KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Mindaugas Radžius

DBVS duomenų struktūrų ir jų apdorojimo algoritmų tyrimas bei optimizavimas

Magistro darbas

Darbo vadovas
doc. dr. V. Pilkauskas

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Mindaugas Radžius

DBVS duomenų struktūrų ir jų apdorojimo algoritmų tyrimas bei optimizavimas

Magistro darbas

Kalbos konsultantė	Vadovas
Lietuvių k. katedros dr.	doc. dr. V. Pilkauskas
I. Mickienė	2006-05-
2006-05-	

Recenzentas	Atliko
doc. dr. R. Butleris	IFM-0/1 gr. stud.
2006-05-	Mindaugas Radžius
	2006-05-

Kaunas, 2006

Turinys

1. Įvadas ... 8
2. Duomenų suspaudimo, apdorojimo algoritmų analizė ir optimizavimas ... 9
2.1. Medžio struktūra ... 11
2.2. Medžio elementų struktūra ... 12
2.3. Medžio konstravimas ... 13
2.4. Duomenų atstatymas ... 17
2.5. Paieška medyje ... 20
2.6. STL konteineriai ... 22
2.6.1. Vektorius ... 24
2.6.2. Dekas ... 24
2.6.3. Sąrašas ... 25
2.6.4. Aibė ... 25
2.6.5. Žemėlapis ... 26
2.6.6. Bendras konteinerių palyginimas ... 28
3. Duomenų manipuliacijos algoritmai ... 28
3.1. Kursoriai ... 28
3.1.1. Vidinė kursoriaus struktūra ... 29
3.1.2. Rūšiavimo kursorius ... 32
3.1.3. Grupavimo kursorius ... 36
3.1.4. Apjungimo kursorius ... 37
3.2. Paieškos filtras ... 38
3.3. Bitų filtras ... 40
4. SQL užklausų gramatinis nagrinėjimas ... 40
4.1.1. Skaičiuojamosios išraiškos ... 41
4.1.2. Duomenų išrinkimas ... 41
4.1.3. Lentelių sąrašas ... 42
4.1.4. Paieškos kriterijus ... 42
4.1.5. Rūšiavimo kriterijus ... 45
4.1.6. Grupavimo kriterijus ... 46
4.2. Sąlygų tikrinimas ir lentelių apjungimo algoritmas ... 46
4.2.1. Vienos lentelės sąlygų tikrinimas ... 47

4.2.2. Kelių lentelių sąlygų tikrinimas 48
5. DBVS sistemų užklausų vykdymo testai 51
6. Išvados 54
7. Literatūra 56
8. Data structures and processing algorithms of DBMS research and optimization 57
9. Santrumpų ir terminų žodynas 58
10. Priedai 59
10.1. TPCH testų SQL užklausos 59
10.2. Straipsnis 68

Lentelės

1 lentelė – Pavyzdinė duomenų lentelė „Table1“ 16

2 lentelė – Bendras konteinerių palyginimas 28

3 lentelė – DBVS prototipo užklausų vykdymo greitis 51

4 lentelė – ORACLE užklausų vykdymo greitis 52

5 lentelė – Microsoft SQL Server užklausų vykdymo greitis 52

6 lentelė – NPBase užklausų vykdymo greitis 52

Paveikslai

1 pav. Duomenų suspaudimo priklausomybė nuo laiko [2] 10
2 pav. Duomenų išpakavimo priklausomybė nuo laiko [2] 10
3 pav. Bendra medžio struktūra 11
4 pav. Ryšio ir duomenų elementų struktūra 12
5 pav. Medžio struktūros pavyzdys 13
6 pav. Medžio poaibiai, turintys po 4 ir 8 duomenų elementus 14
7 pav. Medžio poaibių skirstymas 15
8 pav. Medžio pavyzdys su duomenimis 17
9 pav. Duomenų surinkimo pavyzdys 18
10 pav. Trumpiausio kelio sekos radimo pavyzdys 20
11 pav. Unikalių reikšmių ir įrašų susiejimas 21
12 pav. Sąrašo iteratoriaus veikimo pavyzdys [3, 81p] 23
13 pav. Vektoriaus struktūra [3, 132p] 24
14 pav. Vidinė deko struktūra [3, 143p] 24
15 pav. Deko struktūra [3, 143p] 25
16 pav. Sąrašo vidinė struktūra [3, 148p] 25
17 pav. Elementų įterpimo ir pašalinimo operacijos sąraše [3, 153p] 25
18 pav. Aibės dvejetainis konstravimo medis [3, 157p] 26
19 pav. Aibių pavyzdys [3, 156p] 26
20 pav. Žemėlapio (su unikaliais ir pasikartojančiais raktais) pavyzdys [3, 172p] 27
21 pav. Žemėlapio su unikaliais raktais dvejetainis medis [3, 173p] 27
22 pav. Pagrindiniai kursoriaus struktūros elementai 30
23 pav. Kursoriaus buferių pavyzdys 30
24 pav. Kursoriaus duomenų saugojimas operatyviojoje atmintyje ir faile 31
25 pav. Blokų įrašų išsidėstymas prieš ir po rūšiavimo 33
26 pav. Buferio dydžio įtaka rūšiavimo greičiui 36
27 pav. Apjungimo kursoriaus ir jo deskriptoriaus struktūra 38
28 pav. Užklausos intervalų apjungimo pavyzdys 39
29 pav. Paieškos kriterijaus sąlygų hierarchijos medžio pavyzdys 44
30 pav. Paieškos kriterijaus hierarchinių sąlygų medžio vektoriaus struktūra 45
31 pav. Stulpelių priklausomybės žemėlapis 47

32 pav. Dviejų lentelių apjungimo skirtingais būdais palyginimų kiekių santykis 49
33 pav. ORACLE ir DBVS prototipo užklausų vykdymo greičių santykis 53

1. Įvadas

Daugumoje dabartinių programinių paketų duomenims saugoti ir apdoroti yra naudojamos reliacinės duomenų bazių valdymo sistemos (pvz.: *Microsoft SQL Server*, *MySQL*, *ORACLE*, *DB2* ir kt.). Visos šios duomenų bazių valdymo sistemos (DBVS) yra diskinės, t.y. duomenis saugo pastovioje atmintyje (diske), o kai reikia užsikrauna duomenis į operatyviąją atmintį. Toks duomenų laikymo ir apdorojimo būdas yra sąlyginai pigus lyginat su pastoviosios disko ir operatyviosios atminties kainų santykio. Pastovioji disko atmintis yra žymiai pigesnė už operatyviąją atmintį. Šis kainų santykis gali siekti 200 kartų ir dar daugiau. Laikyti reikiamus duomenų bazės duomenis operatyviojoje atmintyje yra santykinai brangu, tačiau tai suteikia daug didesnį DBVS darbo greitį. Šiuolaikinės pastoviosios disko ir operatyviosios atminties duomenų nuskaitymo greičių santykis gali siekti 100 kartų. Operatyviosios atminties didelis darbo greitis iš dalies kompensuoja operatyviosios atminties santykinai didelę kainą. Šiame darbe yra kuriamas DBVS prototipas, kuris duomenis darbo metu laiko tik operatyviojoje atmintyje. Lyginant pastoviosios disko ir operatyviosios atminčių darbų greičių santykį galima tikėtis, kad šio kuriamo prototipo duomenų apdorojimo greitis gali būti iki 100 karto didesnis. DBVS prototipas bus pritaikytas tik duomenų nuskaitymui. Duomenų modifikavimo funkcijos jis neturės. Taip maksimaliai galima koncentruotis į duomenų suspaudimo, paieškos ir kitus manipuliacijos duomenimis algoritmus maksimaliai išnaudojant atminčių greičių santykį.

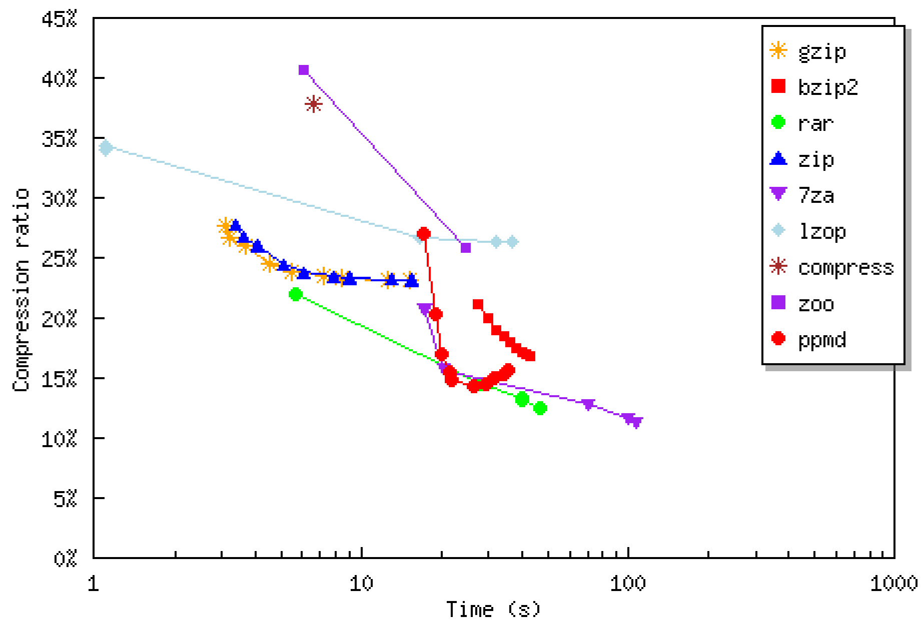
DBVS prototipui yra pritaikomi, optimizuojami arba sukuriami nauji duomenų suspaudimo, apdorojimo, paieškos ir pan. algoritmai. DBVS kuriamas prototipas duomenis darbo metu laiko tik operatyvioje atmintyje. Kadangi operatyviosios atminties santykinai yra mažai, tai labai didelės įtakos turi duomenų suspaudimo algoritmas, nuo kurio priklauso ne vien tik užimamos atminties kiekis, bet ir duomenų nuskaitymo greitis.

Realiai tokių DBVS sistemų yra palyginti mažai. Jos yra skirtos duomenų analizei, kai duomenų kiekiai yra labai dideli. *Clearpace* kompanija yra sukūrusi *NPBase* sistemą [1], kuri turi kuriamo DBVS prototipo funkcionalumą. Šiame darbe atliekama analizė įvairių disko DBVS sistemų, kartu įtraukiant *NPBase* sistemą, palyginimui su kuriamu DBVS prototipu.

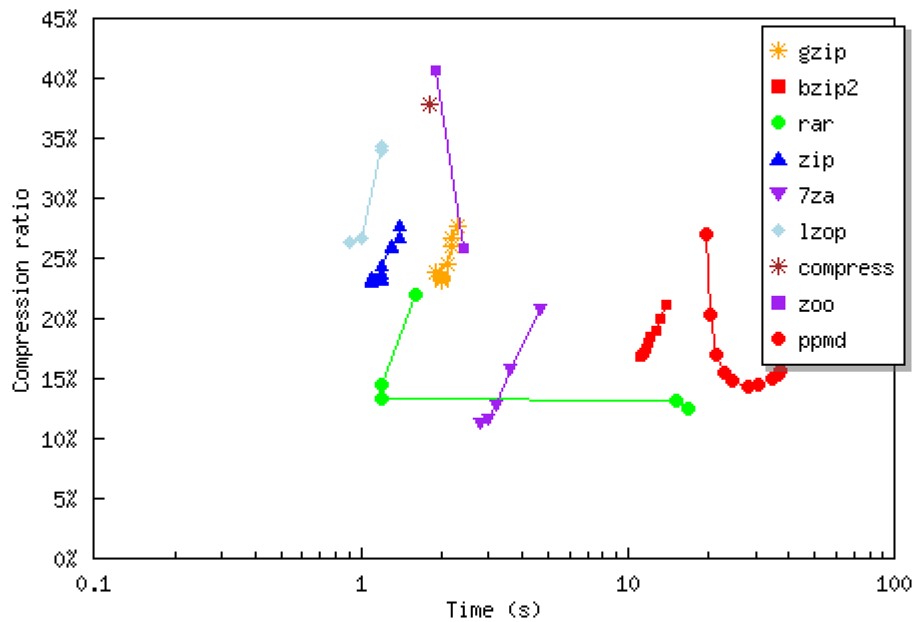
2. Duomenų suspaudimo, apdorojimo algoritmų analizė ir optimizavimas

DBVS prototipo projektavimo tikslai yra šie: labai greita duomenų paieška, didelis duomenų suspaudimas, įvairūs manipuliavimo duomenimis algoritmai. Pagrindiniai faktoriai, lemiantys projektuojamą DBVS prototipą yra siejami su duomenų laikymu operatyvioje atmintyje, atminties fragmentacija, virtualios atminties įtaka DBVS darbui, algoritmų efektyvumu. Pastarasis faktorius labai glaudžiai yra susijęs su duomenų struktūromis. Taip galima atsiriboti nuo „lėtos“ disko atminties, kurios įtaka gali labai iškraipyti įvairių algoritmų veikimo spartą. Tačiau virtualios atminties įtaka turi būti įvertinta, nes kad ir kiek daug bus operatyviosios atminties kompiuteryje, bendru atveju jos visada gali pritrūkti.

Duomenų suspaudimo algoritmas turi pasižymėti labai dideliu duomenų išpakavimo greičiu. Įvairių suspaudimo programų RAR, ZIP ir pan. naudojami algoritmai yra daugiau orientuoti į suspaudimo laipsnį, nei į išpakavimo greitį. 1 pav. pavaizduotas GIMP (angl. *GNU Image Manipulation Program*) išeities kodo suspaudimas įvairiais suspaudimo laipsniais. Šios programos kodas nesuspaustas užima 72MB. 2 pav. pavaizduota išpakavimo priklausomybė nuo laiko. Iš šio piešinėlio galima daryti išvadą, kad pritaikyti RAR, ZIP programų naudojamus algoritmus netikslinga dėl santykinai lėto išpakavimo greičio. Greičiausiai išpakuota 72MB buvo per maždaug 1 sekundę.



1 pav. Duomenų suspaudimo priklausomybė nuo laiko [2]



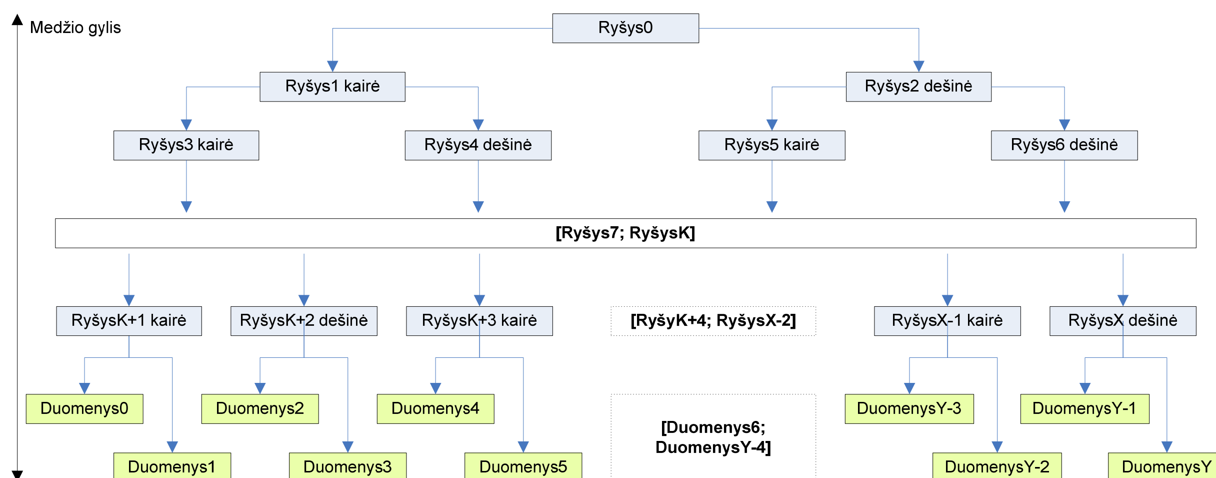
2 pav. Duomenų išpakavimo priklausomybė nuo laiko [2]

DBVS per 1 sekundę gali būti įvykdoma šimtai užklausų, kai duomenų bazės dydis viršija 1GB, todėl tokie algoritmai nėra tinkami. Kadangi DBVS išskirtinai dirbs tik operatyviojoje atmintyje, todėl yra svarbus ir duomenų suspaudimo laipsnis. Taigi reikia laviruoti tarp pakankamo

duomenų suspaudimo laipsnio ir duomenų išpakavimo greičio. Naudojamas suspaudimo algoritmo principas yra paprastas: išsaugojamos tik unikalios duomenų reikšmės, kurios nėra kaip nors specialiai suspaudžiamos (tai sutaupytų nemažai atminties jeigu būtų suspaudžiami eilutės tipo duomenys, nes jie dažniausiai užima apie 80% visos duomenų bazės užimamos vietos, tačiau išpakavimo laikas labai pailgėtų). Išsaugant tik unikalias duomenų reikšmes reikia jas tarpusavyje susieti. Taip pat reikia atsižvelgti į tai, kad būtų galima labai efektyviai vykdyti duomenų paiešką ir įvairias operacijas susijusias su duomenų manipuliavimu. Taigi vienos duomenų bazės lentelės duomenų priklausomybės bus saugojamos medžio struktūroje.

2.1. Medžio struktūra

Medis yra sudaromas iš dviejų tipų elementų: ryšių ir duomenų. Kiekvienas ryšio ar duomenų elementas atitinkamai turi savo numerį nuo 0 iki x ir nuo 0 iki y. Kad būtų aiškesni algoritmai susiję su medžio manipuliacija, visi ryšių elementai, išskyrus viršutinįjį, pagal savo poziciją grafinėje medžio imitacijoje yra arba kairieji, arba dešinieji. Toliau į kiekvieną medžio elementą bus kreipiamasi jo pavadinimu ir numeriu (pvz.: *Ryšys0*). 3 pav. pateikta bendra medžio struktūra, kai medis kiekviename lygyje turi po kairįjį ir dešinįjį ryšius.



3 pav. Bendra medžio struktūra

Šis medis yra simetrinis centro atžvilgiu. Bendru atveju medis yra asimetrinis, t.y. kairėje pusėje esančių ryšių skaičiui nėra lygus dešinėje pusėje esančiam ryšių skaičiui. Konstruojant medį

vienas iš svarbiausių parametrų yra medžio gylis. Jis rodo kiek yra elementų lygių medyje kartu su duomenų elementais

Reliacinėje duomenų bazėje (toliau duomenų bazė) vienas medis atitinka vieną duomenų lentelę (toliau lentelė). Kiekvienas duomenų elementas atitinka kiekvieną lentelės duomenų stulpelį (toliau stulpelis). Ryšio elementai neturi sąsajų su duomenų bazės objektais.

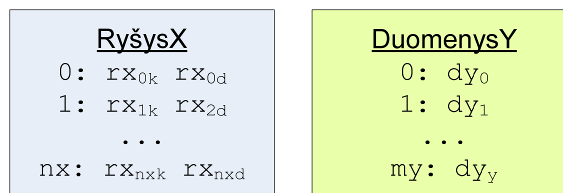
2.2. Medžio elementų struktūra

Kiekvienas medžio elementas susideda iš įrašų. Įrašų kiekis elemente yra skirtingas, bet bendru atveju gali ir sutapti.

Ryšio elementų įrašą sudaro pora skaičių: ryšiai į gilesnio lygio kairįjį ir dešinįjį elementų (ryšio arba duomenų) įrašus. Tie ryšiai yra nurodomi skaičiumi nuo 0 iki n. Kiekvienam medžio ryšio elementui n yra skirtingas (pvz.: penktas ryšio elementas turi n5 įrašus).

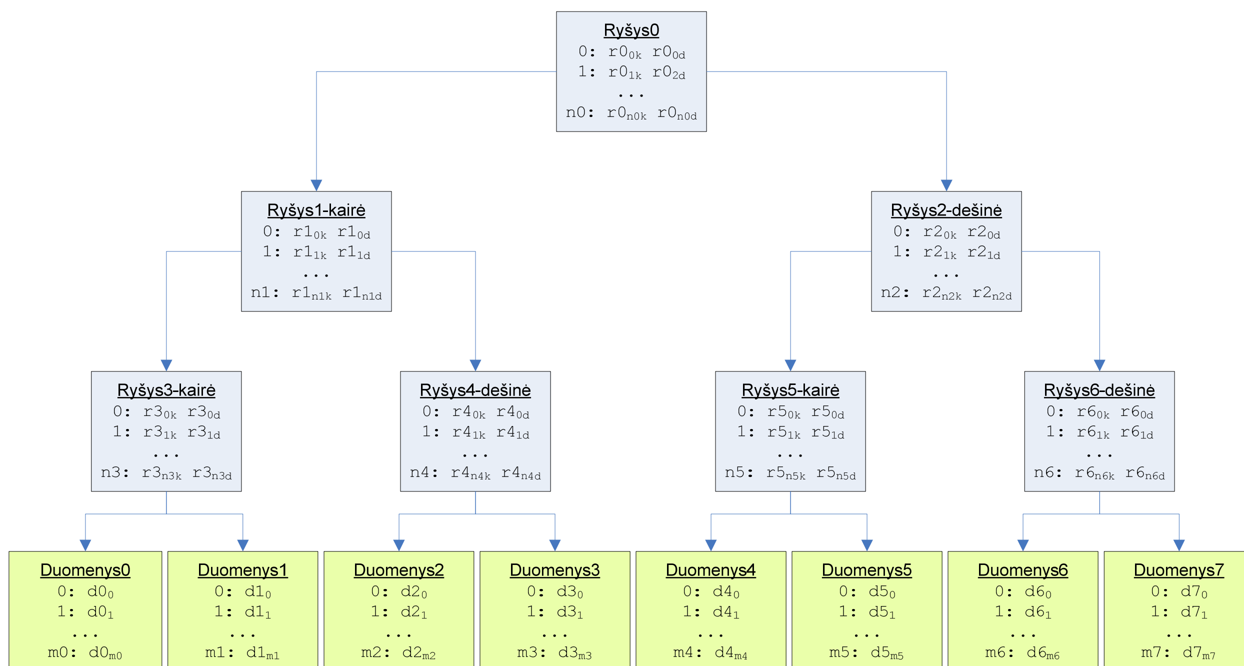
Duomenų elemento įrašus sudaro stulpelio duomenų **unikalios** reikšmės išrūšiuotos unikalumo didėjimo tvarka. Kadangi šiuose elementuose saugojamos tik unikalios reikšmės todėl pasiekiamas duomenų bazės suspaudimas.

4 pav. pavaizduotas ryšio ir duomenų elementai. Juose yra atitinkamai nx ir my įrašų. Ryšio elemente nulinio įrašo ryšys į gilesnio lygio kairįjį ir dešinįjį elementus įrašus yra atitinkamai rx_{0k} (nulinis, kairė) ir rx_{0d} (nulinis, dešinė). Duomenų elemente nulinis įrašas yra dy_0, t.y. unikali stulpelio reikšmė.



4 pav. Ryšio ir duomenų elementų struktūra

5 pav. pateikta konkreti medžio struktūra. Šis medis saugo lentelę, kuri turi 8 stulpelius (numeruojami nuo 0 iki 7 imtinai). Medis yra simetriškas.



5 pav. Medžio struktūros pavyzdys

2.3. Medžio konstravimas

Aprašant medžio konstravimo algoritmą, pirmiausiai reikia aptarti pagrindinius medžio konstravimo parametrus. Medžio konstravimo metu medis yra apdorojamas dalimis, t.y. iš vientisos medžio struktūros išskiriami tam tikri poaibiai. Poaibiai yra ne kas kita kaip stulpelių poaibis. Poaibiai yra vaizduojami pomedžiais (angl. *sub-tree*).

Prieš medžio konstravimą turi būti atliktas kiekvieno stulpelio skirtingų reikšmių (f, angl. *Frequency*) skaičiavimas. Šis procesas yra vadinamas dažnių lentelės konstravimu. Po to skaičiuojamas kiekvieno stulpelio reikšmingumas (m, angl. *Magnitude*). Jis skaičiuojamas pagal (1) formulę.

$$m_i = \left\{ \begin{matrix} 0, f_i \le 1 \\ \lfloor \log_{10}(f_i) \rfloor, f_i > 1 \end{matrix} \right\}; \quad (1)$$

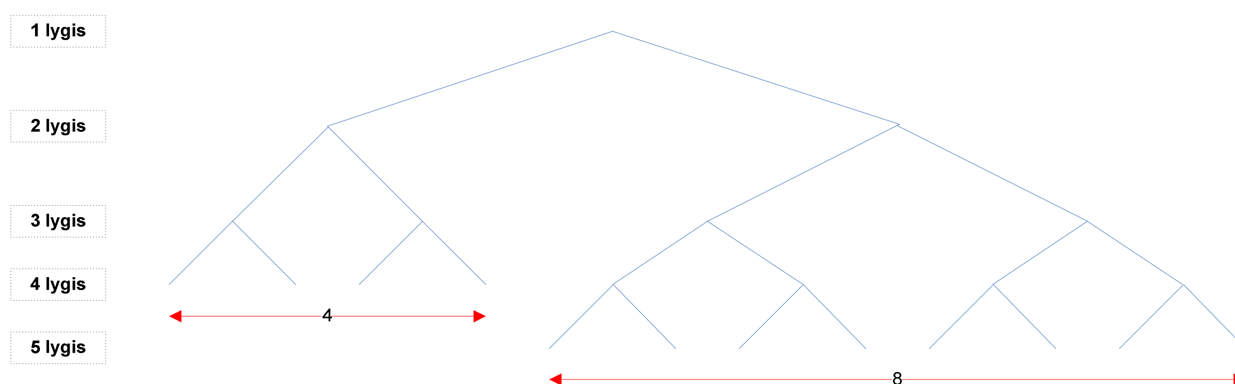
čia f_i – i - ojo stulpelio skirtingų reikšmių kiekis;

m_i – i - ojo stulpelio reikšmingumas.

Po to stulpeliai yra sugrupuojami pagal apskaičiuotus reikšmingumus, t.y. stulpeliai, turintys tą patį reikšmingumą, priklauso tai pačiai grupei. Medžio konstravimo metu stulpeliai iš tos pačios

reikšmingumo grupės bus tame pačiame medžio lygyje. Kuo didesnis stulpelių reikšmingumas, tuo labiau reikia kelti juos į medžio viršų. Taip galima sutaupyti kelis ryšio elementus. Norint tai automatizuoti reikia išrikiuoti stulpelius pagal jų reikšmingumo mažėjimą ir pradėti medžio konstravimą nuo stulpelių, turinčių didžiausią reikšmingumą.

Konstravimo metu medis konstruojamas iš atskirų pomedžių, kurie atstovauja tam tikrą stulpelių poaibį. Pomedžiams yra skaičiuojami parametrai, susiję su stulpelių padėtimi pomedyje, t.y. medžio gylis (d, angl. *tree depth*), kairės ir dešinės pomedžio pusės pločius (wl, wr, angl. *width left/right*), nuo kurių priklauso kiek daug reikės ryšio elementų, norint apjungti visus stulpelius iš tos pačios reikšmingumo grupės, medžio lygį, kuriame pomedis prasideda ir požymį, kuris rodo, ar tai giliausia medžio vieta. 6 pav. pateiktas pomedis, kurio penktame lygyje yra 8 stulpeliai iš vienos reikšmingumo grupės ir ketvirtame lygyje 4 stulpeliai iš kitos (didesnės, turinčios daugiau unikalių reikšmių) reikšmingumo grupės.



