiršti tikslus įvertinti ne tik šiandienai, bet ir numatyti ateičiai. Poreikiai ir vartotojų lūkesčiai bei jų kiekis dažnai auga bėgant laikui. Kad ateityje nekiltų tų pačių spartos bėdų, verta pasilikti rezervą. Juk sistema tikriausiai plėsis ir bus prilipdomas papildomas funkcionalumas, kuris veikiausiai paveiks esamą spartą.

3.1.2. Esamos situacijos analizė

Kitas etapas prasideda nuo problemų įvardijimo ir priežasčių identifikavimo. Reikia nustatyti kas ir kodėl jas sukelia. Šis etapas yra sunkus, jei nėra geros dokumentacijos (ypač UML diagramų).

Rekomenduojama dokumentaciją sukurti ar papildyti, kad būtų galima nustatyti kas sukelią trikdžius (pagal literatūros šaltinį [15]). Sistemos architektūros supratimas ir įvertinimas padeda pasiekti keliamus tikslus. Programinės įrangos tikslų supratimas, architektūros suvokimas ir veikimo procesų išmanymas leidžia:

- nustatyti ar architektūra leidžia išspręsti esamas bėdas
- nustatyti ar problemos gali būti išspręstos derinant ir konfigūruojant, bei kurias dalis derinti (nedaryti bandymų, nes jie neatneša sėkmės)
- apskaičiuoti spartos naudingumo išeigą, jei bus atlikti pakeitimai projekte
- įvertinti ar pakeitimai suteikia pakankamą spartos prieaugį

Architektūros patobulinimai suteikia didesnes spartos pagerinimo galimybes nei aplikacijos derinimas. Peržvelgus architektūrą galima identifikuoti ką būtų įmanoma pakeisti esamoje situacijoje. Tai galbūt visai ne esminiai patobulinimai, leidžiantis pakeisti kas yra daroma. Programos kodas nusako kaip tai yra daroma. Pavyzdžiui, sudėtingus skaičiavimus galima vykdyti foniniame rėžime ir vartotojas nebus priverstas laukti, o galės toliau naudotis sistema. Tai nepakeičia sistemos paskirties, o ir vartotojas tikriausiai bus labiau patenkintas.

Norint nustatyti pačias problematiškiausias vietas būtina su programa įvykdyti įvairaus lygio apkrautumo testavimus (vartojimo scenarijus). Išmatavus visą procesą, galima nustatyti mažiausią pralaidumą (srauto, skaičiavimų) turinčias vietas. Kai jau tampa žinoma kas ir kodėl kelia didžiausias bėdas, reikia jas šalinti. Tikėtina jog išsprendus didžiausias bėdas, pašalinus problematiškiausias vietas, bendra sistemos sparta pakils labiau, nei atsitiktinai taikant įvairius būdus.

3.1.3. Išsikeltų tikslų įgyvendinimo vertinimas

Dar prieš užsiimant spartos kliūčių šalinimų reikia įvertinti ar įmanoma pasiekti nusistatytus tikslus (skyrius 3.1.1 „Optimizacijos tikslų nustatymas“). Jei reikalingas spartos skirtumas yra nedidelis, tikriausiai prieš tai surastų problemų išsprendimas leist pasekti norimus rezultatus. Tam netgi gali užtekti aplikacijos konfigūravimo . Taip pat reikia įvertinti kiek kainuos pakeitimai. Tikėtina, kad pigios priemonės (automatinis optimizavimas) duos mažą naudą nei brangūs pakeitimai (programos perdarymas), kurie duoda didesnę grąžą [16]. Optimaliausia pasirinkti vidutines priemones ir pasiekti vidutinių rezultatų.

Nustačius esamas problemines vietas galima paskaičiuoti ar įmanoma pasiekti pagerėjimo jas ištaisius. Skaičiavimus būtina įvertinti realistiškai ir atsižvelgti į naudojamą techninę įranga. Neįmanoma architektūriškai išspręsti problemų, kurios susidaro dėl techninės įrangos kaltės. Tokias problemas galima spręsti tik gerinant turimą techninę įrangą arba didinant jos kiekį.

3.1.4. Siekiamų tikslų įgyvendinimo plano sukūrimas

Identifikavus visas problemas ir žinant priemones joms spręsti reikia susikurti įgyvendinimo planą. Tikslingiausia planą pradėti vykdyti nuo naudingiausių sprendimų [14]. Todėl ir įgyvendinimo žingsnius vertinga išsirikiuoti pagal atsiperkamumą, kad pritrūkus laiko ar lėšų planas neliktų neįgyvendintas. Sudarant sąrašą pagal tikėtinus spartos rezultatus ir kaštus jiems pasiekti galima atrinkti tinkamiausius būdus (vaistus) spartos problemoms išspręsti, kuriuos ir reikia pirma įgyvendinti.

Iš sąrašo pirmųjų pozicijų reikėtų pašalinti tas priemones, kurių turi pašalinės įtakos projektui. Tai galėtų būti rizikingos dalys, kurių įgyvendinimui reikia neišbandytos technologijos, ir tos, kurios reikalautų didelių pakeitimų esamojo struktūroje, taip įveliant naujų klaidų tikimybę.

Į plano sudarymą įeina ir reikalingų pastangų kiekio įvertinimas. Nereikia pulti vykdyti plano kuris yra neaiškiai apibrėžtas ar jam nėra reikalingų resursų. Sudarius planą turi būti aišku ką ir kada reikės padaryti ir ko iš to tikimąsi.

3.1.5. Greitaveikos patobulinimų ekonominis įvertinimas

Atlikus susiplanuotus apkeitimus yra naudinga surinkti duomenis apie atliktą procesą. Į tai įeina laikas ir kaštai skirti spartos analizei, laikas skirtas programinės įrangos modifikavimui (kodo rašymas, testavimas), pasikeitę techninės įrangos kaštai ir kiti kaštai atsiradę dėl optimizavimo proceso. Norint įvertinti atliktus patobulinimus taip pat reikia surinkti duomenis apie proceso atneštą naudą. Į tai galima įtraukti sutaupytas lėšas techninei įrangai, personalui, programos vertės padidėjimą ir panašiai.

Žinant apie atliktus pakeitimus galima peržvelgti projekto pradinę versiją ir pasižiūrėti ar buvo įmanoma tuos pokyčius numatyti iš anksto. Jei taip, vertinga palyginti kaštus ir naudą jei jie būtų buvę atlikti pradinėse stadijose. Tikėtina, kad kaštai būtų buvę mažesni.

3.1.6. Greitaveikos užtikrinimo proceso apibendrinimas

Programų kūrėjai žino, kad sparta gali nulemti programos kokybę į gerąją arba blogąją pusę. Prastas programos veikimo greitis atsiliepia patiems programų kūrėjams - visai industrijai tai kainuoja milijonus kasmet. Į tai įskaičiuojama visi atsirandantys papildomi kaštai skirti spartos gerinimui arba patirsiami nuostoliai dėl reputacijos rinkoje praradimo.

Beveik visi ankščiau ar vėliau susiduria su programinės įrangos spartos problemomis. Dabar programų kūrimo kompanijos yra priverstos „padaryti daugiau už mažiau“. Reikalavimai didėja, kūrimo trukmė mažėja, o sudėtingumas ir programų dydis auga. Norima, kad programinė įranga atitiktų šiandieninius reikalavimus (pavyzdžiui būtų pasiekiama per naršyklę), kad kai kuriais atvejais sukuria ypač išaugusius kaštus, nes priverčia perdirbti programą nuo pagrindų.

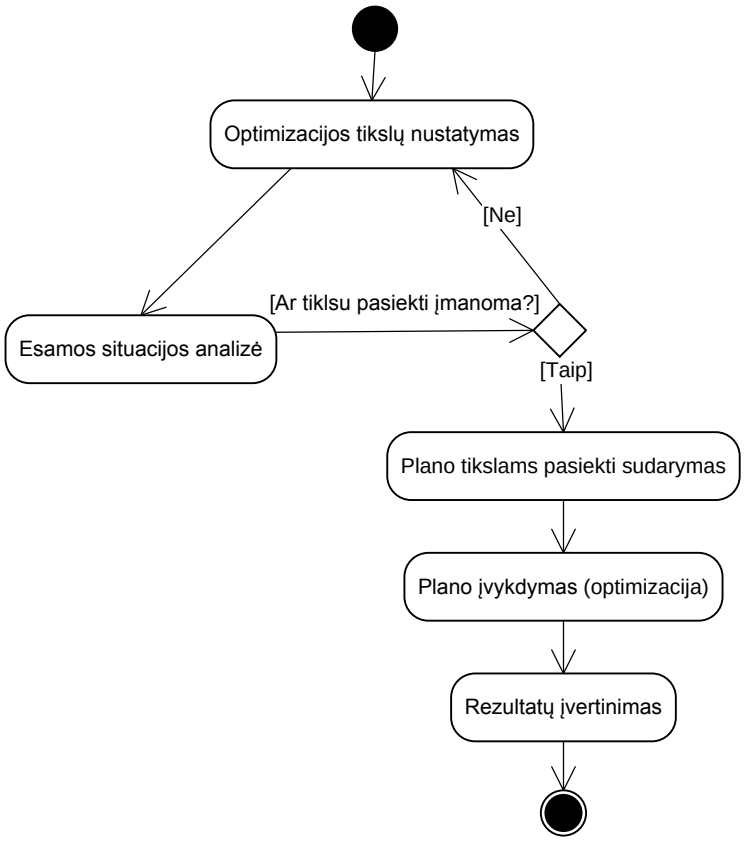
Itin kokybiškos programinės įrangos kūrimo metu, spartos problemos jau būtų numatytos ankstyvose stadijose ir ten pat išspręstos. Deja, realiame programinės įrangos kūrimo cikle dažnai naudojami būdai tik kaip greičiau ir pigiau ją sukurti.

Aprašytas procesas panaudotinas sudarytoje greitaveikos užtikrinimo metodikoje. Siūloma veiksmų atlikimo tvarka padeda siekti pagrindinio tikslo – optimizuoti sistemą. Toliau aprašyta metodika aprašo priemones kurios turi būti taikomos šio greitaveikos užtikrinimo proceso metu.

3.2. Programų sistemos greitaveikos užtikrinimo metodika

Siūloma programų sistemos greitaveikos užtikrinimo metodika leidžia sistemingai tobulinti programų spartą. Siūlomi žingsniai padeda greitai identifikuoti problemas, surasti tinkamus sprendimus joms pašalinti, paskirstyti pagal prioritetus išsikeltus tikslus ir tikslingai paskirstyti dedamas jėgas, kad būtų pasiekti geriausi rezultatai. Naudojantis šia metodika galima pasiekti gerų rezultatų esant mažiausiems kaštams.

Prieš tai buvo aprašytas proceso tvarka – veiksmai ir jų atlikimo seka kiekviename optimizavimo etape. Schematiškai proceso tvarka pavaizduota 22 pav.



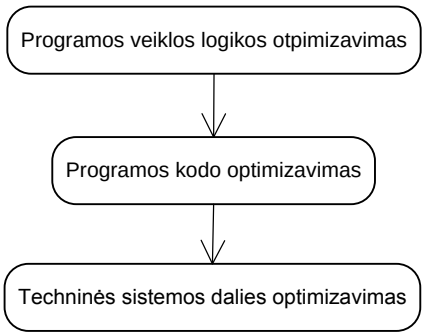
22 pav. *Greitaveikos užtikrinimo proceso veiklos diagrama*

Proceso žingsniai:

1. Nusistatyti optimizacijos tikslus
2. Išanalizuoti esamą situaciją
3. Įvertinti ar galima pasiekti išsikeltus tikslus
4. Sudaryti planą tikslams pasiekti

5. Įvykdyti užsibrėžtus tikslus
6. Įvertinti pasiektus rezultatus

Metodika – tai siūlomas priemonių rinkinys ir tvarka. Tad toliau aprašysime siūlomų priemonių rinkinį ir jų taikymo tvarką. Programų sistemos greitaveikos užtikrinimo metodiką galima išskaidyti į tris pagrindinius etapus (23 pav.):



23 pav. *Optimizavimo etapai pagal greitaveikos užtikrinimo metodiką*

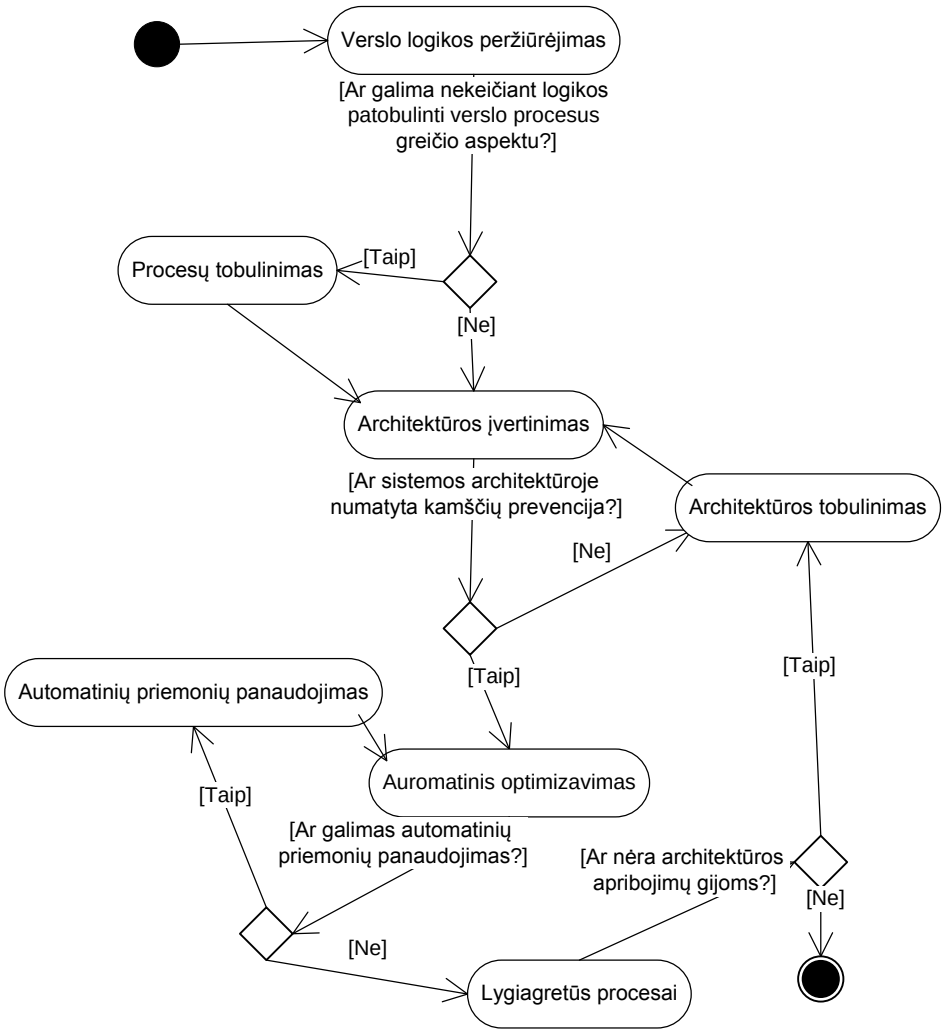
Toliau išdėstomi kiekvieno etapo detalūs žingsniai.

3.2.1. Programos veiklos logikos optimizavimas

Programos veiklos logikos optimizavimo etape peržiūrimas programos karkasas. Akcentuojami šie aspektai:

1. Verslo logikos peržiūrėjimas
2. Sistemos architektūra įvertinimas
3. Kamščių prevencija
4. Numatomos automatinio optimizavimo galimybės
5. Įvertinamas galimas gijų panaudojimas

Grafiškai šio etapo procesas pavaizduotas 24 pav.



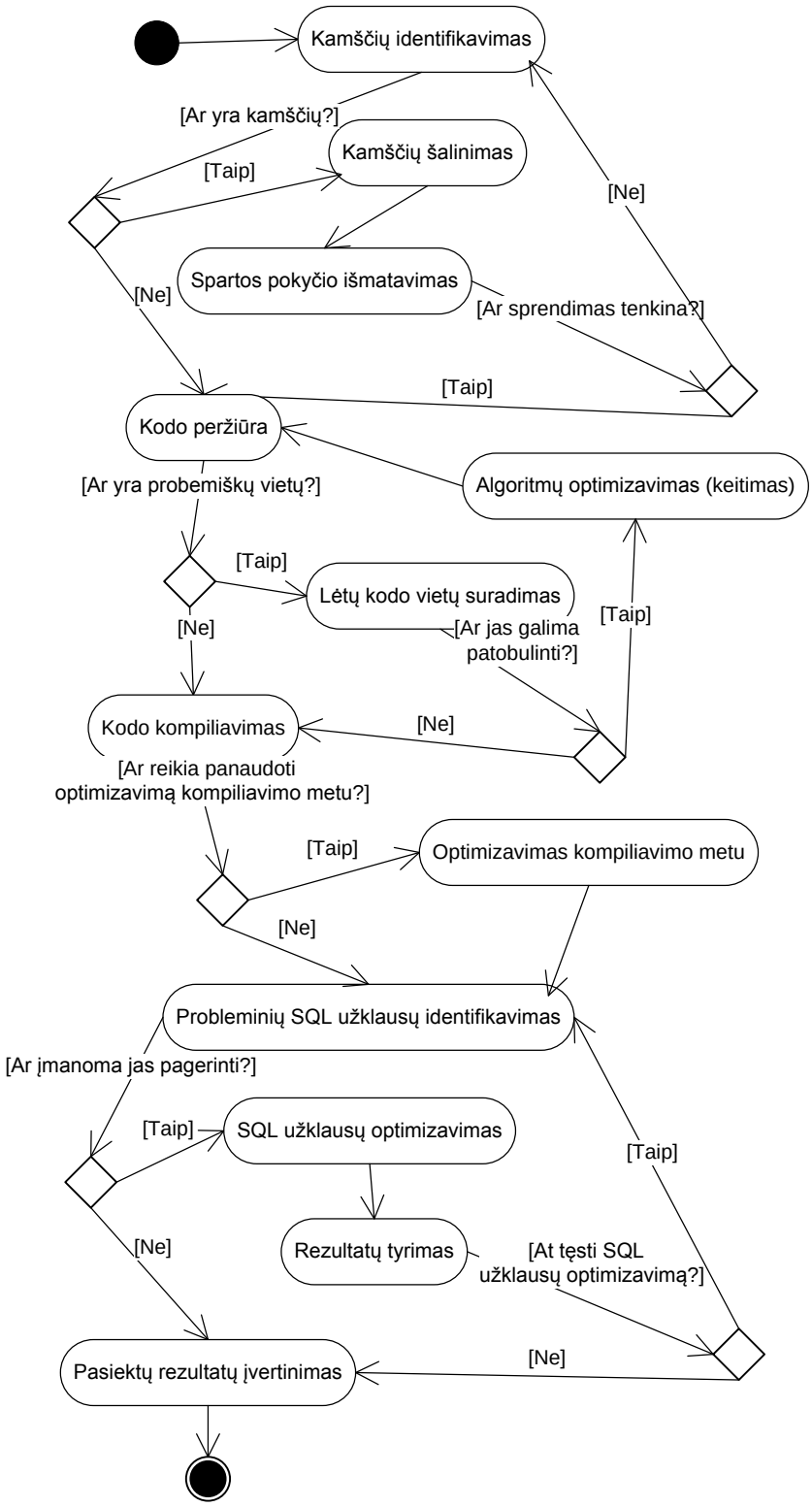
24 pav. *Programos veiklos logikos optimizavimo etapo veiklos diagrama*

3.2.2. Programos kodo optimizavimas

Programos kodo optimizavimo etape didžiausias dėmesys skiriamas šiems aspektams:

1. Kamščių pašalinimas
2. Naudojamų algoritmų optimizavimas (pakeitimas)
3. Optimizavimas kompiliavimo metu
4. SQL užklausų optimizavimas

Grafiškai šio etapo procesas pavaizduotas 25 pav.



