TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Infotehnoloogia eriala

Ragnar Ziugand

Geoandmebaasid ja indeksid

Bakalaureusetöö (6 EAP)

Juhendaja: Sven Laur, PhD

Autor: ..... „.....“ august 2013

Juhendaja: ..... „.....“ august 2013

Lubada kaitsmisele

Professor: ..... „.....“ august 2013

Tartu 2013

Sisukord

Sissejuhatus

Autoril polnud enne ülikooli astumist teadmisi andmebaasidest ja isegi peale andmebaasidega seotud ainete läbimist jäi praktilises osas vajaka. Tööga seoses tuli lisaks andmebaasidele õppida ka geoinfosüsteeme. Teema valik tuligi praktilisest ja isiklikust huvist saada rohkem teadmisi andmebaasidest. Ühelt poolt üritabki autor saada uusi teadmisi ning täiendavat praktilist kogemust, aga teiselt poolt soovib mõista, milliseid tehnikad või häid tavasid peaks andmete indekseerimisel tähele panema.

Tänapäeva infotehnoloogilises maailmas on meie jaoks andmed olulise tähtsusega ja seda näitab ka varajaste infosüsteemide tekkimine. Iga infosüsteemi südameks on andmebaas, kus andmeid hoiustatakse. Kindlasti oleme kõik vahetult või kaudselt kokku puutunud mõne andmebaasiga. Mis need andmebaasid üldse on? Andmebaas on kõige lihtsama definitsiooni järgi struktureeritud ja korrastatud andmekogum. Tehnilisemalt väljendades on andmebaas struktuur, kuhu salvestatakse lõppkasutajate andmed ja metaandmed ehk andmed andmete kohta. Infosüsteemide kasutamisel märkasid inimesed, et oluline on salvestada ka geograafilisi andmeid ning seoses sellega pidid muutuma ka andmebaasid. Geograafilised andmed on tavapärastest andmetest erinevad nii struktuuri kui mahu poolest (Harjumaa metsaeraldiste Shape-fail on 30 MB, katastrite Shape-fail on 400 MB). Tänapäeval enamlevinud andmebaaside haldussüsteemid (PostgreSQL, MySQL, Oracle, MS SQL Server jt.) on enamasti kas relatsioonilised või objekt-relatsioonilised ning toetavad ka geograafilisi andmeid ning nendest kolme vaatame ka antud töös lähemalt.

Meile on oluline, et andmebaas tagastaks andmed võimalikult kiiresti ning selleks otstarbeks kasutataksegi indekseid. Sarnaselt raamatukoguga, kus raamatud on indekseeritud pealkirja, teema või autori järgi, kasutavad ka andmebaasid indekseid, mis võimaldavad andmeid kiiremini leida. Indekseid kasutades tuleb mõelda põhimõttele, et vähem on parem, sest nende ümberorganiseerimine on ressursimahukas ja indeksite vääriti kasutamisel võib-olla kiirusele vastupidine efekt.

Esialgne plaan andmebaaside ehitust uurida osutus liiga mahukaks ning juhendajaga koostöös sai bakalaureusetöö uuritavaks teemaks „Geoandmebaasid ja indeksid“. Töö raames uuritakse andmebaase, geoandmebaase, indekseid ning geoindekseid - viimaste eesmärki, tüüpe, kasutusvaldkonda ning efektiivsust. Selgus, et antud teemast leidub üsna vähe emakeelset materjali. Eelnev kehtib eriti indeksite ja seotud andmestruktuuride kohta ning see ajendas sellist materjali looma. Tööalaselt

on minult korduvalt küsitud, kuidas indekseid õigesti rakendada ning see andis põhjuse kasutada antud tööd ka õppematerjalina.

Bakalaureusetöö jaguneb kolmeks põhiosaks, kus esimene kirjeldab andmebaase ja indekseid, teine geoandmebaase ja geoindekseid ja kolmas keskendub indeksite rakendamisele läbi jõudluskatsete. Viimases üritatakse uurida kolme andmebaasi opereerimist kattuvuspäringute sooritamisel ja indeksite kasutust. Võrreldakse ka ajalist erinevust indeksiga ja indeksita päringute vahel, tehakse põhjalik analüüs ning võetakse tulemused kokku.