6 pav. Medžio poaibiai, turintys po 4 ir 8 duomenų elementus

Jeigu yra pomedis i, kurio stulpelių aibė yra s_i, tai medžio gylį galima apskaičiuoti pagal (2) formulę.

$$d_i = \begin{cases} 0, s_i = 1 \\ \lfloor \log_2(s_i - 1) + 1 \rfloor, s_i > 1 \end{cases}; \tag{2}$$

čia s_i – pomedžio aibės dydis (stulpelių skaičius).

Dešinės ir kairės pomedžio pusės pločiai apskaičiuojami pagal (3) ir (4) formules atitinkamai.

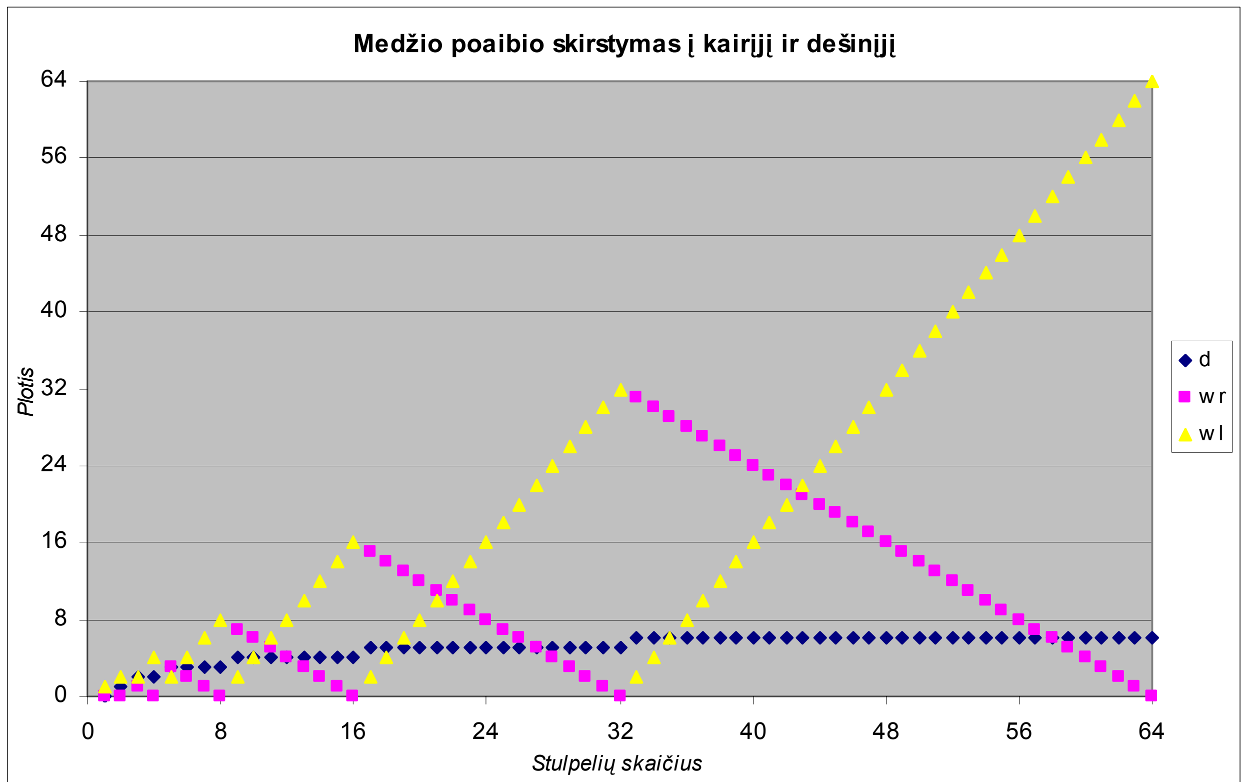
$$wr_i = 2^{d_i} - s_i \quad (3)$$

$$wl_i = s_i - wr_i \quad (4)$$

čia s_i – pomedžio aibės dydis (stulpelių skaičius);

d_i – i - ojo pomedžio gylis.

(3) ir (4) formulės išrinktos taip, kad kairėje medžio pusėje visada būtų lyginis stulpelių skaičius išskyrus atvejį, kai yra tik vienas stulpelis. Jeigu stulpelių skaičius yra 2^{si}, tai visi stulpeliai turi pereiti į kairiąją pusę. Dešinėje pusėje yra stulpeliai atlikę nuo kairės pusės. 7 pav. pavaizduota kairės ir dešinės medžio pusės pločių priklausomybė nuo stulpelių skaičiaus.



7 pav. Medžio poaibių skirstymas

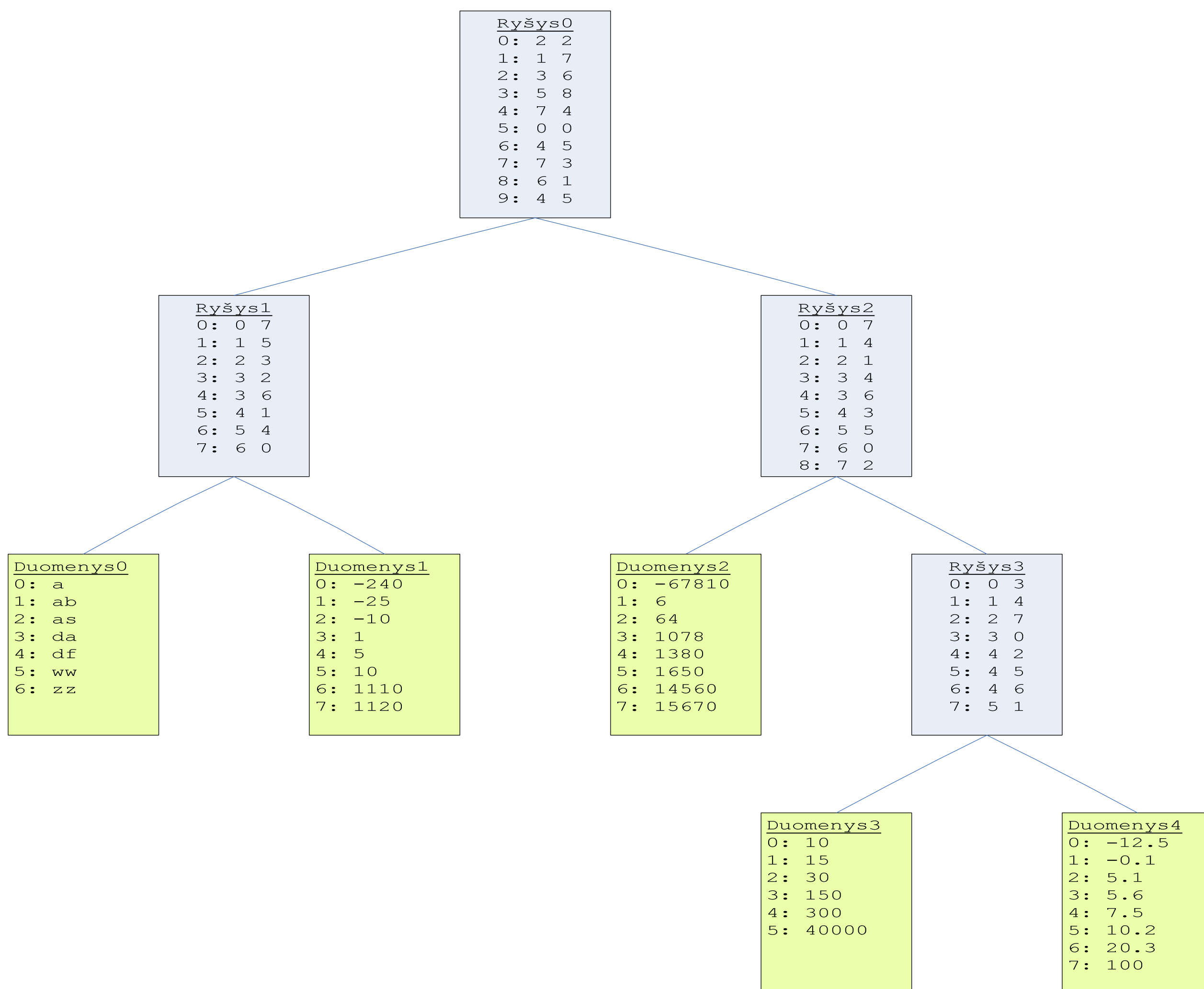
Suskaičiavus pomedžio (medžio) kairės ir dešinės pusės pločius, atliekamas duomenų elementų (stulpelių) apjungimas, naudojant ryšio elementus. Pirmiausiai apdorojami tie medžio poaibiai (pomedžiais), kurie turi mažiausiai unikalių reikšmių. Sujungus pomedžio vienos pusės duomenų elementus ryšio elementais, toliau sujungiami ryšio elementai su kitais ryšio elementais tol, kol galiausiai pasiekiama pomedžio viršūnė, kuri yra vienas ryšio elementas.

Vėliau apdorojama dešinė pomedžio pusė. Sukonstravus vieną pomedį su tam tikru unikalių reikšmių kiekiu vėliau tęsiamas viso medžio kūrimas analogišku principu. Sukonstruoti pomedžiai yra naudojami pagal jų aukščiausią ryšio elementą. 1 lentelėje pateikta lentelė, turinti 5 stulpelius.

1 lentelė – Pavyzdinė duomenų lentelė „Table1“

A (int)	B (double)	C (char *)	D (short)	E (int)
15	7.5	as	1	64
10	5.6	ab	10	14560
300	10.2	da	-10	1650
30	100	df	-25	15670
300	20.3	zz	-240	1078
40000	-0.1	a	1120	-67810
150	-12.5	da	1110	1380
300	5.1	zz	-240	1078
300	5.1	ww	5	6
150	-12.5	da	1110	1380

8 pav. pavaizduotas medžio pavyzdys, atlikus 1 lentelės medžio konstravimą.



8 pav. Medžio pavyzdys su duomenimis

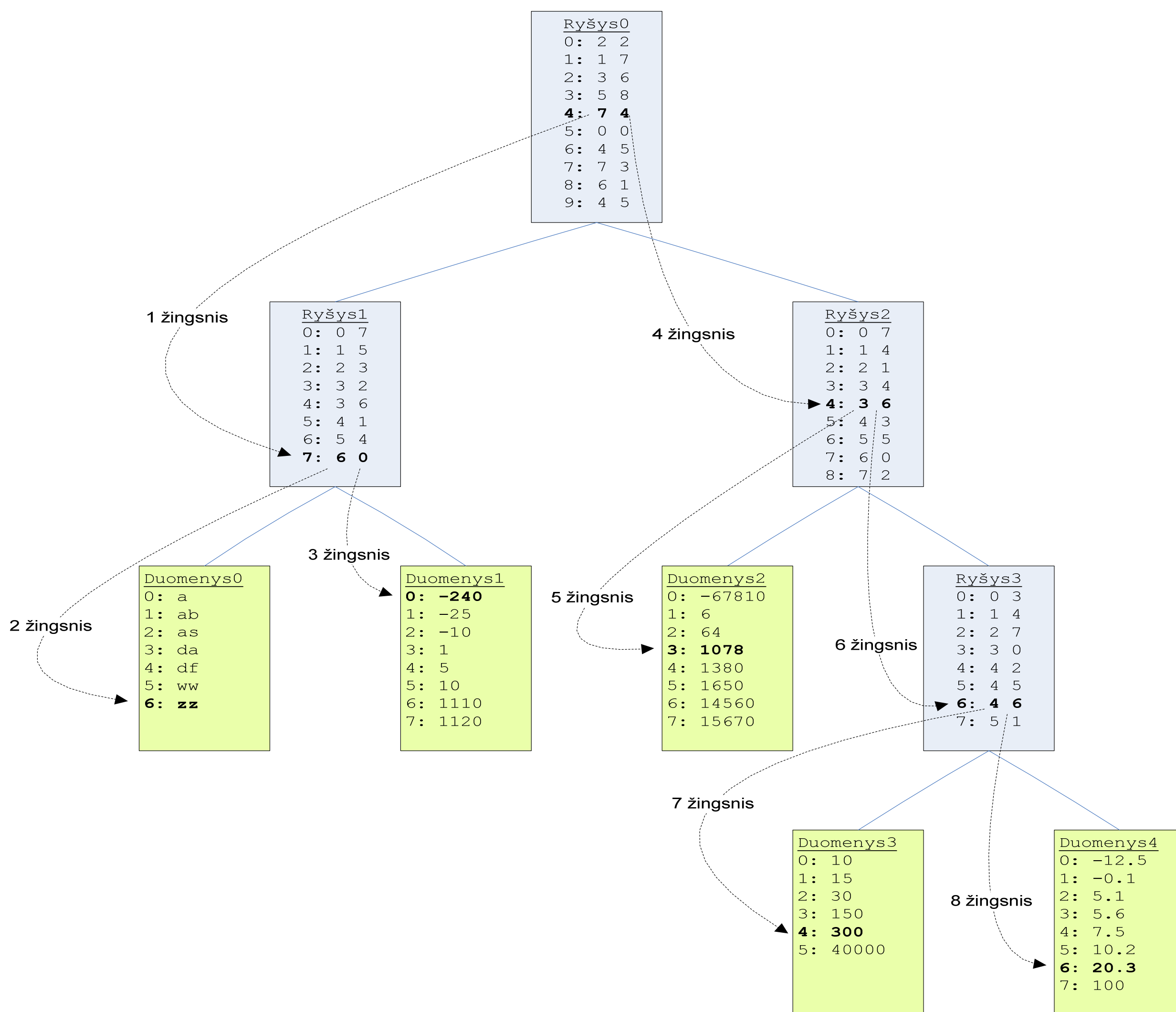
Sukūrus lentelės medį gauname supakuotą lentelės modelį. Kitas žingsnis, naudojant duomenų saugojimą medžio struktūroje, yra duomenų atstatymas.

2.4. Duomenų atstatymas

Duomenis atstatyti galima iš visiškai sukonstruoto medžio. Pirmasis medžio ryšio elementas turi tiek įrašų, kiek yra įrašų lentelėje. Pvz.: jeigu reikia iš medžio gauti penktą įrašą lentelėje, tai reikia imti penktą įrašą aukščiausiame medžio ryšio elemente ir nuosekliai surinkinėti duomenis. Ryšio elementais einama gilyn į kairę arba dešinę tol, kol pasiekiamas duomenų elementas. Pasiekę duomenų elementą, turime vieną stulpelio reikšmę iš atstatomo įrašo. Toks perėjimas nuo aukščiausio ryšio elemento iki duomenų elementų yra pakankamai greitas, nes tai yra kreipinys į operatyvios atminties vietą su tam tikru poslinkiu. Šiuo atveju ryšio

elemento kairysis ar dešinysis indeksas yra indeksas į gilesnio ryšio arba duomenų elemento indeksą. Atstačius vieną įrašo reikšmę, atstatomi likusieji. Kad nereikėtų „vaikščioti“ per medį jau būtose vietose, naudojamas rekursinis duomenų surinkimo algoritmas. Jo idėja aprašoma žemiau.

Medis apeinamas iš kairės į dešinę. Reikia „eiti“ gilyn iki pirmojo duomenų elemento. Po to „pakilti“ vienu lygiu į viršų iki ryšio elemento. Ryšio elemente nuskaitomas dešinysis indeksas ir taip pagal jį einama vėl žemyn iki kito duomenų elemento. Taip kartojami žingsniai tol kol apeinami visi duomenų elementai. Taip turime visiškai atstatytą lentelės įrašą. 9 pav. pavaizduoti penktojo įrašo rekursinio atstatymo algoritmo veiksmai. Punktyrinė linija su virš ja nurodyto žingsnio numeriu rodo algoritmo žingsnius. Atstatytas įrašas identiškas 1 lentelės penktam įrašui. Duomenų elementų ir lentelės stulpelių ryšys yra taip pat išsaugojamas. 9 pav. duomenų elementas pavadinimu „Duomenys0“ atitinka 1 lentelės stulpelį „C“.



9 pav. Duomenų surinkimo pavyzdys

Aprašytame algoritme atstatomas pilnas lentelės įrašas. Tačiau daugeliu atvejų, ypač tais atvejais kaip yra vykdomos įvairios užklausos, kai reikia įrašų tik tam tikrų stulpelių reikšmių šis algoritmas atlieka per daug nereikalingų žingsnių. Bendru atveju jeigu reikia atstatyti ne visus įrašo elementus, tai nėra reikalo „eiti“ tam tikrais medžio ryšio elementais, kad „būtų prieiti“ tie duomenų elementai, kurių mums nereikia.

Trumpiausio maršruto algoritmo idėja: surašomi visų reikalingų duomenų elementų maršrutai iš duomenų elemento iki aukščiausio medžio ryšio elemento į vieną seką, po to gauta seka apverčiama (sukeičiami visi elementai vietomis) ir optimizuojama. Sekos optimizacija paremta jau aplankytų medžio ryšio elementų atmetimu (atmetami ryšio elementai išskyrus pirmus du, kurių yra daugiau negu du sekoje), išsaugant einamojo ryšio elemento indeksą, jei jis dar gali būti naudojamas (nesaugojamas tuo atveju, kai į ryšio elementą kreipiamasi tik vieną kartą).

Algoritmo žingsniai pavaizduoti 10 pav. Šiame pavyzdyje reikia atkurti ne pilną įrašą – 3 iš 5 stulpelių (paryškintus duomenų elementus). Patogumo dėlei ryšio elementai pervardinami iš „RyšysX“ į „R_x“, o duomenų elementus – iš „DuomenysY“ į „D_y“. Reikia surašyti seką iš duomenų elementų į viršutinį ryšio elementą:

$$D_0\ R_1\ R_0 \quad D_1\ R_1\ R_0 \quad D_2\ R_2\ R_0.$$

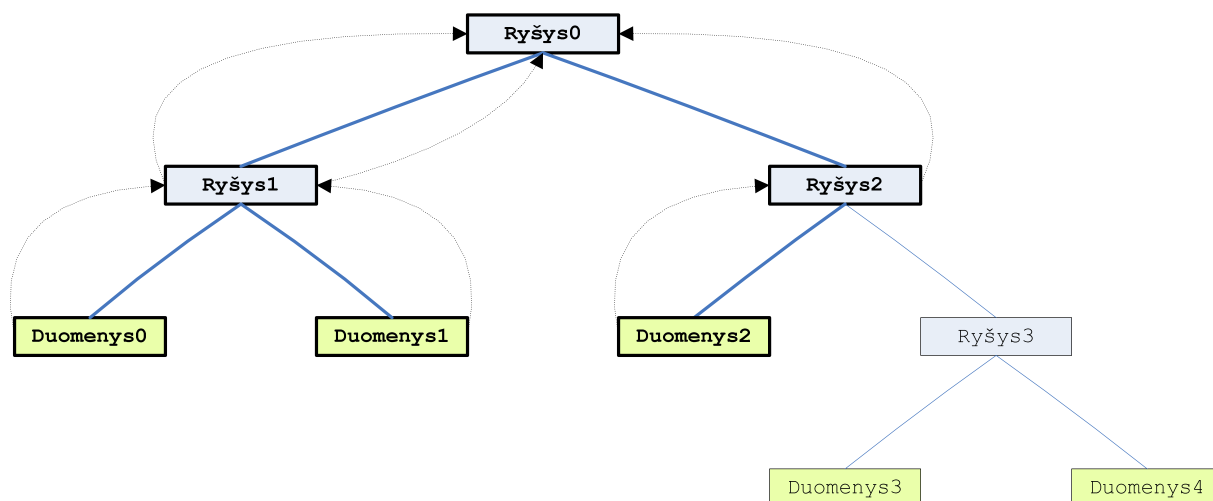
Po sekos apvertimo gaunama seka:

$$R_0\ R_2\ D_2 \quad R_0\ R_1\ D_1 \quad R_0\ R_1\ D_0.$$

Atliekama sekos optimizacija. Įvedami žymėjimus: „+“ reiškia, kad išsaugojamas ryšio elemento indeksas, „-“ – nuskaitomas išsaugotas ryšio elemento indeksas, perbrauktas ryšio elementas – pašalintas iš sekos. Po optimizacijos gauta seka:

$$+R_0\ R_2\ D_2 \quad -R_0\ +R_1\ D_1 \quad \cancel{R_0}\ -R_1\ D_0.$$

Žodinis sekos perėjimas: norint atstatyti įrašą reikia išsaugoti R_0 ryšio elemento indeksą, eiti į R_2 elemento indeksą (pastaba: nereikia išsaugoti indekso, nes jis nebus naudojamas), eiti į D_2 duomenų elementą, nuskaityti R_0 ryšio elemento indeksą, kuris buvo išsaugotas, išsaugoti R_1 ryšio elemento indeksą, eiti į D_1 duomenų elementą, nuskaityti R_1 ryšio elemento indeksą, kuris buvo išsaugotas, eiti į D_0 duomenų elementą. Iš gautos optimizuotos sekos eiti per nereikalingus ryšio elementus (R_5) nereikėjo.



10 pav. Trumpiausio kelio sekos radimo pavyzdys

Atliekant užklausą, tokia seka sukonstruojama vieną kartą. Ji yra naudojama tiek kartų, kiek reikia nuskaityti įrašų iš sukonstruoto lentelės medžio.

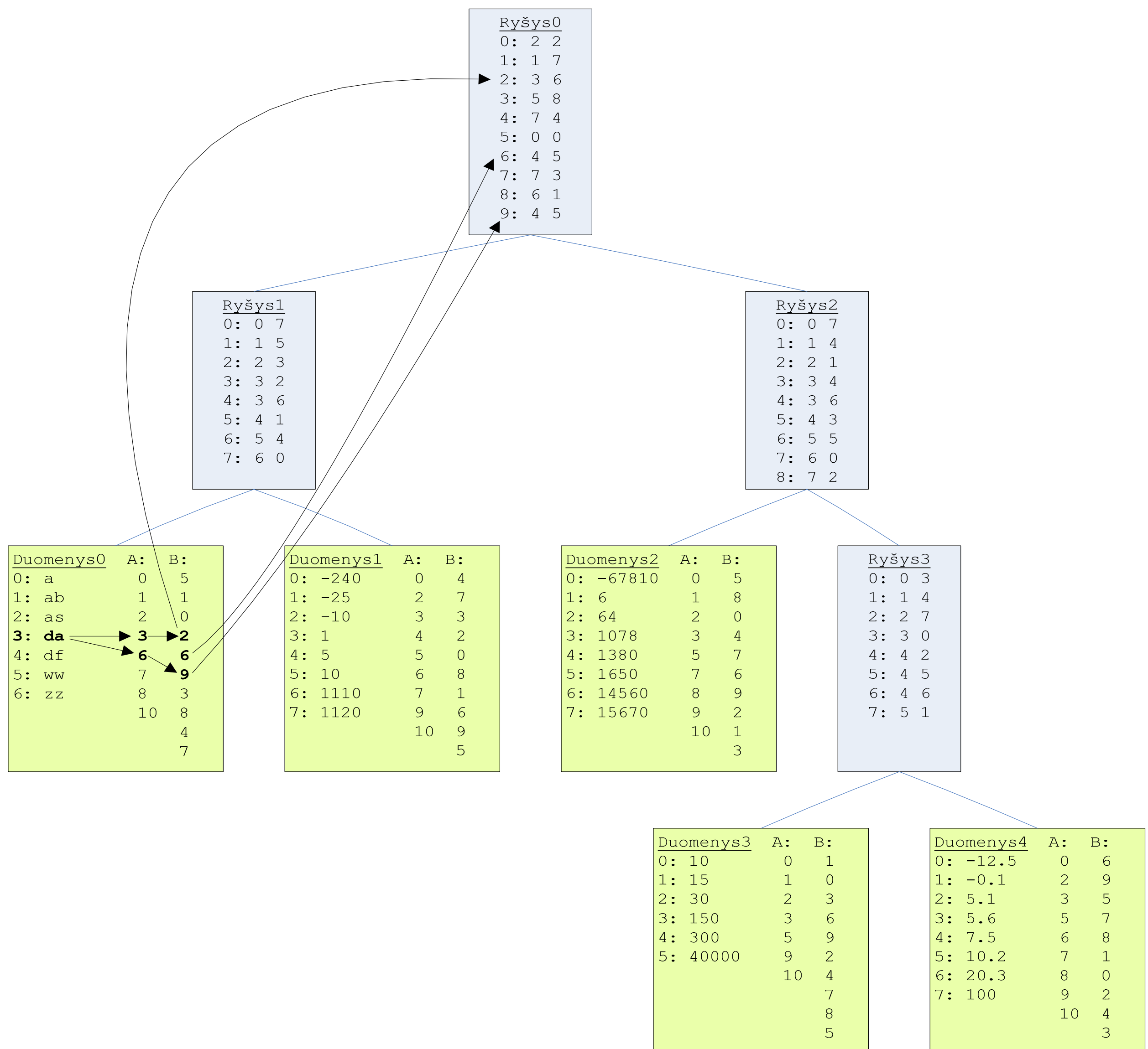
2.5. Paieška medyje

Projektuojant duomenų bazių valdymo sistemos prototipą, vienas iš svarbiausių uždavinių yra greita duomenų paieška. Duomenų lentelės medyje saugojami duomenų elementai, kuriuose yra tik unikalios juos atitinkančių stulpelių reikšmės. Be to, tos reikšmės yra išrikiuotos didėjimo (mažėjimo) tvarka. Jeigu reikia surasti visus lentelės įrašus santykyje su tam tikra reikšme, panaudojant rūšiuotas unikalias reikšmes, galima lengvai nustatyti, kurios reikšmės tenkina sąlygą, tačiau reikia turėti duomenis apie unikalių reikšmių buvimą lentelės įrašuose. Tam yra naudojami kiekvienam duomenų elementui du unikalių reikšmių indeksavimo masyvai: A ir B.

Masyvas B turi tiek elementų, kiek yra lentelėje įrašų. Masyve B saugojami lentelės įrašų numeriai. Pagal šiuos įrašų numerius susiejamos unikalios reikšmės su lentelės įrašais.

Masyvas A bendru atveju turi mažiau elementų už masyvą B. Masyve A saugojama pozicija masyve B, nuo kurios reikia pradėti skaityti įrašų numerius tam tikrai unikaliai reikšmei. Kitais žodžiais tariant, unikalios reikšmės pozicija masyve B yra gaunama iš masyvo A pagal tą patį indeksą duomenų elemente, pagal kurį ieškomas unikalios reikšmės buvimas lentelės įrašuose.

11 pav. pavaizduotas įrašų numerių radimas pagal unikalią reikšmę iš 1 lentelės. Paryškintas tekstas rodo, kuri informacija mus domina. Rodyklės žymi, nuo kurios vietos reikia imti reikšmes. Iš duomenų elemento „Duomenys0“ ieškoma, kuriuose įrašuose yra unikali reikmė „da“. Šios reikšmės indeksas duomenų elemente yra 3. Pagal šią reikšmę nuskaitomi du gretimi A masyvo elementai: pirmas rodo masyvo B indeksų intervalo pradžią (imtinai), antras – pabaigą (neimtinai). Taigi iš masyvo A nuskaitomi du elementai: 3, 6. Jie yra masyvo B intervalas: [3, 6). Iš masyvo B nuskaitomos visos reikšmė, kurių indeksai gauti iš masyvo A: 2, 6, 9. Šios reikšmės ir yra lentelės įrašų numeriai. Kadangi aukščiausias ryšio elementas „Ryšys0“ yra tiesiogiai susijęs su lentelės įrašais, todėl grafiškai rodyklės rodo nuo duomenų elemento iki aukščiausio ryšio elemento.