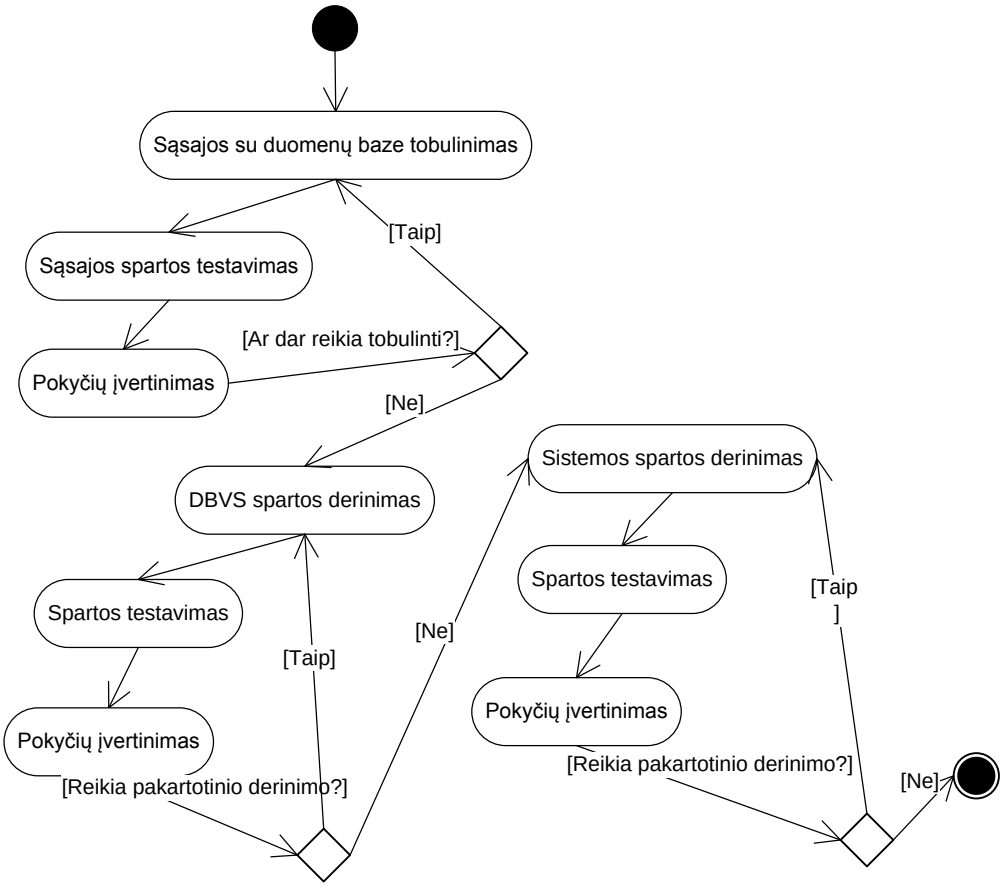
25 pav.*Programos kodo optimizavimo etapo veiklos diagrama*

3.2.3. Techninės sistemos dalies optimizavimas

Techninės sistemos dalies optimizavimo etape išskiriami šie aspektai:

1. Programinės sąsajos su duomenų baze optimizavimas
2. DBVS spartos derinimas
3. Sistemos spartos derinimas (konfigūravimas)

Grafiškai šio etapo procesas pavaizduotas 26 pav.



26 pav. *Techninės sistemos dalies optimizavimo etapo veiklos diagrama*

4. EKSPERIMENTINĖS PROGRAMŲ SISTEMOS SPECIFIKACIJA

4.1. Projekto tikslas ir adresatas

4.1.1. Projekto tikslas

Kauno teritorinės ligonių kasos (KTLK), kaip ir kitų kasų, priedermė – privalomojo sveikatos draudimo fondo biudžeto lėšas panaudoti kuo geriau tenkinant vartotojų, t.y. pacientų poreikius - visiems apdraustiesiems sudaryti finansines, ekonomines ir vadybines prielaidas laiku gauti kokybiškas, prieinamas sveikatos priežiūros paslaugas.

Lietuvoje iš viso yra 10 teritorinių ligonių kasų. Kiekviena apskritis turi savo teritorinę kasą. Visos jos neturi vieningos informacijos apsikeitimo, apdorojimo ir teikimo sistemos. Tiesa, 1999 m. pradėjo veikti kompiuterinė statistinių ir ekonominių duomenų surinkimo bei apdorojimo sistema „Sveidra“, kuri atsakinga už statistinių – epidemiologinių duomenų surinkimą ir analizę bei už ekonominių duomenų rinkimą ir analizę. Bet mes nežadame konkuruoti su šia programa, t.y. pasisavinti šios programos funkcijų.

Į kuriamą programų sistemą būtų galima įtraukti šias teritorinės ligonių kasos funkcijas:

1. nustatyta tvarka kompensuoti draudžiamiesiems (t.y. apdraustiesiems asmenims) galūnių, sąnarių ir organų protezų įsigijimo ir protezavimo, vaistų ir medicinos pagalbos priemonių įsigijimo bei sanatorinio – kurortinio gydymo išlaidas;
2. analizuoti ir įvertinti duomenis apie apskrities savivaldybių gyventojų sveikatos būklę ir gyventojų sudėtį pagal amžių bei lytį, TLK teritorijoje veikiančių įmonių, įstaigų, organizacijų komercinės ir ūkinės ar kitokios veiklos poveikį sveikatai, teikti siūlymus šiais klausimais TLK stebėtojų tarybai;
3. kontroliuoti asmens sveikatos priežiūros paslaugų, apmokamų iš privalomojo sveikatos draudimo fondo biudžeto kiekį ir kokybę bei atlikti finansinę privalomojo sveikatos draudimo fondo biudžeto lėšų naudojimo analizę;
4. kontroliuoti Sveikatos apsaugos ministerijos nustatyta tvarka ir sąlygomis draudžiamiesiems teikiamų asmens sveikatos priežiūros paslaugų prieinamumą ir tinkamumą;
5. skelbti informaciją apie savo veiklą, informuoti draudžiamuosius apie teikiamas asmens sveikatos priežiūros paslaugas, jų teikimo tvarką ir sąlygas.

Pagal šias funkcijas reikia sukurti KTLK informacinę sistemą, apimančią daugelį KTLK prižiūrimų sričių ir padedančią atlikti šias funkcijas:

- Ataskaitų priėmimo – priimti ataskaitas iš gydymo įstaigų (vartotojų), jas apdoroti, nukreipti į atitinkamus TLK skyrius. Pirmasis etapas – sukurti dantų protezavimo duomenų ataskaitas iš gydymo įstaigų ir jas apdoroti.
- Ataskaitų generavimo – vartotojams pateikti norimas duomenų ataskaitas. Duomenų šaltiniai: Kauno teritorinės ligonių kasos informacinė sistema ir Sveidra.
- Analizės ir statistikos – leidžiantis analizuoti konkrečią gydymo įstaigą ar jų grupę įvairiais apsektais: pagal suteiktas paslaugas, išleistus pinigus ir pan. Skirtingai nuo prieš tai esančio modulio, rezultatai pateikiami diagramomis, grafikais ar vaizdiniais žemėlapyje.
- Komunikavimo – sudaryti galimybę vartotojams bendrauti tarpusavyje: siųsti ir gauti pranešimus. Kartu su pranešimais galima persiųsti failus.

Dabar darbuotojai atlikdami darbo funkcijas susiduria su šiomis pagrindinėmis problemomis:

- dideli popierinių dokumentų kiekiai (jų tikrinimas, suvedimas į kompiuterines sistemas);
- neautomatizuotas procesas;
- nėra funkcijų pasiskirstymo – daug vartotojų atlieka daug funkcijų, jas kartodami.

Nors rinka ir nėra didelė – tebūtų vos 10 potencialių klientų, galinčių nupirkti programą, tačiau ateityje būtų galima plėsti sistemą, papildant ją naujomis posistemėmis. Pinigai ūtų gaunami taip pat ir už sistemos priežiūrą.

Kadangi pagrindinis duomenų šaltinis yra dabar visose Lietuvos asmens sveikatos priežiūros įstaigose naudojama programa „Sveidra“ (duomenys kaupiami „Oracle“ duomenų bazėje jau 10 metų), būtina galimybė pasiekti šiuos duomenis per informacinę sistemą. Iš šių duomenų ir bus formuojamos ataskaitos. Taip pat sistemą turi būti galima išplėsti, integruojant į ją papildomus modulius (didinant funkcionalumą). Tai ateityje leistų sistemai suteikti papildomo pageidaujamo funkcionalumo.

4.1.2. Informacija apie užsakovo organizaciją

Apie

Kauno teritorinės ligonių kasos, kaip ir kitų kasų, priedermė – Privalomojo sveikatos draudimo fondo biudžeto lėšas panaudoti kuo geriau tenkinant vartotojų, t.y. pacientų poreikius - visiems apdraustiesiems sudaryti finansines, ekonomines ir vadybines prielaidas laiku gauti kokybiškas, prieinamas sveikatos priežiūros paslaugas.

Vizija

Institucija, vykdanti ne tik privalomąjį sveikatos draudimą, bet dalyvaujanti valdant sveikatos sistemą bei siekianti geresnės paslaugų kokybės ir efektyvesnio sistemos funkcionavimo.

Misija

Privalomojo Sveikatos Draudimo Fondo (PSDF) biudžeto lėšas panaudoti maksimaliai atsižvelgiant į vartotojų poreikius.

Tikslai

1. Sudaryti finansines ir ekonomines prielaidas draudžiamųjų sveikatos priežiūrai užtikrinti.
2. Tenkinti paslaugų gavėjų ir jų tiekėjų poreikius, atsižvelgiant į PSDF biudžetą, žmogiškųjų išteklių bei infrastruktūros galimybes.
3. Užtikrinti kokybės valdymo sistemos funkcionavimą laikantis LST EN ISO 9001:2001 standarto reikalavimų.
4. Siekti profesionalios bei kolegialios tarp institucinės veiklos didinant draudžiamųjų pasitikėjimą.

Uždaviniai

1. Pastoviai analizuoti sveikatos priežiūros paslaugų poreikį.
2. Skaidriai paskirstyti PSDF biudžeto lėšas.

3. Tobulinti sutarčių sudarymo procedūrą ir vykdyti sutartinių įsipareigojimų kontrolę.
4. Gerinti vartotojų aptarnavimą diegiant naujas elektronines paslaugas.

Institucijos vadovybės įsipareigojimai

1. Gerinti institucijos vidinę komunikaciją.
2. Sudaryti galimybes nuolat kelti valstybės tarnautojų bei darbuotojų kvalifikaciją.
3. Nuolat gerinti dirbančiųjų darbo sąlygas, didinti darbuotojų motyvaciją.

4.1.3. Duomenų srautų įstaigoje įvertinimas

Lietuvoje didelių sistemų nėra daug. Pačias didžiausias sistemas turi bankai, kurie naudoja savus spartinimo būdus. Deja, bankų naudojami sprendimai nėra skelbiami viešai. Lietuvoje priskaičiuojama per 100 įvairių registrų[12]: nekilnojamo turto, gyventojų, miškų ir t.t. Dauguma jų yra nedideli. Ir tik tokie kaip adresų registras savo apimtimi viršija 1.000.000 įrašų. Kadangi Lietuvoje nėra daug didelių sistemų, nėra didelio poreikio jų spartai gerinti pasaulyje naudojamais metodais.

Kalbant apie sveikatos priežiūrą, kiekviena teritorinė ligonių kasa aptarnauja gydymo įstaigas, esančias jų valdymo zonoje (toje apskrityje). Gydymo įstaigos savo ruožtu teikia paslaugas apskrities gyventojams. Toliau einančioje lentelėje galite matyti gyventojų pasiskirstymą pagal apskritis[13].

1 lentelė. *Gyventojų pasiskirstymas pagal apskritis*

Apskritis	Gyventojų skaičius
Alytaus	**187769**
Kauno	**701529**
Klaipėdos	**385768**
Marijampolės	**188634**
Panevėžio	**299990**

[12] Informacija iš http://www.registrai.lt

[13] 2005 m. duomenys http://www.stat.gov.lt/lt/pages/view/?id=285

Šiaulių	**370096**
Tauragės	**134275**
Telšių	**179885**
Utenos	**185962**
Vilniaus	**850064**

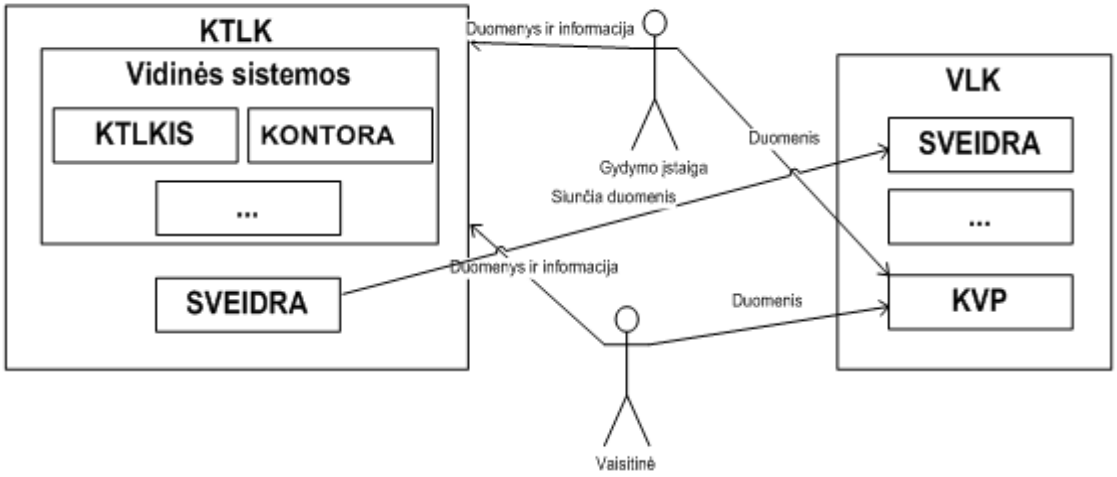
Gyventojų aptarnavimas sveikatos priežiūros įstaigose, paslaugų ir produktų teikimas generuoja sistemose milijonus duomenų įrašų per metus. Kauno apskrityje per metus suteikiama paslaugų už daugiau nei 600 milijonų litų (vien kompensuojamiems vaistams per mėnesį skiriama 15 mln. Litų, kurie vidutiniškai apmoka 264000 receptus). Norint efektyviai sukontroliuoti visą procesą ir peržiūrėti šią galybę duomenų, reikia daug žmogiškųjų išteklių. Todėl daugelis įmonių stengiasi procesą kiek galima automatizuoti. Deja, kiekvieną kartą sistemą reikia derinti ir kurti specifiniams vartotojų poreikiams.

4.2. Sistemos veiklos konteksto aprašymas

Kauno teritorinė ligonių kasa (KTLK) yra pavaldi Valstybinei ligonių kasai (VLK). KTLK aptarnauja apskrities gyventojus ir įstaigas. Gydymo įstaigos savo ruožtu, pasiraišusios paslaugų teikimo sutartis su teritorine ligoniu kasa (TLK[14]), teikia gydymo paslaugas gyventojams.

KTLK, VLK ir medicinos įstaigos bendrauja per įvairias sistemas: Sveidra, Kontora, Kompensuojamų vaistų sistema (KVP) ir kitos. Vienos sistemos priklauso VLK ir yra skirtos visoms susijusioms įstaigoms Lietuvoje, kitos yra sukurtos tik KTLK iniciatyva (pavyzdžiui, Kauno teritorinės ligonių kasos informacinė sistema – KTLKIS) ir skirtos vidiniams tikslams.

[14] Kiekviena apskritis Lietuvoje turi savo TLK.



27 pav.*Sistemos veiklos konteksto diagrama*

Sistemų aprašymai:

- **Kontora** – e-administracijos ir dokumentų valdymo sistema[15]. Tarp naujųjų vadybos sąvokų – e-valdžia, e-verslas, e-vyriausybė vis dažniau vartojama e-administracija – administravimas naudojantis kompiuterinėmis technologijomis. Administravimas yra valdymo proceso dalis – veikla, kuria remiantis rengiamos veiklos programos, keičiamasi informacija, reguliuojama veiklos eiga ir tikrinama, kiek ji atitinka užsibrėžtus planus bei užduotis. E-administracijos informacinės sistemos pagrindiniai principai:
  - administracinės hierarchijos skaidrumas;
  - prioritetas organizacinei struktūrai;
  - atributinių duomenų klasifikavimas;
  - informacijos paieškos universalumas;
  - veikiančios sistemos išplečiamumas.

  Pasaulio ir Lietuvos rinkose yra įvairių e-administracijos sistemų ir daugelis jų atitinka šiuos principus. Sistemos ar technologijos pasirinkimas kiekvienu konkrečiu atveju yra pakankamai daug patirties, žinių bei fantazijos

[15] Daugiau informacijos apie sistemą Kontora http://www.iterija.lt/index.php/lt/34375/

reikalaujantis uždavinys. Šių principų svarbą ir pasirinkimo galimybę lemia turimos investicijos į e-administraciją, organizacijos personalo gebėjimai ir principinga vadovybės nuostata: „Reikia, ir mes tai padarysim“.

- **Sveidra** – Privalomojo sveikatos draudimo kompiuterizuota informacinė sistema "Sveidra" Valstybinėje ligonių kasoje pradėjo veikti prieš 10 metų[16]. Svarbiausios jos taikymo sritys: Privalomojo sveikatos draudimo fondo metinių biudžetų projektų sudarymas, patvirtinto biudžeto vykdymo apskaita, kontrolė bei analizė, Lietuvos Respublikos - ES narės - migruojantiems darbuotojams teikiamų sveikatos draudimo paslaugų bei atsiskaitymų už jas apskaita, „įsijungimas“ į Sveikatos apsaugos ministerijos rengiamas programas ir į bendravalstybinius informacinės visuomenės projektus, susijusius su sveikatos draudimu, taip pat - įvairių vidinių sveikatos draudimo sistemos informacinių poreikių tenkinimas.

- **KVP** – sistema, skirta gydytojų tapatybę patvirtinantiems lipdukams (toliau – GTPL) blankams išduoti; gyventojams kompensuojamųjų vaistų pasams (toliau – KVP) išduoti bei patikrinti asmenų privalomojo sveikatos draudimo statusą.

- **KTLKIS** – Kauno teritorinės ligonių kasos sistema yra šio magistrinio darbo eksperimentinė sistema.