11 pav. Unikalių reikšmių ir įrašų susiejimas

Taigi pasinaudojant sukurta medžio struktūra, galima sukonstruoti lentelės įrašus, o naudojant A ir B masyvus, atlikti lentelės įrašų paiešką.

2.6. STL konteineriai

STL (angl. *Standard Template Library*) – C++ programavimo kalbos biblioteka, skirta įvairioms duomenų struktūroms saugoti, keisti ir atlikti įvairias funkcijas [3, 13-14p]. Kadangi C++ kalboje yra sukurtas šablonų (angl. *template*) mechanizmas, tai vienas iš pagrindinių STL bibliotekos privalumų yra tas, kad visos klasės ir funkcijos yra šabloninės. Tai įgalina sukurti vieną bendrą algoritmą bendrai duomenų struktūrai, kad po to būtų galima naudoti bet kokiai kitai išvestinei duomenų struktūrai. STL bibliotekoje duomenys saugomi taip vadinamuose konteineriuose.

Konteineris yra šabloninė klasė [3, 129p], kurios viduje yra saugojami tam tikri elementai. Konteineriai yra kelių tipų, kurie bus aptarti vėliau. Visiems konteineriams galioja tam tikros savybės, jie gali atlikti tam tikras bendras funkcijas.

Konteinerių pagrindinės savybės [3, 129p]:

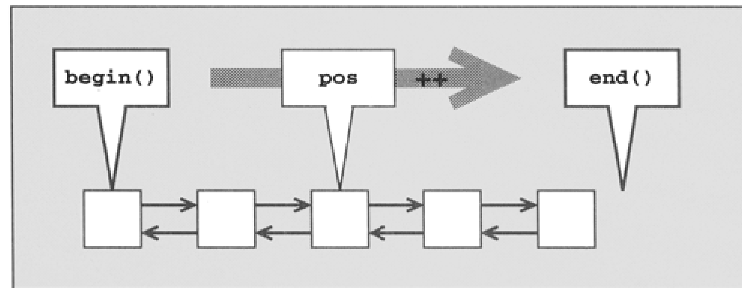
- Visi konteineriai turi mechanizmą, įgalinantį grąžinti reikšmę, o ne nuorodą.
- Konteineriai kopijuoja elementus vietoj to, kad kopijuotų tik nuorodas (angl. *reference*).
- Bendru atveju saugant duomenis konteineriuose, jie yra išsaugomi tam tikra tvarka. Dėl to kiekvieną kartą skaitant duomenis, jie nuskaitomi viena ir ta pačia tvarka.
- Kiekvienas konteineris turi pakartotinio nuskaitymo (angl. *iterator*) mechanizmą. Ši savybė yra pagrindinė.
- Operacijos su konteineriais nėra saugios nes, kad vartotojas pats turi pasirūpinti, kad neskaitytų duomenų su blogu indeksu ar pan. Priešingu atveju gaunama klaida.

Konteinerių pagrindinės funkcijos [3, 130p]:

- Kiekvienas konteineris turi turėti konstruktorių pagal nutylėjimą (angl. *default constructor*), kopijavimo konstruktorių (angl. *copy constructor*) ir griovėją (angl. *destructor*).
- Trys konteinerių dydžių operacijos: *size()* (elementų kiekis), *empty()* (ar tuščias konteineris), *max_size()* (didžiausias elementų saugojimo kiekis).

- Palyginimo operacijos: < (mažiau), > (daugiau), <= (mažiau arba lygu), >= (daugiau arba lygu), == (lygu), != (nelygu). Konteineriai yra lygūs, jei juose esantys elementai ir jų tvarka konteineriuose sutampa.
- Paieškos funkcijos *upper_bound()*, *lower_bound()* grąžina iteratorių su paieškos rezultatais.

Naudojant iteratorių iš bet kokio konteinerio, galima nuskaityti visas reikšmes. Iteratorius turi *begin()* (grąžina konteinerio pirmą elementą), *end()* (grąžina požymį, ar jau bandoma nuskaityti už konteinerio ribų) funkcijas, operatorių „++“ (vienetu padidinama nuskaitoma konteinerio pozicija), operatorių „--“ (vienetu pamažinama nuskaitoma konteinerio pozicija) funkcijas, operatorių „*“ (nuskaitoma reikšmė pagal einamąją iteratoriaus poziciją). 12 pav. pateiktas sąrašo iteratoriaus veikimo pavyzdys.



12 pav. Sąrašo iteratoriaus veikimo pavyzdys [3, 81p]

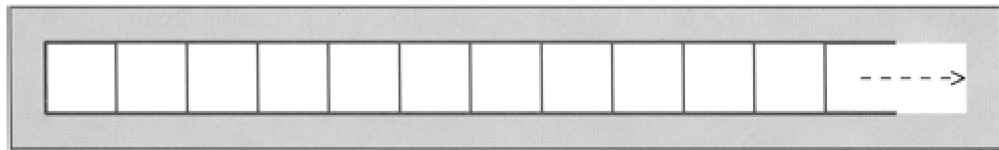
Taip pat yra naudojami atbuliniai iteratoriai (angl. *reverse iterator*), kurių atliekamos operacijos yra vienodos su jau aptartais iteratoriais, tačiau reikšmės nuskaitomos iš kitos pusės. *begin()* ir *end()* funkcijos yra priešingos lyginant su iteratoriaus atitinkamomis funkcijomis.

STL bibliotekoje yra naudojami įvairūs efektyvūs algoritmai. Vienas iš svarbiausių algoritmų kuriant duomenų bazių valdymo sistemą yra dvejetainė paieška ir greito rūšiavimo algoritmai. Visos STL bibliotekos funkcijos yra šabloninės, todėl šiuos algoritmus galima lengvai pritaikyti visiems bet kokios duomenų struktūros konteineriams. Prieš naudojant dvejetainę paiešką, konteinerio elementai turi būti išrikiuoti, jei konteineris nedaro to pats elementų įterpimo metu (pvz. aibė).

STL bibliotekoje konteineriai yra kelių pagrindinių tipų: vektorius (angl. *vector*), dekas (angl. *deque - Double Ended QUEue*), sąrašas (angl. *list*), aibė (angl. *set*) ir žemėlapis (angl. *map*).

2.6.1. Vektorius

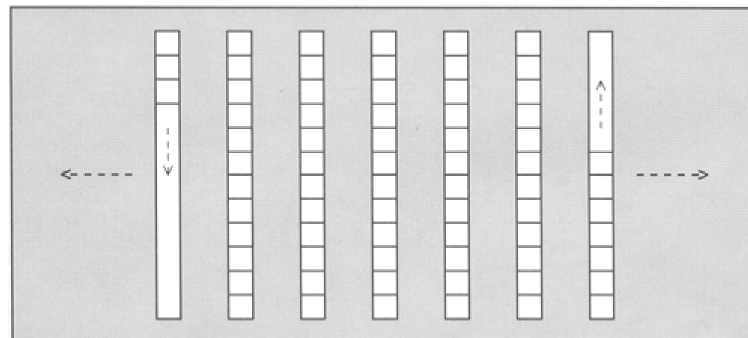
Vektorius - tai dinamiškai besiplečiantis masyvas į vieną pusę. Duomenys rašymo metu yra kopijuojami į tam tikrą vidinį dinaminį masyvą. Duomenų ėmimas pagal indeksą iš vektoriaus yra atsitiktinio paėmimo (angl. *random access*). Tai reiškia kad, reikšmės paėmimas iš vektorius pagal indeksą bus visoms reikšmėms vienodas. Vektoriaus duomenų nuskaitymo ir įrašymo laikinės charakteristikos yra pačios geriausios lyginant su kitų tipų konteineriais. Taip pat labai greitos yra trynimo vektoriaus gale ir įterpimo vektoriaus gale operacijos. Jeigu atliekamos pastarosios operacijos vektoriaus viduje, tai jų atlikimo laikai labai pablogėja, nes reikia perstumti dalį vektoriaus elementų. 13 pav. pavaizduota vektoriaus struktūra.



13 pav. Vektoriaus struktūra [3, 132p]

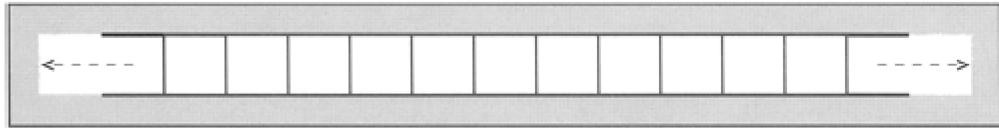
2.6.2. Dekas

Dekas - labai panaši struktūra į vektorių, tačiau galinti plėstis į abu galus. Rašymo ir skaitymo operacijos yra taip pat labai greitos. Kaip ir vektorius, dekas turi atsitiktinio priėjimo nuskaitymo galimybę. Įterpimo ir trynimo operacijos yra greitos deko galuose. Jei atliekamos šios operacijos deko viduje, tai reikia perrašyti dalį deko elementų. Deko vidinė duomenų struktūra yra masyvų masyvas, kuri pavaizduota 14 pav.



14 pav. Vidinė deko struktūra [3, 143p]

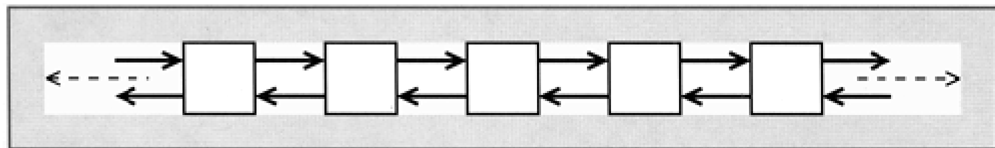
Duomenų nuskaitymas yra lėtesnis nei vektoriaus. 15 pav. pateikta deko struktūra.



15 pav. Deko struktūra [3, 143p]

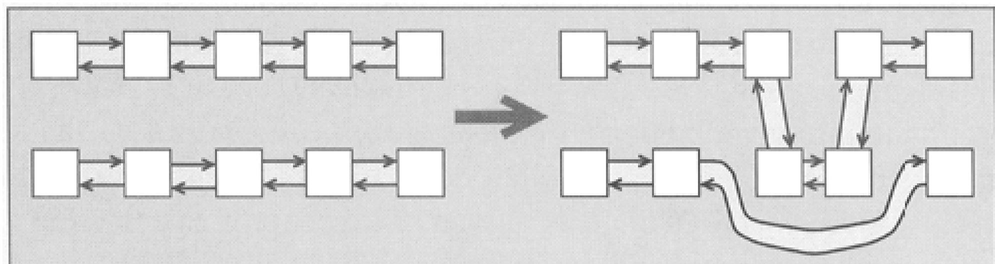
2.6.3. Sąrašas

Sąrašas - tai duomenų struktūra, turinti kiekviename elemente dvi nuorodas: į kairę ir į dešinę. Vidinė sąrašo struktūra yra visiškai skirtinga lyginant su vektoriaus ar deko struktūromis. Sąrašas neturi atsitiktinio priėjimo nuskaitymo galimybės. Tai reiškia, kad norint nuskaityti konkrečią reikšmę iš tam tikros pozicijos, iki jos reikia „eiti“ nuo pirmojo elemento. Įterpimo ir šalinimo operacijos yra labai greitos lyginant su vektoriumi ar deku. Įterpti ar ištrinti galima į bet kurią poziciją per tą patį laiką. 16 pav. pateikta sąrašo vidinė struktūra.



16 pav. Sąrašo vidinė struktūra [3, 148p]

17 pav. pavaizduotos elementų įterpimo ir šalinimo operacijos sąraše.

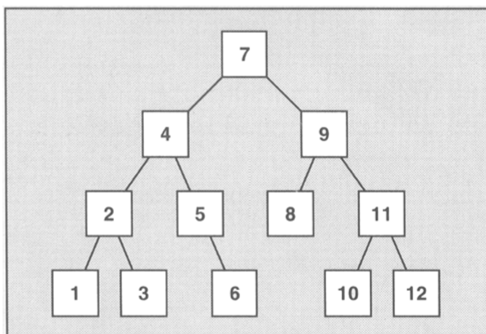


17 pav. Elementų įterpimo ir pašalinimo operacijos sąraše [3, 153p]

2.6.4. Aibė

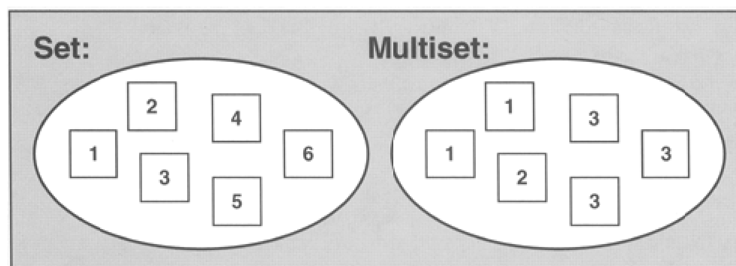
Aibė - tai tokia duomenų struktūra, kurioje nereikia išsaugoti elementų įrašymo pozicijų, reikia atlikti tik paieškos operacijas. Yra du tipai aibių: aibės tik su unikaliomis

reikšmėmis, aibės su pasikartojančiomis reikšmėmis (angl. *multiset*). Įterpiant elementus automatiškai, jie yra išrikiuojami ir yra konstruojamas dvejetainis medis, kuris parodytas 18 pav. Taigi naudojant dvejetainę paiešką, atliekama labai greita elemento paieška aibėje. Jis konstruojamas tam, kad bet kuriuo metu butų galima greitai surasti, ar tam tikras elementas yra aibėje ar ne.



18 pav. Aibės dvejetainis konstravimo medis [3, 157p]

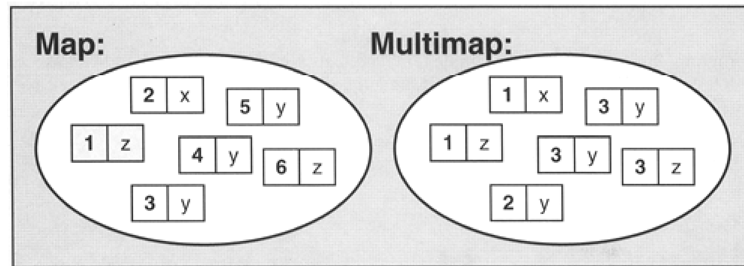
19 pav. pavaizduotas aibės, turinčios tik unikalias reikšmes, ir aibės, turinčios pasikartojančias reikšmes, pavyzdys.



19 pav. Aibių pavyzdys [3, 156p]

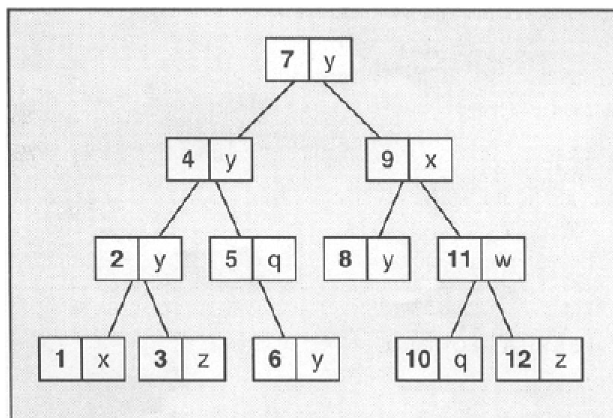
2.6.5. *Žemėlapis*

Žemėlapis - tai tokia duomenų struktūra, kuri saugo rakto ir reikšmės poras. Raktas - tai duomenys, pagal kurį yra ieškoma reikšmė. Yra dviejų tipų žemėlapiai. Vieni saugo tik pagal unikalius raktus, kiti gali saugoti ir pagal pasikartojančius raktus (angl. *multimap*). Įrašant raktų ir reikšmių poras į žemėlapį, jos automatiškai yra rūšiuojamos. Žemėlapių pavyzdys yra pateiktas 20 pav.



20 pav. Žemėlapio (su unikaliais ir pasikartojančiais raktais) pavyzdys [3, 172p]

Žemėlapiuose duomenys saugomi išbalansuotuose dvejetainiuose medžiuose. Kai reikia surasti reikšmę (reikšmes) pagal raktą (raktus), yra vykdoma dvejetainė paieška, kuri yra labai efektyvi, kai duomenys yra išrūšiuoti. Duomenų įterpimas arba pašalinimas yra pakankamai lėtos operacijos lyginant su paieška pagal raktą, nes reikia modifikuoti vidinę dvejetainio medžio struktūrą, kuri pateikta 21 pav.



21 pav. Žemėlapio su unikaliais raktais dvejetainis medis [3, 173p]

Modifikuojant reikšmę, pirmiausiai rakto ir reikšmės pora išmetama iš medžio ir modifikavus vėl įtraukiama į medį.

2.6.6. Bendras konteinerių palyginimas

2 lentelėje pateiktas konteinerių tipų palyginimas. Jie lyginami pagal vidinę duomenų struktūrą, konteinerių elementų tipus, pasikartojančias reikšmes ir reikšmių manipuliavimo funkcijų efektyvumą.

2 lentelė – Bendras konteinerių palyginimas

	Vektorius	**Dekas**	**Sąrašas**	**Aibė**	**Aibė**[1]	**Žemėlapis**	**Žemėlapis**[2]
Vidinė duomenų struktūra	Dinaminis masyvas	Masyvų masyvas	Dvikryptis sąrašas	Dvejetainis medis	Dvejetainis medis	Dvejetainis medis	Dvejetainis medis
Elementas	Reikšmė	Reikšmė	Reikšmė	Reikšmė	Reikšmė	Raktas/ Reikšmė	Raktas/ Reikšmė
Pasikartojantys elementai leidžiami	Taip	Taip	Taip	Ne	Taip	Ne (raktui)	Taip
Paieška	Lėta	Lėta	Labai lėta	Greita	Greita	Greita raktui	Greita raktui
Įterpimas, pašalinimas greitas	Gale	Pradžioje ir gale	Bet kur	Niekur	Niekur	Niekur	Niekur

3. Duomenų manipuliacijos algoritmai

3.1. Kursoriai

Bendrąja prasme reliacinių duomenų bazių kontekste, kursorius yra užklausos rezultatai (įrašai). Šiame darbe kursorius turi kiek kitokią prasmę. Kursorius, tai toks objektas, kuris gali saugoti savyje bet kokį kiekį stulpelių ir įrašų. Stulpelių tipai gali skirtis. Kursorius, kuriant duomenų bazių valdymo sistemos prototipą, yra vienas iš pagrindinių sistemos elementų, tiesiogiai darantis įtaką sistemos darbo spartai. Jie bus naudojami duomenų surinkimui iš lentelės medžio, rūšiavimui bei grupavimui pagal tam tikrus kriterijus. Kadangi kursoriai tiesiogiai

---

[1] Aibė, turinti pasikartojančių elementų.

[2] Žemėlapis, turintis pasikartojančių raktų.

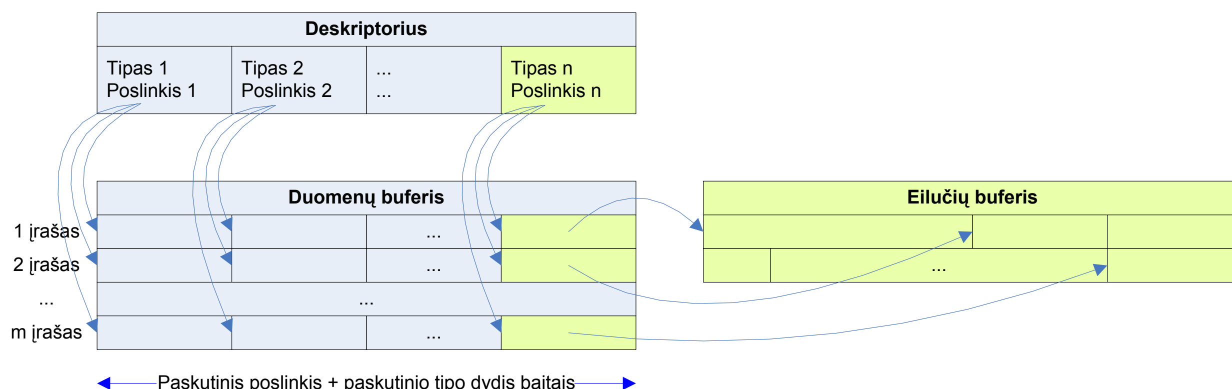
įtakoja sistemos darbo spartą, jie turi būti realizuoti pakankamai žemame lygyje. Kursorius turi savybę saugoti duomenis tiek operatyvioje, tiek pastovioje atmintyse. Tai yra būtina, jeigu reikia taupyti operatyviąją atmintį. Tai pat negalima visų duomenų laikyti pastovioje atmintyje, nes labai nukenčia duomenų nuskaitymo/įrašymo greitis. Naudojami kelių tipų kursoriai: bazinis kursorius, rūšiavimo kursorius, grupavimo kursorius, apjungimo kursorius. Bazinis kursorius yra visų kitų kursorių bazinė klasė, kuri tai pat gali būti naudojama kaip atskiras kursorius.

3.1.1. Vidinė kursoriaus struktūra

Visų kursorių tipų duomenų laikymo vidinė struktūra yra labai panaši. Kursoriuose naudojamus duomenų tipus reikia išskirti į dvi grupes: fiksuoto ilgio ir kintamo ilgio. Fiksuoto ilgio duomenų tipai - tai labai gerai žinomi įvairūs skaičių ir adresų tipai (*char* (1 baitas), *short* (2 baitai), *int* (4 baitai), *float* (6 baitai), *double* (8 baitai), *__int64* (8 baitai), *void* (4 baitai) ir pan.). Kintamo ilgio duomenų tipai yra tekstinės eilutės ir dvejetainiai masyvai (*char **). Šiems tipams kiekvieno duomenų lemento saugojimui turi būti išsaugoma ne tik pati reikšmė, bet ir duomenų elemento kiekis baitais. Taigi kursoriaus duomenys yra dvejopai apdorojami nuo įvedamų duomenų tipų. Kiekvienas kursorius atmintyje turi tam tikro dydžio išskirtą vientisą buferį, į kurį yra rašomi duomenys. Jis vadinamas duomenų buferiu. Jeigu kursorius turi stulpelių, kurių tipai yra kintamo ilgio, tai papildomai dar yra naudojamas kitas buferis saugoti kintamo ilgio duomenų reikšmėms. Jis vadinamas eilučių buferiu. Kursoriaus konstravimo metu yra nurodami abiejų buferių maksimalūs dydžiai, kad iš anksto būtų paruošta vieta duomenims saugoti.

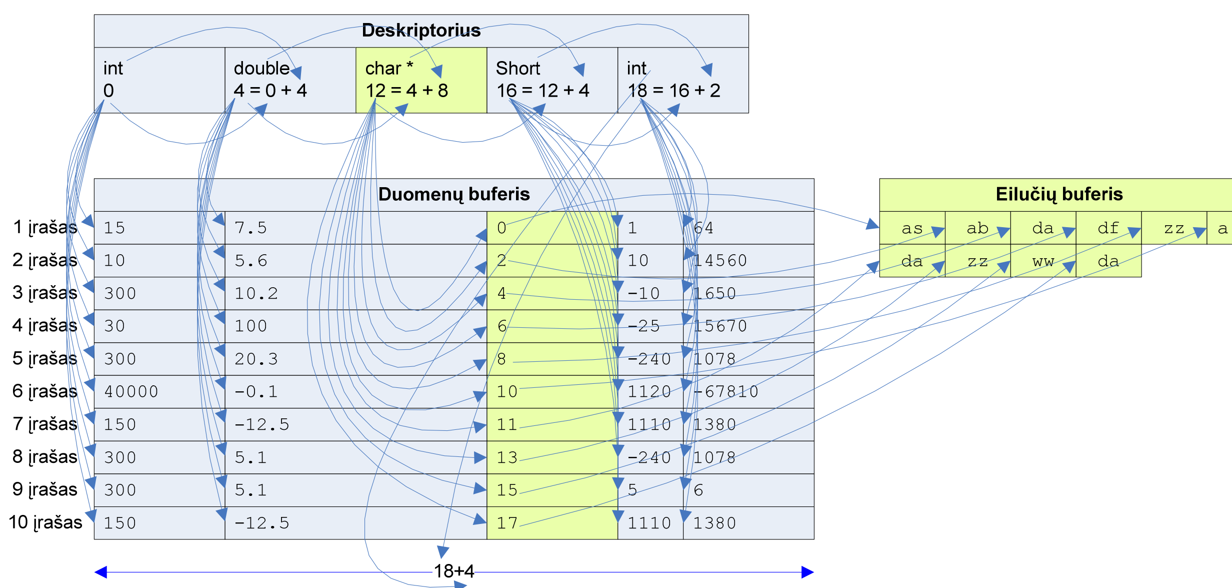
Kursoriaus saugomų duomenų tipų aprašymui yra naudojami deskriptoriai. Deskriptoriai, tai STL bibliotekos vektoriai, saugantys duomenų tipą, poslinkį nuo einamojo įrašo kursoriuje pradžios baitais ir dar papildomą tarnybinę informaciją. Deskriptoriaus pradžioje gali būti saugoma papildoma informacija, susijusi su įrašo pozicija kursoriuje (pvz. tikrasis įrašo numeris). Į kursoriaus duomenų buferį yra rašomi duomenys, kurių ilgis yra fiksuotas. Jeigu kursorius turi kintamo ilgio duomenis, tai į eilučių buferį yra rašomos viena paskui kitą eilutės arba dvejetainiai masyvai, o į duomenų buferį įrašomas fiksuoto ilgio (4 baitų) poslinkis eilučių buferyje (ne nuo einamojo įrašo pradžios, bet nuo eilučių buferio pradžios). Naudojant du buferius išvengiama duomenų atstatymo problemos, kai duomenis reikia atstatyti pagal tam tikrus įrašų numerius (ne nuosekliai). 22 pav. pateiktas kursoriaus deskriptorius ir kursoriaus duomenų ir eilučių buferiai. Rodyklės vaizduoja poslinkius baitais: duomenų buferyje nuo einamojo įrašo

pradžios, eilučių buferyje nuo buferio pradžios. Melsva spalva simbolizuoja duomenų buferį, o žalsva - eilučių buferį.



22 pav. Pagrindiniai kursoriaus struktūros elementai

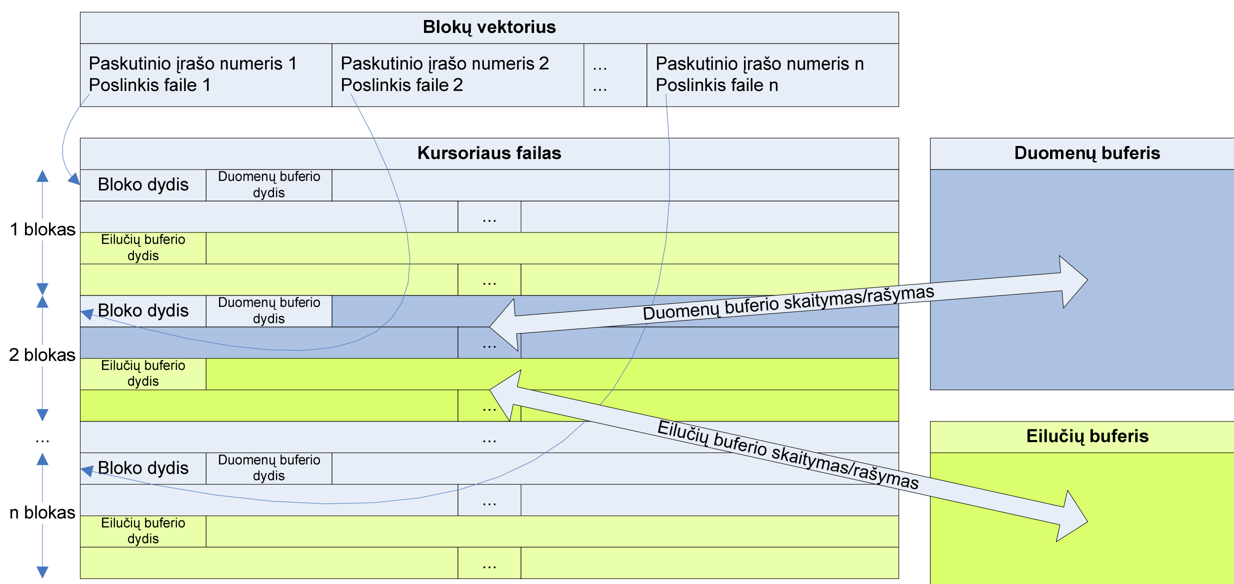
23 pav. pateikiamas 1 lentelės duomenų saugojimas kursoriuje.



23 pav. Kursoriaus buferių pavyzdys

Kiekvieno įrašo įrašymo metu į kursorių yra tikrinama, ar duomenys tilps į kursoriaus buferius. Tam tikslui kursoriaus sukūrimo metu yra apibrėžiamas didžiausias eilutės ilgis. Jeigu duomenys netilps į kursoriaus buferius, tai buferiai yra užrašomi į failą, o buferiai atmintyje yra atlaisvinami. Duomenų dalys, kurios telpa buferiuose, o vėliau užrašomos į failus yra vadinamos blokais. Norint iš kursoriaus nuskaityti bet kurį įrašą pirmiausiai reikia išsiaiškinti, ar jis yra

operatyviojoje atmintyje, ar faile. Šiam tikslui yra naudojamas blokų vektorius, kurio kiekvienas elementas saugo bloko paskutinio įrašo numerį ir poslinkį faile nuo jo pradžios baitais iki einamojo bloko pradžios. Kadangi įrašant duomenis į kursorių įrašų numeriai nuosekliai didėja, blokų vektorius automatiškai tampa išrūšiuotas pagal paskutinio įrašo numerį. Vėliau galima atlikti dvejetainę paiešką, ieškant kuriame bloke yra įrašas. Suradus bloką, jis yra nuskaitomas iš failo į operatyviąją atmintį ir atmintyje, panaudojant kursoriaus deskriptorių, nuskaitomi kiekvieno stulpelio duomenys. Faile bloką sudaro dvi dalys: duomenų buferis ir eilučių buferis. Bloko pradžioje įrašomas abiejų buferių dydis pridedant suminio buferių ir atskirų buferių dydžius baitas (po 4 baitus), po to saugomas duomenų buferio dydis baitais (4 baitai), pats duomenų buferis, eilučių buferio dydis baitais (4 baitai) ir pats eilučių buferis. Kursoriaus duomenų saugojimas operatyviojoje atmintyje ir faile pateiktas 24 pav.



24 pav. Kursoriaus duomenų saugojimas operatyviojoje atmintyje ir faile

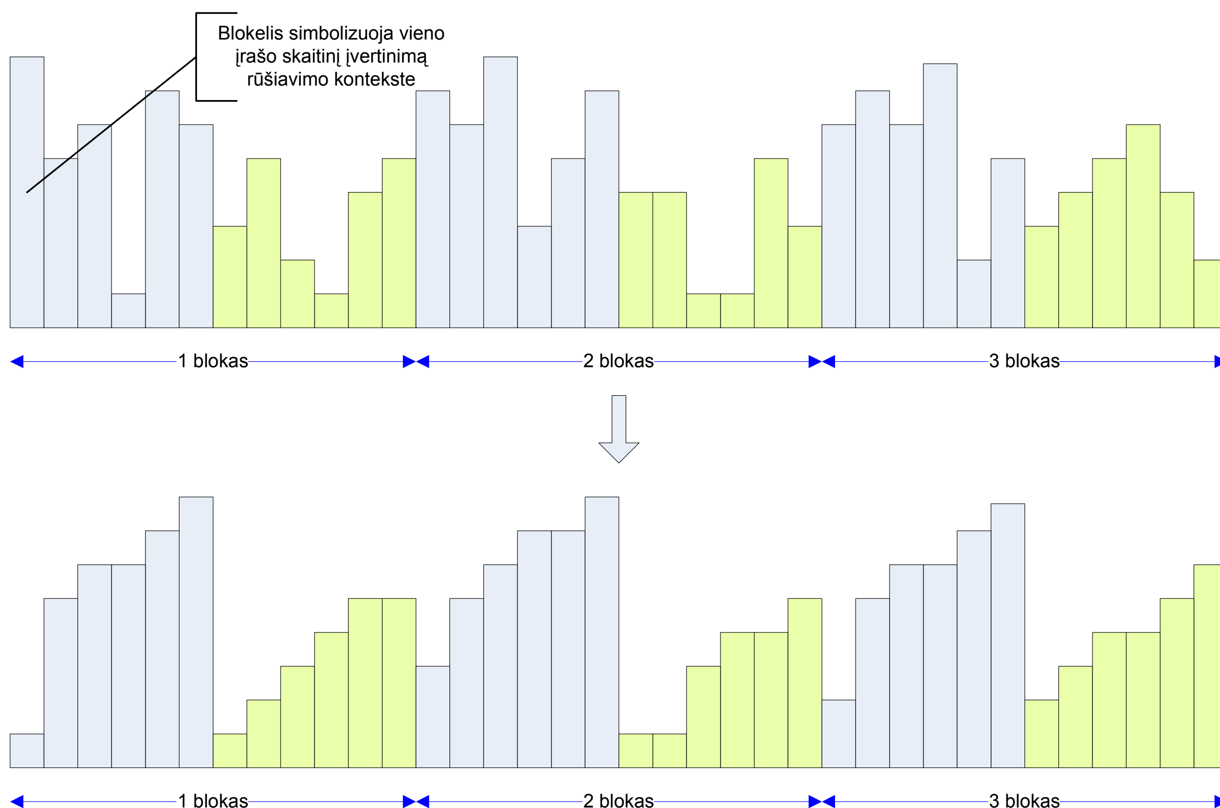
Aprašytas funkcionalumas galioja visų rūšių kursoriams. Jeigu duomenų nereikia rūšiuoti, grupuoti ar apjungti, tai šio funkcionalumo visiškai pakanka, kad būtų galima surinkti duomenis iš lentelės medžio. Toks funkcionalumas yra realizuotas baziniame kursoriuje, kurį paveldi rūšiavimo, grupavimo ir apjungimo kursoriai. Kursorių funkcionalumo pavyzdžiams bus naudojami SQL (angl. *Structured Query Language*) kalbos sakiniai [5].

3.1.2. Rūšiavimo kursorius

Rūšiavimo kursorius be visų bazinio kursoriaus savybių turi galimybę išrūšiuoti duomenis duomenų rašymo į kursorių metu. Rūšiavimo kriterijus yra nurodomas kursoriaus sukūrimo metu, kartu su stulpelių tipų nustatymu. Rūšiuoti galima didėjančiai arba mažėjančiai (rūšiavimo tipas) pagal stulpelius nurodant rūšiavimo prioritetą (pagal kurį stulpelį reikia išrūšiuoti pirmiausiai). Rūšiavimo tipą ir prioritetą galima saugoti viename skaičiuje. Skaičiaus ženklas rodo rūšiavimo tipą (<0 – rūšiuojama mažėjančiai, =0 – nerūšiuojama, >0 – rūšiuojama didėjančiai), o skaičiaus modulis – rūšiavimo prioritetą (kuo didesnis modulis, tuo mažesnis rūšiavimo prioritetas).

Tarkime reikia išrūšiuoti 1 lentelę pagal B stulpelį mažėjančiai, o pagal A didėjančiai. Simbolinis rūšiavimo kriterijus atrodys taip: B (-1) (pirmiausiai mažėjančiai), A (+2) (jei yra pasikartojančių B stulpelio reikšmių, tai didėjančiai pagal A).

Nustačius kursoriaus rūšiavimo kriterijus, duomenų rašymo į kursorių metu rūšiavimas nėra atliekamas. Rūšiavimas atliekamas operatyviojoje atmintyje, kai kursoriaus blokas užrašomas į failą arba kai pabaigiamas kursoriaus duomenų rašymas (buferiai yra tokio dydžio, kad nereikia naudoti failo buferių saugojimui). Bloko rašymo į failą metu yra išrūšiuojamas ne vien duomenų buferis, bet ir eilučių buferis, atnaujinant duomenų buferio poslinkius į eilučių buferį. Tai atliekama dėl to, nes kursoriaus duomenų įrašymo pabaigoje reikės apjungti visus išrūšiuotus blokus į vieną (nuskaitant duomenis iš failo, iš dviejų blokų nėra galimybės nuskaityti abu pilnus blokus, nes yra ribojamas kursoriaus buferių dydis). Blokų apjungimas yra būtinas, nes skaitymo metu visi kursoriuje esantys įrašai turi būti išrūšiuoti. 25 pav. pateikiamas tarpinis blokų išrūšiavimo metodas.



25 pav. Blokų įrašų išsidėstymas prieš ir po rūšiavimo

Atlikus rūšiavimą stulpelių kiekis kursoriuje gali sumažėti, jeigu kai kurie stulpeliai prieš rūšiavimą yra reikalingi tik duomenų išrūšiavimui, bet jų pačių saugoti nereikia. Kursoriaus blokų rūšiavimas atliekamas greito rūšiavimo (angl. *quick sort*) algoritmu. Šis algoritmas, kaip ir daugelis kitų standartinių algoritmų yra realizuoti tam tikrose C++ bibliotekose. *qsort()* funkcija išrūšiuoja bet kokio dydžio duomenis, kai jiems yra aprašyta palyginimo funkcija. Ši funkcija pagal greito rūšiavimo algoritmą tiesiog kviečia su skirtingais duomenų įrašais palyginimo funkciją ir gauna -1 (pirmas įrašas mažesnis už antrąjį), 0 (įrašai lygūs, +1 (pirmas įrašas didesnis už antrąjį).

Greito rūšiavimo algoritmo vidutinis atliekamų veiksmų skaičius yra apskaičiuojamas pagal (5) formulę [4].

$$k = n \cdot \log_2 n \quad (5)$$

čia k – vidutinis atliekamų veiksmų skaičius;

n – įrašų kiekis bloke (n = 1, 2, ..., N).

Atlikus blokų greitą rūšiavimą reikia blokus apjungti, kad duomenys būti visiškai išrūšiuoti. Blokų apjungimas atliekamas suliejimo algoritmu, kuris yra nesudėtingas. Apjungiant du blokus reikia imti įrašą iš pirmo ir jį lyginti su įrašu iš antro, tas kuris tenkina rūšiavimo kriterijų rašomas pirmasis ir taip kartojami veiksmus tol kol „pereinami“ abiejų blokų įrašai. Suliejimo algoritmą galima vykdyti dviem būdais kai turim tam tikrą blokų kiekį. Sulieti gretimus blokus poromis arba sulieti visus blokus iš karto.

Pirmasis būdas reikalauja pakartotinio blokų suliejimo. Pvz.: jeigu yra 8 blokai, tai po pirmo gretimų blokų suliejimo gaunami 4, suliejus poromis keturis blokus gaunami 2 ir suliejus du blokus gaunamas pilnai išrūšiuotas kursorius. Tai buvo pasiekta per tris suliejimo etapus.

Antram suliejimo tipui reikia tik vieno suliejimo etapo, tačiau reikia daug palyginimo operacijų. Kiekvieną įrašą reikia palyginti su kiekvieno bloko pirmu įrašu ir tik tada nuspręsti, kurį įrašą užrašyti. Dėl mažesnio palyginimų kiekio bus naudojamas pirmasis suliejimo būdas.

Kyla klausimas, ar rūšiuojant blokus atskirai ir po to suliejant po du yra efektyvesnis algoritmas negu išrūšiuoti vieną didelį bloką turint labai didelį duomenų ir eilučių buferį. Ištikrųjų abu algoritmai yra labai artimi savo vidutiniu atliekamų veiksmų skaičiumi. Suliejant blokus poromis reikia atlikti $\log_2 m$ etapų, kai m blokų kiekis. Iš (6) formulių gauname, kad nesvarbu kuriame esame suliejimo etape, vidutinis veiksmų kiekis yra tas pats. Tai galima paaiškinti tuo, kad po vieno etapo suliejimo yra dvigubai mažiau blokų, bet nauji blokai yra dvigubai didesni.

$$
\begin{aligned}
l_1 &= \frac{m}{2} \cdot (n + n) = m \cdot n; \\
l_2 &= \frac{m}{4} \cdot (2n + 2n) = m \cdot n \\
&\dots \\
l_i &= \frac{m}{2^i} \cdot (2^{i-1} n + 2^{i-1} n) = m \cdot n
\end{aligned}
\qquad (6)
$$

čia l_i – i – ojo suliejimo etapo vidutinis veiksmų skaičius;

i – suliejimo etapo numeris (i = 1, 2, ..., N);

m – blokų kiekis (m = 2, 4, …, 2^N);

n – bloko dydis (įrašų kiekis bloke).

Vidutinį veiksmų skaičių per visus suliejimo etapus galima apskaičiuoti pagal (7) formulę, kai blokai suliejami poromis.

$$k = m \cdot n \cdot \log_2 m \qquad (7)$$

čia k – bendras suliejimo etapų atliekamų veiksmų skaičius.

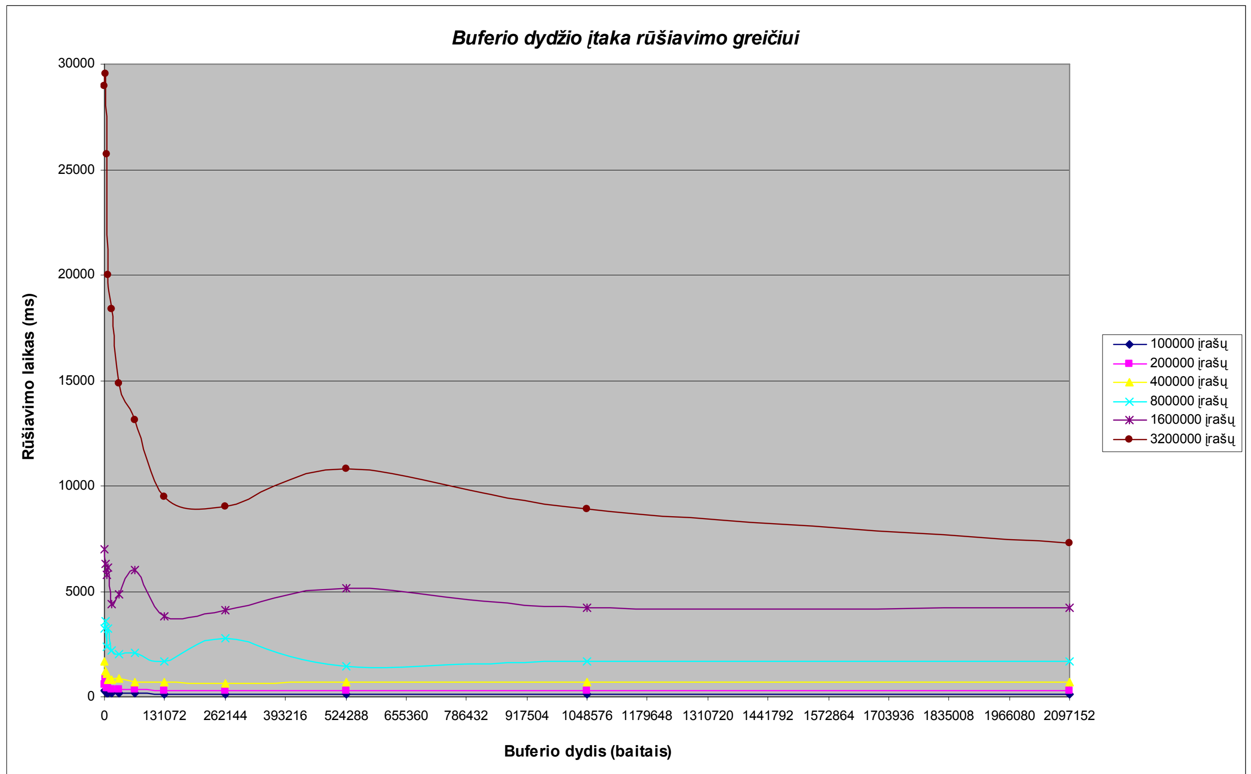
Bendrą viso algoritmo suliejimo etapais atliekamų veiksmų skaičių sudaro m kartų atliekamas greitas rūšiavimas kiekvienam blokui ir bendras suliejimo etapų atliekamų veiksmų skaičius, apskaičiuojamas pagal (7) formulę. Bendras viso algoritmo suliejimo etapais atliekamų veiksmų skaičius apskaičiuojamas pagal (8) formulę.

$$t = m \cdot n \cdot \log_2 n + m \cdot n \cdot \log_2 m \qquad (8)$$

čia t – bendras algoritmo atliekamų veiksmų skaičius.

Naudojant suliejimo etapais algoritmą sutaupoma operatyviosios atminties, bet prarandama laiko saugant ir nuskaitant išrūšiuotus blokus pastoviojoje atmintyje. Naudojant vieną didelį bloką sunaudojama labai daug atminties, bet nereikia lėtų rašymo į diską ir skaitymo iš disko operacijų. Bendru atveju bloko dydis gali būti didesnis už operatyviosios atminties dydį. Tokiu atveju operacinė sistema pradės naudoti virtualiąją atmintį, kas savo ruožtu labai stipriai įtakos sistemos efektyvumą. Taigi kursoriaus rūšiavimui yra naudojamas suliejimo etapais algoritmas.

Naudojant suliejimo etapais algoritmą reikia parinkti optimalų kursoriaus buferio dydį. Nuo to priklauso sunaudojamos operatyviosios atminties kiekis ir rūšiavimo greitis. Buvo atliktas testas, kurio metu nustatytas optimalus kursoriaus buferio dydis. Testuojamas kursorius, kuriame yra tik vienas stulpelis. Stulpelio tipas yra sveikasis skaičius (*int*, 4 baitai). Buferio dydis keičiamas nuo 1024 (2^{10}) iki 2097152 (2^{21}) baitų dvigubinant ankstesnįjį buferio dydį, o įrašų kiekis nuo 100000 iki 3200000 dvigubinant ankstesnįjį įrašų kiekį. 26 pav. pateikiama diagrama, rodanti kursoriaus dydžio įtaką rūšiavimo greičiui. Iš diagramos nustatomas optimalus buferio dydis. Buferio dydžio intervalas yra nuo 262144 (2^{18}) iki 1048576 (2^{20}) baitų. Naudojant tokį buferio dydį sunaudojama santykinai mažai atminties, o rūšiavimo greitis yra labai artimas 2 MB ir didesnių buferių kursoriaus rūšiavimo greičiams.



26 pav. Buferio dydžio įtaka rūšiavimo greičiui

3.1.3. Grupavimo kursorius

Grupavimo kursorius skirtas grupuoti tam tikrus duomenis pagal tam tikrus kriterijus. Norint optimaliai grupuoti duomenis iš pradžių juos reikia išrūšiuoti pagal grupavimo kriterijų. Pvz.: jeigu grupuojami 1 lentelės A ir C stulpeliai, tai lentelės duomenys turi būti išrūšiuoti pagal A stulpelį, po to jeigu yra pasikartojančių reikšmių stulpelyje A, tai turi būti išrūšiuoti ir pagal C stulpelį. Kai duomenys yra išrūšiuoti, jie nuosekliai peržiūrimi ir vienodas reikšmes turintys įrašai pagal grupavimo prioritetus yra sugrupuojami: atliekama konkreti agregatinė funkcija (pvz.: surandamas minimumas, maksimumas, suma, vidurkis, kiekis tarp tų įrašų kurie yra vienodi pagal grupavimo prioritetus [5, 151p]).

Skirtingai nuo rūšiavimo kursoriaus, grupavimo kursorius po grupavimo gali turėti mažiau arba daugiau stulpelių lyginant su stulpelių kiekiu grupavimo pradžioje. Rūšiavimo kursoriaus stulpelių kiekis po rūšiavimo gali tik sumažėti. Grupavimo metu, kai kurie stulpeliai gali būti naudojami tik grupavimui atlikti, bet patys stulpeliai nebus saugojami po grupavimo operacijos. Taip pat stulpelių kiekis gali ir padidėti, nes gali reikėti išsaugoti sugrupuotų stulpelių funkcijų rezultatus (pvz.: *select A, B, sum(B) from Table1*).

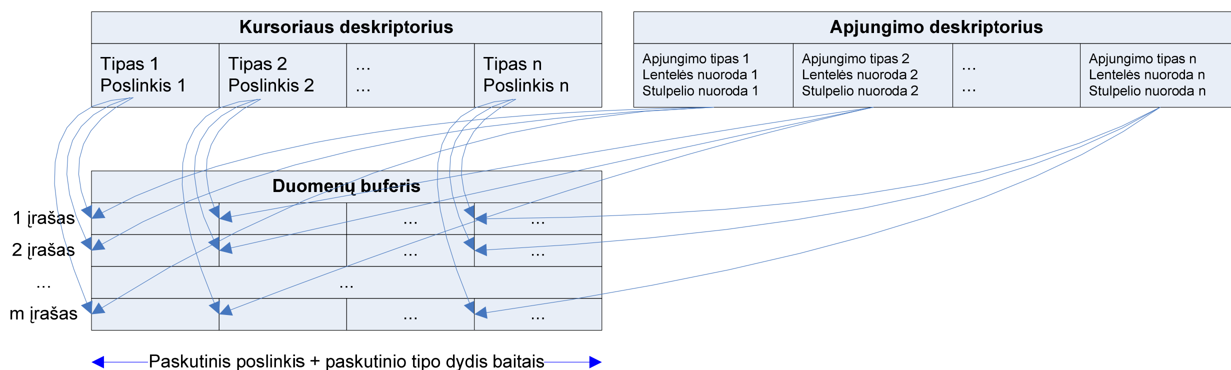
Grupavimo metu pagal SQL92 standartą [5, 153p] galima nurodyti kuriuos įrašus reikia grupuoti. Tai nurodoma tam tikra sąlyga (pvz.: *select A, B, sum(B) from Table1 having A > 0*). Grupavimo metu tikrinant įrašus reikia patikrinti grupavimo sąlygą ir pagal gautą rezultatą įtraukti įrašą į grupę arba ne.

Grupavimo metu duomenys saugojami duomenų ir eilučių buferiuose kaip ir anksčiau aprašytuose kursoriuose. Kai buferiai užpildomi, jie perkeliami i failą. Skaitymo metu galima nuskaityti iš failo bet kurį bloką pagal jo indeksą ir taip nuskaityti sugrupuotą įrašą pagal tam tikrą poziciją.

3.1.4. Apjungimo kursorius

Apjungimo kursorius yra naudojamas dviejų lentelių apjungimo (angl. *join*) operacijai atlikti. Apjungimo kursorius iš esmės yra labai panašus į rūšiavimo kursorių. Jis neturi galimybės saugoti kintamo ilgio duomenis, t.y. apjungimo kursorius turi tik duomenų buferį, bet neturi eilučių buferio. Tokio tipo kursorius sukurtas tam, kad būtų galima greičiau ir paprasčiau atlikti rūšiavimą nei įprastame rūšiavimo kursoriuje. Tai pasiekiama nenaudojant įvairių tikrinimo sąlygų dėl kintamo ilgio duomenų. Naudojami tik sveikųjų skaičių tipai (*char* (1 baitas), *short* (2 baitai), *int* (4 baitai), *__int64* (8 baitai)).

Apjungimo kursoriuje yra saugojami lentelių įrašų numeriai (*int* (4 baitai)) iš abiejų apjungiamų lentelių. Bendru atveju apjungiamų lentelių skaičius gali būti didesnis už 2. Apjungimo kursorius turi specialų deskriptorių, kuriame saugojama informacija apie kursoriaus duomenis. Kursoriuje be lentelių įrašų indeksų gali būti saugojamos stulpelių unikalių reikšmių indeksai (suspaustos lentelės stulpelį atitinkančio duomenų elemento unikalių reikšmių indeksai), kurie naudojami apjungimo sąlygoms patikrinti. Dėl to apjungimo kursoriaus deskriptorius turi nuorodą į lentelės arba stulpelio objektą. Stulpelių reikšmės priklausomai nuo apjungiamų lentelių eiliškumo gali būti dviejų tipų: naudojamos apjungimui einamuoju metu ir reikšmės, kurios bus naudojamos ateityje apjungiant jau apjungtas lenteles su kitomis (apjungtomis arba ne) lentelėmis). Apjungimo kursoriaus vidinė struktūra ir jo deskriptorius pateiktas 27 pav. Apjungimo algoritmas bus aptartas kitame skyriuje.