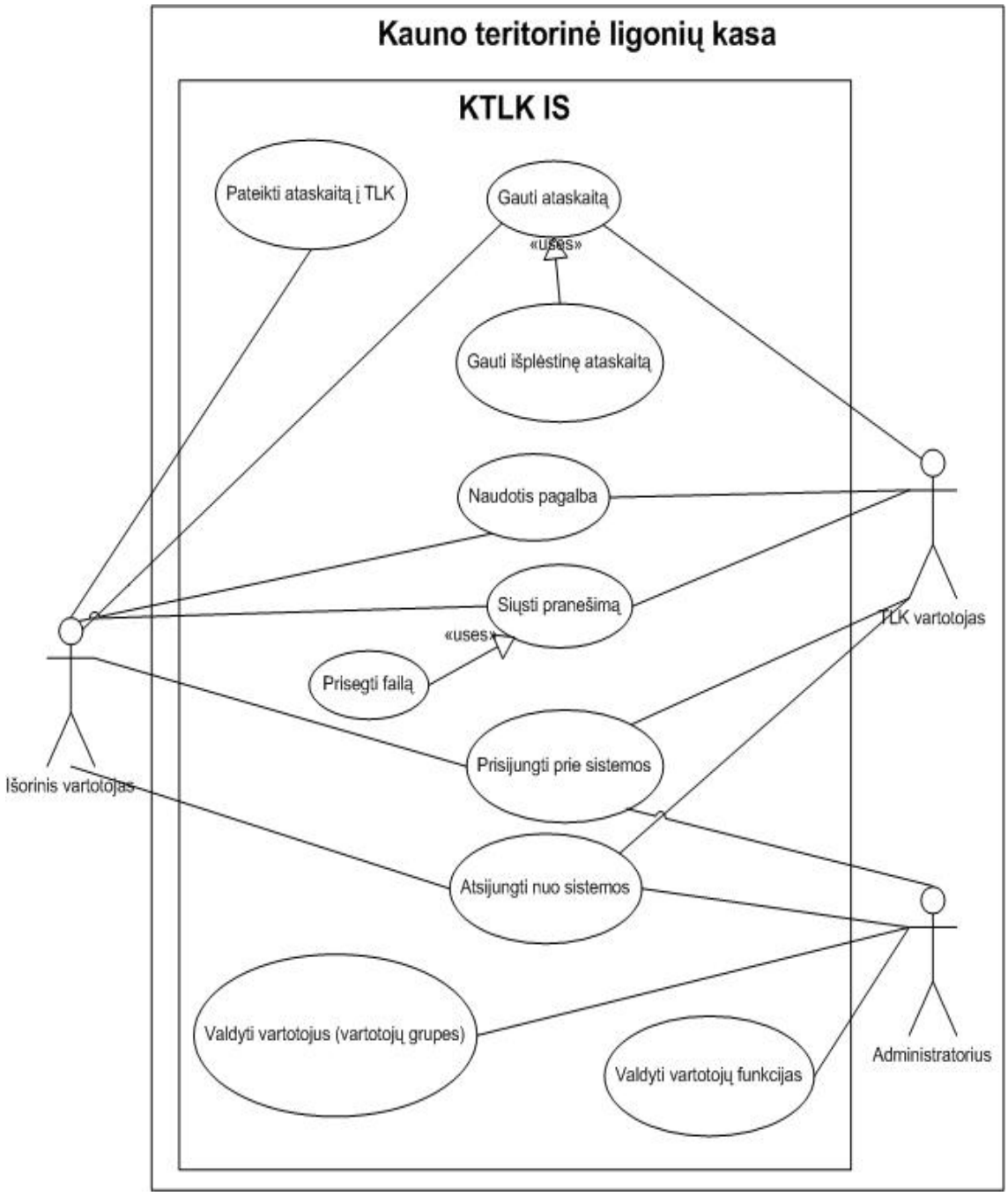
Konteksto schemos diagrama rodo, kad nėra sukurtos vieningos sistemų infrastruktūros (decentralizuota). Yra kelios valstybinės reikšmės sistemos, kurias prižiūri VLK (centrinis Sveidros serveris, KVP – tai centralizuotos sistemos). Šias sistemas galima pasiekti internetu. Kitos sistemos yra išdėstytos TLK skyriuose ir naudoja lokalius resursus, kurie nėra identiški kiekvienai ligonių kasai. Jos taip pat dažniausiai yra skirtos naudotis tik pačios TLK darbuotojams.

[16] Daugiau informacijos apie sistemą Sveidra http://www.vlk.lt/vlk/pr/?page=item&kat_id=5&date=2005-06-01&item_id=1361 ir http://www.alna.lt/as/veiklos-sritys-ir-sprendimai/viesasis_administravimas/igyvendinti-sprendimai/#sveidra

4.3. Panaudojimo atvejų specifikacija

4.3.1. Panaudos atvejų diagrama

Šioje diagramoje pavaizduoti sistemos vartotojai ir jų funkcijos sisteminiu požiūriu. Kaip matyti, TLK vartotojas ir administratorius yra kasos darbuotojai, o išorinis vartotojas gali būti bet kuris ne vidaus vartotojas (vaistinė, ligoninė, poliklinika).



28 pav.*Panaudojimo atvejų diagrama*

4.3.2. Sistemos vartotojų aprašymas

- **TLK vartotojas** – Teritorinės ligonių kasos darbuotojas, priklausantis vienam iš skyrių, ir sistemos pagalba atliekantis jam priskirtas funkcijas. Funkcijos: įstaigų

ataskaitų peržiūra, statistinė ir teritorinė duomenų analizė skirta viešosioms įstaigoms kontroliuoti.

- **Išorinis vartotojas** – vartotojas, prisijungiantis iš išorės (ne TLK darbuotojas) ir galintis atlikti sistemoje jam leidžiamus veiksmus. Potencialūs vartotojai: gydymo įstaigos, vaistinės, VLK (Valstybinė ligonių kasa), kitos TLK (Teritorinės ligonių kasos). Sistemos pagalba jie gali pateikti ataskaitas į TLK, gauti ataskaitas iš TLK pusės, gauti ir siųsti informacinius pranešimus.
- **Administratorius** – vartotojas (-jai), prižiūrintis sistemą, ir galintis keisti jos nustatymus. Jis taip pat įtraukia ir šalina vartotojus, bei keičia sistemos funkcionalumą: pagal reikalavimus reguliuoja vartotojams ir jų grupėms leidžiamas atlikti funkcijas.

4.3.3. Panaudos atvejų aprašymas

- Pateikti ataskaitą į KTLK - gydymo įstaiga, vaistinė ar kitas išorinis vartotojas siunčia ar perduoda ataskaitos pobūdžio dokumentą.
- Gauti ataskaitą - pagal pasirinktą sritį vartotojui suformuojama viena iš galimų ataskaitų (lentelės forma).
- Gauti išplėstinę ataskaitą - pagal pasirinktą sritį vartotojui suformuojama viena iš galimų ataskaitų (diagrama).
- Naudotis pagalba - vartotojas atsiverčia sistemos žinyną – sistemos naudojimosi paaiškinimus ir instrukcijas.
- Siųsti pranešimą - sistemos vartotojai gali siųsti tekstinius pranešimus vieni kitiems.
- Prisegti failą - prie siunčiamos tekstinės žinutės vartotojas gali prisegti dokumentą.
- Prisijungti prie sistemos - vartotojas įveda savo vartotojo vardą ir slaptažodį į prisijungimo formą.
- Atsijungti nuo sistemos - vartotojas atsijungia nuo sistemos.
- Valdyti vartotojus (vartotojų grupes) - administratorius mato vartotojų sąrašą, patvirtina naujus vartotojus, atblokuoja užblokuotus, šalina ir riboja priėjimą atskiriems vartotojams.

- ❖ Valdyti vartotojų funkcijas - administratorius kontroliuoja, ką konkretūs vartotojai gali atlikti sistemoje. Kiekvienas vartotojas turi savo atliekamas funkcijas.

4.4. Sistemos reikalavimų specifikacija

Reikalavimai sistemos vartotojo sąsajai:

- lengvai skaitoma sąsaja;
- paprastas (nesudėtingas) panaudojimas;
- prieinamumas, kad vartotojas nesivaržytų naudodamas sistemą;
- panašumas į kitas KTLK naudojamas sistemas (bendri bruožai)

Reikalavimai naudojamumui:

- sistemos panaudojimas bet kokio asmens be apmokymo (90% sėkmingas pasinaudojimas pirmu bandymu);
- nacionalinės kalbos naudojimas;
- veiklos našumo prieaugis dėl sistemos diegimo.

Reikalavimai sistemos operacijų spartai

- sistemos vartotojas negali laukti bet kurios operacijos įvykdymo ilgiau nei 30s.
- jei neįmanoma, sudaryti sąlygas leisti vartotojui naudotis sistema operacijos vykdymo metu.

Reikalavimai sistemai prižiūrėti:

- nesudėtingas pakeitimų įgyvendinimas;
- konfigūruojami saugumo nustatymai;
- vartotojų valdymas ir kontroliavimas;
- galimybė ateityje pritaikyti sistemą daugiakalbei aplinkai;
- galimybė prie sistemos prijungti vis naujus dalykinės srities modulius (jei bus poreikis).

Sistemos saugumo reikalavimai

Tai vienas iš sudėtingiausių reikalavimų ir susijęs su didele rizika, jei jo nepaisoma.

Reikia išskirti tris saugumo aspektus:

- duomenų apsauga ir vientisumas;
- apsauga nuo ne sankcionuoto priėjimo prie sistemos;
- apsauga nuo kito vartotojo teisių pasisavinimo.

Reikalavimai duomenims

Šioje sistemoje reikalavimai duomenims yra svarbūs tik pirmojo panaudojimo atveju (žiūrėti 28 pav.) – „Pateikti ataskaitą į TLK“. Eksperimentinėje sistemoje numatoma priimti tik vieno tipo ataskaitas. Ateityje tikimasi priimamų ataskaitų tipų skaičių padidinti. Kol kas numatoma realizuoti protezavimo ataskaitų priėmimą iš gydymo įstaigų. Kadangi tam naudojamas tam tikras standartizuotas protokolas, jo visas aprašymas čia nepridedamas[17].

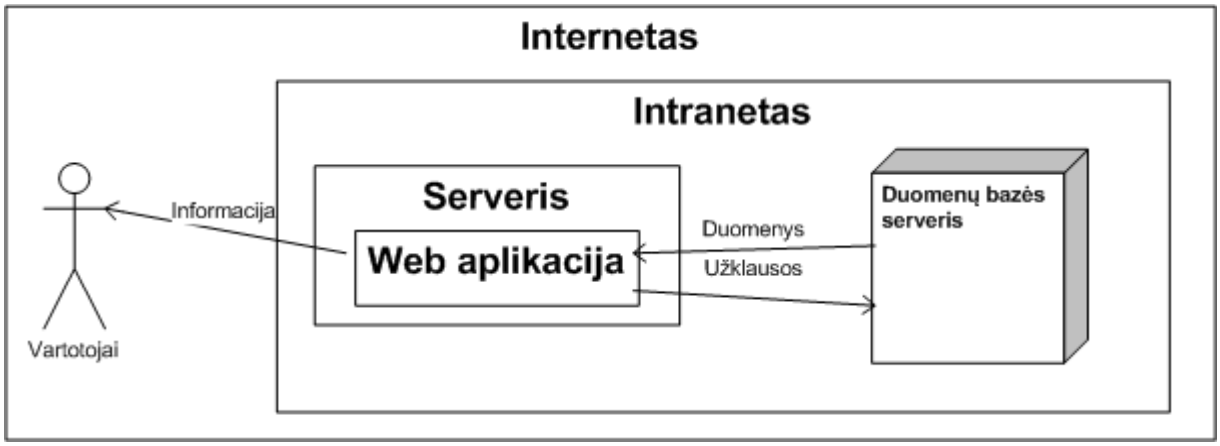
Sveidros duomenų bazėje esamų duomenų negalima kopijuoti (replikuoti) į kitas duomenų bazes ar sistemas. Taip pat negalima duomenų keisti. Priėjimas tik skaitymo teisėmis.

4.5. Sistemos architektūros specifikacija

4.5.1. Sistemos architektūra

Sistemos architektūra ir išdėstymo vaizdas pateikiamas kitame paveikslėlyje:

[17] Protezavimo ataskaitų iš gydymo įstaigų priėmimo protokolas http://www.ktlk.lt/doc/asmenu_saraso_protokolas_V2_0.doc



29 pav.*KTLK informacinės sistemos dalių išdėstymas*

Sistemos architektūros aprašymas

Kauno teritorinės ligonių kasos sistema (KTLKIS) yra internetinė aplikacija, prieinama per šiuo metu populiariausias interneto naršykles, ir yra visiškai su jomis suderinta. Web serveris talpina visą sistemą ir leidžia vartotojams iš išorės ją pasiekti. Vartotojai, esantys lokaliame tinkle (intranete) ją pasiekia vietiniu tinklu. KTLKIS sistemos duomenys saugomi Microsoft SQL Server 20005 duomenų bazėje. Duomenys medicininių paslaugų ataskaitoms imami iš Sveidros duomenų bazės serverio (naudojamas Oracle 8.1.7). KTLKIS sistemai prie Sveidros duomenų bazės leidžiama prisijungti tik skaitymo teisėmis. Kiekviena duomenų bazė gali turėti atskirą serverį.

Kuriamos sistemos architektūros tikslai:

- užtikrinti greičiausią duomenų paėmimo iš duomenų bazės, jų apdorojimo ir pateikimo laiką;
- užtikrinti, kad ateityje sistemą būtų galima plėsti ir vystyti.

Kuriamos sistemos apribojimai:

- kuriamą sistemą planuojama integruoti į Microsoft „SharePoint“ serverio programų paketą;
- sistema neturi reikalauti daug sisteminių resursų

Naudojami projektavimo šablonai ir komponentai

- Web klientui realizuoti naudojama Microsoft siūlomas projektavimo šablonas WCSF[18] (angl. Web Client Software Factory). Šablono naudojimas suteikia visas galimybes sistemai plėsti ateityje, turi lengvai integruojamų servisų galimybę ir leidžia sugrupuoti sistemos funkcionalumą į funkcinius modulius.
- MVP (angl. Model – View - Presenter) šablonas[19] – skirtas internetiniams puslapiams projektuoti ir atvaizduoti vartotojo naršyklėje.
- AmCharts[20] – nemokamas flash diagramų generatorius. Naudojamas atvaizduoti suformuotas ataskaitas diagramomis ir grafikais.

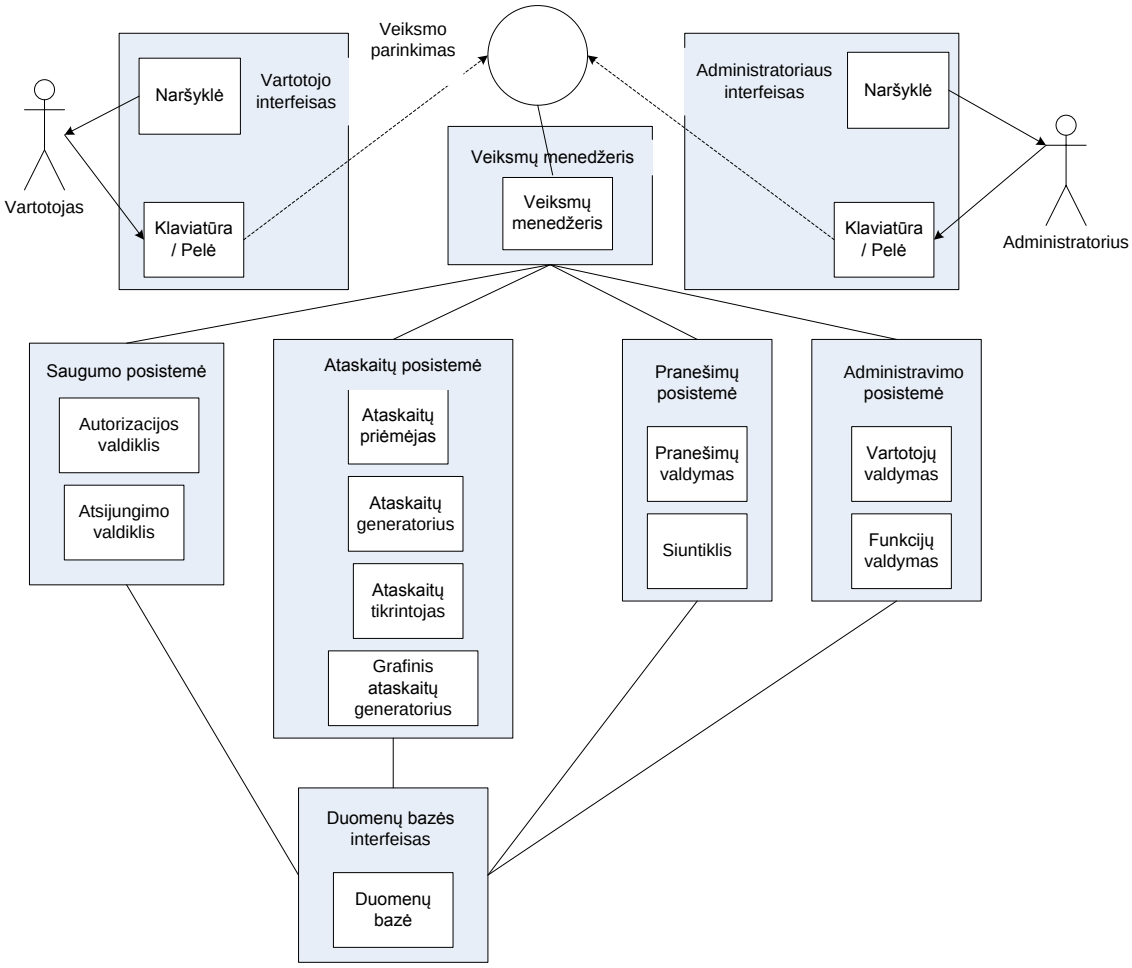
4.5.2. Sistemos posistemių diagrama

Naudojamos architektūros šablonas WCSF visą funkcionalumą sugrupuoja į funkcinius modulius. Sistemos modulių diagrama parodyta 30 paveikslėlyje.

[18] Daugiau informacijos apie šabloną galima pasiskaityti http://www.codeplex.com/websf ir http://msdn.microsoft.com/en-us/library/bb264518.aspx

[19] Daugiau informacijos apie MVP galite rasti http://msdn.microsoft.com/en-us/magazine/cc188690.aspx ir http://en.wikipedia.org/wiki/Presenter_First

[20] Daugiau informacijos galima rasti svetainėje kūrėjų svetainėje http://www.amcharts.com/



30 pav.*Sistemos posistemių diagrama*

4.5.3. Sistemos posistemių aprašymas

- Vartotojo interfeisas – tai grafinė vartotojo aplinka, per kurią jis gali atlikti veiksmus su sistema. Vartotojo sąsaja sąveikauja su veiksmų menedžeriu (kaip parodyta 26 pav.). Veiksmų menedžeris reguliuoja vartojo veiksmus ir reaguoja į sukeltus įvykius. Tai tarpinis bendravimo elementas visoje sistemoje.
- Veiksmų menedžeris – visų vartotojo veiklų sistemoje valdiklis. Apdoroja vartotojų atliekamus veiksmus ir per interfeisus pakviečia reikalingas posistemes. Veiksmų menedžeris riboja vartotoją nuo jam negalimų veiksmų atlikimo. Taip saugomasi nuo neteisėto sistemos panaudojimo.

- Administratoriaus interfeisas – administratoriaus sąsaja, per kurią administratorius atlieka savo funkcijas. Tai grafinė administratoriaus aplinka, per kurią jis gali atlikti veiksmus, keisti įvairius sistemos nustatymus.
- Saugumo posistemė – atsakinga už autorizacijos ir autentifikacijos užtikrinimą, vartotojų ir jų teisių kontroliavimą, bei sistemos duomenų saugumą.
- Ataskaitų posistemė – posistemė atsakinga už duomenų pateikimą vartotojui, ataskaitų suformavimą.
- Pranešimų posistemė – atsakinga už pranešimų siuntimą ir gavimą.
- Administravimo posistemė – atsakinga už administratoriaus funkcijų užtikrinimą. Leidžia valdyti vartotojus ir jų funkcijas sistemoje bei saugumo nustatymus.
- Duomenų bazės interfeisas – leidžia bendrauti su duomenų baze: gauti, išsaugoti, atnaujinti ir ištrinti duomenis.

4.6. Sistemos klasių diagramos

Toliau pateikiama ataskaitų posistemės, ataskaitų apdorojimo mechanizmo klasių diagrama. Tai kritinė sistemos ir tyrimo dalis.

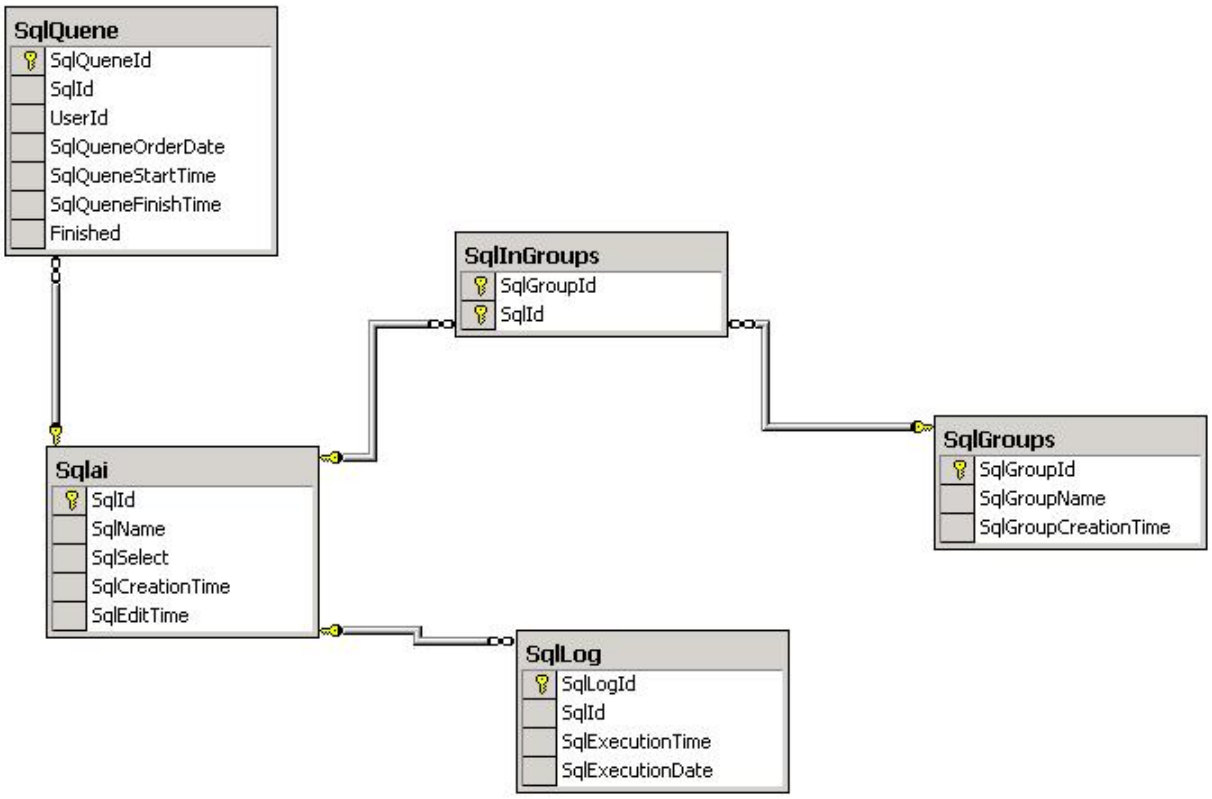
31 pav.*Ataskaitų posistemės, ataskaitų apdorojimo dalies klasių diagrama*

Kitos klasių diagramos pateikiamos prieduose.

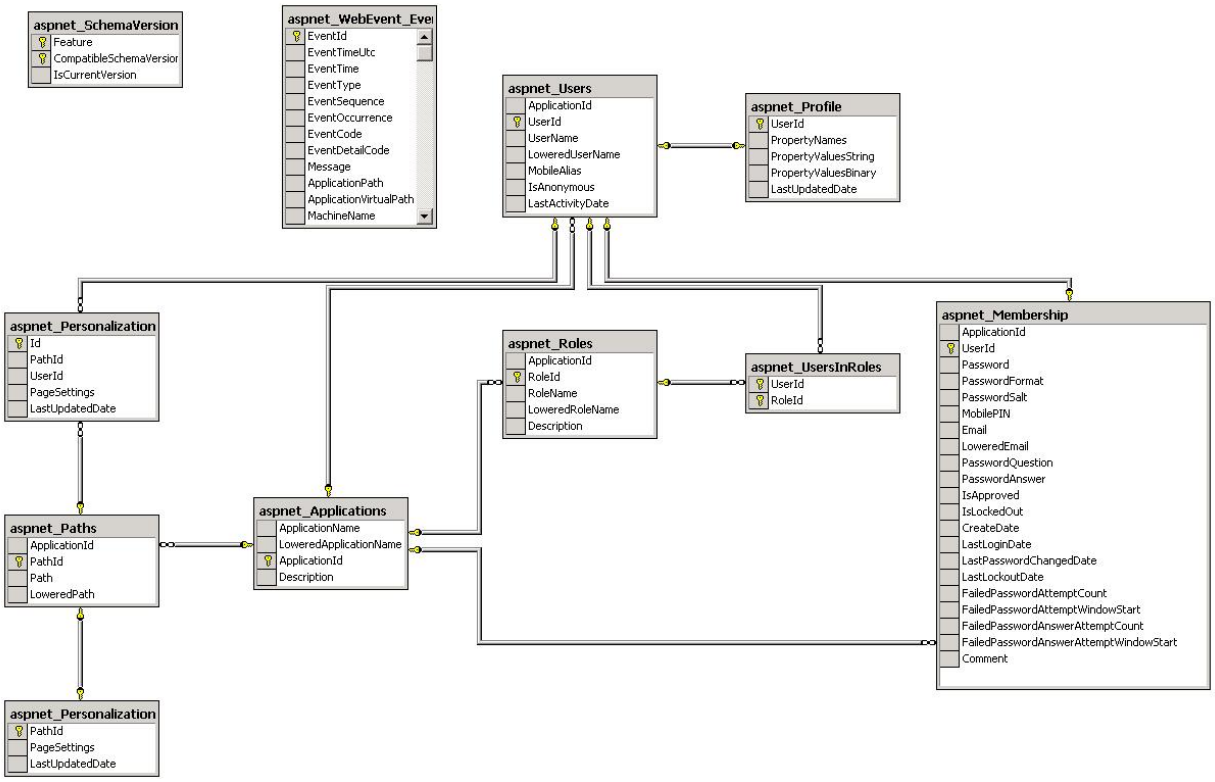
4.7. Duomenų bazės struktūra

KTLKIS sistema gali būti sukonfigūruota naudotis duomenimis skirtingų duomenų bazių. Naudojamas duomenų pasiekiamumo sluoksnis leidžia informaciją gauti iš skirtingų tipų duomenų bazių (Oracle, MS SQL, MySql ar kitos populiarios duomenų bazių valdymo sistemos) nesirūpinant pačių duomenų gavimų. KTLKIS pagrindinis duomenų šaltinis bus Oracle duomenų bazė. Joje sutalpinti KTLK ir visų jos aptarnaujamų įstaigų naudojamos programos Sveidra duomenys (apie sistema Sveidrą galima pasiskaityti 4.2 skyriuje „Sistemos veiklos konteksto aprašymas“). Sveidros duomenų bazę sudaro 153 lentelės. Pagrindinės duomenų lentelės pateiktos prieduose esančiose diagramose (žiūrėti priedus). Duomenys iš šių lentelių bus naudojami ataskaitoms formuoti. Taip pat ten bus išsaugomi ir iš ataskaitų apdorojimo atėję duomenis.

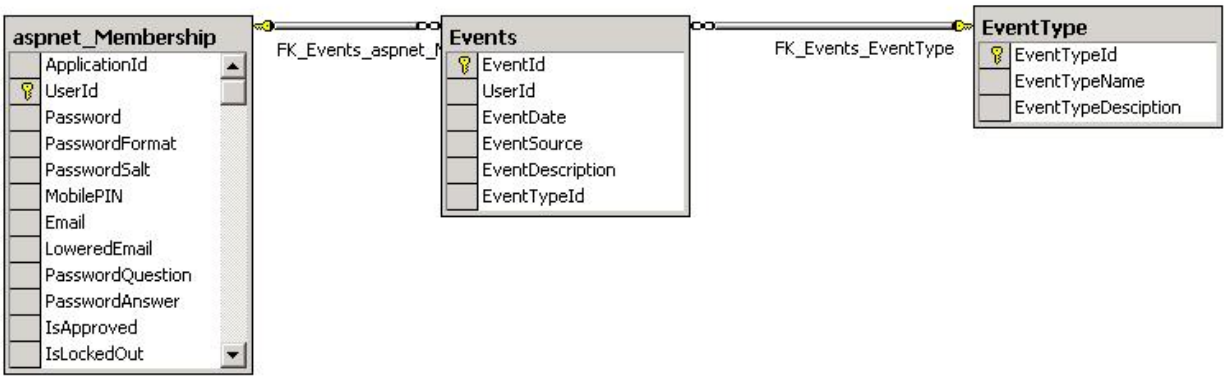
Toliau pateikiama įvairiems KTLKIS funkcionalumams realizuoti sukurtos papildomos duomenų bazės lentelės, saugomos MS SQL 2005 duomenų bazėje.



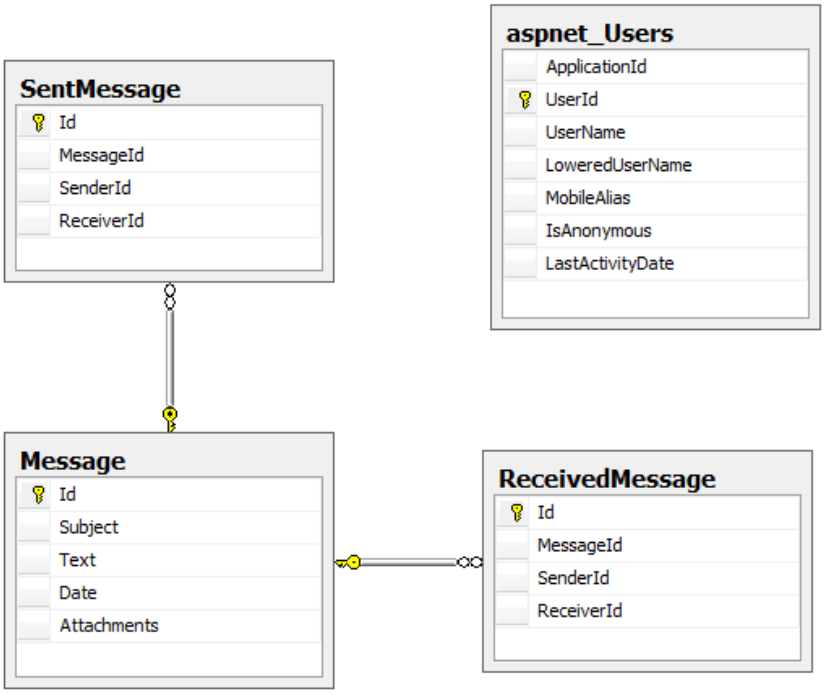
32 pav.*Duomenų lentelė „Sqlai“ ir susijusios lentelės. Struktūra naudojama saugoti ir sekti SQL užklausų vykdymą.*



33 pav.*Duomenų lentelė „aspnet_Users“ ir susijusios lentelės. Duomenų struktūra naudojama vartotojams administruoti ir saugoti.*



34 pav.*Duomenų lentelė „Events“ ir susijusios lentelės. Duomenų struktūra naudojama sistemos įvykius registruoti.*



35 pav. *Duomenų lentelė „Message“ ir susijusios lentelės. Duomenų struktūra naudojama pranešimams saugoti.*

5. GREITAVEIKOS UŽTIKRINIMO METODIKOS EKSPERIMENTINIS TYRIMAS

Šiame skyriuje aprašytas greitaveikos užtikrinimo metodikos (3 skyrius „Programų sistemos greitaveikos užtikrinimo metodika“) eksperimentinis tyrimas. Eksperimento tikslas – pagrįsti siūlomos metodikos naudingumą ir parodyti jos efektyvumą. Atliekamo eksperimento sąlygos aprašytos 5.1 skyriuje.

5.1. Eksperimentinio tyrimo sąlygos

Magistrinio darbo programų sistema (Kauno teritorinės ligonių kasos informacinė sistema – KTLKIS) buvo kuriama nuo 2006 m. rugsėjo mėnesio iki 2008 liepos mėnesio. Aktualiausias dalykas viso kūrimo ir tobulinimo metu buvo koncentravimasis į programos spartą. Jau reikalavimų ir projektavimo etapuose buvo bandoma iš anksto numatyti sistemos greitaveikos bėdas.

Magistrinio darbo programų sistema buvo integruota valstybinėje įstaigoje – Kauno teritorinėje ligonių kasoje. Tai leido sistemos tyrimui naudoti įstaigos turimus technikos ir duomenų resursus.

Su valstybine įstaiga buvo susitarta, kad iš pradžių bus sukurtas sistemos prototipas ir užpildytas duomenimis. Atlikus sistemos spartos įvertinimą ir pradinį testavimą bei palyginus rezultatus su spartos reikalavimais pateiktais (4.4 skyriuje „Sistemos reikalavimų specifikacija“) bus užsibrėžti optimizavimo tikslai, kuriuos bus bandoma įvykdyti.

Sistemos greitaveikos tyrimas vyko 10 mėnesių (nuo 2008-07-01 iki 2009-04-30). Taip buvo suplanuota, kad būtų galima stebėti optimizavimo rezultatus etapais. Kiekvieno mėnesio pradžioje buvo atliekami sistemos greitaveikos patobulinimai, o mėnesio eigoje renkama veikimo statistika. Šis procesas buvo pakartotas keletą kartų. Todėl esant reikalui galima pateikti smulkią kiekvieno etapo patobulinimo įtaką bendrai sistemos spartai.

Eksperimento pradžioje nusistatytos sąlygos:

1) techninė sistemos įranga tyrimo metu nekeičiama;

2) tariama, kad visu eksperimento laikotarpiu duomenų srautai yra panašūs: apdorojamų atskaitų sudėtingumas nekinta; formuojamų atskaitų tipai išlieka tie patys;
3) matuojant įvairių sistemos vietų greitaveiką neatsižvelgiama į Sveidros duomenų bazės apkrautumą. Tariama, kad apkrautumas tyrumo laikotarpiu visada pastovus.

Ataskaitų kūrimo posistemė duomenis ataskaitoms naudoja iš valstybinės reikšmės sistemos Sveidra. Informacijos srautams generuoti buvo naudojami duomenys iš Sveidros duomenų bazės. Jos dydis 200 GB, o bendras įrašų skaičius visose lentelėse yra apie 400 milijonų.

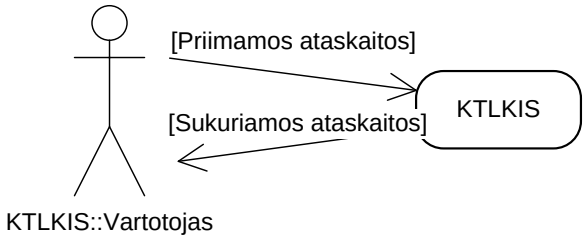
Eksperimento metu vykdyti spartos matavimai buvo atliekami su kompanijos „JetBrains“[21] įrankiu „dotTrace Profiler“[22]. SQL užklausų vykdymo greičiui nustatyti buvo naudojamas „SQL Navigator“[23] įrankis.

5.2. Eksperimentinio tyrimo aprašymas

5.2.1. Pradinis sistemos spartos įvertinimas

Sukūrus sistemos prototipą ir paleidus jį veikti, buvo atliktas pirminis spartos matavimas ir vykdomas vieno mėnesio trukmės stebėjimas. Matuojami buvo du duomenų srautai sistemoje:

1) ataskaitų priėmimo srautas,
2) ataskaitų kūrimo srautas.



36 pav.*KTLKIS ataskaitų srautai*

[21] Kompanijos svetainė http://www.jetbrains.com

[22] dotTrace Profiler įrankio aprašymas http://www.jetbrains.com/profiler/

[23] SQL Navigator įrankio aprašymas http://www.quest.com/sql-navigator/

Kalbant apie priimamas ataskaitas, pagal ataskaitų priėmimo protokolą (žiūrėti 4.4 skyrių „Sistemos reikalavimų specifikavimas“ - reikalavimai duomenims) viena ataskaita laikoma vieno žmogaus duomenys (kas dažniausiai būna viena duomenų eilutė). Kadangi žmogaus duomenys pagal naudojamą protokolą eina per 4 failus, tai laikoma, kad vienos ataskaitos apdorojimo laikas susideda iš visų keturių duomenų failų apdirbimo laikų.

Sukurta ataskaita – tai iš duomenų bazės duomenų pagal SQL užklausa suformuoti duomenys. Šios ataskaitos sukūrimo laikas skaičiuojamas nuo vartotojo mygtuko paspaudimo iki rezultatų vartotojui pateikimo.

Po atliktų stebėjimų buvo gauti šie rezultatai, pateikti lentelėje žemiau. Sukurtų ataskaitų skaičius lygus įvykdytų SQL užklausų skaičiui.

2 lentelė. *Nulinio etapo rezultatai*

	Ataskaitų skaičius	Bendras laikas (s)	Vidutinis laikas (s)
Priimta	453	3312	7,3
Sukurta	134	5861	43,7

Apdorotų operacijų (bendras ataskaitų skaičius) skaičius nėra didelis, nes pirmajame etape buvo taisomos pastebėtos sistemos klaidos ir dėl to kurį laiką sistema neveikė. Taip pat sistema nebuvo iškart pradėta naudoti visų potencialių vartotojų (įstaigos darbuotojų).

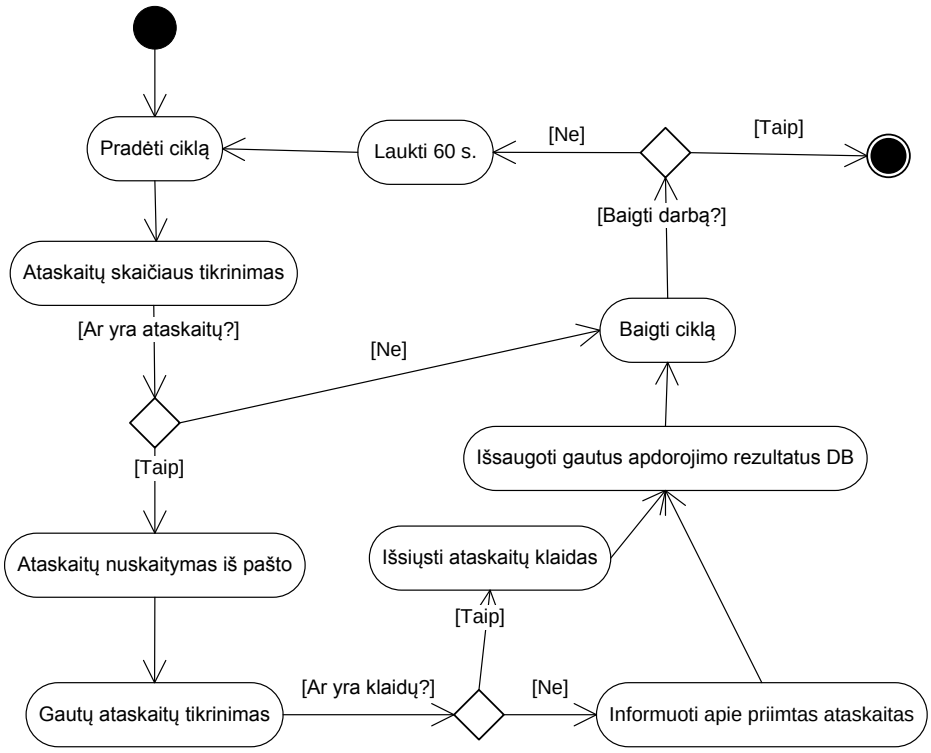
Gauti rezultatai rodo, kad operacijų vidutinis laikas santykinai nėra laba didelis. Toliau bus bandoma taikyti siūlomą greitaveikos užtikrinimo metodiką ir stebėti, kaip keičiasi rezultatai nuo įvairių sprendimų taikymo.

5.2.2. Programos veiklos logikos optimizavimas

Pirmiausia peržvelgus sistemos architektūrą buvo bandoma ieškoti mažiausiai pralaidžių vietų. Surastos dvi vietos:

1) ataskaitų apdorojimo posistemėje potenciali problema gali būti dėl per didelio laiškų (ataskaitų) kiekio. Kaip matyti 37 pav. naudojamas ataskaitų apdorojimo mechanizmas yra nuoseklus ir ciklišKas, todėl, srautui išaugus, jis gali nebespėti susidoroti su darbu.
2) Ataskaitų kūrimo modulio problema yra neatsparumas apkrautumui. Esant dideliam apkrautumui piko valandomis yra tikimybė, kad sistema smarkiai sulėtės arba išvis

nustos veikusi, nes sistemos pralaidumas yra ribojamas naudojamos techninės įrangos.