27 pav. Apjungimo kursoriaus ir jo deskriptoriaus struktūra

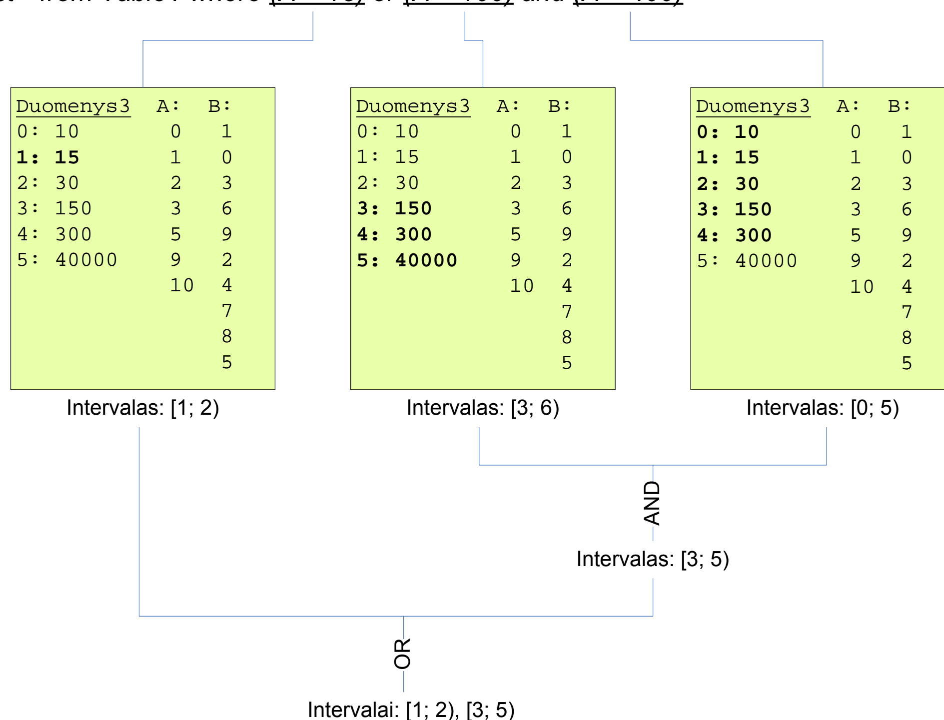
3.2. Paieškos filtras

Lentelės medžio konstravimo metu suformuojami A ir B masyvai, atsakingi už duomenų elementų unikalių reikšmių pozicijas lentelės įrašuose. Surasti tam tikrą kriterijų tenkinančius įrašus naudojami A ir B masyvų duomenys. Duomenų paieškos kriterijus gali būti paprastas, kai ieškoma pagal vieno stulpelio reikšmes iš vieno intervalo (pvz.: *E > 10*) arba sudėtinis, kai ieškoma pagal vieno stulpelio reikšmes iš kelių intervalų ir šie intervalai tarpusavyje gali būti jungiami loginėmis *AND*, *OR*, *NOT* operacijomis (pvz. *(E < 100) OR (E > 200) AND (E < 300)*). Paieškos algoritmas aprašomas paieškos filtru. Algoritmo darbo metu yra saugojami unikalių reikšmių indeksai medžio duomenų elementuose, kurių unikalios reikšmės tenkina paieškos kriterijų. Paieškos filtre išsaugojamas unikalių reikšmių indeksų intervalas (intervalai). Kai paieškos kriterijus yra sudėtinis, tai intervalus reikia apjungti atitinkamomis loginėmis operacijomis, t.y. atlikti intervalų sankirtą arba sąjungą. Apjungiant intervalus reikia laikytis loginių operacijų prioritetų. Pirmiausiai atliekama *NOT* operacija, po to *AND* ir galiausiai *OR*. Paieškos kriterijus intervalų apjungimo atžvilgiu turi būti pradėtas nagrinėti nuo logine prasme „giliausių“ sąlygų atsižvelgiant į loginių operacijų prioritetus. Intervalams saugoti yra naudojamas dviejų skaitmenų vektorius. Kiekviename lygyje atlikus intervalų apjungimą intervalų kiekis gali pasikeisti. Dėl šios priežasties vektoriuje reikia įterpti arba ištrinti intervalus, o tai nėra numatyta vektoriaus realizacijoje. Tam naudojama žemo lygio funkcijos atliekančios operatyvios atminties blokų perkėlimą iš vienos vietos į kitą (pvz.: *memmove()*, *memcpy()* ir pan.).

28 pav. pavaizduotas SQL užklausos paieškos kriterijaus sąlygų intervalų apjungimas. Duomenys naudojami iš 1 lentelės. Pabrauktas sąlygos tenkinantys unikalių reikšmių

indeksai yra paryškinti. Po to atliekamas hierarchinis intervalų apjungimas priklausomas nuo paieškos kriterijaus.

SQL užklausa: *select * from Table1 where (A = 15) or (A > 100) and (A < 400)*



28 pav. Užklausos intervalų apjungimo pavyzdys

Intervalo pradžia yra įtraukiama, o intervalo galas netraukiamas į intervalą. Tai palengvina intervalų paiešką naudojant STL bibliotekos paieškos funkcijas apjungiant intervalus.

Unikalių reikšmių indeksų intervalų sukūrimas nesuteikia galutinio paieškos rezultato, nes tai nėra lentelės įrašų numeriai, o tik vidinis sistemos duomenų išsaugojimas. Norint pakeisti intervalus į lentelės įrašų indeksus reikia intervalus perindeksuoti panaudojant A ir B masyvus. Vienas unikalios reikšmės indeksas turi atitikmenį A masyve pagal savo indeksą. Po to, vienas ar daugiau elementų iš masyvo A yra naudojami surasti masyvo B reikšmėms. B reikšmės yra lentelės įrašų numeriai. Atliekant tokią procedūrą kiekvieną kartą reikia išsaugoti filtro būseną, t.y. intervalo numerį, A ir B masyvų pozicijas, A masyvo paskutinę reikšmę (intervalo pabaiga).

SQL užklausų sąlygų (paieškos kriterijaus) išskaidymas į hierarchinę struktūrą bus aptariamas kitame skyriuje.

3.3. Bitų filtras

Paieškos filtras yra naudojamas tada, kai tikrinamose sąlygose yra tik vienas stulpelis. Jeigu sąlygoje yra keli skirtingi stulpeliai iš tos pačios lentelės paieškos filtras netinkama, nes nėra būdo, kaip susieti skirtingų stulpelių unikalius duomenis, bei A ir B masyvus. Dėl šios priežasties naudojamas bitų filtras. Jo veikimo principas paremtas bitinėmis *AND*, *OR* operacijomis. Bito numeris bitų filtre atitinka lentelės įrašo numerį. Bitų filtre bitai yra saugojami vektoriuje, kurio elementai yra pastovaus dydžio sveikųjų skaičių tipo. Naudojamas didžiausias sveikųjų 64 bitų skaičių tipas *__int64*, nes atlikti logines operacijas galima su didesnėmis bitų grupėmis naudojant vieną operaciją. Bitų filtro sukūrimo metu nustatomi pradiniai bitai, o po to atliekamos bitinės operacijos su pavieniais bitais arba su kitais bitų filtrais. Atlikus bitines operacijas bitų filtras gali būti paverstas į sveikųjų skaičių vektorių.

4. SQL užklausų gramatinis nagrinėjimas

Reliacinėse duomenų bazėse labiausiai paplitę SQL tipo užklausos. Duomenų bazių valdymo sistemos prototipe šios užklausos yra palaikomos. Kadangi DBVS prototipas yra skirtas tik duomenų skaitymui, tai naudojama tik dalis SQL galimybių. Nenaudojamos duomenų įrašymo, lentelių struktūrų keitimo, lentelių pašalinimo ir kitos funkcijos (pvz.: *update*, *alter*, *delete* ir pan.). Naudojamos tik duomenų išrikimo ir manipuliacijos duomenimis funkcijos (pvz.: *select*). Užklausų pateikimas DBVS prototipui ir rezultatų nuskaitymas yra atliekamas per ODBC (angl. *Open DataBase Connectivity*) tvarkyklę [7]. SQL užklausų vykdymas yra atliekamas tik tada, kai atlikta sintaksinė ir semantinė analizė. Šiai analizei atlikti yra naudojamas gramatinis nagrinėtojas (angl. *parser*). Gramatinis nagrinėtojas yra sukurtas naudojant ANTLR (angl. *ANother Tool Language Recognition*) įrankį [6]. ANTLR įrankis pagal turimą SQL gramatiką [8] sugeneruoja gramatinį nagrinėtoją. Gramatinis nagrinėtojas yra tam tikros programavimo kalbos kodas. Šis kodas atspindi SQL gramatikos [8] gramatinį medį.

DBVS prototipo SQL užklausų gramatinis nagrinėjimas yra suskaidytas į atskiras dalis:

- *SELECT* dalis, kurioje nurodomas ieškomų duomenų arba jų tam tikrų skaičiuojamųjų išraiškų sąrašas,
- *FROM* dalis, nurodomas lentelių sąrašas iš kurių yra naudojami duomenys,
- *WHERE, ON* dalis, nurodomas įrašų paieškos ir lentelių apjungimo kriterijai,

- *ORDER BY* dalis, rezultatų rūšiavimo kriterijus,
- *GROUP BY*, *HAVING* dalis, duomenų ir rezultatų grupavimas nurodant grupavimo kriterijų,
- vidinės užklausos (angl. *sub-query*), naudojamos kaip pagalbinė priemonė formuojant pagrindinės užklausos paieškos kriterijų.

Kiekvienos dalies gramatinis nagrinėjimas yra aprašomas atskirame skyriuje. Skaičiuojamųjų išraiškų vykdymas yra naudojamas daugelyje dalių.

4.1.1. Skaičiuojamosios išraiškos

Skaičiuojamosios išraiškos, tai tokios išraiškos, kurias reikia gramatiškai išnagrinėti ir po to atlikti tam tikras matematines ar kitas funkcijas. Išraiškos gali būti sudėtingos, t.y. turėti daugiau nei vieną skaičiavimo lygį (pvz.: *(A+B+(C+D)*A)*B*), kurie atskiriami skliausteliais. Tokioms išraiškoms apskaičiuoti yra naudojama lenkiška forma (angl. *polish notation*) [9]. Tokia išraiškos forma leidžia pakeisti skaičiuojamąją išraišką iš kelių skaičiavimo lygių į vieną. Sukonstruotą išraišką reikia vykdyti nuosekliai iš kairės į dešinę. Pvz.: išraiškos *(0 + 1) * (2 + 3)* lenkiška forma yra ** + 0 1 + 2 3*. Lenkiška forma yra nepatogi, tuo kad jos priekyje yra operacija, o tik po to operandai. DBVS prototipe naudojama atvirkštinė lenkiška forma (angl. *reverse polish notation*) [10], kurioje pirmiausiai yra operandai, o tik po to operacija. Pvz.: išraiškos *((1 + 2) * 4) + 3* atvirkštinė lenkiška forma yra *1 2 + 4 * 3 +*. SQL užklausų gramatinis nagrinėtojas nekintamas skaičiuojamas išraiškas (neturinčias kintamųjų) pakeičia į konstantę prieš tai apskaičiavęs išraiškos rezultatą. Taip sutaupoma nemažai laiko atliekant kiekvieno grąžinamo įrašo suformavimą. Kintamos skaičiuojamosios išraiškos suformuojamos taip, kad vietoj operacijos ženklo yra įterptas adresas į funkciją, kurią reikia iškviesti su tam tikrais operandais.

4.1.2. Duomenų išrinkimas

SELECT dalyje pateikiamas ieškomų duomenų sąrašas. Jame yra išvardinami lentelių stulpeliai (pvz.: *select A, B, C ...*), įvairios skaičiuojamosios išraiškos (pvz.: *select A+B+1, A*D, ...*), konstantos (pvz.: *select 1, 2 ...*). Šiam sąrašui talpinti yra naudojamas vektoriaus konteineris. Vektoriaus elementai yra objektai, galintys saugoti savyje tiek funkcijos adresą, tiek operandą, tiek ir suskaičiuotą rezultatą. Grąžinant užklausos rezultatus, kiekvienas

įrašas turi būti perskaičiuojamas. Perskaičiavimas gali būti visiškai elementarus, t. y. tam tikrų stulpelių reikšmių perrašymas neatliekant jokių skaičiavimų.

4.1.3. Lentelių sąrašas

Užklausos šaltinis gali būti viena arba daugiau lentelių. Jeigu užklausos naudojami duomenys yra tik iš vienos lentelės, tai duomenų apjungimas tarp skirtingų lentelių nėra atliekamas ir *FROM* dalyje yra tik vienos lentelės pavadinimas. Kai užklausoje yra naudojama daugiau nei viena lentelė, tai užklausos vykdymo metu turės būti atliekamas duomenų apjungimas tarp visų naudojamų lentelių ir *FROM* dalyje yra išvardinamos visos užklausoje naudojamos lentelės. Lentelių sąrašui saugoti yra naudojamas vektorius. Vektoriaus elementai yra adresai į lentelių objektus.

Atliekant lentelių apjungimą *FROM* dalyje yra naudojamas lentelių apjungimo tipą nurodantis raktažodis *JOIN*. Apjungimo tipai yra kelių rūšių: *INNER JOIN* (*JOIN*) (rezultatas visos galimos lentelių įrašų kombinacijos), *LEFT JOIN* (kairiosios lentelės visi įrašai ir pakartojami tiek kartų kiek yra sąlygą tenkinančių įrašų iš dešiniosios lentelės), *RIGHT JOIN* (dešiniosios lentelės visi įrašai ir pakartojami tiek kartų kiek yra sąlygą tenkinančių įrašų iš kairiosios lentelės), *OUTER JOIN* (pirmų trijų tipų unikalių įrašų aibė).

4.1.4. Paieškos kriterijus

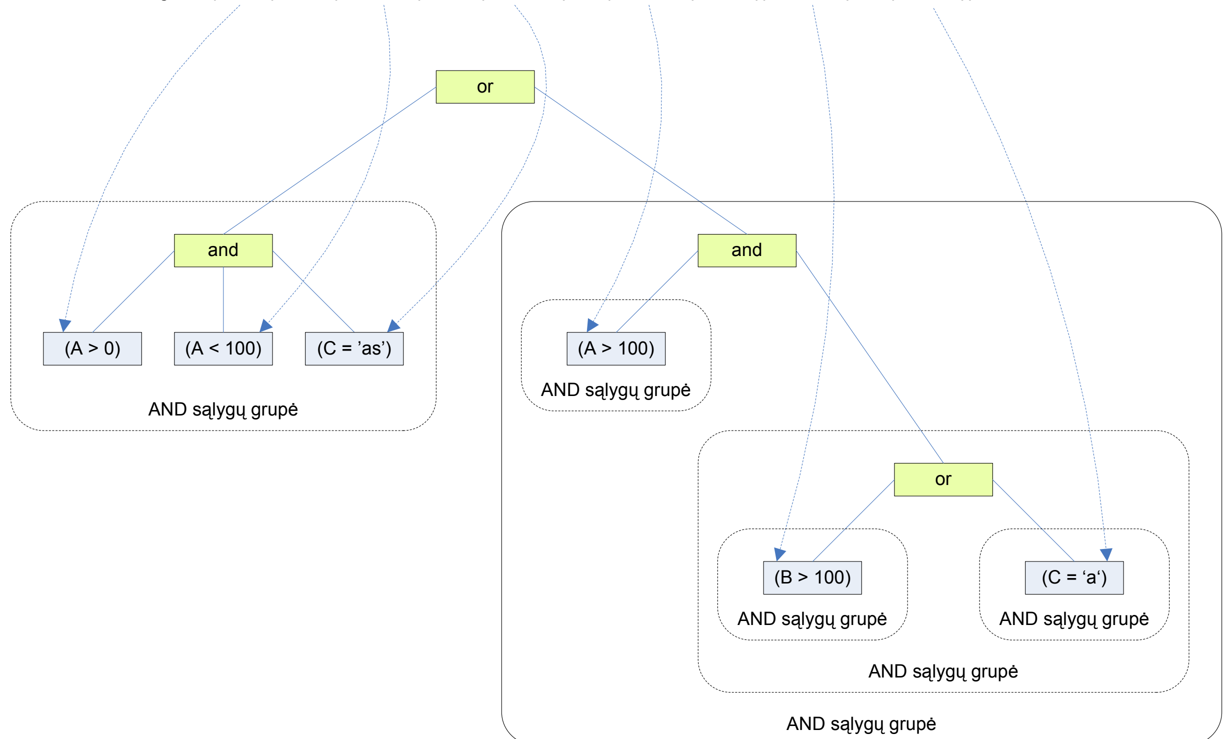
Paieškos kriterijus nurodo, kokie duomenys iš lentelės (lentelių) turi būti naudojami suformuojant užklausos rezultatą. Jeigu naudojama tik viena lentelė arba naudojama kelios lentelės, kai apjungimo tipas yra *INNER JOIN* arba *OUTER JOIN*, tai paieškos kriterijus yra nurodomas po raktažodžio *WHERE*. Jeigu naudojamos daugiau nei viena lentelė ir apjungimo tipas yra *LEFT JOIN* arba *RIGHT JOIN*, tai paieškos kriterijus rašoma po *ON* raktažodžio. Iš esmės paieškos kriterijai niekuo nesiskiria vykdymo prasme vertinant ar jie yra po *WHERE* ar *ON* raktažodžių.

Gramatinis nagrinėtojas paieškos kriterijų išskaido į hierarchinį sąlygų medį. Sąlygos yra skirstomos į dvi pagrindines grupes, kurios savo ruoštu į dar smulkesnes. Sąlygų grupės:

- Palyginimo sąlygos (nereikia atlikti jokių skaičiavimo operacijų, naudojamos tik palyginimo operacijos):
  - stulpelio lyginimas su konstante (pvz.: $A > 0$),

    - stulpelio lyginimas su stulpeliu iš tos pačios lentelės (nenaudojamas lentelių apjungimas) (pvz.: *A > B*),
    - stulpelio lyginimas su stulpeliu iš kitos lentelės (naudojamas lentelių apjungimas) (pvz.: *Table1.A > Table2.A)*.
- Skaičiavimo sąlygos (reikia atlikti skaičiavimo operacijas):
    - išraiška iš tos pačios lentelės stulpelių (pvz.: *sin(A) > sin(B)*),
    - išraiška iš skirtingų lentelių stulpelių (pvz.: *sin(Table1.A) > sin(Table2.B)*).

Paieškos kriterijuje sąlygos yra sujungtos loginėmis *AND*, *OR* operacijomis. Kadangi pirmiausiai vykdomos tos sąlygos, kurios sujungtos *AND* loginėmis operacijomis, tai paieškos kriterijus yra skaidomas į *AND* loginėmis operacijomis sujungtas sąlygų grupes (pvz.: paieškos kriterijus *(A > 0) and (B > 0) or (C > 0)* turi dvi *AND* sąlygų grupes: *(A > 0) and (B > 0)* ir *(C > 0)*. Šis skaidymas paremtas tuo, kad vykdant sąlygas iš vienos *AND* loginių operacijų sąlygų grupės (toliau *AND* sąlygų grupės) bent vienai sąlygai grąžinant neigiamą rezultatą iškart galima nustoti tikrinti likusias sąlygas, nes visos *AND* sąlygų grupės rezultatas vis vien bus neigiamas. Kiekviena *AND* sąlygų grupė gali turėti viduje kitas *AND* sąlygų grupes. Sąlygų vykdymas atliekamas nuo sąlygų giliausio lygio kylant į viršų. Taip gaunamas *AND* sąlygų grupių hierarchinis medis. 29 pav. pateiktas paieškos kriterijaus sąlygų hierarchijos medžio pavyzdys. Punktyrinės rodyklės žymi paieškos kriterijaus sąlygų padėtį sąlygų hierarchijos medyje. Užapvalinti stačiakampiai gaubia *AND* sąlygų grupes.

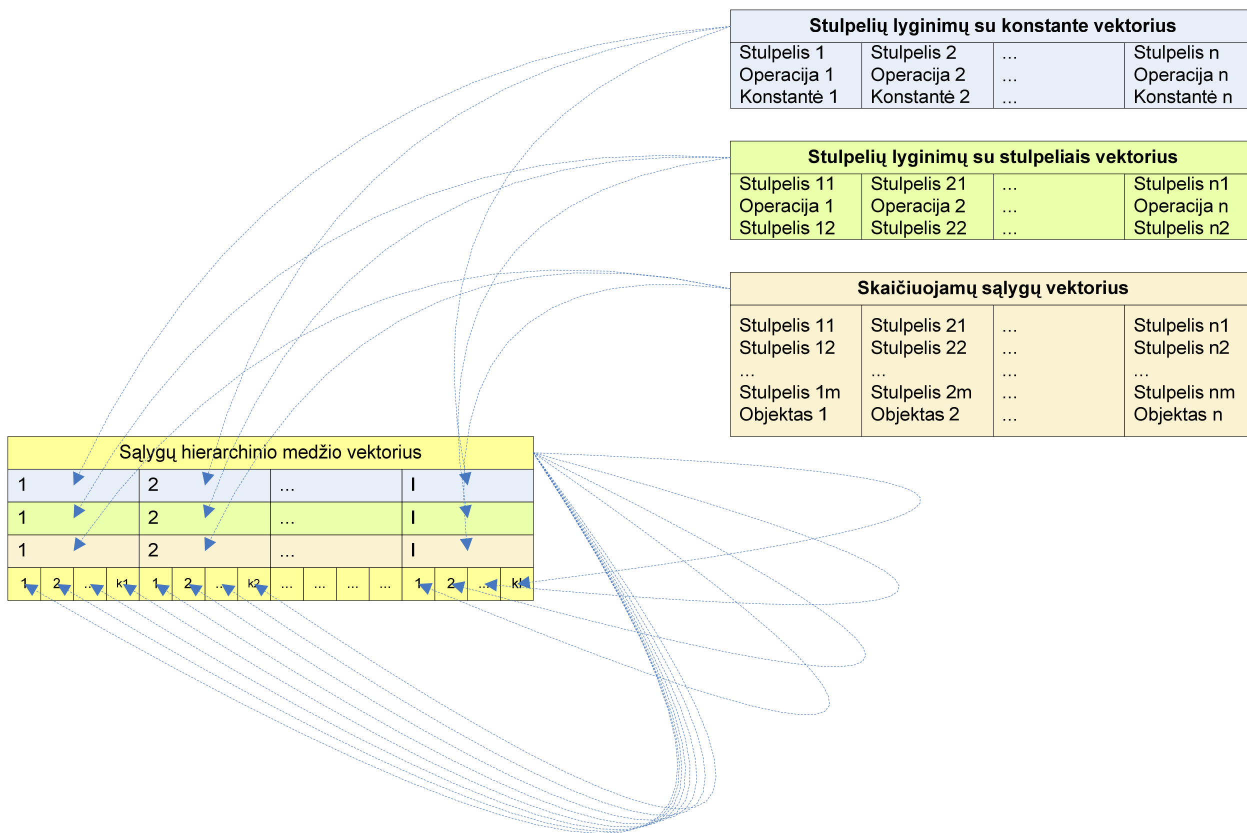


29 pav. Paieškos kriterijaus sąlygų hierarchijos medžio pavyzdys

Gramatinis nagrinėtojas sąlygų hierarchinį medį saugo vektoriuje, kurio elementai yra kitų sąlygų vektoriai. Struktūra gaunasi rekursyvi, tačiau vykdymo metu yra lengvai apdorojama. Sąlygų hierarchinio medžio vektoriuje yra saugojami šie vektoriai:

- stulpelių lyginimų su konstante *AND* grupės sąlygų vektorius,
- stulpelių lyginamų su stulpeliais *AND* grupės sąlygų vektorius,
- skaičiuojamų sąlygų *AND* grupės sąlygų vektorius (saugojami tik stulpelių adresai, visa informacija reikalinga apskaičiuoti sąlygą yra saugojama atvirkštinėje lenkiškoje formoje atskirame objekte),
- sąlygų hierarchinio medžio vektoriaus *AND* grupės sąlygų vektorius (rekursinis *AND* sąlygų grupių saugojimas).

30 pav. pateikta paieškos kriterijaus hierarchinių sąlygų medžio vektoriaus struktūra.



30 pav. Paieškos kriterijaus hierarchinių sąlygų medžio vektoriaus struktūra

Toks sąlygų grupavimas yra naudojamas tam, kad būtų galima kuo anksčiau nustatyti viso paieškos kriterijaus reikšmę. Detalesnis šių sąlygų vykdymo eiliškumas pateikiamas lentelių apjungimo skyriuje.

4.1.5. Rūšiavimo kriterijus

ORDER BY dalyje pateikiamas lentelių stulpelių (pvz.: *order by A desc, C asc*) arba skaičiuojamų išraiškų sąrašas (pvz.: *order by 1+1 desc, A+1 asc, B+E desc*). Kiekvienas rūšiavimo kriterijaus elementas (stulpelis arba skaičiuojama išraiška) turi didėjančio ar mažėjančio rūšiavimo požymį, kuris naudojamas rūšiavimo metu.. Rūšiavimo kriterijaus sąrašui talpinti yra naudojamas vektoriaus konteineris. Vektoriaus elementai yra objektai, galintys saugoti savyje tiek lentelės stulpelio adresą, tiek skaičiuojamos išraiškos, perverstos į lenkišką atvirkštinę formą, objekto adresą. Atliekant duomenų rūšiavimą naudojamas rūšiavimo kursorius, kuris buvo aptartas anksčiau.

4.1.6. Grupavimo kriterijus

GROUP BY dalyje nurodoma pagal kokius stulpelius ar skaičiuojamąsias išraiškas grupuoti duomenis. Priklausomai nuo grupavimo kriterijaus, duomenų išrinkimo *SELECT* dalyje turi būti arba sugrupuoti rezultatai arba jų skaičiuojamosios išraiškos. Grupavimo kriterijus sąrašui talpinti yra naudojamas vektoriaus konteineris. Vektoriaus elementai yra objektai, galintys saugoti savyje tiek lentelės stulpelio adresą, tiek skaičiuojamos išraiškos, perverstos į lenkišką atvirkštinę formą, objekto adresą. Atliekant duomenų grupavimą naudojamas grupavimo kursorius, kuris buvo aptartas anksčiau.

4.2. Sąlygų tikrinimas ir lentelių apjungimo algoritmas

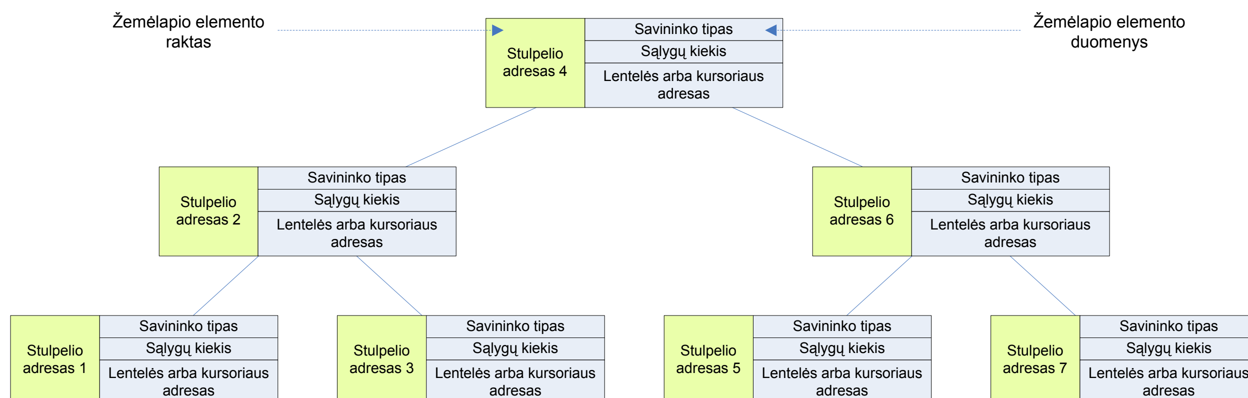
Sąlygų tikrinimas atliekamas visada, kai nurodomas paieškos kriterijus. Jeigu naudojama daugiau negu viena lentelė, tai atliekamas lentelių apjungimas. Vienos sąlygos gali būti tik iš tos pačios lentelės, kai tuo tarpu kitos gali būti ir iš skirtingų lentelių. Dėl šios priežasties sąlygų tikrinimas ir lentelių apjungimo algoritmas yra vienas nuo kito neatsiejami.

Sąlygos yra suskirstytos į penkias grupes, kurios buvo aptartos „Rūšiavimo kriterijus“ skyriuje. Kiekviena sąlygų grupė turi prioritetą prieš kitas sąlygų grupes, todėl sąlygų tikrinimo eiliškumas nustatomas pagal tam tikrą algoritmą.