37 pav.*Ataskaitų apdorojimo mechanizmo veiklos diagrama*

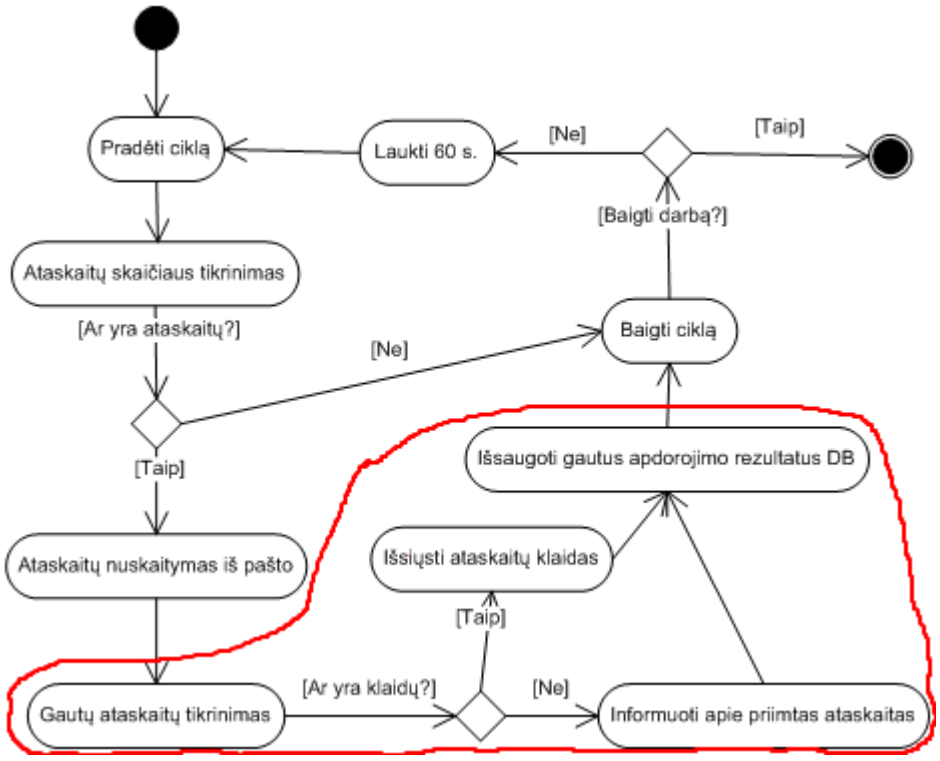
Pirmajame optimizavimo etape pasinaudojus metodika bus panaudotos šios priemonės:

1. sistemos architektūros optimizavimas;
2. kompromisinis optimizavimas;
3. lygiagrečiųjų procesų panaudojimas.

Kartu panaudojus šių priemonių rinkinį galima spręsti pirmąją problemą. Siūloma kai kuriuos nuoseklaus apdorojimo procesus keisti į lygiagrečiuosius, taip pat panaudojus kompromisinį optimizavimą paspartinti ataskaitų apdorojimą laikant visus duomenys atmintyje. Atminties problemų kilti netūrėtų, nes vienu metu sistema apdoroja ne daugiau 100 ataskaitų. 38 pav. pavaizduota, kuriuos procesus galima išlygiagretinti. Tai būtų:

1. ataskaitų tikrinimas – tikrinti ne po vieną, o visas iš karto. Panaudoti esamą atminties kiekį ir joje laikyti duomenis. Taip greičiau būtų nuskaitomi duomenys, sumažėtų skaitymo operacijų ir failų (praktikoje jos yra labai lėtos);

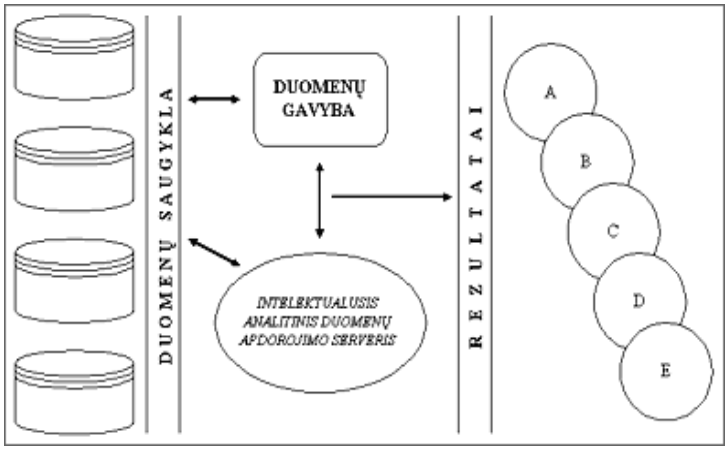
2. apdorojimo rezultatus siuntėjams siųsti lygiagrečiai;
3. rezultatus saugoti į duomenų bazę lygiagrečiai.



38 pav. *Patobulintas ataskaitų apdorojimo mechanizmo siūlymas*

Reikalingų pakeitimų projekte sąnaudos nėra labai didelės. Naudotina procesų lygiagretumo technologija nėra nauja ar sudėtinga.

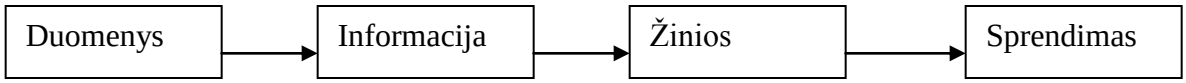
Antroji problema gali būti sprendžiama atsisakant realaus laiko sistemos ir dirbant su laike agreguotais duomenimis. Pavyzdžiui, daugumą reikalingų ataskaitų galima iš anksto suformuoti sistemai nedirbant (angl. idle state). Tai būtų galima atlikti naktimis arba esant nedideliam sistemos apkrautumui. Tai ypač aktualu didelėms ir sudėtingoms ataskaitoms, kurių formavimas sunaudoja daug sistemos resursų ir laiko.



39 pav. *Integruotos duomenų gavybos architektūra*

Sprendžiant šią problemą taip pat galima pasinaudoti duomenų gavybos metodika. Duomenų gavyba – tai procesas, naudojantis įvairius duomenų analizės įrankius, kurie padeda atrasti tokias duomenų struktūras ir ryšius, kurie būtų panaudojami realioms išvadoms ir sistemos rezultatams apibrėžti. Ši technologija sėkmingai taikoma versle, medicinoje ir kitose gyvenimo srityse, kur reikia apdoroti labai didelius informacijos kiekius [3] [9]. Duomenų gavybos tikslas – informacijos išgavimas iš didelių duomenų bazių.

Schemoje visa tai galima pavaizduoti duomenų transformacijos grandine (40 pav.):



40 pav. *Duomenų transformacijos grandinė*

Atlikus pirmojo optimizavimo etapo pakeitimus ir atlikus sistemos spartos stebėjimus buvo gauti šie rezultatai (3 ir 4 lentelės):

3 lentelė. *Pirmojo etapo rezultatai. Priimtos ataskaitos.*

Ataskaitų skaičius	Bendras laikas (s)	Vidutinis laikas (s)
741	5214	7,0

Kaip matoma iš rezultatų, vienos ataskaitos apdorojimo laikas sumažėjo 0,3 s. Šiuo atveju leido sutaupyti $0{,}3 * 741 = 222{,}3s$ sistemos darbo laiko. Lygiagrečiųjų procesų panaudojimas leido pagreitinti ataskaitų apdorojimą, bet labai nedaug. Tiesa, sutaupytas laikas

bus tuo didesnis, kuo daugiau ataskaitų bus priimama. Tai taip pat leistų patikslinti vienos ataskaitos apdorojimo laiką.

4 lentelė. *Pirmojo etapo rezultatai. Sukurtos ataskaitos.*

Užsakyta ataskaitų	SQL užklausų	Bendras laikas (s)	Vidutinis laikas (s)
746	589	26152	44,4

Perėjus nuo „realaus laiko“ sistemos ir vartotojui leidus dirbti su laike agreguotais duomenimis, buvo gauti rezultatai 4 lentelėje. Užsakytų ataskaitų ir SQL užklausų kiekiai skiriasi, nes čia jos turi atskiras sąvokas. SQL užklausa yra sakinys, pagal kurį buvo įvykdyta duomenų gavyba. Užsakytų ataskaitų skaičius šiuo atveju yra: kiek kartų vartotojai pageidavo gauti ataskaitą iš sistemos. Po atliktų architektūrinių pakeitimų, kai vartotojas užsisako ataskaitą, yra patikrinama, ar ši ataskaita jau yra paruošta (ankščiau buvo įvykdyta SQL užklausa), ar ne. Jei ataskaita yra paruošta, ji pateikiama vartotojui nesikreipiant į duomenų bazę. Jei ne – sistema kreipiasi į duomenų bazę, siunčia SQL užklausą įvykdyti ir gautus rezultatus pateikia vartotojui.

Šis panaudotas informacijos gavybos modelis leido sutaupyti 157 ataskaitų sukūrimo laiką. Iš viso yra sutaupyta sistemos darbo laiko: $(746-589)*44{,}4s=6971s$. Jei lygintume su pradiniu įvertinimu, kurio SQL užklausos vykdymo laikas buvo 43,7 s, pokytis yra +0,7 s. Taip gali būti dėl to, kad išaugus sistemos srautams buvo gautas tikslesnis ataskaitos sukūrimo vidutinis laikas.

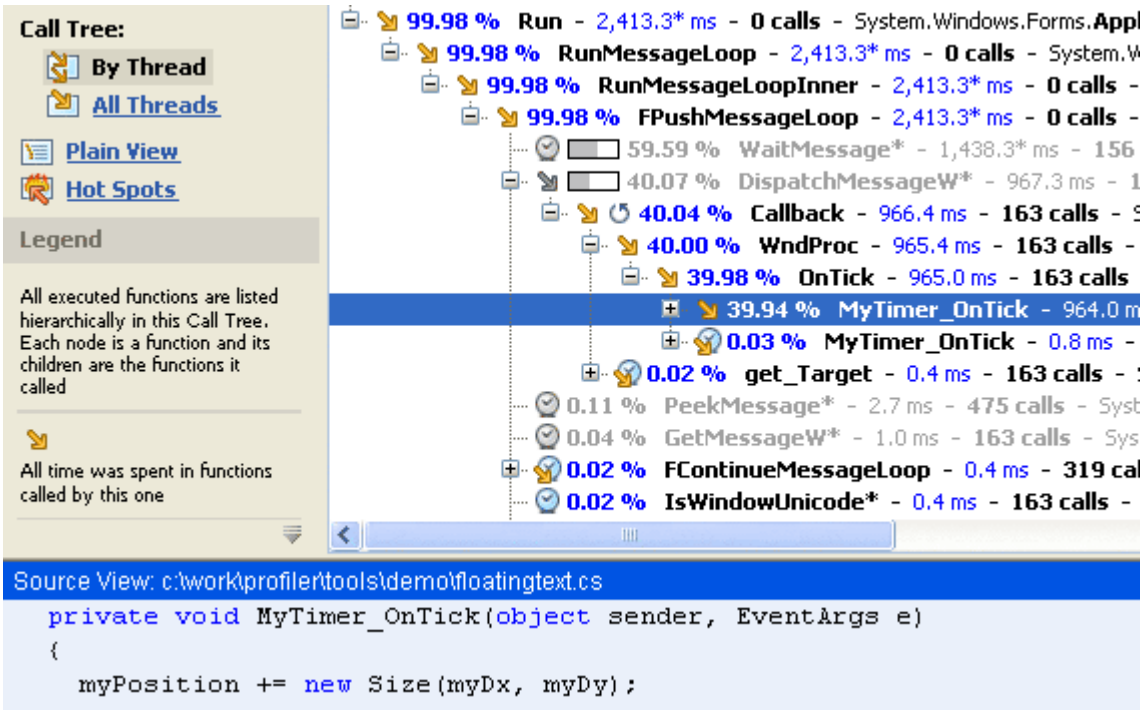
Taip pat galima apskaičiuoti vidutinį ataskaitos sukūrimo laiką (kas nėra tas pats, kas vidutinis SQL užklausos vykdymo laikas), kuris yra: $26152s/746=35s$. Grafiškai rezultatus galite matyti 44 pav.

5.2.3. Programos kodo optimizavimas

Norint dar pagerinti pirmajame etape pasiektus rezultatus (5.2.5 skyrius „Programos veiklos logikos optimizavimas“), reikia nuodugniai atlikti visos sistemos profiliavimą. Tai yra, reikia išmatuoti visų atliekamų operacijų vykdymo laikus ir nustatyti, kurios iš jų kritinės. Taip pat įvertinti, ar tai yra kodo, architektūros, ar logikos problema. Pagal greitaveikos

užtikrinimo metodikos gaires, šiame etape reikia identifikuoti problemines kodo vietas ir jas pašalinti.

Šiems tikslams atlikti naudosime dotTrace Profiler įrankį (aprašytas 5.1 skyriuje „Eksperimentinio tyrimo sąlygos“). Pavyzdinis profiliavimo programos langas su rezultatai parodytas 41 pav. Jame matyti, kurių kodo vietų vykdymas užtrunka ilgiausiai. Visas tiriamo laikotarpio laikas (vartotojo atliekamos operacijos, pavyzdžiui, ataskaitos užsakymas) yra 100%.



41 pav. *Profiliavimo analizės rezultatai*

Kaip matyti 41 pav., identifikuotos dvi ilgai užtrunkančios operacijos (vykdymo laikas 1s ir daugiau). Suradus daugiau tokių vietų programos kode, reiktų jas peržiūrėti ir nustatyti priežastis. Jei įmanoma, šias kodo vietas reiktų optimizuoti, kad jų vykdymo laikas užimtų mažiausiai laiko.

Atlikus antrojo etapo išsikeltus tikslus ir atlikus sistemos stebėjimą gauti rezultatai pavaizduoti 5 ir 6 lentelėse.

5 lentelė. *Antrojo etapo rezultatai. Priimtos ataskaitos.*

Ataskaitų skaičius	Bendras laikas (s)	Vidutinis laikas (s)
1048	6847	6,5

Antrojo etapo optimizavimo rezultatai, palyginus su pirmuoju etapu, yra panašūs. Vidutinis ataskaitos apdorojimo laikas sumažėjo 0,5 s. Apdorojamų ataskaitų skaičius taip pat augo, nes vis daugiau gydymo įstaigų siųsdamos ataskaitas naudojasi būtent šia sistema.

6 lentelė. *Antrojo etapo rezultatai. Sukurtos ataskaitos.*

Užsakyta ataskaitų	SQL užklausų	Bendras laikas (s)	Vidutinis laikas (s)
1967	1438	63480	44,1

Kaip matyti iš lentelėje pateiktų rezultatų, vidutinis SQL užklausos vykdymo laikas praktiškai nepakito (pokytis -0,3 s). Pokyčiui įtakos turėjo išaugę informacijos srautai (ataskaitų užsakymo dydžiai). Apskaičiavę sistemos darbo laiką, gauname 17,6 h. Šis laikas buvo skirtas vien tik SQL užklausoms įvykdyti duomenų bazės serveryje. Kitame etape bus stengiamasi sumažinti serverio darbo laiką vykdant SQL užklausas.

5.2.4. Sąsajos su duomenų baze ir SQL užklausų optimizavimas

Trečiajame etape reikia optimizuoti SQL užklausas ir KTLKIS sąsają su duomenų baze. Iš prieš tai buvusio etapo (skyrius 5.2.3 „Programos kodo optimizavimas“) matyti, jog neatlikus šio etapo patobulinimų ir augant ataskaitų kiekiams sistema gali patirti rimtų spartos sutrikimų.

Prieš tai buvusiame etape atlikus vykdymo laiko matavimus nustatyta, kad teorijoje numatytos problemos tikrai egzistuoja ir praktiškai. Kaip rašoma 2.3.1 skyriuje „Sistemos spartos derinimas“, šio etapo tikslas bus optimizuoti rašymo ir skaitymo operacijas, bei serverio apkrovas. I/O operacijos laiko požiūriu yra pačios brangiausios. Paprastai tai ir yra pirmasis „butelio kakliukas“ duomenų bazės veikloje [17].

Taip pat išmatuota, kiek užtrunka rezultatų saugojimas duomenų bazėje. Vidutinė trukmė yra 1,3 s. Tai yra 20% viso ataskaitos apdorojimo laiko (6,5 s).

Pirma reikia suplanuoti, kokius sąsajos du duomenų baze patobulinimus atliksime. Ataskaitų apdorojimo mechanizme (33 pav.) žingsnį „išsaugoti gautus apdorojimo rezultatus DB“ galime išskaidyti į šiuos etapus:

1. prisijungti prie duomenų bazės;

2. įvykdyti SQL užklausą (įterpimo arba atnaujinimo);
3. atsijungti nuo duomenų bazės.

Kadangi apdorojamų ataskaitų skaičius yra didelis, pagal [18] literatūros šaltinį būtų galima rezultatus pasiųsti į duomenų bazę vienu kreipiniu. Tai leistų sutaupyti išeities ir įeities operacijų skaičių. Deja, to paties negalima padaryti su užklausų kūrimo mechanizmu.

Atlikus sąsajos su duomenų baze optimizavimą ir tyrimą, gauti rezultatai pavaizduoti 7 lentelėje.

7 lentelė. *Trečiojo etapo rezultatai. Priimtos ataskaitos.*

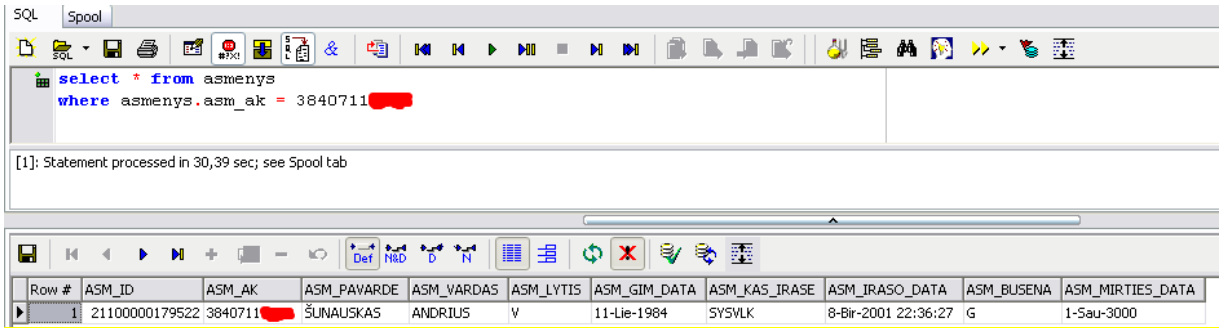
Ataskaitų skaičius	Bendras laikas (s)	Vidutinis laikas (s)
1139	7082	6,2

Matyti jog šis patobulinimas sutrumpino vidutinį ataskaitos apdorojimo laiką 0,3 s. Tiksliau, vidutinis apdorotos ataskaitos saugojimo laikas į duomenų bazę sutrumpėjo nuo 1,3 s. iki 1 s. Įvertinant trečiojo etapo ataskaitų kiekį pavyko sutaupyti $1139 * 0{,}3s = 342s$ programų sistemos darbo laiko.

Ataskaitų kūrimo mechanizmui pagreitinti optimizuosime SQL užklausas. Kaip pavyzdį aprašysiu mažos ir didelės užklausos optimizavimo rezultatus. Taip optimizavus visą ataskaitų kūrimo posistemę būtų galima vidutinį užklausos laiką sumažinti iki 30 s.

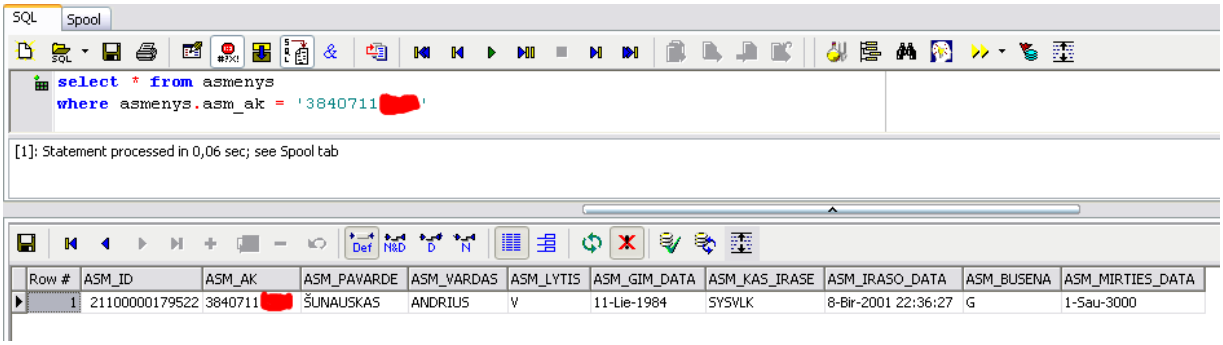
Deja, negalima įvertinti kiekvienos SQL užklausos optimizavimo rezultatų bendroje sistemos spartoje. Todėl įvertinimas bus padarytas kiekvienai SQL užklausai atskirai.