Sąlygų tikrinimo ir lentelių apjungimo algoritmas prasideda nuo papildomų struktūrų sukūrimo ir jų užpildymo. Kiekviena lentelė turi bitų filtrą, kuris yra aprašytas „Bitų filtras“ skyriuje. Šiame filtre nustatomi bitai į vienetą, jeigu sąlygose naudojamų stulpelių įrašų reikšmės tenkina sąlygas. Taip kiekvienoje lentelėje nustatomi sąlygas tenkinantys įrašų numeriai. Bet kuris lentelių stulpelis gali būti naudojamas keliose sąlygose. Jeigu sąlygose naudojami stulpeliai yra iš skirtingų lentelių, tai reikia atlikti lentelių apjungimą, naudojant apjungimo kursorių. Šis kursorius yra aptartas skyriuje „Apjungimo kursorius“. Naudojant šį kursorių yra saugojami lentelių įrašų numeriai ir sąlygose naudojamų stulpelių unikalių reikšmių indeksai lentelės medžio duomenų elemente. Tokiu atveju stulpelis gali būti jau įtrauktas į apjungimo kursorių arba gali būti dar neįtrauktas ir priklausyti vienai iš lentelių. Kad nereikėtų pakartotinai nuskaitinėti stulpelių unikalių reikšmių indeksų, yra naudojamas stulpelių priklausomybės žemėlapis. Jo struktūra pateikta 31 pav.

Stulpelių priklausomybės žemėlapis yra konteineris, kurio tipas yra žemėlapis. Toks konteinerio tipas parinktas dėl to, kad labai dažnai sąlygų tikrinimo metu reikia žinoti, kam priklauso stulpelis. Žemėlapio raktas yra stulpelio adresas, o duomenys yra stulpelio savininko

tipas (lentelė arba kursorius), sąlygų kiekis, kuriose šis stulpelis dar bus naudojamas ir priklausomai nuo stulpelio tipo lentelės arba kursoriaus adresas. Apdorojant to paties stulpelio kitą sąlygą, sąlygų kiekis yra mažinamas vienetu. Taip galima nustatyti, kad sąlygoje naudojamų stulpelių unikalių reikšmių indeksų galima nesaugoti. Jeigu naudojami duomenys ne iš vienos lentelės tai lentelių apjungimo rezultatas yra kursorius, priešingu atvejų rezultatas yra lentelėje nustatytas paieškos arba bitų filtras.



31 pav. Stulpelių priklausomybės žemėlapis

Pirmiausiai tikrinamos elementariausios sąlygos (stulpelio lyginimas su konstante, stulpelio lyginimas su stulpeliu iš tos pačios lentelės), po to tikrinamos sudėtingos sąlygos, kurioms reikalingas lentelių apjungimas. Anksčiau tikrinamos tos sąlygos, kurios potencialiai gali labiau sumažinti rezultatų aibę už kitas sąlygas.

4.2.1. Vienos lentelės sąlygų tikrinimas

Pačios paprasčiausios sąlygos yra stulpelių lyginimas su konstante. Šios sąlygos yra tikrinamos naudojant lentelės medį. Pagal sąlygos konstantę lentelės medžio duomenų elemente nustatomas unikalių reikšmių indeksų intervalas. Šiam intervalui pritaikant A ir B masyvus iš medžio duomenų elemento gaunami lentelės įrašų numeriai. Tai atliekama naudojant paieškos filtrą. Jeigu po šio tipo sąlygų apdorojimo bus naudojamos dar kitos sąlygos iš tos pačios lentelės, tai įrašų numeriai surašomi į bitų filtrą.

Toliau apdorojamos sąlygos, kuriose naudojami stulpeliai iš tos pačios lentelės. Vienos sąlygos yra tik stulpelių reikšmių palyginimas, kitos sąlygos yra tam tikra stulpelių skaičiuojamoji išraiška. Pirmiau yra patikrinamos palyginimo sąlygos, nes tam pakanka tik vienos palyginimo operacijos, vėliau tikrinamos skaičiuojamosios sąlygos, nes bendru atveju

suskaičiuoti vieno lentelės įrašo sąlygos teisingumą, gali reikėti daug operacijų. Palyginimo ir skaičiuojamų sąlygų tikrinimas atliekamas bitų filtrų nustatymu ir loginių operacijų atlikimu su jais. Šie tikrinimai atliekami su kiekvienu lentelės įrašu atsižvelgiant į bitų filtrą, kuris galėjo būti nustatytas apdorojant sąlygas, kai lyginami stulpeliai su konstantomis.

Sąlygų patikrinimo rezultatas yra arba paieškos filtras arba bitų filtras.

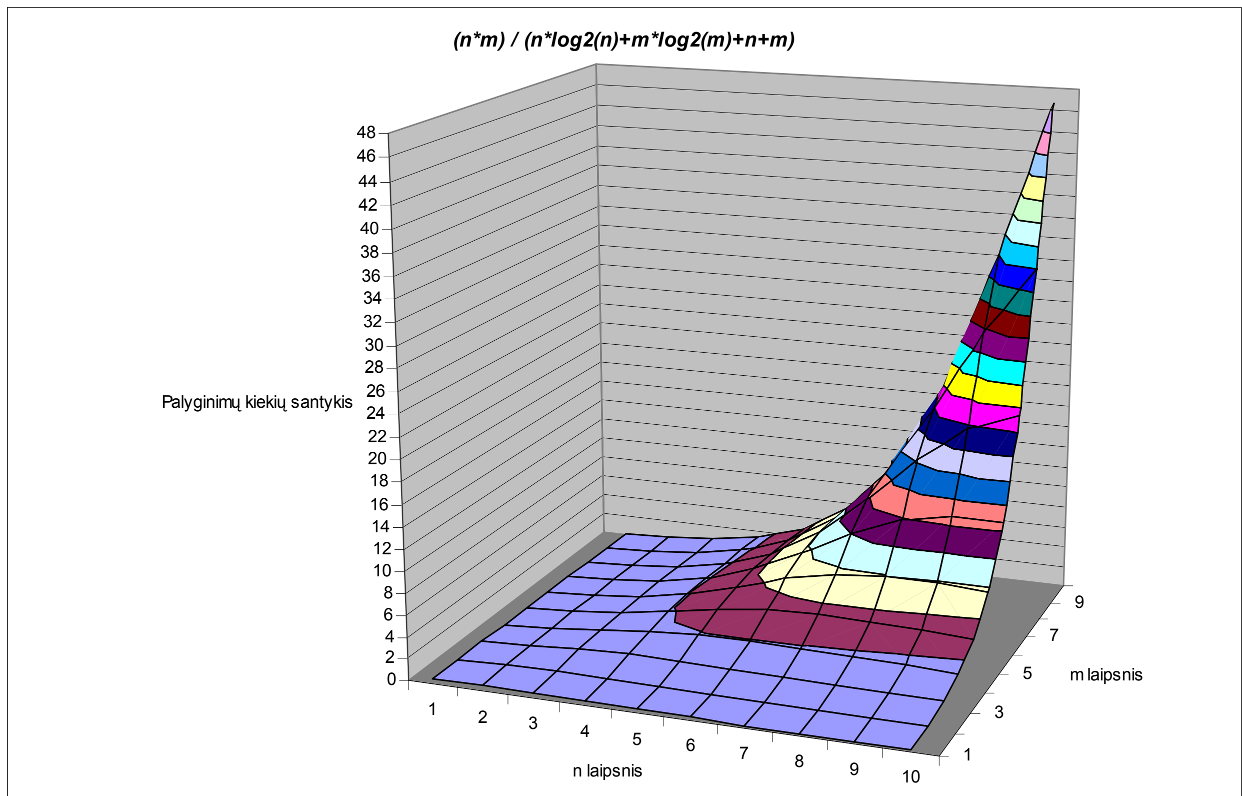
4.2.2. Kelių lentelių sąlygų tikrinimas

Tai sudėtingiausias sąlygų tikrinimo būdas. Šių sąlygų patikrinimas reikalauja santykinai daug laiko lyginat su sąlygomis iš tos pačios lentelės. Naudojant šias sąlygas nustatoma, kurie lentelių įrašai bus apjungti. Jeigu reikia apjungti dvi lenteles yra pagrindiniai du būdai tai atlikti:

- peržiūrint pirmosios lentelės įrašus kiekvieną kartą peržiūrinėti antros lentelės įrašus ir tikrinti apjungimo sąlygų teisingumą,
- išrūšiuoti lenteles pagal sąlygoje naudojamus stulpelius ir nuosekliai peržiūrint abiejų lentelių įrašus tikrinti apjungimo sąlygų teisingumą.

Pirmas būdas yra paprastas, ta prasme, kad nereikia atlikti jokių paruošiamųjų veiksmų, tačiau bendru atveju reikalauja labai daug palyginimo operacijų. Jeigu dviejų lentelių įrašų skaičius yra atitinkamai m ir n , tai bendras palyginimų kiekis yra jų sandauga (m*n).

Antras būdas reikalauja dviejų rūšiavimo procedūrų, tačiau palyginimų skaičius yra daug mažesnis. Jeigu dviejų lentelių įrašų skaičius yra atitinkamai m ir n , tai bendras palyginimų kiekis yra jų suma (m+n). Pirmą būdą tikslinga naudoti, kai lentelės turi mažai įrašų, o antrą – priešingu atveju. 32 pav. pateiktas abiejų metodų palyginimų kiekio santykis (pirmo metodo palyginimų skaičius padalintas iš antrojo metodo palyginimų skaičiaus) priklausomai nuo lentelių įrašų skaičiaus (m ir n). Kad schema būtų mažiau apkrauta dideliais skaičiais, tai įrašų kiekis išreiškiamas logaritminėje (pagrindas 2) skalėje. Iš šio paveikslėlio matome, kad antras būdas yra daug kartų greitesnis už pirmąjį, kai lentelėse yra virš 50 įrašų.



32 pav. Dviejų lentelių apjungimo skirtingais būdais palyginimų kiekių santykis

Kadangi DBVS prototipas skirtas dirbti su dideliais duomenų kiekiais, tai pirmas apjungimo būdas nebus naudojamas apskritai. Apjunginėjant daugiau nei dvi lenteles apjungiama gali būti ne vien tik lentelė su lentele, bet ir lentelė su apjungimo kursoriumi, apjungimo kursorius su apjungimo kursoriumi. Atliekant sąlygų tikrinimą tam tikra seka galima sumažinti lentelių arba apjungimo kursorių rūšiavimų kiekį, t.y. sekti, pagal kokius stulpelius buvo išrikiuotos lentelės arba apjungimo kursoriai ir pakartotinai nerūšiuoti jeigu rūšiavimo tvarka nepasikeitė.

Prieš atliekant sudėtingų sąlygų tikrinimą ir kartu lentelių apjungimą sąlygos turi būti išrūšiuotos tam tikra seka. Ši seka priklauso nuo palyginamų sąlygų operatorių (=, <, >, <=, >=, <>), apjungiamų lentelių arba apjungimo vektorių logaritminio įrašų kiekių skirtumo, mažiausio lentelių arba apjungimo kursorių rūšiavimo kiekio. Sekos gale yra skaičiuojamosios sąlygos, kuriose reikia apskaičiuoti sąlygos rezultatą.

Sąlygos gali būti tikrinamos tuo pačiu metu jeigu jose naudojami stulpeliai yra iš tų pačių lentelių arba apjungimo kursorių. Toks tikrinimas kartu paremtas tuo, kad kitą kartą nereikėtų daryti papildomų operacijų, kad pertikrinti visus arba dalį dviejų lentelių arba apjungimo kursorių įrašų. Po kiekvienos sąlygos arba sąlygų grupės iš tos pačios lentelės arba

apjungimo kursoriaus sąlygų tikrinimo seka turi būti iš naujo patikrinama, nes kai kurių lentelių ar kursorių duomenys galėjo apsijungti į naujus apjungimo kursorius. Ši seka yra realizuota kaip vektorius, kurio elementai yra struktūra. Struktūrą sudaro:

- palyginimo sąlygų adresų vektorius, kurios gali būti tikrinamos kartu,
- kairiosios ir dešiniosios sąlygų pusės stulpelių savininkų (lentelių arba apjungimo kursorių) tipai,
- kairiosios ir dešiniosios sąlygų pusės rūšiavimo kriterijus (vektorius), rezultatų rūšiavimo kriterijus (nurodoma kaip turėtų būti išrūšiuotas kursorius po duomenų apjungimo, kuris bus naudojamas vėliau apjungti pagal tuos stulpelius, pagal kuriuos yra išrūšiuojama),
- logaritminis lentelių arba apjungimo kursorių (toliau vadinama objektų) įrašų kiekio skirtumas apskaičiuojamas pagal (9) formulę.

  $$\lg_diff = \left|\lg m - \lg n\right|; \quad (9)$$

  čia m – pirmojo objekto įrašų kiekis;

  n – antrojo objekto įrašų kiekis.

- požymis, rodantis, ar galima peržiūrėti lentelės arba apjungimo kursoriaus įrašus negrįžtant prie jau peržiūrėtų įrašų (pvz.: jeigu naudojamas palyginimo sąlygos operatorius <>, tai antros lentelės arba apjungimo kursoriaus įrašai turi būti peržiūrimi pakartotinai, nes tokie įrašai gali būti praleisti jeigu bus peržiūrimi tik neperžiūrėti įrašai).

Apdorojant kiekvieną sąlygą reikia atlikti arba sąlygos patikrinimą arba lentelių ar apjungimo vektorių apjungimą pagal tam tikrą sąlygą.

Dažniausiai apjungimo sąlygose yra naudojamas = operatorius. Kai apjungimo kursoriai yra išrūšiuoti pagal apjungiamus stulpelius, tai naudojant = operatorių vienu praėjimu per abu apjungimo kursorius galima atlikti pilną tų kursorių apjungimą. Jeigu naudojami kiti operatoriai arba skaičiuojamosios sąlygos, tuomet bendru atveju negalima atlikti kursorių apjungimo vienu praėjimu. Reikia pakartotinai grįžti prie jau peržiūrėtų įrašų. Apjunginėjant apjungimo kursorius, tarpiniai apjungimo kursoriai yra saugojami aibės tipo konteineriuose. Taip galima labai greitai surasti, ar tam tikras apjungimo kursorius yra jau sukurtas.

5. DBVS sistemų užklausų vykdymo testai

Norint įvertinti kiek efektyvūs sukurti, optimizuoti algoritmai sukurtam DBVS prototipui reikia palyginti įvairių užklausų atlikimo greitį su kitomis gerai žinomomis diskinėmis DBVS ir pasirinkta *NPBase* sistema. Užklausos turi būti labai skirtingos, kad pamatuoti įvairius sistemos funkcionalumo aspektus. Duomenų bazė ir atliekamos užklausos, kurios bus naudojamos visose DBVS sistemose, yra sugeneruojamos naudojant TPCH (angl. *Transaction Processing Performance Council (ad-Hoc)*) įrankį [11].

TPCH testuose yra aprašomos 22 skirtingos užklausos. 5 iš 22 užklausų yra naujesnės nei SQL92 standartas, taigi jos nebus vykdomos. Likusios 17 užklausų bus vykdomos. Kelios užklausos iš 17 nebus įvykdytos su tam tikromis DBVS sistemomis dėl tam tikro funkcionalumo nepalaikymo arba netinkamos SQL sintaksės. Testai atliekami su trijų skirtingų dydžių duomenų bazėmis: 100MB, 400MB ir 1GB. Testų rezultatų palyginimui buvo pasirinkta *ORACLE*, *Microsoft SQL Server* ir *NPBase* DBVS sistemos. Su *NPBase* DBVS sistema buvo atliekami testai tik su 100MB dydžio duomenų baze, nes šios sistemos sunaudojamas operatyvios atminties kiekis išaugo 3 kartus lyginant su *ORACLE* ar *Microsoft SQL Server* DBVS sistemomis. 3, 4, 5, 6 lentelės pateikiami testų rezultatai. Užklausų vykdymo laikai pateikiami minutėmis ir tūkstantosiomis sekundės dalimis. Pilnos SQL užklausos pateikiamos priede, o lentelėse naudojami tik užklausų numeriai.

3 lentelė – DBVS prototipo užklausų vykdymo greitis

	Prototipas, 100 MB		**Prototipas, 400 MB**		**Prototipas, 1 GB**	
Užklausos nr.	Laikas (mm:ss.ms)	Laikas (ms)	Laikas (mm:ss.ms)	Laikas (ms)	Laikas (mm:ss.ms)	Laikas (ms)
1	00:18,5	18549	01:08,2	68218	03:20,0	200000
2	00:01,7	1721	-	-	-	-
3	00:01,4	1408	00:04,9	4876	00:13,2	13457
4	02:32,7	152740	-	-	-	-
5	00:02,6	2587	-	-	-	-
6	00:00,1	108	00:00,4	421	00:01,3	1300
7	00:06,7	6687	00:26,0	25984	01:44,1	104083
8	00:04,9	4930	00:17,4	17358	01:08,6	68637
9	00:04,1	4114	00:05,2	5181	00:14,3	14275
10	00:00,4	392	00:01,7	1705	00:04,7	4698
11	01:19,7	79700	-	-	-	-
12	00:03,0	2984	00:12,1	12088	00:43,8	43796
13	-	-	-	-	-	-
14	00:00,9	949	00:03,4	3359	00:08,7	8668
15	-	-	-	-	-	-
16	-	-	-	-	-	-
17	00:00,3	298	00:01,1	1117	00:02,8	2762

4 lentelė – ORACLE užklausų vykdymo greitis

	ORACLE, 100 MB		ORACLE, 400 MB		ORACLE, 1G	
Užklausos nr.	Laikas (mm:ss.ms)	Laikas (ms)	Laikas (mm:ss.ms)	Laikas (ms)	Laikas (mm:ss.ms)	Laikas (ms)
1	00:28,5	28547	02:28,0	148025	07:49,8	469806
2	00:04,5	4534	00:14,3	14338	00:30,8	30814
3	00:35,1	35089	02:25,0	144589	05:46,9	346852
4	00:32,9	32929	02:33,7	153658	05:14,5	314519
5	01:25,9	85878	03:45,2	225219	44:33,0	2672974
6	00:26,9	26883	02:08,8	128846	03:45,3	225283
7	-	-	-	-	-	-
8	-	-	-	-	-	-
9	00:35,7	35719	02:24,2	144227	20:52,5	1252471
10	00:12,8	12777	00:42,6	42569	01:38,0	97961
11	00:03,3	3253	00:11,6	11595	00:22,3	22296
12	00:26,1	26100	01:59,5	119509	03:54,1	234122
13	-	-	03:53,8	233823	04:30,4	270420
14	00:28,1	28124	02:07,1	127105	03:54,5	234456
15	00:33,2	33212	02:33,0	152961	04:12,8	252800
16	01:18,4	78394	21:47,4	1307370	12:25,4	745358
17	-	-	-	-	-	-

5 lentelė – Microsoft SQL Server užklausų vykdymo greitis

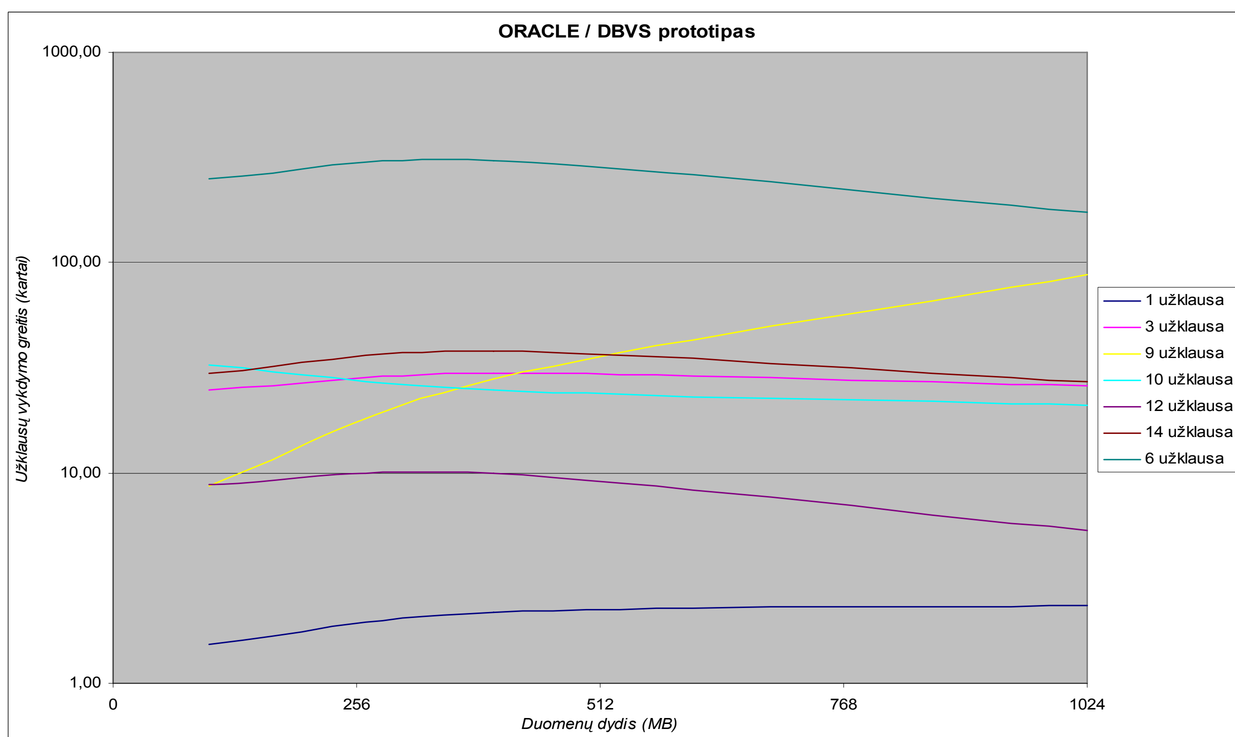
	MSSQL, 100 MB		MSSQL, 400 MB		MSSQL, 1G	
Užklausos nr.	Laikas (mm:ss.ms)	Laikas (ms)	Laikas (mm:ss.ms)	Laikas (ms)	Laikas (mm:ss.ms)	Laikas (ms)
1	00:15,6	15630	01:00,6	60551	01:05,2	65194
2	00:04,4	4398	00:01,8	1773	00:17,4	17374
3	00:06,0	6036	00:07,6	7597	01:19,3	79266
4	00:01,1	1055	00:06,1	6118	01:06,6	66557
5	00:00,6	610	00:04,1	4114	01:45,7	105697
6	00:00,6	633	00:03,9	3871	00:47,7	47712
7	-	-	-	-	-	-
8	-	-	-	-	-	-
9	00:00,8	797	00:06,7	6688	01:06,1	66127
10	00:00,2	184	00:01,0	984	00:12,3	12257
11	00:00,8	779	00:02,1	2118	00:08,0	7998
12	00:01,0	966	00:04,4	4448	00:54,3	54315
13	00:02,5	2481	00:26,1	26122	01:11,8	71821
14	00:01,8	1795	00:06,5	6494	00:54,3	54310
15	00:01,2	1209	00:05,1	5071	01:00,5	60451
16	00:02,5	2493	00:14,2	14236	01:50,4	110411
17	-	-	-	-	-	-

6 lentelė – NPBase užklausų vykdymo greitis

	NPBase, 100 MB	
Užklausos nr.	Laikas (mm:ss.ms)	Laikas (ms)
1	00:07,5	7525
2	-	-
3	-	-
4	-	-

5	-	-
6	00:00,5	496
7	-	-
8	-	-
9	-	-
10	-	-
11	-	-
12	-	-
13	-	-
14	00:00,9	922
15	-	-
16	-	-
17	-	-

33 pav. pavaizduotas *ORACLE* ir DBVS prototipo užklausų vykdymo greičių santykis, t.y. kuo didesnis šis santykis, tuo DBVS prototipo užklausų vykdymo greitis yra didesnis (kartais). Užklausos pagal jų numerį pateiktos priede su detalesniu aprašymu. Iš paveikslėlio matome, kad šis santykis svyruoja priklausomai nuo užklausų tipo, tačiau vidurkis yra apie 30-40 kartų (atliekamas sudėtingas lentelių apjungimas). Paprasčiausios paieškos ir paprasčiausių lentelių apjungimo užklausų vykdymo greitis yra labai didelis prie mažesnių duomenų bazių – DBVS prototipas šias užklausas įvykdo 100-200 kartų greičiau nei *ORACLE*, tačiau prie didesnių duomenų bazių šis santykis ima mažėti.



33 pav. ORACLE ir DBVS prototipo užklausų vykdymo greičių santykis

6. Išvados

Šiame darbe buvo sukurtas DBVS prototipas. Sistemos duomenų suspaudimo, apdorojimo, paieškos ir kito funkcionalumo algoritmai yra eksperimentiniai ir pritaikyti konkrečiam duomenų saugojimo modeliui. Gautas darbo išvadas galima suskirstyti į tam tikras grupes.

- Duomenų saugojimo modelis pagrįstas medžio struktūra. Kiekviena duomenų bazės lentelė turi savo atitikmenį – medį.
- Naudojant lentelės medį buvo sukurtos efektyvios duomenų paieškos struktūros (paieškos filtrai) ir jas apdorojantys algoritmai.
- Paieškos filtrai tiesiogiai naudoja medžio elementų indeksų reikšmes, taip maksimaliai sumažinantys paieškos laiką.
- Atlikti lentelės įrašų indeksų filtravimui naudojamas bitų filtras, kurio pagalba galima atlikti logines operacijas tarp lentelės įrašų indeksų.
- Naudojant ANTLR įrankį buvo sugeneruotas SQL užklausų gramatinis nagrinėtojas, kuris SQL užklausų paieškos, rūšiavimo, grupavimo kriterijus išskaido į hierarchinę struktūrą.
- Skaičiuojamosioms išraiškoms apdoroti naudojama atvirkštinė lenkiška forma.
- Paieškos kriterijų galima tikrinti dalimis (loginių operacijų *AND* grupėmis), taip minimizuojant tikrinamų sąlygų kiekį. Jei bent viena sąlyga grąžina neigiamą rezultatą, tai visos *AND* sąlygų grupės rezultatas taip pat bus neigiamas.
- Suskirstytos sąlygos į tam tikras „sudėtingumo“ grupes. Tai leidžia iš pradžių tikrinti, tas sąlygas, kurios maksimaliai gali apriboti rezultatų aibę.
- Duomenų rūšiavimui, grupavimui ir apjungimui atlikti sukurti kursoriai, kurie naudoja greito rūšiavimo algoritmą dalimis, taip apribojant naudojamos operatyviosios atminties kiekį.
- Duomenų apjungimo algoritmas buvo orientuotas į maksimaliai greitą duomenų apjungimą, bet ne į operatyviosios atminties taupymą.
- Duomenų apjungimo algoritmas yra suskaidytas į apjungimo sąlygų rūšiavimą, optimizavimą, tikrinimą (duomenų apjungimą) ir apdorotų sąlygų pašalinimą.

- Atlikti DBVS prototipo užklausų vykdymo testai parodė, kad sukurti ir optimizuoti algoritmai yra gana efektyvus (30-40 kartų greičiau atliekamos sudėtingos apjungimo užklausos), tačiau nepasiektas 100 kartų greitesnis užklausų vykdymo greitis lyginant su diskinėmis DBVS sistemomis (*ORACLE*, *Microsoft SQL Server*).
- Paprastos duomenų paieškos ir lentelių apjungimo užklausų vykdymo greitis DBVS prototipo buvo apie 100-200 kartų didesnis nei *ORACLE*. Tai visiškai paaiškinama, nes DBVS prototipas buvo projektuojamas būtent tokio tipo užklausoms.
- Dirbant su didesnėmis duomenų bazėmis DBVS prototipo pranašumas ima mažėti. Tai paaiškinama per dideliu operatyviosios atminties kiekio išnaudojimu (labai padidėja atminties fragmentacija, pradedama naudoti operacinės sistemos virtuali atmintis).
- Lyginat sukurto DBVS prototipo užklausų vykdymo greitį su analogiška DBVS sistema *NPBase*, kuri taip pat skirta tik duomenų skaitymui ir duomenys laikomi tik operatyviojoje atmintyje darbo metu, paaiškėjo, kad duomenų paieška yra vykdoma kelis kartus greičiau, bet duomenų grupavimas yra lėtesnis.