Turime SQL užklausą pavaizduotą 42 pav. Tai yra labai paprasta SQL užklausa. Jos įvykdymo laikas yra 30,39 s. (bandymas atliktas 100 kartų). Ji pagal duotą asmens kodą suranda žmogaus duomenis.



42 pav. *Neoptimali SQL užklausa*

Lentelės aprašymas pateikiamas prieduose 4 pav. Kaip matyti lentelės laukas ASM_AK yra tekstinio tipo (varchar). Jei mes SQL užklausoje asmens kodą įkelsime į kabutes, gausime:



43 pav. *Optimizuota SQL užklausa*

Kaip matyti iš rezultatų, laikas sutrumpėjo iki 0,06 s. SQL užklausos vykdymo laikui įtakos gali turėti ir toks paprastas dalykas kaip kabutės, nurodančios, kad įrašyta reikšmė yra tekstinė. Vykdymo laiko pokytis dramatiškas – 507 kartai. Ar tai yra užklausos optimizavimas? Taip. Kartais nereikia stengtis atlikti sudėtingų pakeitimų, užtenka ypač paprastų.

Turint didelę SQL užklausą (žiūrėti prieduose „Suteiktų paslaugų neatitikimas“), ją optimizuoti yra begalė būtų. Netgi atlikę užklausos optimizavimą negalime teigti, kad ji optimali. Pagal greitaveikos užtikrinimo metodiką svarbu nusistatyti, kokio rezultato mes siekiame.

8 lentelė. *Trečiojo etapo rezultatai. SQL užklausų optimizavimo rezultatai.*

	Prieš (s)	Po (s)
Maža	30,39	0,06
Didelė	4861	4503

Matome, kad netgi nedidelės SQL užklausos vykdymo laikas sutrumpėjo 30 s., o didelės net 6 min. Dirbant su didelėmis duomenų bazėmis, kur SQL užklausos turi būti ypač optimizuotos, šie rezultatai duoda teigiamos įtakos sistemos spartai.

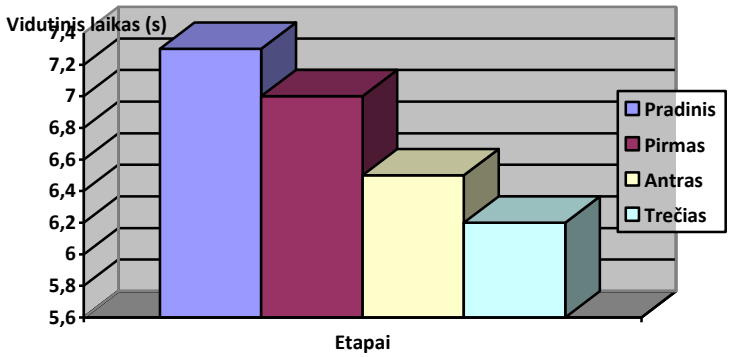
5.3. Eksperimentinio tyrimo rezultatai

Atlikus greitaveikos užtikrinimo metodikos eksperimentinį tyrimą, t.y. pritaikius ją tiriamai sistemai, buvo gauti rezultatai 9 ir 10 lentelėje.

9 lentelė. *Eksperimentinio tyrimo rezultatai. Priimtos ataskaitos.*

Etapas	Ataskaitų skaičius	Bendras laikas (s)	Vidutinis laikas (s)
Pradinis	453	3312	7,3
Pirmas	741	5214	7,0
Antras	1048	6847	6,5
Trečias	1139	7082	6,2

Iš šių rezultatų matyti, kad po kiekvieno etapo optimizacijos vidutinis ataskaitos apdorojimo laikas mažėjo (38 pav.).



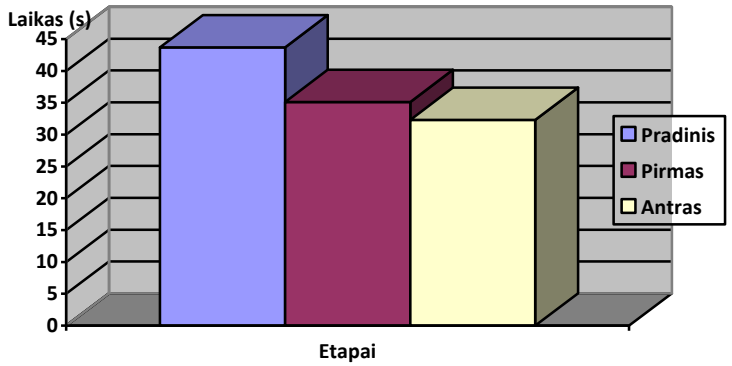
44 pav. *Eksperimentinio tyrimo rezultatai. Priimtų ataskaitų vidutinis apdorojimo laikas*

Vadinasi, pasirinktos priemonės davė teigiamų pokyčių programos spartai. Pasirinktos metodikos taikymas padėjo ataskaitų apdorojimo laiką pagerinti 1,1 s (15 %).

Posistemės, atsakingos už ataskaitų kūrimą, rezultatai pavaizduoti 10 lentelėje ir 45 pav.

10 lentelė.*Eksperimentinio tyrimo rezultatai. Sukurtos ataskaitos.*

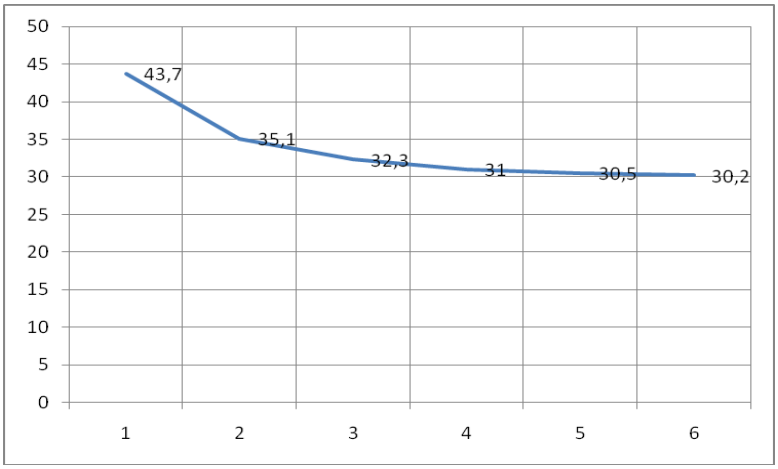
Etapas	Užsakyta ataskaitų	Vidutinis užsakytos ataskaitos įvykdymo laikas	Įvykdyta SQL užklausų	Bendras SQL užklausų vykdymo laikas (s)	Vidutinis SQL užklausos vykdymo laikas (s)
Pradinis	134	43,7	134	5861	43,7
Pirmas	746	35,1	589	26152	44,4
Antras	1967	32,3	1438	63480	44,1



45 pav. *Eksperimentinio tyrimo rezultatai. Sukurtų ataskaitų vidutinis sukūrimo laikas*

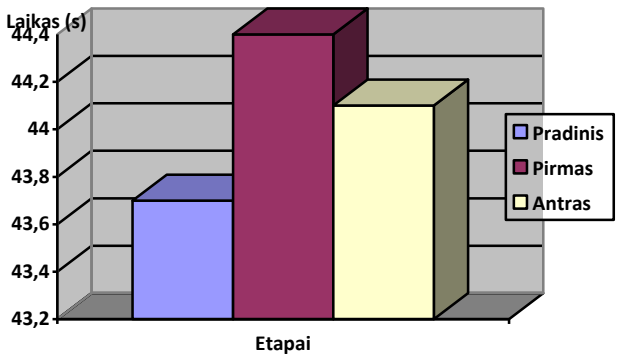
Iš 45 pav. grafiškai atvaizduotų rezultatų matyti, kad kiekviename etape vidutinis atskaitos sukūrimo laikas mažėjo. Įvertinus antrojo etapo sukurtų ataskaitų skaičių, pavyko sutaupyti $(43{,}7-32{,}3)*1967=6{,}2h$ sistemos darbo laiko. Taip pat šį valandų skaičių galime pavadinti laiku, kurio nereikėjo vartotojams sugaišti, norint gauti jiems reikiama informaciją. Todėl galime teigti, kad sistema padėjo sutaupyti ir darbuotojų laiko sąnaudas.

46 pav. pavaizduota tikėtina ataskaitų sukūrimo laiko priklausomybė nuo ataskaitų kiekio. Kuriant šią diagramą naudotasi pradinio, pirmojo ir antrojo etapų rezultatais ir kitų etapų tendencija. Matyti, kad didžiausi laiko pokyčiai įvyksta pirmuosiuose etapuose. Įdėtam darbui įvertinti galime pasiremti Pareto taisykle[24].



46 pav. *Laiko ir ataskaitų kiekio priklausomybės tendencija.*

[24] Pareto taisyklė http://lt.wikipedia.org/wiki/Pareto_principas



47 pav.*Eksperimentinio tyrimo rezultatai. Sukurtų ataskaitų vidutinis sukūrimo laikas*

Iš gautų rezultatų matyti, kad vidutinis SQL užklausos laikas nebuvo mažėjantis. Čia nebuvo įvertintas 3 etape atliktas SQL užklausų optimizavimas, nes nėra tikslinga vertinti tai kaip suminį rezultatą. Yra labai sunku nustatyti konkrečios SQL užklausos įtaką bendrai sistemos spartai. Prieš tai buvo aprašytas konkretus vienos SQL užklausos optimizavimo pavyzdys ir gauti rezultatai.

Atlikti greitaveikos užtikrinimo metodikos tyrimo rezultatai leidžia teigti, kad metodika yra efektyvi. Dviejų informacijos srautų tyrimų (priimamų ir sugeneruojamų ataskaitų) rezultatai rodo, kiek darbo laiko sutaupo programų sistema ir patys vartotojai.

5.4. Eksperimentinio tyrimo išvados

Atlikus eksperimentą ir įvertinus gautus rezultatus galima teigti, kad programų sistemos spartos tyrimas, remiantis greitaveikos užtikrinimo metodika, pavyko. Eksperimento metu išplaukusios išvados:

1) pritaikius greitaveikos užtikrinimo metodiką bendra sistemos sparta padidėjo. Tai rodo gautų rezultatų įvertinimas, jų palyginimas su pradine sistemos sparta.
2) Didžiausias našumas buvo pasiektas pirmajame tyrimo etape atlikus programos veiklos logikos optimizavimą pagal metodikos siūlomas priemones. Pasiektas 26% sistemos spartos pagerinimas.
3) Sistemoms, dirbančioms su didelėmis duomenų bazėmis itin efektyvu taikyti metodikos priemonių rinkinį, susijusį su SQL užklausų ir sąsajų su duomenų baze optimizavimu.

4) Greitaveikos užtikrinimo metodikos pritaikymas tobulinant programų sistemą leido sutaupyti keletą valandų sistemos darbo ir sumažinti vartotojų laiko sąnaudas naudojantis sistema.

6. IŠVADOS

Pagrindinės šio darbo išvados yra:

1. Analizės metu išanalizuotos programų sistemos greitaveikos didinimo priemonės ir optimizavimo būdai. Tai apima veiklos logikos, programinio kodo, kamščių, sistemos spartos ir duomenų bazės derinimo problemas, automatinio optimizavimo galimybes, lygiagrečiųjų procesų panaudojimą, algoritmų efektyvumo analizę, kodo kompiliavimo veiksmingumą ir SQL užklausų svarbą.
2. Išanalizavus programų sistemos optimizavimo būdus ir priemones nustatyta, kad dažniausiai autoriai akcentuoja skirtingas metodikas ir būdus greitaveikai užtikrinti. Bet visi autoriai sutinka, kad imant minėtų priemonių įvairias kombinacijas ir jas pritaikant programų sistemoms galima programų greitaveiką padidinti. Rezultatai ir pasiektas efektyvumas priklauso nuo pasirinktų priemonių, išsikeltų tikslų ir kaštų optimizavimui atlikti.
3. Sukurta greitaveikos užtikrinimo metodika sudaryta iš 3 etapų: programos veiklos logikos, kodo ir techninės sistemos dalies optimizavimo. Metodika leidžia užtikrinti programinės įrangos kokybę, sistemos greičio, resursų panaudojimo ir našumo aspektais.
4. Sukurta eksperimentinė sistema, veikianti viešojoje valstybinėje įstaigoje – Kauno teritorinėje ligonių kasoje. Kūrimo procese naudotos Microsft .NET technologijos bei duomenų bazių valdymo sistemos MS SQL ir Oracle. Sistema akcentuota į darbą su dideliais duomenų srautais įstaigoje – ataskaitų apdorojimu ir jų kūrimu. Sistema taip pat operuoja valstybinės reikšmės didelės apimties duomenų baze. Pagal šios sistemos informacijos srautus atliktas pilnavertis metodikos eksperimentinis tyrimas.
5. Eksperimento metu taikytas optimizavimo būdas naudojant lygiagrečiuosius procesus nepasiteisino. Gauti prasti rezultatai galėjo būti dėl eksperimentinės sistemos darbo procesų nepakankamo suderinamumo su daug atskirų gijų.
6. Eksperimento būdu ištyrus programų sistemos greitaveikos užtikrinimo metodiką gauta, kad optimizavimas atliktas programos veiklos logikos optimizavimo etape duoda didžiausią naudą iš visų optimizavimo etapų. Tai patvirtina teorijoje ir praktikoje minimą faktą, kad programinės įrangos kūrimo proceso pirminiame

etape – projektavimo metu – pastebėtos ir ištaisytos klaidos – duoda didžiausią naudą ir kainuoja mažiausiai.

7. Apibendrinus darbo rezultatus, pasiektas iki 26% sistemos greičio padidėjimas. Tai leidžia teigti, kad pasiūlytas priemonių rinkinys ir tvarka, kuria jas reikia taikyti, yra pakankamai efektyvi. Naudojantis šia metodika galima gauti gerų rezultatų esant mažiems kaštams. Pasiekti rezultatai – tai sistemos ir jos vartotojų darbo laiko sąnaudų, resursų ir lėšų sumažinimas.

7. SANTRUMPŲ IR TERMINŲ ŽODYNAS

Santrumpos:

- KTLK – Viešoji įstaiga Kauno teritorinė ligonių kasa.
- TLK – teritorinė ligonių kasa.
- VLK – Viešoji įstaiga Valstybinė ligonių kasa prie Sveikatos apsaugos ministerijos.
- Sveidra - 1999 m. visoje Lietuvoje pradėjo veikti kompiuterinė statistinių ir ekonominių duomenų surinkimo bei apdorojimo sistema „Sveidra“, kuri atsakinga už statistinių-epidemiologinių duomenų surinkimą ir analizę bei už ekonominių duomenų rinkimą ir analizę.
- KTLKIS – Kauno teritorinės ligonių kasos informacinė sistema yra šio magistro darbo eksperimentinė dalis.
- KVP – Kompensuojamų vaistų pasų išdavimo ir kontrolės sistema.

Terminai:

- SQL (angl. Structured Query Language) – struktūrizuota užklausų kalba.
- OS (angl. Operating system) – operacinė sistema.
- UML (angl. Unified Modeling Language) – unifikuota modeliavimo kalba.
- GUI (angl. Graphical User Interface) – grafinė vartotojo sąsaja.
- CSS (angl. Cascading Style Sheets) - kalba, skirta nusakyti kita struktūrine kalba aprašyto dokumento vaizdavimą. Dažniausiai CSS aprašomas HTML dokumentų pateikimas, tačiau ją galima taikyti ir įvairiems kitiems XML dokumentams (tarp jų SVG ir XUL)[25].
- MVP (angl. Model – View – Presneter) – architektūrinis šablonas siūlomas Microsoft ir skirtas realizuoti klientinei programos daliai.
- CRM[26] (angl. Customer Relationship Management, “santykių su klientais valdymas”) – būdas analizuojant ir panaudojant rinkodaros duomenų bankų duomenis bei įdarbinant komunikacines technologijas sukurti bendrą įmonės praktiką ir metodus, kurie maksimaliai padidintų kiekvieno individualaus kliento ilgalaikę vertę (ang. life

[25] Apibrėžimas paimtas iš http://lt.wikipedia.org/wiki/CSS

[26] Apibrėžimas paimtas iš http://lt.wikipedia.org/wiki/CRM

time value) įmonei. CRM suplanuoja verslo filosofiją, kurios centre yra klientas, o prie jo poreikių derinama veikla bei kultūra, reikalinga efektyviai rinkodarai, pardavimams ir paslaugų teikimui. CRM prasideda nuo verslo strategijos, kuri lemia pokyčius organizacijoje bei jos darbinėje veikloje ir yra susijusi su informacijos technologija. CRM technologija įgalina sistemingai valdyti santykius su klientais.

- DB (angl. Database) – duomenų bazė.
- DBVS (angl. database management system (DBMS)) – duomenų bazių valdymo sistema yra kompiuterinė programa ar programų paketas, skirtas duomenų bazei valdyti. Paprastai DBVS sugeba valdyti milžiniškus struktūrizuotų duomenų kiekius bei vienu metu palaikyti daugelį lygiagrečiai dirbančių vartotojų. Geriausiai žinomos DBVS: Oracle, DB2, Microsoft Access, Microsoft SQL Server, Microsoft Visual FoxPro, PostgreSQL, MySQL.
- Greitaveika – terminas, dažniausiai suprantamas kaip kiekybinė našumo išraiška. Artimiausias sinonimas sparta.
- Duomenų gavyba (angl. Data mining) - būdas išrinkti aktualią informaciją iš didelių duomenų kiekių. Šis būdas paprastai vartojamas įvairių verslo intelektualinių organizacijų ir finansų analitikų, bet vis daugiau vartojamas ir moksle išrinkti informaciją iš ypač didžiulių duomenų rinkinių, sugeneruotų modernių eksperimentinių ir stebėjimo metodų. Duomenų gavyba buvo apibūdinta kaip: „numanomas reikšmingas informacijos išrinkimas iš prieš tai nežinotos ir potencialiai naudingos informacijos“ ir „naudingos informacijos išrinkimo mokslas iš didelių duomenų rinkinių ar duomenų bazių“ [2].
- Duomenų saugykla (angl. Data warehouse) yra pagrindinė talpykla bendriems organizacijos duomenims laikyti. Ją sudaro neapdirbta medžiaga, skirta valdymo sprendimų pagalbos sistemai. Kritinis faktorius, kodėl naudojama duomenų saugykla yra tai, kad duomenų analitikas gali įvykdyti sudėtingas užklausas ir analizes, tokias kaip duomenų gavyba, ir išgauti informacijos nesulėtindamas operacinės sistemos [5].
- Duomenų bazės derinimas (angl. Database tuning) - apibūdina grupę veiklų, naudojamų optimizuoti ir homogenizuoti duomenų bazės spartą. Dažnai duomenų bazės derinimas susitapatina su užklausų optimizavimu. Bet tai kartu susiję ir su duomenų bazės failų nustatymais, duomenų bazės valdymo sistema, operacine sistema ir technine įranga, su kuria DBVS veikia. Duomenų bazės derinimo tikslas yra

maksimizuoti sistemos resursų panaudojimą, kad darbus atlikti būtų kaip galima efektyviau bei progresyviau. Daugelis sistemų yra sukurtos valdyti darbus gana efektyviai. Bet taip pat galima žymiai padidinti spartą, parenkant tinkamus nustatymus ir konfigūraciją duomenų bazei, ir derinant DBVS [8].

- SQL užklausų derinimas (angl. SQL statements tuning) – SQL užklausų optimizavimas. Daugiau informacijos literatūros šaltiniuose [10] ir [11].
- Amdahlo dėsnis naudojamas apskaičiuoti maksimaliam teoriniam pagerėjimui, kai tobulinama sistemos dalis. Šis dėsnis taip pat dažnai naudojamas lygiagrečiuose procesuose nuspėti teorinį spartos padidėjimą, naudojant daugiau nei vieną procesorių. Santykinis pagerėjimas pagal Amdahlo dėsnį apskaičiuojamas pagal šią formulę:

$$\frac{1}{\sum_{k=0}^{n}(\frac{P_k}{S_k})}$$

P_k – instrukcijų dalis, kuri gali būti pagreitinta (arba sulėtinta), S_k – pagreitėjimo daugiklis (kur 1 reiškia nepakitusia būsena), k - žymė kiekvienai iteracijai , n - yra bendras sistemos pagreitėjimų /sulėtėjimų skaičius.