7. Literatūra

1. „*NPBase*“, 2006 [žiūrėta 2005-08-09]. Prieiga per internetą http://www.clearpace.com/products_npbase.htm.
2. „*Practical Compressor Test*“, 2004 [žiūrėta 2006-04-09]. Prieiga per internetą: http://www.elis.ugent.be/~wheirman/compression/.
3. *Nicolai M. Josuttis, „The C++ Standard Library, A Tutorial and Reference“, 1999, Addison Wesley*
4. *„Quicksort“,* 2006 [žiūrėta 2006-05-04]. Prieiga per internetą: http://en.wikipedia.org/wiki/Quicksort.
5. *American National Standard for Information Systems, „Database Language – SQL“, 1992, American National Standards Institute, Inc.*
6. „*ANTLR: A Predicated Parser Generator*“, 2003 [žiūrėta 2006-04-09] . Prieiga per internetą: http://www.antlr.org/article/1055550346383/antlr.pdf.
7. „*Open Database Connectivity*“ 2006 [žiūrėta 2006-04-02]. Prieiga per internetą http://en.wikipedia.org/wiki/Open_Database_Connectivity.
8. „*MS SQL Server 2000 SELECT statement*“, 2003 [žiūrėta 2006-04-13]. Prieiga per internetą: http://www.antlr.org/grammar/1062280680642/MS_SQL_SELECT.html.
9. „*Polish notation*“, 2006 [žiūrėta 2006-04-14]. Prieiga per internetą: http://en.wikipedia.org/wiki/Polish_notation.
10. „*Reverse Polish notation*“, 2006 [žiūrėta 2006-04-14]. Prieiga per internetą: http://en.wikipedia.org/wiki/Reverse_Polish_notation.
11. „*TPC-H*“, 2004 [žiūrėta 2006-04-20]. Prieiga per internetą: http://www.tpc.org/tpch/.

8. Data structures and processing algorithms of DBMS research and optimization

Summary

Most of the common DBMS (*ORACLE*, *Microsoft SQL Server*, *MySQL*, *DB2*, etc.) use disk storage as run time memory. When data becomes necessary from database, DBMS reads them from disk and loads it to RAM (Random Access Memory). This approach uses small amount of RAM, but DBMS efficiency is relative poor.

In this work was created prototype of DBMS. This prototype is read-only DBMS and holds all database data in RAM. The purpose of this work is to compare efficiency of common DBMS against created prototype.

For this prototype was developed and optimized bunch of data searching, sorting, grouping, joining algorithms. All of these algorithms are based on the main prototype data model idea: database table stores all data into table tree.

Prototype and other DBMS were tested with TPCH test, which consists of very different 22 SQL queries to test DBMS efficiency. The test result produced good results for prototype: prototype was 30-40 times faster against *ORACLE* on complex joining queries and 100-200 times faster against *ORACLE* on simple joining and searching queries.

9. Santrumpų ir terminų žodynas

DBVS – Duomenų bazių saldymo sistema (angl. *DBMS - DataBase Management System*)

STL – C++ programavimo kalbos šablonų biblioteka (angl. *Standard Template Library*)

SQL – Struktūrizuota užklausų kalba (angl. *Structured Query Language*)

ODBC – Duomenų perdavimo iš duomenų bazės tam tikrai programai standartas (angl. *Open DataBase Connectivity*)

ANTLR – Gramatinio nagrinėtojo pagal tam tikrą gramatiką generavimo įrankis (angl. *ANother Tool Language Recognition*)

TPCH – Duomenų bazių ir jų užklausų generatorius, skirtas pamatuoti bendrą DBVS efektyvumą (angl. *Transaction Processing Performance Council (ad-Hoc)*)

10. Priedai

10.1. TPCH testų SQL užklausos

1. Didžiausios lentelės duomenų grupavimas

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= '1998-09-03'
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

2. Duomenų apjungimas (5 lentelės) su parametrizuota vidine užklausa (4 lentelės, minimumo paieška)

```
select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
from
    parts,
    supplier,
    partsupp,
    nation,
    region
where
    p_partkey = ps_partkey
    and s_suppkey = ps_suppkey
    and p_size = 42
    and p_type like '%NICKEL'
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'MIDDLE EAST'
    and ps_supplycost =
        (
            select
                min(ps_supplycost)
            from
                partsupp,
                supplier,
                nation,
                region
            where
                p_partkey = ps_partkey
```

```
                and s_suppkey = ps_suppkey
                and s_nationkey = n_nationkey
                and n_regionkey = r_regionkey
                and r_name = 'MIDDLE EAST'
        )
order by
    s_acctbal desc,
    n_name,
    s_name,
    p_partkey;
```

3. Duomenų apjungimas (3 lentelės) be vidinių užklausų

```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = 'FURNITURE'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < '1995-03-27'
    and l_shipdate > '1995-03-27'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate;
```

4. Duomenų paieška su parametrizuota vidine užklausa (netuščios aibės tikrinimas)

```
select
    o_orderpriority,
    count(*) as order_count
from
    orders
where
    o_orderdate >= '1997-04-01'
    and o_orderdate < '1997-07-01'
    and exists
        (
            select
                *
            from
                lineitem
            where
                l_orderkey = o_orderkey
                and l_commitdate < l_receiptdate
        )
group by
    o_orderpriority
order by
    o_orderpriority;
```

5. Duomenų apjungimas (6 lentelės) be vidinių užklausų

```
select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    customer,
```

```
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'MIDDLE EAST'
    and o_orderdate >= '1995-01-01'
    and o_orderdate < '1996-01-01'
group by
    n_name
order by
    revenue desc;
```

6. Duomenų paieška be vidinių užklausų

```
select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_shipdate >= '1995-01-01'
    and l_shipdate < '1996-01-01'
    and l_discount between 0.02 - 0.01 and 0.02 + 0.01
    and l_quantity < 25;
```

7. Duomenų apjungimas (6 lentelės), kai *FROM* dalyje yra vidinė užklausa su *OR* logine sąlyga

```
select
    supp_nation,
    cust_nation,
    l_year,
    sum(volume) as revenue
from
    (
        select
            n1.n_name as supp_nation,
            n2.n_name as cust_nation,
            extract(year from l_shipdate) as l_year,
            l_extendedprice * (1 - l_discount) as volume
        from
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2
        where
            s_suppkey = l_suppkey
            and o_orderkey = l_orderkey
            and c_custkey = o_custkey
            and s_nationkey = n1.n_nationkey
            and c_nationkey = n2.n_nationkey
            and
            (
                (
                    n1.n_name = 'BRAZIL'
                    and n2.n_name = 'SAUDI ARABIA'
                )
                or
                (
                    n1.n_name = 'SAUDI ARABIA'
```

```
                    and n2.n_name = 'BRAZIL'
                )
            )
            and l_shipdate between '1995-01-01' and '1996-12-31'
    ) as shipping
group by
    supp_nation,
    cust_nation,
    l_year
order by
    supp_nation,
    cust_nation,
    l_year;
```

8. Duomenų apjungimas (6 lentelės), kai *FROM* dalyje yra vidinė užklausa

```
select
    nation,
    o_year,
    sum(amount) as sum_profit
from
    (
        select
            n_name as nation,
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
        from
            parts,
            supplier,
            lineitem,
            partsupp,
            orders,
            nation
        where
            s_suppkey = l_suppkey
            and ps_suppkey = l_suppkey
            and ps_partkey = l_partkey
            and p_partkey = l_partkey
            and o_orderkey = l_orderkey
            and s_nationkey = n_nationkey
            and p_name like '%grey%'
    ) as profit
group by
    nation,
    o_year
order by
    nation,
    o_year desc;
```

9. Duomenų apjungimas (4 lentelės) be vidinių užklausų

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= '1993-06-01'
    and o_orderdate < '1993-09-01'
```

```
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
order by
    revenue desc;
```

10. Duomenų apjungimas (3 lentelės) su parametrizuota vidine užklausa (3 lentelės, rezultatų sumavimas)

```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    partsupp,
    supplier,
    nation
where
    ps_suppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = 'UNITED KINGDOM'
group by
    ps_partkey
having
    sum(ps_supplycost * ps_availqty) >
        (
            select
                sum(ps_supplycost * ps_availqty) * 0.0000000
            from
                partsupp,
                supplier,
                nation
            where
                ps_suppkey = s_suppkey
                and s_nationkey = n_nationkey
                and n_name = 'UNITED KINGDOM'
        )
order by
    value desc;
```

11. Duomenų apjungimas (2 lentelės) su neparametrizuota vidine užklausa

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct ps_suppkey) as supplier_cnt
from
    partsupp,
    parts
where
    p_partkey = ps_partkey
    and p_brand <> 'Brand#43'
    and p_type not like 'MEDIUM BURNISHED%'
    and p_size in (12, 31, 21, 3, 18, 39, 42, 44)
    and ps_suppkey not in
        (
            select
                s_suppkey
            from
                supplier
            where
                s_comment like '%Customer%Complaints%'
```

```
        )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size;
```

12. Duomenų apjungimas (2 lentelės) su parametrizuota vidine užklausa

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    parts
where
    p_partkey = l_partkey
    and p_brand = 'Brand#32'
    and p_container = 'LG BOX'
    and l_quantity <
        (
            select
                0.2 * avg(l_quantity)
            from
                lineitem
            where
                l_partkey = p_partkey
        );
```

13. Duomenų apjungimas (3 lentelės) su neparametrizuota vidine užklausa

```
select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    customer,
    orders,
    lineitem
where
    o_orderkey in
        (
            select
                l_orderkey
            from
                lineitem
            group by
                l_orderkey
            having
                sum(l_quantity) > 313
        )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate;
```

14. Duomenų apjungimas (2 lentelės) su 3 loginėm *OR* sąlygom

```
select
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    lineitem,
    parts
where
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#24'
        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= 9
        and l_quantity <= 9 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#22'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        and l_quantity >= 20
        and l_quantity <= 20 + 10
        and p_size between 1 and 10
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#43'
        and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        and l_quantity >= 30
        and l_quantity <= 30 + 10
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    );
```

15. Duomenų apjungimas (2 lentelės), kai *FROM* dalyje yra neparametrizuota vidinė užklausa, kurioje atliekama neparametrizuota ir parametrizuota vidinės užklausos

```
select
    s_name,
    s_address
from
    supplier,
    nation
where
    s_suppkey in
        (
            select
                ps_suppkey
            from
                partsupp
            where
                ps_partkey in
                    (
                        select
                            p_partkey
                        from
                            parts
                        where
                            p_name like 'navy%'
                    )
                and ps_availqty >
                    (
                        select
```

```
                            0.5 * sum(l_quantity)
                        from
                            lineitem
                        where
                            l_partkey = ps_partkey
                            and l_suppkey = ps_suppkey
                            and l_shipdate >= '1994-01-01'
                            and l_shipdate < '1995-01-01'
                    )
            )
    and s_nationkey = n_nationkey
    and n_name = 'ETHIOPIA'
order by
    s_name;
```

16. Duomenų apjungimas (4 lentelės) su dviem parametrizuotom vidinėm užklausom

```
select
    s_name,
    count(*) as numwait
from
    supplier,
    lineitem l1,
    orders,
    nation
where
    s_suppkey = l1.l_suppkey
    and o_orderkey = l1.l_orderkey
    and o_orderstatus = 'F'
    and l1.l_receiptdate > l1.l_commitdate
    and exists
        (
            select
                *
            from
                lineitem l2
            where
                l2.l_orderkey = l1.l_orderkey
                and l2.l_suppkey <> l1.l_suppkey
        )
        and not exists
            (
                select
                    *
                from
                    lineitem l3
                where
                    l3.l_orderkey = l1.l_orderkey
                    and l3.l_suppkey <> l1.l_suppkey
                    and l3.l_receiptdate > l3.l_commitdate
            )
        and s_nationkey = n_nationkey
        and n_name = 'ETHIOPIA'
group by
    s_name
order by
    numwait desc,
    s_name;
```

17. Duomenų paieška, kai *FROM* dalyje yra neparametrizuota vidinė užklausa, kurioje atliekamos neparametrizuota ir parametrizuota vidinės užklausos

```
select
    cntrycode,
    count(*) as numcust,
    sum(c_acctbal) as totacctbal
from
    (
        select
```