- Valdančioji lentelė (angl. Driving table) – tai lentelė, kuri Oracle DBVS sujungiant keletą lentelių per JOIN sakinį, yra apdorojama pirmiausia.

8. LITERATŪRA

1) Whang K.Y., Brady S. High-Performance Expert System-DBMS Interface For Network Management and Control. In IEEE journal onr selected areas in communications, volume 7, Issue 3, April 1989, p. 408 – 417, [žiūrėta 2009-02-10]. Prieiga per internetą: Http://Ieeexplore.Ieee.Org/Xpls/Abs_All.Jsp?Arnumber=16873

2) Hand D., Mannila H., Smyth P. Principles of Data Mining. ISBN 0-262-08290-X. MIT Press, Cambridge, MA, 2001.

3) Ian H., Frank E. *Practical Machine Learning Tools and Techniques*. Academic Press, Published 2005, ISBN 0120884070 [žiūrėta 2009-03-25]. Prieiga per internetą: http://books.google.com/books?id=QTnOcZJzlUoC&dq=data+mining

4) Alex A. F., Lavington S. H. Mining Very Large Databases with Parallel Processing. Published 1998 Springer, Database management, ISBN 0792380487, [žiūrėta 2009-02-10]. Prieiga per internetą: http://books.google.com/books?id=CvF3rxep5dQC&dq=large+databases+speed

5) Kimball R., Reeves L., Thornthwaite W., Ross M., Thornwaite W. The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing and Deploying Data Warehouses with CD Rom, 1st edition. Publisher: John Wiley & Sons, Inc. New York, NY, USA, Textbook Paperback, Year of Publication: 1998. ISBN:0471255475 [žiūrėta 2009-03-03]. Prieiga per internetą: http://portal.acm.org/citation.cfm?id=551694

6) Eder J., Missikoff M., Contributor J. Eder. Advanced Information Systems Engineering: 15th International Conference. Springer, computer-aided software engineering/ Congresses, Published 2003, ISBN 3540404422, [žiūrėta 2009-03-06] Prieiga per internetą: http://books.google.com/books?id=3r-vREFcj7cC&dq=subject:%22Computers+/+Artificial+Intelligence%22

7) Whalen E., Garcia M., Adrien DeLuca S., Thompson D. *Microsoft SQL Server 2000 Performance Tuning Technical Reference*, , ISBN 0-7356-1270-6, Microsoft Press, 2001, p. 438.

8) Agrawal S., Chaudhuri S., Kollar L., Marathe A., Narasayya V., Syamala M. Database Tuning Advisor for Microsoft SQL Server 2005. Microsoft Corporation, One Microsoft Way, Redmond, WA 98052., USA. [žiūrėta 2009-02-15]. Prieiga per internetą: http://www.cs.toronto.edu/vldb04/protected/eProceedings/contents/pdf/IND4P3.PDF

9) Inmon W. H., Rock C. The data warehouse and data mining. Communications of the ACM archive, Volume 39, Issue 11 (November 1996), Pages: 49 – 50. Publisher ACM New York, NY, USA 1996, ISSN:0001-0782, [žiūrėta 2009-03-09]. Prieiga per internetą: http://portal.acm.org/citation.cfm?id=240470

10) Dam Sajal. SQL Server Query Performance Tuning Distilled. ISBN 1590594215, 9781590594216, Edition: 2, Published by Apress, 2004. [Žiūrėta 2009-04-20] Prieiga per internetą: http://books.google.com/books?id=yK9wRfYX9nkC&dq=SQL+statements+timing+analysis&source=gbs_summary_s&cad=0

11) Tow D. SQL Tuning. ISBN 0596005733, 9780596005733, Published by O'Reilly, 2003. [Žiūrėta 2009-04-20]. Prieiga per internetą: http://books.google.lt/books?id=MW0RI-I5W4oC&dq=sql+tuning&source=gbs_summary_s&cad=0

12) Moré Jorge J., Wright Stephen J. *Optimization software guide*. ISBN 0898713226, 9780898713220, Edition: 2, published by SIAM, 1993.

13) Smith C. U., Performance Engineering of Software Systems. MA, Addison-Wesley, 1990. [Žiūrėta 2009-05-15]. Prieiga per internetą: www.perfeng.com.

14) Smith C. U. and Williams L. G. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. MA, Addison-Wesley, Boston, 2002. [Žiūrėta 2009-05-15]. Prieiga per internetą: www.perfeng.com.

15) Williams L. G. and Smith C. U., PASASM: A Method for the Performance Assessment of Software Architectures. (Submitted for publication) 2002 . [Žiūrėta 2009-05-15]. Prieiga per internetą: www.perfeng.com.

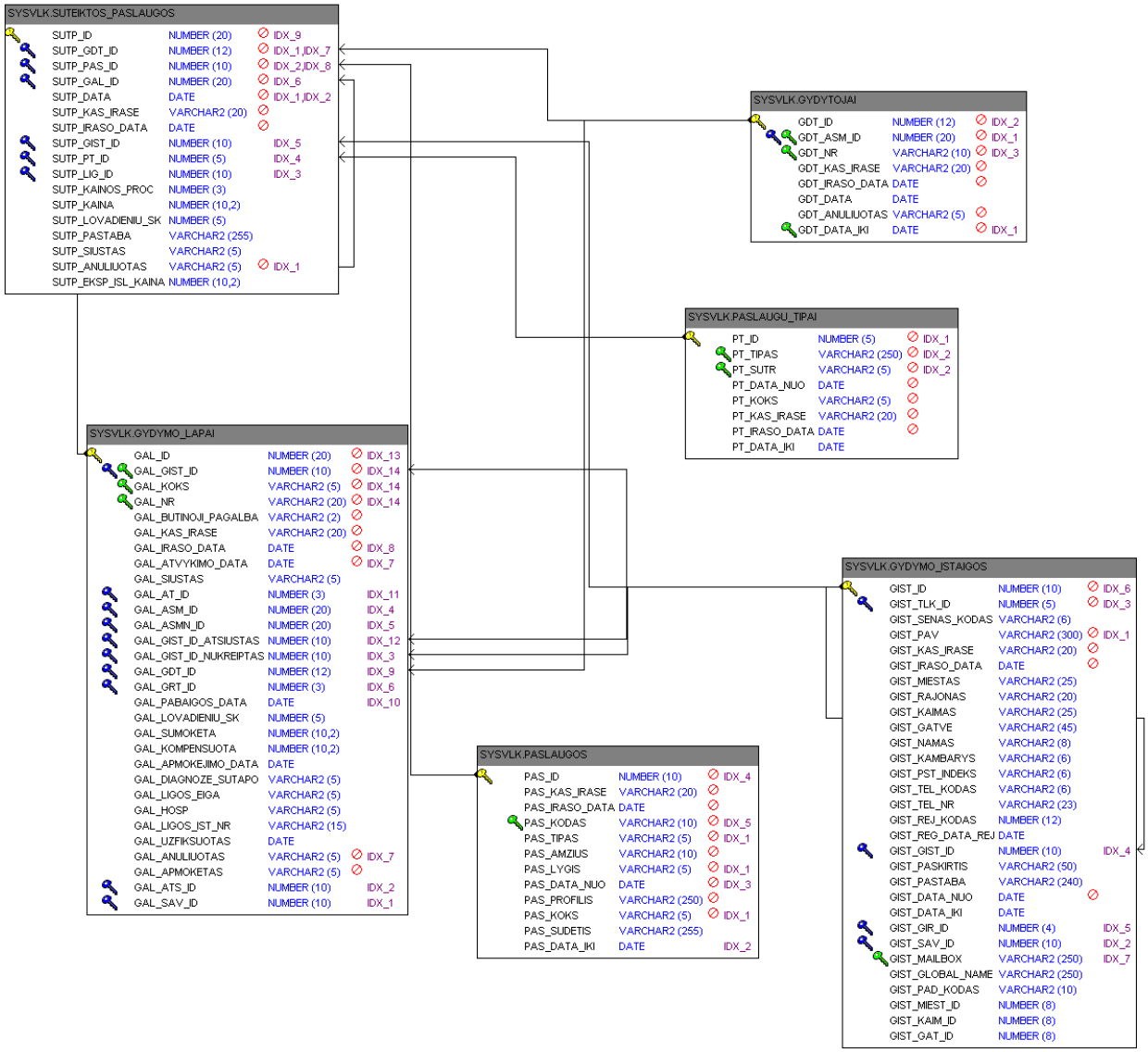
16) Williams L. G. and Smith C. U. The Business Case for Software Performance Engineering. (Submitted for publication) 2002. [Žiūrėta 2009-05-15]. Prieiga per internetą: www.perfeng.com.

17) Lindh B. *Application Performance Optimization*. Sweden: Sun BluePrints™OnLine – March, 2002.

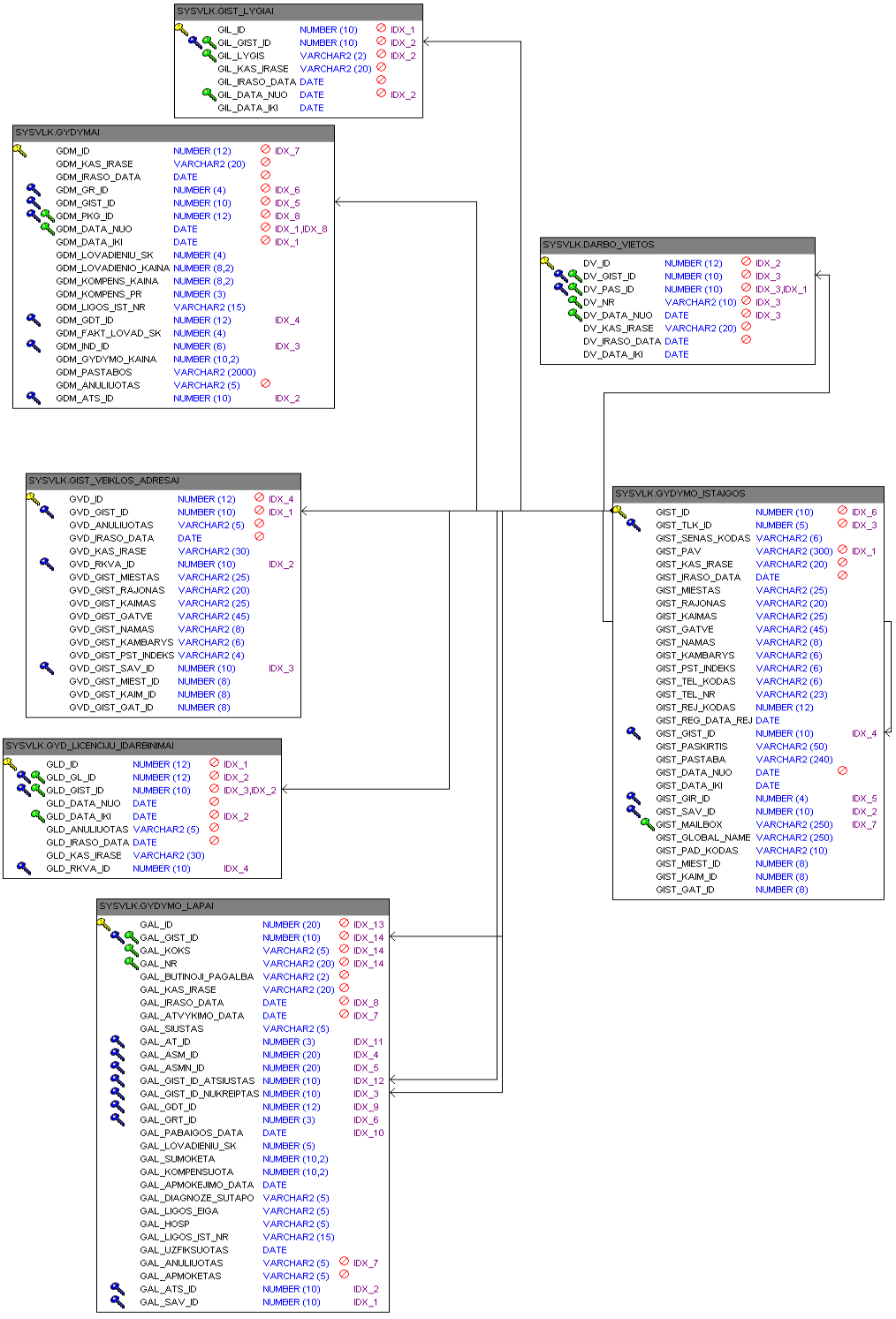
18) Charvet F., Pande A. Database Performance Study. [Žiūrėta 2008.04.09]. Prieiga per internetą: http://www.umsl.edu/divisions/business/mis/bov/TuningPaperV5.pdf

PRIEDAI

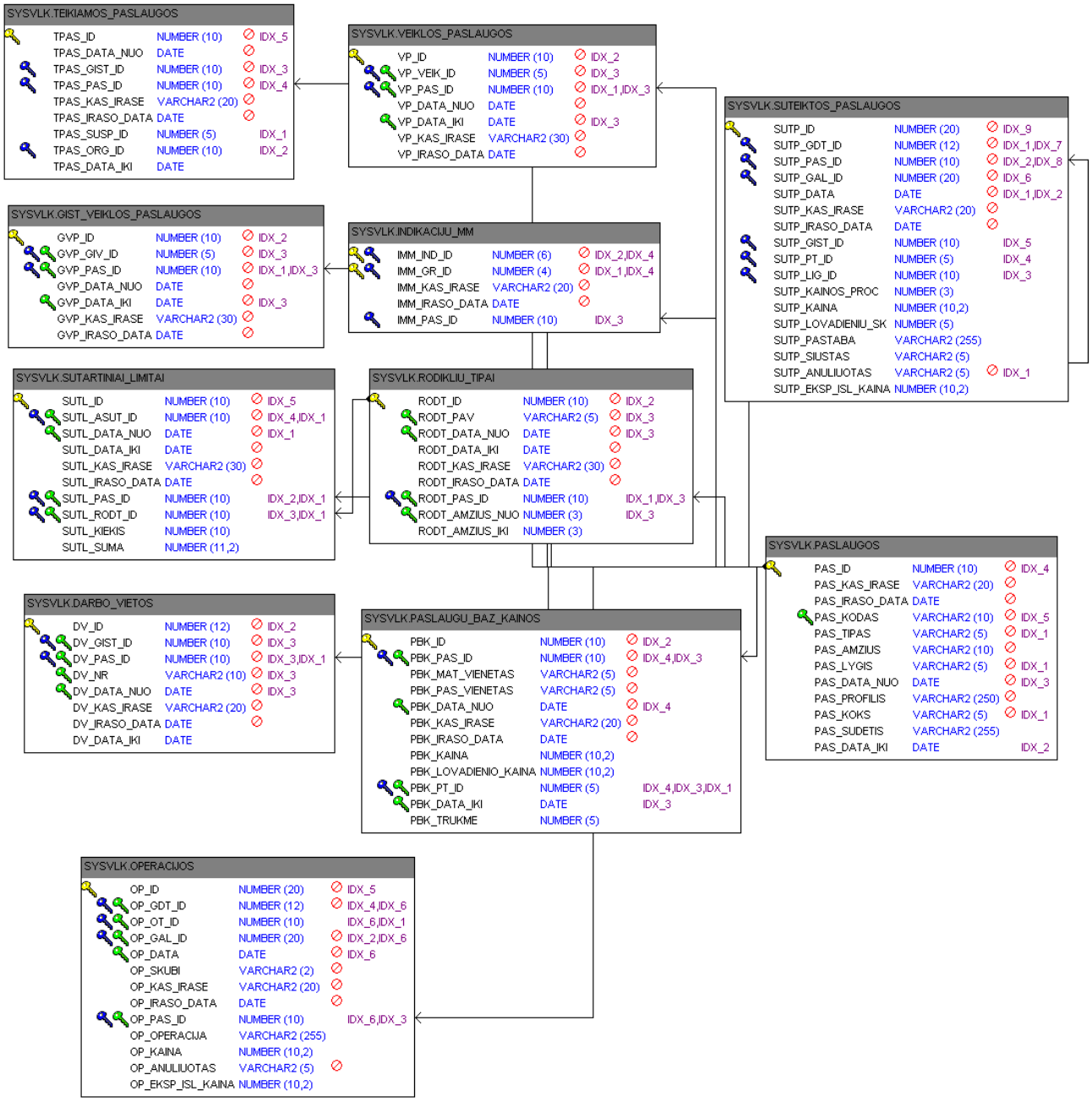
Sveidros duomenų bazės lentelių schemos pagal funkcines grupes



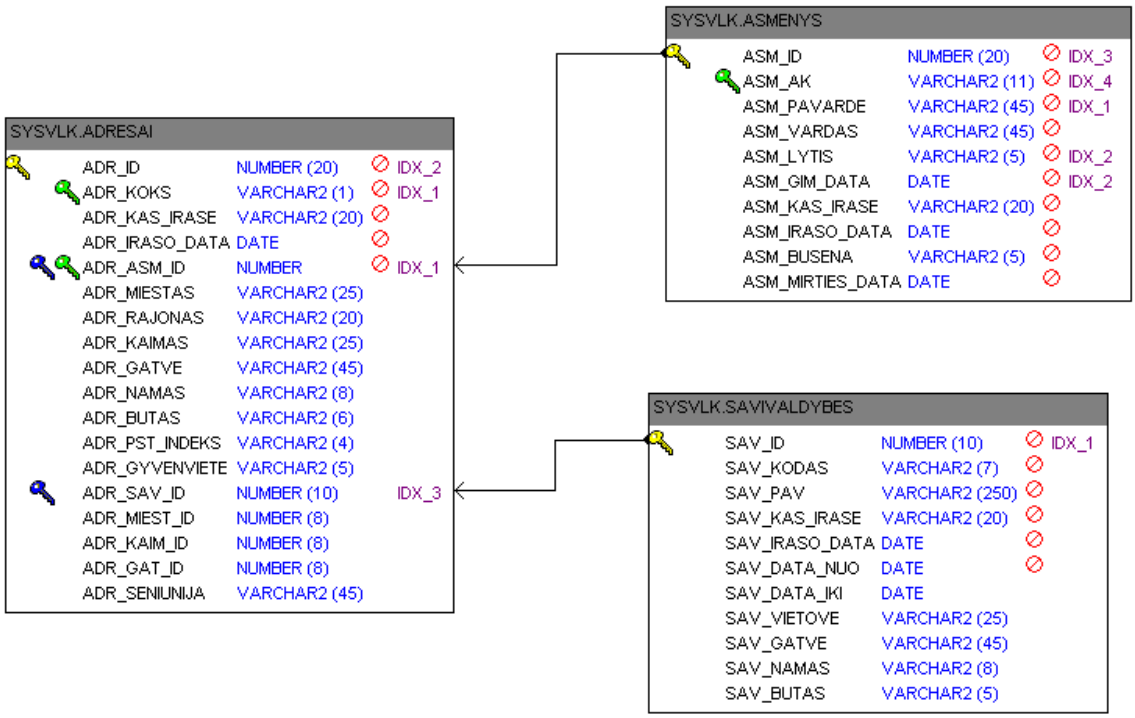
1. pav. Duomenų lentelė „Suteiktos paslaugos“ ir susijusios lentelės



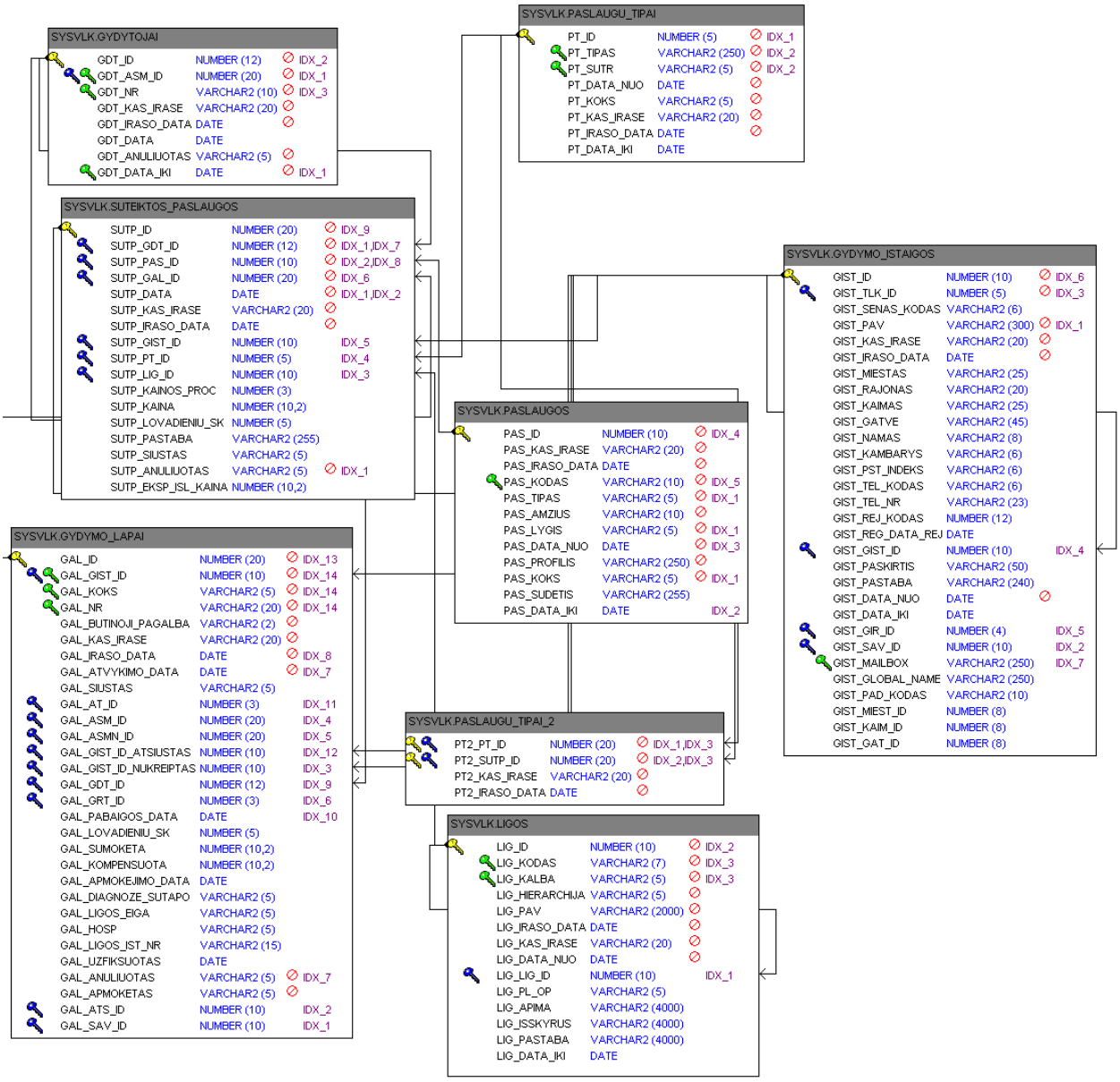
2 pav. Duomenų lentelė „Gydymo lapai“ ir susijusios lentelės



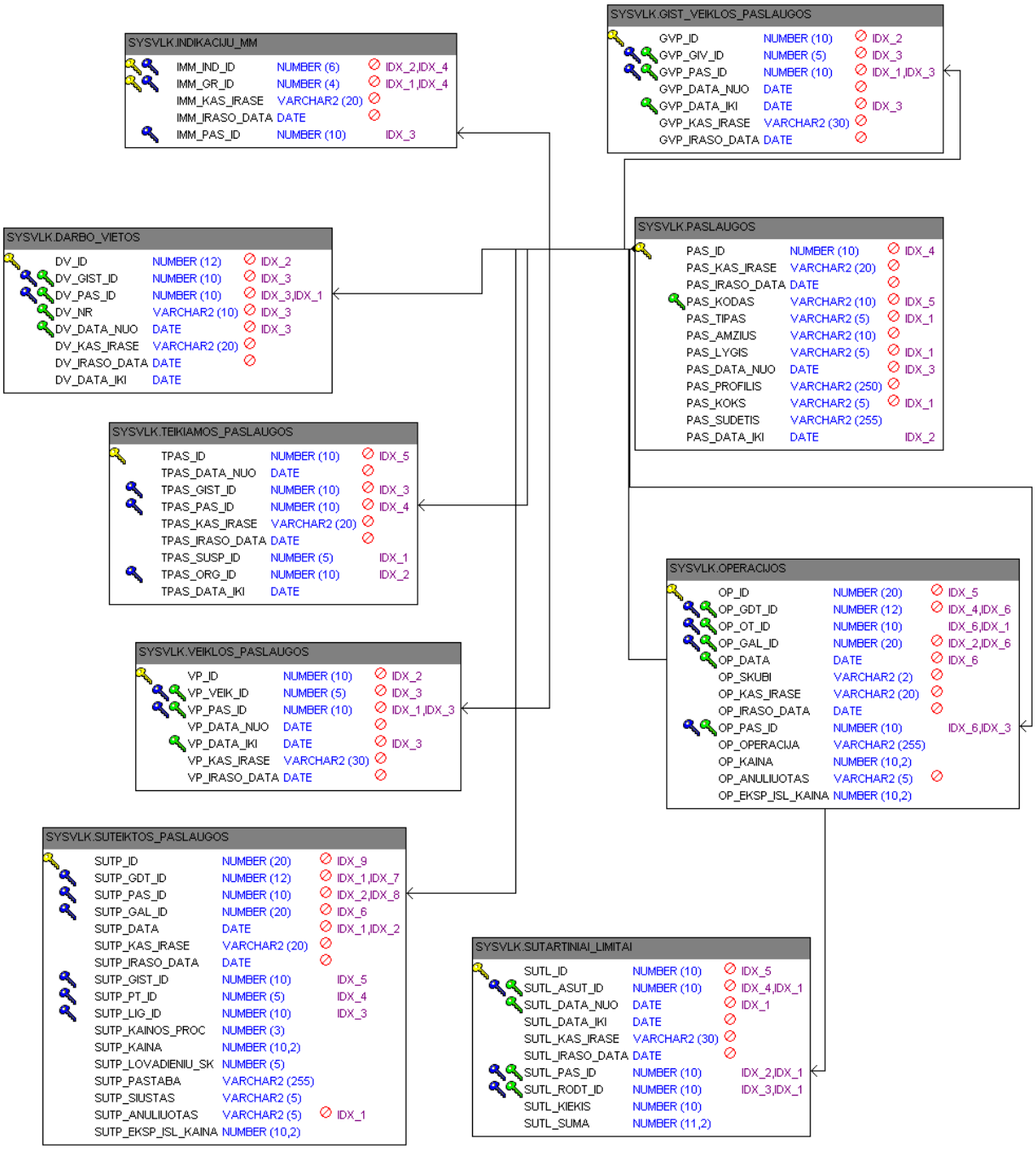
3 pav. Duomenų lentelė „Rodiklių tipai“ ir susijusios lentelės



4 pav. Duomenų lentelė „Asmenys“ ir susijusios lentelės



5 pav. Duomenų lentelė „Gydymo įstaigos“ ir susijusios lentelės



6 pav. Duomenų lentelė „Paslaugos“ ir susijusios lentelės

Pavyzdinės SQL užklausos iš Sveidros duomenų bazės

1) Suteiktų paslaugų neatitikimas (klaidos paslaugų grupėse)

Neoptimizuota:

```
select gal_gist_id as gist_id,gal_nr,apmok_tlk,sutp_kaina,viskas.pas_kodas,viskas.pas_profilis,viskas.lig_kodas,tip_apras from
(select reik_pas.*,tip_gr_id,tip_apras from
(select  * from
(select laps.*,sutp_pas_id,sutp_kaina,sutp_lig_id,sutp_anuliuotas,sutp_data,lig_kodas from
(select lap.*,lks_tlk_id ,decode(tlk_tlk_id,16,tlk_id,tlk_tlk_id) tlk,tlk_miestas as apmok_tlk from
(select  gal_id,gal_gist_id,gal_nr,gal_sav_id from gydymo_lapai
where gal_pabaigos_data>=TO_DATE('#data_nuo#','YYYY.MM.DD')
and gal_pabaigos_data<TO_DATE('#data_iki#','YYYY.MM.DD')
and gal_koks='S'
and gal_gist_id in ( #gist_id#)

and gal_anuliuotas='N') lap ,sysvlk.lik_kasos_sav,ligoniu_kasos
where gal_sav_id+0=lks_sav_id(+)
and lks_tlk_id=tlk_id(+))laps,suteiktos_paslaugos,ligos
where gal_id+0=sutp_gal_id
and sutp_lig_id+0=lig_id(+))lap_pas,
(select pas_id,pas_kodas,to_number(pas_tipas) as pas_tipas,pas_profilis from paslaugos where
 to_number(pas_tipas) not  in (11,12,20,21,22,27,29,33,34,39,41,42,40,48,4,5,7,8,9,51,52,53)
 and to_number(pas_kodas) not in (701,702,703,704,836,837,838,839,840,841,1254,1255,

1394,1396,1412,1150,1160,1378,1152,1154,1162,1164,1422,1424,1489,1491,2008,2011,2010
))pasl
where sutp_pas_id+0=pas_id
and sutp_anuliuotas='N'
and nvl(sutp_kaina,0)>0)reik_pas,pasl_tip
where pas_tipas=pasl_tip.tip_tipas
and trunc(sutp_data) between tip_nuo and tip_iki )viskas,pasl_grupes
where viskas.tip_gr_id=pasl_grupes.gr_gr_id(+)
and viskas.sutp_lig_id=pasl_grupes.gr_lig_id(+)
and trunc(sutp_data)>=gr_nuo(+)
and trunc(sutp_data)<gr_iki(+)
and gr_lig_id is null

union all

select gal_gist_id as gist_id,gal_nr,apmok_tlk,sutp_kaina,viskas.pas_kodas,viskas.pas_profilis,viskas.lig_kodas,tip_apras from
(select ilgi.*,tip_gr_id,tip_apras from
```

```
(select  lap_pas.*,sysvlk.lik_kasos_sav.lks_tlk_id,
decode(tlk_tlk_id,16,tlk_id,tlk_tlk_id) tlk,tlk_miestas as apmok_tlk from
(select  gal_id,gal_gist_id,gal_nr,gal_sav_id,lig_kodas,a.*
from gydymo_lapai ,ligos,
(select  suteiktos_paslaugos.*,pas_kodas,to_number(pas_tipas) as pas_tipas,pas_profilis
from suteiktos_paslaugos,paslaugos
where   sutp_pas_id=pas_id
and TO_NUMBER(pas_tipas) in (33,34,39,41,48,11)
and sutp_data>=TO_DATE('#data_nuo#','YYYY.MM.DD')
and sutp_data<TO_DATE('#data_iki#','YYYY.MM.DD')
and sutp_anuliuotas='N'
and nvl(sutp_kaina,0)>0) a
where a.sutp_gal_id+0=gal_id
and a.sutp_lig_id+0=lig_id(+)
and gal_anuliuotas='N'
and gal_gist_id in ( #gist_id#)


)lap_pas,sysvlk.lik_kasos_sav,ligoniu_kasos
where gal_sav_id+0=lks_sav_id(+)
and lks_tlk_id+0=tlk_id(+))ilgi,pasl_tip
where pas_tipas=pasl_tip.tip_tipas
and trunc(sutp_data) between tip_nuo and tip_iki )viskas,pasl_grupes
where viskas.tip_gr_id=pasl_grupes.gr_gr_id(+)
and viskas.sutp_lig_id=pasl_grupes.gr_lig_id(+)
and trunc(sutp_data)>=gr_nuo(+)
and trunc(sutp_data)<gr_iki(+)
and gr_lig_id is null

order by gist_id,tip_apras,pas_kodas
```

Optimizuota

```
        select m_gist_id,gist_id,gal_nr,decode(tlk_miestas,null,'Kaunas',tlk_miestas) as
apmok_tlk ,sutp_kaina,pas_kodas,pas_profilis,sutp_data from
        (select www.* from
        (select istaigos.gist_gist_id as m_gist_id,viskas.* from
        (select gal_gist_id as
gist_id,gal_nr,gal_sav_id,sutp_kaina,pasl.pas_kodas,pasl.pas_profilis,sutp_data,sutp_pas_id
from
        (select lap.*,sutp_pas_id,sutp_kaina,sutp_lig_id,sutp_anuliuotas,sutp_data from
        (select  gal_id,gal_gist_id,gal_nr,gal_sav_id from gydymo_lapai
        where gal_pabaigos_data>=TO_DATE('#data_nuo#','YYYY.MM.DD')
        and gal_pabaigos_data<TO_DATE('#data_iki#','YYYY.MM.DD')

        and gal_anuliuotas='N') lap ,suteiktos_paslaugos
        where gal_id+0=sutp_gal_id )lap_pas,
```

```
        (select pas_id,pas_kodas,to_number(pas_tipas) as pas_tipas,pas_profilis from paslaugos
where
          to_number(pas_tipas) not in
(7,8,9,12,20,21,22,27,29,33,34,39,40,41,42,48,11,53,65,66,76))pasl
        where sutp_pas_id+0=pas_id
        and sutp_anuliuotas='N'

        union all

        select gal_gist_id as
gist_id,gal_nr,gal_sav_id,sutp_kaina,pas_kodas,pas_profilis,sutp_data,sutp_pas_id from
gydymo_lapai,
        (select  sutp_gal_id,sutp_pas_id,sutp_kaina,sutp_lig_id,sutp_anuliuotas,sutp_data,
        pas_kodas,to_number(pas_tipas) as pas_tipas,pas_profilis
        from suteiktos_paslaugos,paslaugos
        where   sutp_pas_id=pas_id
        and TO_NUMBER(pas_tipas) in (11,33,34,39,40,41,42,48,65,66,76)
        and sutp_data>=TO_DATE('#data_nuo#','YYYY.MM.DD')
        and sutp_data<TO_DATE('#data_iki#','YYYY.MM.DD')
        and sutp_anuliuotas='N' ) a
        where a.sutp_gal_id+0=gal_id
        and gal_anuliuotas='N'


        union all

        select gal_gist_id as
gist_id,gal_nr,gal_sav_id,sutp_kaina,pas_kodas,pas_profilis,sutp_data,sutp_pas_id from
        (select lap.*,op_pas_id as sutp_pas_id,op_kaina as sutp_kaina,op_data as sutp_data
from
        (select  gal_id,gal_gist_id,gal_nr,gal_sav_id from gydymo_lapai
        where gal_pabaigos_data>=TO_DATE('#data_nuo#','YYYY.MM.DD')
        and gal_pabaigos_data<TO_DATE('#data_iki#','YYYY.MM.DD')


        and gal_koks='S'
        and gal_anuliuotas='N') lap ,operacijos
        where gal_id+0=op_gal_id
        and op_pas_id is not null
        and op_anuliuotas='N')lap_pas,paslaugos
        where sutp_pas_id+0=pas_id)viskas, istaigos
        where viskas.gist_id=istaigos.gist_id(+))www,sut_pasl
        where www.m_gist_id=sut_gist_id(+)
        and www.sutp_pas_id=sut_pas_id(+)
        and trunc(sutp_data)>=sut_nuo(+)
        and trunc(sutp_data)<=sut_iki(+)
```

```
        and ( sut_id is null or trunc(sutp_data)>TO_DATE('#data_iki#','YYYY.MM.DD'))
)blogi,sysvlk.lik_kasos_sav,ligoniu_kasos
        where gal_sav_id+0=lks_sav_id(+)

        and lks_tlk_id=tlk_id(+)
        and   gist_id in ( #gist_id#)
        order by m_gist_id,gist_id,to_number(pas_kodas),sutp_data
```

2) Dienos stacionaro paslaugos

Neoptimizuota

```
Select gydymo_istaigos.gist_pav as gist_pav, menuo, sum (asm) as asm, sum(kaina) as kaina,
sum(lovad_sk) as lovad_sk
from(
Select gal_gist_id, substr(trunc (gal_pabaigos_data, 'MM'),0,7) as menuo  , count(distinct
sutp_gal_id) asm, sum(sutp_kaina) as kaina, sum(sutp_lovadieniu_sk) as lovad_sk,

sum(mok) as mok, sum (nemok) as nemok

from(

select sutp_gal_id, nvl(sutp_kaina,0) as sutp_kaina, nvl(sutp_lovadieniu_sk,0) as
sutp_lovadieniu_sk,
decode (nvl(sutp_kaina,0), 0, 0, 1) as mok , decode (nvl(sutp_kaina,0), 0, 1, 0) as nemok  ,
sutp_anuliuotas
from(

Select * from paslaugos where pas_tipas in (44,46) /*51,  44,46*/
) aaa, suteiktos_paslaugos

where aaa.pas_id+0 = sutp_pas_id
) bbb,

(select * from gydymo_lapai,lik_kasos_sav
where gal_sav_id = lks_sav_id
and lks_tlk_id = 5) gydymo_lapai
where bbb.sutp_gal_id = gal_id
and   ((gal_pabaigos_data >=TO_DATE('#data_nuo#','YYYY.MM.DD')
and gal_pabaigos_data <TO_DATE('#data_iki#','YYYY.MM.DD') ) or

(  gal_pabaigos_data >=add_months(TO_DATE('#data_nuo#','YYYY.MM.DD') ,-12 )
and gal_pabaigos_data < add_months(TO_DATE('#data_iki#','YYYY.MM.DD') , -12)))

and gal_anuliuotas = 'N'
and sutp_anuliuotas = 'N'
```

```
group by gal_gist_id, substr(trunc (gal_pabaigos_data, 'MM'),0,7)

) ccc, istaigos,  gydymo_istaigos

where ccc.gal_gist_id = istaigos.gist_id
and istaigos.gist_gist_id = gydymo_istaigos.gist_id

group by gydymo_istaigos.gist_pav, menuo
```

Optimizuota

```
Select gydymo_istaigos.gist_pav as istaiga,
sum(decode(metai, 2003,lovad_sk,0)) as m2003_lov
, sum(decode(metai, 2004,lovad_sk,0)) as m2004_lov,
 sum(decode(metai, 2005,lovad_sk,0)) as m2005_lov,
 sum(decode(metai, 2006,lovad_sk,0)) as m2006_lov,

sum(decode(metai, 2003,mok,0)) as m2003_psl
, sum(decode(metai, 2004,mok,0)) as m2004_psl,
 sum(decode(metai, 2005,mok,0)) as m2005_psl,
 sum(decode(metai, 2006,mok,0)) as m2006_psl,

sum(decode(metai, 2003,asm,0)) as m2003_asm
, sum(decode(metai, 2004,asm,0)) as m2004_asm,
 sum(decode(metai, 2005,asm,0)) as m2005_asm,
 sum(decode(metai, 2006,asm,0)) as m2006_asm

/*asm,  kaina,  lovad_sk, mok,  nemok*/
from(
Select gal_gist_id, substr(trunc (gal_pabaigos_data, 'YYYY'),0,4) as metai  , count(distinct
sutp_gal_id) asm, sum(sutp_kaina) as kaina, sum(sutp_lovadieniu_sk) as lovad_sk,

sum(mok) as mok, sum (nemok) as nemok

from(
select sutp_gal_id, nvl(sutp_kaina,0) as sutp_kaina, nvl(sutp_lovadieniu_sk,0) as
sutp_lovadieniu_sk,
decode (nvl(sutp_kaina,0), 0, 0, 1) as mok , decode (nvl(sutp_kaina,0), 0, 1, 0) as nemok  ,
sutp_anuliuotas
from(

Select * from paslaugos where pas_tipas in (44,46)
) aaa, suteiktos_paslaugos

where aaa.pas_id+0 = sutp_pas_id
) bbb, gydymo_lapai
where bbb.sutp_gal_id = gal_id
and   gal_pabaigos_data >=TO_DATE('#data_nuo#','YYYY.MM.DD')
```

```
and gal_pabaigos_data <TO_DATE('#data_iki#','YYYY.MM.DD')
and gal_anuliuotas = 'N'
and sutp_anuliuotas = 'N'
group by gal_gist_id, trunc (gal_pabaigos_data, 'YYYY')

) ccc, istaigos,  gydymo_istaigos

where ccc.gal_gist_id = istaigos.gist_id
and istaigos.gist_gist_id = gydymo_istaigos.gist_id

group by gydymo_istaigos.gist_pav
```