```
            substring(c_phone from 1 for 2) as cntrycode,
            c_acctbal
        from
            customer
        where
            substring(c_phone from 1 for 2) in ('15', '28', '20', '16', '13', '27', '12')
            and c_acctbal >
                (
                    select
                        avg(c_acctbal)
                    from
                        customer
                    where
                        c_acctbal > 0.00
                        and substring(c_phone from 1 for 2) in ('15', '28', '20', '16',
'13', '27', '12')
                )
                and not exists
                    (
                        select
                            *
                        from
                            orders
                        where
                            o_custkey = c_custkey
                    )
    ) as custsale
group by
    cntrycode
order by
    cntrycode;
```

10.2. Straipsnis

DBVS duomenų struktūrų ir jų apdorojimo algoritmų tyrimas bei optimizavimas

Konferencijos „Informacinės technologijos 2006“ pranešimų leidinys

Mindaugas Radžius

Kauno Technologijos Universitetas, Verslo informatikos katedra

1. Įvadas

Daugumoje dabartinių programinių paketų duomenims saugoti ir apdoroti yra naudojamos reliacinės duomenų bazių valdymo sistemos (pvz.: *Microsoft SQL Server*, *MySQL*, *ORACLE*, *DB2* ir kt.). Visos šios duomenų bazių valdymo sistemos (DBVS) yra diskinės, t.y. duomenis saugo pastovioje atmintyje (diske), o kai reikia užsikrauna duomenis į operatyviąją atmintį. Toks duomenų laikymo ir apdorojimo būdas yra sąlyginai pigus lyginat su pastoviosios disko ir operatyviosios atminties kainų santykio. Pastovioji disko atmintis yra žymiai pigesnė už operatyviąją atmintį. Šis kainų santykis gali siekti 200 kartų ir dar daugiau. Laikyti reikiamus duomenų bazės duomenis operatyviojoje atmintyje yra santykinai brangu, tačiau tai suteikia daug didesnį DBVS darbo greitį. Šiuolaikinės pastoviosios disko ir operatyviosios atminties duomenų nuskaitymo greičių santykis gali siekti 100 kartų. Operatyviosios atminties didelis darbo greitis iš dalies kompensuoja operatyviosios atminties santykinai didelę kainą. Šiame darbe yra kuriamas DBVS prototipas, kuris duomenis darbo metu laiko tik operatyviojoje atmintyje. Lyginant pastoviosios disko ir operatyviosios atminčių darbų greičių santykį galima tikėtis, kad šio kuriamo prototipo duomenų apdorojimo greitis gali būti iki 100 karto didesnis. DBVS prototipas bus pritaikytas tik duomenų nuskaitymui. Duomenų modifikavimo funkcijos jis neturės. Taip maksimaliai galima koncentruotis į duomenų suspaudimo, paieškos ir kitus manipuliacijos duomenimis algoritmus maksimaliai išnaudojant atminčių greičių santykį.

DBVS prototipui yra pritaikomi, optimizuojami arba sukuriami nauji duomenų suspaudimo, apdorojimo, paieškos ir pan. algoritmai. DBVS kuriamas prototipas duomenis darbo metu laiko tik operatyvioje atmintyje. Kadangi operatyviosios atminties santykinai yra mažai, tai labai didelės įtakos turi duomenų suspaudimo algoritmas, nuo kurio priklauso ne vien tik užimamos atminties kiekis, bet ir duomenų nuskaitymo greitis.

Realiai tokių DBVS sistemų yra palyginti mažai. Jos yra skirtos duomenų analizei, kai duomenų kiekiai yra labai dideli. *Clearpace* kompanija yra sukūrusi *NPBase* sistemą [1], kuri turi kuriamo DBVS prototipo funkcionalumą. Šiame darbe atliekama analizė įvairių disko DBVS sistemų, kartu įtraukiant *NPBase* sistemą, palyginimui su kuriamu DBVS prototipu.

2. Duomenų suspaudimo, apdorojimo algoritmų analizė ir optimizavimas

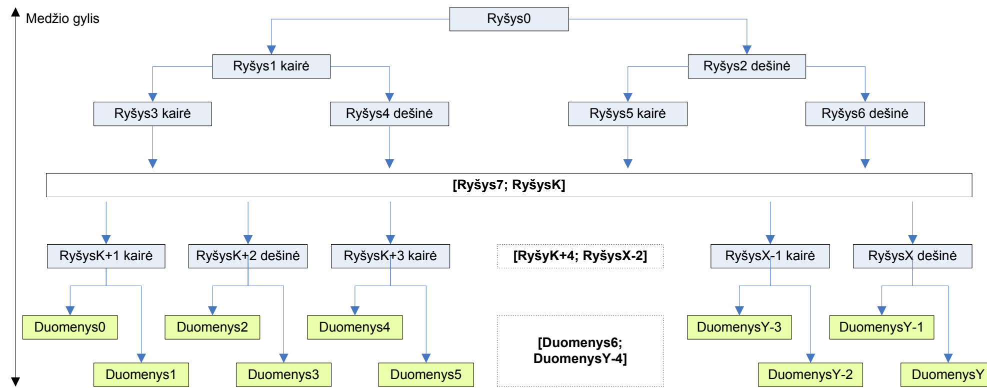
DBVS prototipo projektavimo tikslai yra šie: labai greita duomenų paieška, didelis duomenų suspaudimas, įvairūs manipuliavimo duomenimis algoritmai. Pagrindiniai faktoriai, lemiantys projektuojamą DBVS prototipą yra siejami su duomenų laikymu operatyvioje atmintyje, atminties fragmentacija, virtualios atminties įtaka DBVS darbui, algoritmų efektyvumu. Pastarasis faktorius labai glaudžiai yra susijęs su duomenų struktūromis. Taip galima atsiriboti nuo „lėtos“ disko atminties, kurios įtaka gali labai iškraipyti įvairių algoritmų veikimo spartą. Tačiau virtualios atminties įtaka turi būti įvertinta, nes kad ir kiek daug bus operatyviosios atminties kompiuteryje, bendru atveju jos visada gali pritrūkti.

Duomenų suspaudimo algoritmas turi pasižymėti labai dideliu duomenų išpakavimo greičiu. Įvairių suspaudimo programų RAR, ZIP ir pan. naudojami algoritmai yra daugiau orientuoti į suspaudimo laipsnį, nei į išpakavimo greitį.

DBVS per 1 sekundę gali būti įvykdoma šimtai užklausų, kai duomenų bazės dydis viršija 1GB, todėl tokie algoritmai nėra tinkami. Kadangi DBVS išskirtinai dirbs tik operatyviojoje atmintyje, todėl yra svarbus ir duomenų suspaudimo laipsnis. Taigi reikia laviruoti tarp pakankamo duomenų suspaudimo laipsnio ir duomenų išpakavimo greičio. Naudojamas suspaudimo algoritmo principas yra paprastas: išsaugojamos tik unikalios duomenų reikšmės, kurios nėra kaip nors specialiai suspaudžiamos (tai sutaupytų nemažai atminties jeigu būtų suspaudžiami eilutės tipo duomenys, nes jie dažniausiai užima apie 80% visos duomenų bazės užimamos vietos, tačiau išpakavimo laikas labai pailgėtų). Išsaugant tik unikalias duomenų reikšmes reikia jas tarpusavyje susieti. Taip pat reikia atsižvelgti į tai, kad būtų galima labai efektyviai vykdyti duomenų paiešką ir įvairias operacijas susijusias su duomenų manipuliavimu. Taigi vienos duomenų bazės lentelės duomenų priklausomybės bus saugojamos medžio struktūroje.

2.1. Medžio struktūra

Medis yra sudaromas iš dviejų tipų elementų: ryšių ir duomenų. Kiekvienas ryšio ar duomenų elementas atitinkamai turi savo numerį nuo 0 iki x ir nuo 0 iki y. Kad būtų aiškesni algoritmai susiję su medžio manipuliacija, visi ryšių elementai, išskyrus viršutinįjį, pagal savo poziciją grafinėje medžio imitacijoje yra arba kairieji, arba dešinieji. Toliau į kiekvieną medžio elementą bus kreipiamasi jo pavadinimu ir numeriu (pvz.: *Ryšys0*). 1 pav. pateikta bendra medžio struktūra, kai medis kiekviename lygyje turi po kairįjį ir dešinįjį ryšius.



1 pav. Bendra medžio struktūra

Šis medis yra simetrinis centro atžvilgiu. Bendru atveju medis yra asimetrinis, t.y. kairėje pusėje esančių ryšių skaičiui nėra lygus dešinėje pusėje esančiam ryšių skaičiui. Konstruojant medį vienas iš svarbiausių parametrų yra medžio gylis. Jis rodo kiek yra elementų lygių medyje kartu su duomenų elementais

Reliacinėje duomenų bazėje (toliau duomenų bazė) vienas medis atitinka vieną duomenų lentelę (toliau lentelė). Kiekvienas duomenų elementas atitinka kiekvieną lentelės duomenų stulpelį (toliau stulpelis). Ryšio elementai neturi sąsajų su duomenų bazės objektais.

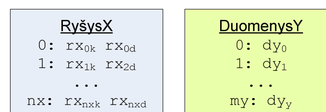
2.2. Medžio elementų struktūra

Kiekvienas medžio elementas susideda iš įrašų. Įrašų kiekis elemente yra skirtingas, bet bendru atveju gali ir sutapti.

Ryšio elementų įrašą sudaro pora skaičių: ryšiai į gilesnio lygio kairįjį ir dešinįjį elementų (ryšio arba duomenų) įrašus. Tie ryšiai yra nurodomi skaičiumi nuo 0 iki n. Kiekvienam medžio ryšio elementui n yra skirtingas (pvz.: penktas ryšio elementas turi n5 įrašus).

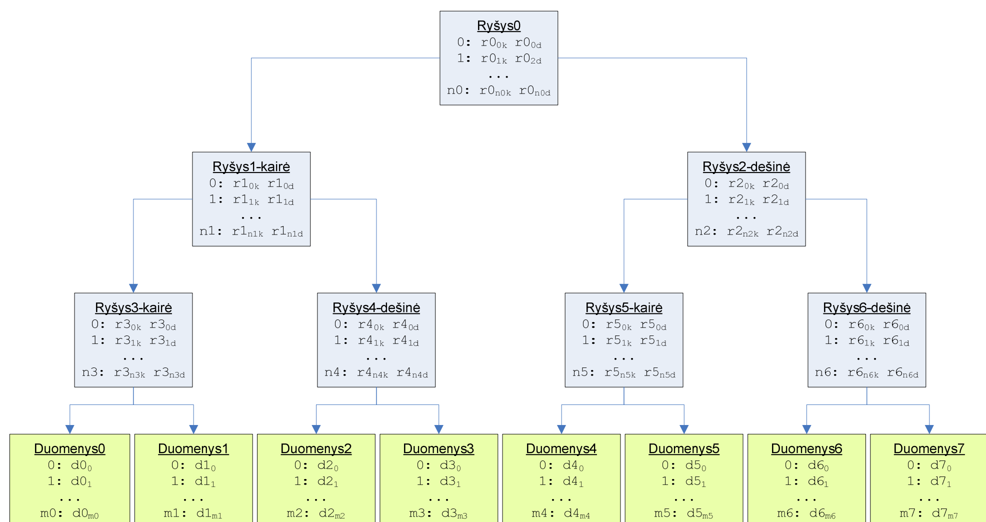
Duomenų elemento įrašus sudaro stulpelio duomenų **unikalios** reikšmės išrūšiuotos unikalumo didėjimo tvarka. Kadangi šiuose elementuose saugojamos tik unikalios reikšmės todėl pasiekiamas duomenų bazės suspaudimas.

2 pav. pavaizduotas ryšio ir duomenų elementai. Juose yra atitinkamai nx ir my įrašų. Ryšio elemente nulinio įrašo ryšys į gilesnio lygio kairįjį ir dešinįjį elementus įrašus yra atitinkamai rx_{0k} (nulinis, kairė) ir rx_{0d} (nulinis, dešinė). Duomenų elemente nulinis įrašas yra dy_0, t.y. unikali stulpelio reikšmė.



2 pav. Ryšio ir duomenų elementų struktūra

3 pav. pateikta konkreti medžio struktūra. Šis medis saugo lentelę, kuri turi 8 stulpelius (numeruojami nuo 0 iki 7 imtinai). Medis yra simetriškas.



3 pav. Medžio struktūros pavyzdys

2.3. Medžio konstravimas

Aprašant medžio konstravimo algoritmą, pirmiausiai reikia aptarti pagrindinius medžio konstravimo parametrus. Medžio konstravimo metu medis yra apdorojamas dalimis, t.y. iš vientisos medžio struktūros išskiriami tam tikri poaibiai. Poaibiai yra ne kas kita kaip stulpelių poaibis. Poaibiai yra vaizduojami pomedžiais (angl. *sub-tree*).

Prieš medžio konstravimą turi būti atliktas kiekvieno stulpelio skirtingų reikšmių (f, angl. *Frequency*) skaičiavimas. Šis procesas yra vadinamas dažnių lentelės konstravimu. Po to skaičiuojamas kiekvieno stulpelio reikšmingumas (m, angl. *Magnitude*). Jis skaičiuojamas pagal (1) formulę.

$$m_i = \begin{Bmatrix} 0, f_i \le 1 \\ \lfloor \log_{10}(f_i) \rfloor, f_i > 1 \end{Bmatrix}; \tag{1}$$

čia f_i – i - ojo stulpelio skirtingų reikšmių kiekis;
m_i – i - ojo stulpelio reikšmingumas.

Po to stulpeliai yra sugrupuojami pagal apskaičiuotus reikšmingumus, t.y. stulpeliai, turintys tą patį reikšmingumą, priklauso tai pačiai grupei. Medžio konstravimo metu stulpeliai iš tos pačios reikšmingumo grupės bus tame pačiame medžio lygyje. Kuo didesnis stulpelių reikšmingumas, tuo labiau reikia kelti juos į medžio viršų. Taip galima sutaupyti kelis ryšio elementus. Norint tai automatizuoti reikia išrikiuoti stulpelius pagal jų reikšmingumo mažėjimą ir pradėti medžio konstravimą nuo stulpelių, turinčių didžiausią reikšmingumą.

Konstravimo metu medis konstruojamas iš atskirų pomedžių, kurie atstovauja tam tikrą stulpelių poaibį. Pomedžiams yra skaičiuojami parametrai, susiję su stulpelių padėtimi pomedyje, t.y. medžio gylis (d, angl. *tree depth*), kairės ir dešinės pomedžio pusės pločius (wl, wr, angl. *width left/right*), nuo kurių priklauso kiek daug reikės ryšio elementų, norint apjungti visus stulpelius iš tos pačios reikšmingumo grupės, medžio lygį, kuriame pomedis prasideda ir požymį, kuris rodo, ar tai giliausia medžio vieta.

Jeigu yra pomedis i, kurio stulpelių aibė yra s_i, tai medžio gylį galima apskaičiuoti pagal (2) formulę.

$$d_i = \begin{Bmatrix} 0, s_i = 1 \\ \lfloor \log_2(s_i - 1) + 1 \rfloor, s_i > 1 \end{Bmatrix}; \tag{2}$$

čia s_i – pomedžio aibės dydis (stulpelių skaičius).

Dešinės ir kairės pomedžio pusės pločiai apskaičiuojami pagal (3) ir (4) formules atitinkamai.

$$wr_i = 2^{d_i} - s_i \tag{3}$$

$$wl_i = s_i - wr_i \tag{4}$$

čia s_i – pomedžio aibės dydis (stulpelių skaičius);
d_i – i - ojo pomedžio gylis.

(3) ir (4) formulės išrinktos taip, kad kairėje medžio pusėje visada būtų lyginis stulpelių skaičius išskyrus atvejį, kai yra tik vienas stulpelis. Jeigu stulpelių skaičius yra 2^{si}, tai visi stulpeliai turi pereiti į kairiąją pusę. Dešinėje pusėje yra stulpeliai atlikę nuo kairės pusės.

Suskaičiavus pomedžio (medžio) kairės ir dešinės pusės pločius, atliekamas duomenų elementų (stulpelių) apjungimas, naudojant ryšio elementus. Pirmiausiai apdorojami tie medžio poaibiai (pomedžiais), kurie turi mažiausiai unikalių reikšmių. Sujungus pomedžio vienos pusės duomenų elementus ryšio elementais, toliau sujungiami ryšio elementai su kitais ryšio elementais tol, kol galiausiai pasiekiama pomedžio viršūnė, kuri yra vienas ryšio elementas. Vėliau apdorojama dešinė pomedžio pusė. Sukonstravus vieną pomedį su tam tikru unikalių reikšmių kiekiu vėliau tęsiamas viso medžio kūrimas analogišku principu. Sukonstruoti pomedžiai yra naudojami pagal jų aukščiausią ryšio elementą.

3. Duomenų manipuliacijos algoritmai

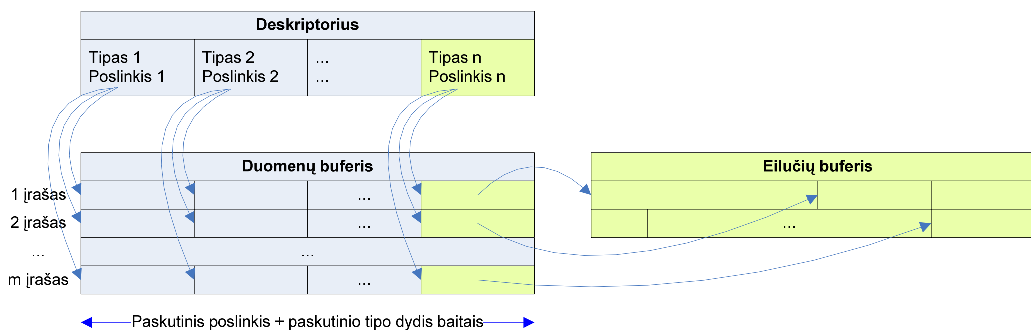
3.1. Kursoriai

Bendrąja prasme reliacinių duomenų bazių kontekste, kursorius yra užklausos rezultatai (įrašai). Šiame darbe kursorius turi kiek kitokią prasmę. Kursorius, tai toks objektas, kuris gali saugoti savyje bet kokį kiekį stulpelių ir įrašų. Stulpelių tipai gali skirtis. Kursorius, kuriant duomenų bazių valdymo sistemos prototipą, yra vienas iš pagrindinių sistemos elementų, tiesiogiai darantis įtaką sistemos darbo spartai. Jie bus naudojami duomenų surinkimui iš lentelės medžio, rūšiavimui bei grupavimui pagal tam tikrus kriterijus. Kadangi kursoriai tiesiogiai įtakoja sistemos darbo spartą, jie turi būti realizuoti pakankamai žemame lygyje. Kursorius turi savybę saugoti duomenis tiek operatyvioje, tiek pastovioje atmintyse. Tai yra būtina, jeigu reikia taupyti operatyviąją atmintį. Tai pat negalima visų duomenų laikyti pastovioje atmintyje, nes labai nukenčia duomenų nuskaitymo/įrašymo greitis. Naudojami kelių tipų kursoriai: bazinis kursorius, rūšiavimo kursorius, grupavimo kursorius, apjungimo kursorius. Bazinis kursorius yra visų kitų kursorių bazinė klasė, kuri tai pat gali būti naudojama kaip atskiras kursorius.

3.2. Vidinė kursoriaus struktūra

Visų kursorių tipų duomenų laikymo vidinė struktūra yra labai panaši. Kursoriuose naudojamus duomenų tipus reikia išskirti į dvi grupes: fiksuoto ilgio ir kintamo ilgio. Fiksuoto ilgio duomenų tipai - tai labai gerai žinomi įvairūs skaičių ir adresų tipai (*char* (1 baitas), *short* (2 baitai), *int* (4 baitai), *float* (6 baitai), *double* (8 baitai), *__int64* (8 baitai), *void* (4 baitai) ir pan.). Kintamo ilgio duomenų tipai yra tekstinės eilutės ir dvejetainiai masyvai (*char* *). Šiems tipams kiekvieno duomenų lemento saugojimui turi būti išsaugoma ne tik pati reikšmė, bet ir duomenų elemento kiekis baitais. Taigi kursoriaus duomenys yra dvejopai apdorojami nuo įvedamų duomenų tipų. Kiekvienas kursorius atmintyje turi tam tikro dydžio išskirtą vientisą buferį, į kurį yra rašomi duomenys. Jis vadinamas duomenų buferiu. Jeigu kursorius turi stulpelių, kurių tipai yra kintamo ilgio, tai papildomai dar yra naudojamas kitas buferis saugoti kintamo ilgio duomenų reikšmėms. Jis vadinamas eilučių buferiu. Kursoriaus konstravimo metu yra nurodami abiejų buferių maksimalūs dydžiai, kad iš anksto būtų paruošta vieta duomenims saugoti.

Kursoriaus saugomų duomenų tipų aprašymui yra naudojami deskriptoriai. Deskriptoriai, tai STL bibliotekos vektoriai, saugantys duomenų tipą, poslinkį nuo einamojo įrašo kursoriuje pradžios baitais ir dar papildomą tarnybinę informaciją. Deskriptoriaus pradžioje gali būti saugoma papildoma informacija, susijusi su įrašo pozicija kursoriuje (pvz. tikrasis įrašo numeris). Į kursoriaus duomenų buferį yra rašomi duomenys, kurių ilgis yra fiksuotas. Jeigu kursorius turi kintamo ilgio duomenis, tai į eilučių buferį yra rašomos viena paskui kitą eilutės arba dvejetainiai masyvai, o į duomenų buferį įrašomas fiksuoto ilgio (4 baitų) poslinkis eilučių buferyje (ne nuo einamojo įrašo pradžios, bet nuo eilučių buferio pradžios). Naudojant du buferius išvengiama duomenų atstatymo problemos, kai duomenis reikia atstatyti pagal tam tikrus įrašų numerius (ne nuosekliai). 4 pav. pateiktas kursoriaus deskriptorius ir kursoriaus duomenų ir eilučių buferiai. Rodyklės vaizduoja poslinkius baitais: duomenų buferyje nuo einamojo įrašo pradžios, eilučių buferyje nuo buferio pradžios. Melsva spalva simbolizuoja duomenų buferį, o žalsva - eilučių buferį.



4 pav. Pagrindiniai kursoriaus struktūros elementai

Kiekvieno įrašo įrašymo metu į kursorių yra tikrinama, ar duomenys tilps į kursoriaus buferius. Tam tikslui kursoriaus sukūrimo metu yra apibrėžiamas didžiausias eilutės ilgis. Jeigu duomenys netilps į kursoriaus buferius, tai buferiai yra užrašomi į failą, o buferiai atmintyje yra atlaisvinami. Duomenų dalys, kurios telpa buferiuose, o vėliau užrašomos į failus yra vadinamos blokais. Norint iš kursoriaus nuskaityti bet kurį įrašą pirmiausiai reikia išsiaiškinti, ar jis yra operatyviojoje atmintyje, ar faile. Šiam tikslui yra naudojamas blokų vektorius, kurio kiekvienas elementas saugo bloko paskutinio įrašo numerį ir poslinkį faile nuo jo pradžios baitais iki einamojo bloko pradžios. Kadangi įrašant duomenis į kursorių įrašų numeriai nuosekliai didėja, blokų vektorius automatiškai tampa išrūšiuotas pagal paskutinio įrašo numerį. Vėliau galima atlikti dvejetainę paiešką, ieškant kuriame bloke yra įrašas. Suradus bloką, jis yra nuskaitomas iš failo į operatyviąją atmintį ir atmintyje, panaudojant kursoriaus deskriptorių, nuskaitomi kiekvieno stulpelio duomenys. Faile bloką sudaro dvi dalys: duomenų buferis ir eilučių buferis. Bloko pradžioje įrašomas abiejų buferių dydis pridedant suminio buferių ir atskirų buferių dydžius baitas (po 4 baitus), po to saugomas duomenų buferio dydis baitais (4 baitai), pats duomenų buferis, eilučių buferio dydis baitais (4 baitai) ir pats eilučių buferis.

Aprašytas funkcionalumas galioja visų rūšių kursoriams. Jeigu duomenų nereikia rūšiuoti, grupuoti ar apjungti, tai šio funkcionalumo visiškai pakanka, kad būtų galima surinkti duomenis iš lentelės medžio. Toks funkcionalumas yra realizuotas baziniame kursoriuje, kurį paveldi rūšiavimo, grupavimo ir apjungimo kursoriai. Kursorių funkcionalumo pavyzdžiams bus naudojami SQL (angl. *Structured Query Language*) kalbos sakiniai [5].

3.3. Rūšiavimo kursorius

Rūšiavimo kursorius be visų bazinio kursoriaus savybių turi galimybę išrūšiuoti duomenis duomenų rašymo į kursorių metu. Rūšiavimo kriterijus yra nurodomas kursoriaus sukūrimo metu, kartu su stulpelių tipų nustatymu. Rūšiuoti galima didėjančiai arba mažėjančiai (rūšiavimo tipas) pagal stulpelius nurodant rūšiavimo prioritetą (pagal kurį stulpelį reikia išrūšiuoti pirmiausiai). Rūšiavimo tipą ir prioritetą galima saugoti viename skaičiuje. Skaičiaus ženklas rodo rūšiavimo tipą (<0 – rūšiuojama mažėjančiai, =0 – nerūšiuojama, >0 – rūšiuojama didėjančiai), o skaičiaus modulis – rūšiavimo prioritetą (kuo didesnis modulis, tuo mažesnis rūšiavimo prioritetas).

Tarkime reikia išrūšiuoti 1 lentelę pagal B stulpelį mažėjančiai, o pagal A didėjančiai. Simbolinis rūšiavimo kriterijus atrodys taip: B (-1) (pirmiausiai mažėjančiai), A (+2) (jei yra pasikartojančių B stulpelio reikšmių, tai didėjančiai pagal A).

Nustačius kursoriaus rūšiavimo kriterijus, duomenų rašymo į kursorių metu rūšiavimas nėra atliekamas. Rūšiavimas atliekamas operatyviojoje atmintyje, kai kursoriaus blokas užrašomas į failą arba kai pabaigiamas kursoriaus duomenų rašymas (buferiai yra tokio dydžio, kad nereikia naudoti failo buferių saugojimui). Bloko rašymo į failą metu yra išrūšiuojamas ne vien duomenų buferis, bet ir eilučių buferis, atnaujinant duomenų buferio poslinkius į eilučių buferį. Tai atliekama dėl to, nes kursoriaus duomenų įrašymo pabaigoje reikės apjungti visus išrūšiuotus blokus į vieną (nuskaitant duomenis iš failo, iš dviejų blokų nėra galimybės nuskaityti abu pilnus blokus, nes yra ribojamas kursoriaus buferių dydis). Blokų apjungimas yra būtinas, nes skaitymo metu visi kursoriuje esantys įrašai turi būti išrūšiuoti.

Atlikus rūšiavimą stulpelių kiekis kursoriuje gali sumažėti, jeigu kai kurie stulpeliai prieš rūšiavimą yra reikalingi tik duomenų išrūšiavimui, bet jų pačių saugoti nereikia.

Kursoriaus blokų rūšiavimas atliekamas greito rūšiavimo (angl. *quick sort*) algoritmu. Šis algoritmas, kaip ir daugelis kitų standartinių algoritmų yra realizuoti tam tikrose C++ bibliotekose. *qsort()* funkcija išrūšiuoja bet kokio dydžio duomenis, kai jiems yra aprašyta palyginimo funkcija. Ši funkcija pagal greito rūšiavimo algoritmą tiesiog kviečia su skirtingais duomenų įrašais palyginimo funkciją ir gauna -1 (pirmas įrašas mažesnis už antrąjį), 0 (įrašai lygūs, +1 (pirmas įrašas didesnis už antrąjį).

Greito rūšiavimo algoritmo vidutinis atliekamų veiksmų skaičius yra apskaičiuojamas pagal (5) formulę [4].

$$k = n \cdot \log_2 n \qquad (5)$$

čia k – vidutinis atliekamų veiksmų skaičius;
n – įrašų kiekis bloke (n = 1, 2, ..., N).

Atlikus blokų greitą rūšiavimą reikia blokus apjungti, kad duomenys būti visiškai išrūšiuoti. Blokų apjungimas atliekamas suliejimo algoritmu, kuris yra nesudėtingas. Apjungiant du blokus reikia imti įrašą iš pirmo ir jį lyginti su įrašu iš antro, tas kuris tenkina rūšiavimo kriterijų rašomas pirmasis ir taip kartojami veiksmus tol kol „pereinami" abiejų blokų įrašai. Suliejimo algoritmą galima vykdyti dviem būdais kai turim tam tikrą blokų kiekį. Sulieti gretimus blokus poromis arba sulieti visus blokus iš karto.

Pirmasis būdas reikalauja pakartotinio blokų suliejimo. Pvz.: jeigu yra 8 blokai, tai po pirmo gretimų blokų suliejimo gaunami 4, suliejus poromis keturis blokus gaunami 2 ir suliejus du blokus gaunamas pilnai išrūšiuotas kursorius. Tai buvo pasiekta per tris suliejimo etapus.

Antram suliejimo tipui reikia tik vieno suliejimo etapo, tačiau reikia daug palyginimo operacijų. Kiekvieną įrašą reikia palyginti su kiekvieno bloko pirmu įrašu ir tik tada nuspręsti, kurį įrašą užrašyti. Dėl mažesnio palyginimų kiekio bus naudojamas pirmasis suliejimo būdas.

Kyla klausimas, ar rūšiuojant blokus atskirai ir po to suliejant po du yra efektyvesnis algoritmas negu išrūšiuoti vieną didelį bloką turint labai didelį duomenų ir eilučių buferį. Ištikrųjų abu algoritmai yra labai artimi savo vidutiniu atliekamų veiksmų skaičiumi. Suliejant blokus poromis reikia atlikti $\log_2 m$ etapų, kai m blokų kiekis. Iš (6) formulių gauname, kad nesvarbu kuriame esame suliejimo etape, vidutinis veiksmų kiekis yra tas pats. Tai galima paaiškinti tuo, kad po vieno etapo suliejimo yra dvigubai mažiau blokų, bet nauji blokai yra dvigubai didesni.

$$\begin{aligned} l_1 &= \frac{m}{2} \cdot (n + n) = m \cdot n; \\ l_2 &= \frac{m}{4} \cdot (2n + 2n) = m \cdot n \\ &\ldots \\ l_i &= \frac{m}{2^i} \cdot (2^{i-1} n + 2^{i-1} n) = m \cdot n \end{aligned} \qquad (6)$$

čia l_i – i – ojo suliejimo etapo vidutinis veiksmų skaičius;
i – suliejimo etapo numeris (i = 1, 2, ..., N);
m – blokų kiekis ($m = 2, 4, \ldots, 2^N$);
n – bloko dydis (įrašų kiekis bloke).

Vidutinį veiksmų skaičių per visus suliejimo etapus galima apskaičiuoti pagal (7) formulę, kai blokai suliejami poromis.

$$k = m \cdot n \cdot \log_2 m \qquad (7)$$

čia k – bendras suliejimo etapų atliekamų veiksmų skaičius.

Bendrą viso algoritmo suliejimo etapais atliekamų veiksmų skaičių sudaro m kartų atliekamas greitas rūšiavimas kiekvienam blokui ir bendras suliejimo etapų atliekamų veiksmų skaičius, apskaičiuojamas pagal (7) formulę. Bendras viso algoritmo suliejimo etapais atliekamų veiksmų skaičius apskaičiuojamas pagal (8) formulę.

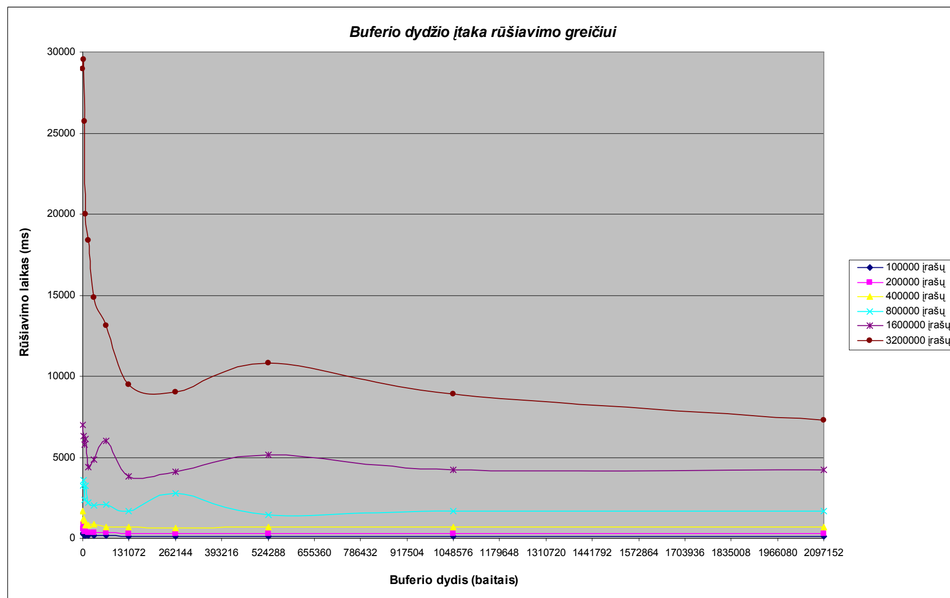
$$t = m \cdot n \cdot \log_2 n + m \cdot n \cdot \log_2 m \qquad (8)$$

čia t – bendras algoritmo atliekamų veiksmų skaičius.

Naudojant suliejimo etapais algoritmą sutaupoma operatyviosios atminties, bet prarandama laiko saugant ir nuskaitant išrūšiuotus blokus pastoviojoje atmintyje. Naudojant vieną didelį bloką sunaudojama labai daug atminties, bet nereikia lėtų rašymo į diską ir skaitymo iš disko operacijų. Bendru atveju bloko dydis gali būti didesnis už operatyviosios atminties dydį. Tokiu atveju operacinė sistema pradės naudoti virtualiąją atmintį, kas savo ruožtu labai stipriai įtakos sistemos efektyvumą. Taigi kursoriaus rūšiavimui yra naudojamas suliejimo etapais algoritmas.

Naudojant suliejimo etapais algoritmą reikia parinkti optimalų kursoriaus buferio dydį. Nuo to priklauso sunaudojamos operatyviosios atminties kiekis ir rūšiavimo greitis. Buvo atliktas testas, kurio metu

nustatytas optimalus kursoriaus buferio dydis. Testuojamas kursorius, kuriame yra tik vienas stulpelis. Stulpelio tipas yra sveikasis skaičius (*int*, 4 baitai). Buferio dydis keičiamas nuo 1024 (2^{10}) iki 2097152 (2^{21}) baitų dvigubinant ankstesnįjį buferio dydį, o įrašų kiekis nuo 100000 iki 3200000 dvigubinant ankstesnįjį įrašų kiekį. 5 pav. pateikiama diagrama, rodanti kursoriaus dydžio įtaką rūšiavimo greičiui. Iš diagramos nustatomas optimalus buferio dydis. Buferio dydžio intervalas yra nuo 262144 (2^{18}) iki 1048576 (2^{20}) baitų. Naudojant tokį buferio dydį sunaudojama santykinai mažai atminties, o rūšiavimo greitis yra labai artimas 2 MB ir didesnių buferių kursoriaus rūšiavimo greičiams.



5 pav. Buferio dydžio įtaka rūšiavimo greičiui

4. Išvados

Šiame darbe buvo sukurtas DBVS prototipas. Sistemos duomenų suspaudimo, apdorojimo, paieškos ir kito funkcionalumo algoritmai yra eksperimentiniai ir pritaikyti konkrečiam duomenų saugojimo modeliui. Gautas darbo išvadas galima suskirstyti į tam tikras grupes.

- Duomenų saugojimo modelis pagrįstas medžio struktūra. Kiekviena duomenų bazės lentelė turi savo atitikmenį – medį.
- Naudojant lentelės medį buvo sukurtos efektyvios duomenų paieškos struktūros (paieškos filtrai) ir jas apdorojantys algoritmai.
- Paieškos filtrai tiesiogiai naudoja medžio elementų indeksų reikšmes, taip maksimaliai sumažinantys paieškos laiką.
- Atlikti lentelės įrašų indeksų filtravimui naudojamas bitų filtras, kurio pagalba galima atlikti logines operacijas tarp lentelės įrašų indeksų.
- Naudojant ANTLR įrankį buvo sugeneruotas SQL užklausų gramatinis nagrinėtojas, kuris SQL užklausų paieškos, rūšiavimo, grupavimo kriterijus išskaido į hierarchinę struktūrą.
- Skaičiuojamosioms išraiškoms apdoroti naudojama atvirkštinė lenkiška forma.
- Paieškos kriterijų galima tikrinti dalimis (loginių operacijų *AND* grupėmis), taip minimizuojant tikrinamų sąlygų kiekį. Jei bent viena sąlyga grąžina neigiamą rezultatą, tai visos *AND* sąlygų grupės rezultatas taip pat bus neigiamas.
- Suskirstytos sąlygos į tam tikras „sudėtingumo“ grupes. Tai leidžia iš pradžių tikrinti, tas sąlygas, kurios maksimaliai gali apriboti rezultatų aibę.
- Duomenų rūšiavimui, grupavimui ir apjungimui atlikti sukurti kursoriai, kurie naudoja greito rūšiavimo algoritmą dalimis, taip apribojant naudojamos operatyviosios atminties kiekį.
- Duomenų apjungimo algoritmas buvo orientuotas į maksimaliai greitą duomenų apjungimą, bet ne į operatyviosios atminties taupymą.

- Duomenų apjungimo algoritmas yra suskaidytas į apjungimo sąlygų rūšiavimą, optimizavimą, tikrinimą (duomenų apjungimą) ir apdorotų sąlygų pašalinimą.
- Atlikti DBVS prototipo užklausų vykdymo testai parodė, kad sukurti ir optimizuoti algoritmai yra gana efektyvus (30-40 kartų greičiau atliekamos sudėtingos apjungimo užklausos), tačiau nepasiektas 100 kartų greitesnis užklausų vykdymo greitis lyginant su diskinėmis DBVS sistemomis (*ORACLE*, *Microsoft SQL Server*).
- Paprastos duomenų paieškos ir lentelių apjungimo užklausų vykdymo greitis DBVS prototipo buvo apie 100-200 kartų didesnis nei *ORACLE*. Tai visiškai paaiškinama, nes DBVS prototipas buvo projektuojamas būtent tokio tipo užklausoms.
- Dirbant su didesnėmis duomenų bazėmis DBVS prototipo pranašumas ima mažėti. Tai paaiškinama per dideliu operatyviosios atminties kiekio išnaudojimu (labai padidėja atminties fragmentacija, pradedama naudoti operacinės sistemos virtuali atmintis).
- Lyginat sukurto DBVS prototipo užklausų vykdymo greitį su analogiška DBVS sistema *NPBase*, kuri taip pat skirta tik duomenų skaitymui ir duomenys laikomi tik operatyviojoje atmintyje darbo metu, paaiškėjo, kad duomenų paieška yra vykdoma kelis kartus greičiau, bet duomenų grupavimas yra lėtesnis.

5. Literatūra

1. „*NPBase*“, 2006 [žiūrėta 2005-08-09]. Prieiga per internetą http://www.clearpace.com/products_npbase.htm.
2. „*Practical Compressor Test*“, 2004 [žiūrėta 2006-04-09]. Prieiga per internetą: http://www.elis.ugent.be/~wheirman/compression/.
3. *Nicolai M. Josuttis, „The C++ Standard Library, A Tutorial and Reference“, 1999, Addison Wesley*
4. *„Quicksort“,* 2006 [žiūrėta 2006-05-04]. Prieiga per internetą: http://en.wikipedia.org/wiki/Quicksort.
5. *American National Standard for Information Systems, „Database Language – SQL“, 1992, American National Standards Institute, Inc.*
6. „*ANTLR: A Predicated Parser Generator*“, 2003 [žiūrėta 2006-04-09] . Prieiga per internetą: http://www.antlr.org/article/1055550346383/antlr.pdf.
7. „*Open Database Connectivity*“ 2006 [žiūrėta 2006-04-02]. Prieiga per internetą http://en.wikipedia.org/wiki/Open_Database_Connectivity.
8. „*MS SQL Server 2000 SELECT statement*“, 2003 [žiūrėta 2006-04-13]. Prieiga per internetą: http://www.antlr.org/grammar/1062280680642/MS_SQL_SELECT.html.
9. „*Polish notation*“, 2006 [žiūrėta 2006-04-14]. Prieiga per internetą: http://en.wikipedia.org/wiki/Polish_notation.
10. „*Reverse Polish notation*“, 2006 [žiūrėta 2006-04-14]. Prieiga per internetą: http://en.wikipedia.org/wiki/Reverse_Polish_notation.
11. „*TPC-H*“, 2004 [žiūrėta 2006-04-20]. Prieiga per internetą: http://www.tpc.org/tpch/.

6. Data structures and processing algorithms of DBMS research and optimization

Summary

Most of the common DBMS (*ORACLE*, *Microsoft SQL Server*, *MySQL*, *DB2*, etc.) use disk storage as run time memory. When data becomes necessary from database, DBMS reads them from disk and loads it to RAM (Random Access Memory). This approach uses small amount of RAM, but DBMS efficiency is relative poor.

In this work was created prototype of DBMS. This prototype is read-only DBMS and holds all database data in RAM. The purpose of this work is to compare efficiency of common DBMS against created prototype.

For this prototype was developed and optimized bunch of data searching, sorting, grouping, joining algorithms. All of these algorithms are based on the main prototype data model idea: database table stores all data into table tree.

Prototype and other DBMS were tested with TPCH test, which consists of very different 22 SQL queries to test DBMS efficiency. The test result produced good results for prototype: prototype was 30-40 times faster against *ORACLE* on complex joining queries and 100-200 times faster against *ORACLE* on simple joining and searching queries.