1 Andmebaasid

Andmebaas on kõige lihtsama definitsiooni järgi struktureeritud ja korrastatud andmekogum. Tehnilisemalt väljendandes on andmebaas struktuur, kuhu salvestatakse:

1) lõppkasutajate andmed;

2) metaandmed – andmed andmete kohta, mille kaudu lõppkasutajate andmed integreeritakse.

Andmebaas võib olla nii kollektsioon faile, mis on ühendatud mingi ühise tunnuse alusel kui ka eraldiseisva failina. DBMS (ingl *Database Management System*) ehk andmebaasi haldamise süsteem on programmide kogum, mis haldab andmebaasi struktuuri ja tagab ligipääsu andmetele, mida sinna salvestatakse. Haldussüsteem võimaldab inimesel lisaks andmete vaatamisele ka luua, muuta, uuendada ning salvestada andmeid andmebaasi.

1970ndatel lõi Edgar F. Codd, kes töötas tollal IBM-s, relatsioonilise mudeli. Mudeli aluseks on relatsioon. Lihtsustamiseks mõtleme, et relatsioon on tabel, mis koosneb ridadest ja veergudest. Iga rida on kirje korteež (ingl *tuple*) ja iga veerg on atribuut (ingl *attribute*). Tollase mudeli kohest kasutuselevõttu takistas fakt, et arvutid olid üsna aeglased. Tänu arvutite kiirele arengule ja relatsioonilise mudeli omadustele muutus see väga populaarseks ja kasutatakse tänapäevani. Lihtsusele aitab kaasa ka relatsioonilise andmebaasi haldussüsteem (ingl *RDBMS – Relational database management system*), mis võimaldab kasutajal lihtsasti hallata andmeid, ilma, et ta peaks teadma taustal toimuvast. Eelnevat toetas ka päringukeel SQL (ingl *Structured Query Language*), mis võimaldab kasutajal pärida lihtsasti andmeid relatsioonilisest andmebaasist. 80ndatel kasvas välja neljanda põlvkonna andmebaasimudel – objektorienteeritud andmemudel (ingl *Object-oriented data model*). Andmestruktuuris „objekt“ sisalduvad nii andmed kui ka andmetevahelised suhted. Atribuudid kirjeldavad objekti omadusi. Sarnased objektid grupeeritakse klassidesse. Klassides on defineeritud meetodid, mis võrdsustatakse protseduuridega. Klassid on organiseeritud klasside hierarhiana, kus igal klassil on vaid üks ülem. Päritavus on omadus, mis võimaldab objektil klassihierarhias pärida atribuudid ja meetodid ülemistest klassidest. Objektorienteeritud andmemudeli haldamiseks on loodud objektorienteeritud andmebaasi haldussüsteem (ingl *Object-oriented database management system*). Objektorienteeritud andmebaasid on loodud ka selleks, et toetada objektorienteeritud programmeerimiskeeli, sest mõlemad kasutavad sarnast ülesehitust. Antud töös keskendutakse samuti kahele eespool

kirjeldatud andmebaasimudelile, millest võib lähemalt lugeda raamatu Database systems: design, implementation, and management 1. peatükist. [CMR11]

1.1 Indeksid

Sarnaselt raamatukoguga, kus raamatud on indekseeritud pealkirja, teema või autori järgi, on ka andmebaasidel olemas omad indeksid, mille järgi on võimalik andmeid organiseerida selliselt, et neid paremini leida ja kasutada. Ka indekseid saame määratleda erinevate parameetrite alusel. Üheks variandiks on jaotada indeksid tüübi järgi kolmeks:

1) primaarindeks (ingl *primary index*) – indeks, mis luuakse tavaliselt automaatselt andmebaasihaldussüsteemi poolt peavõtmega veerule. Primaarindeks on unikaalne, sest igal tabelil võib-olla vaid üks peavõti. Primaarindeks koosneb kahest osast: peavõtmega veerust ja viidast mäluplokile, kus konkreetne väärtus asub. Antud indeksi i-s rida oleks siis kujul: <*väärtus, viit mäluplokile*>. Primaarindeksi hoiustamiseks kuluv andmemaht on palju väiksem kui tabeli enda oma, sest primaarindeksil on vähem kirjeid kui algandmetes ja indeks sisaldab kahte veergu, mistõttu mahub ühte mäluplokki rohkem andmeid;

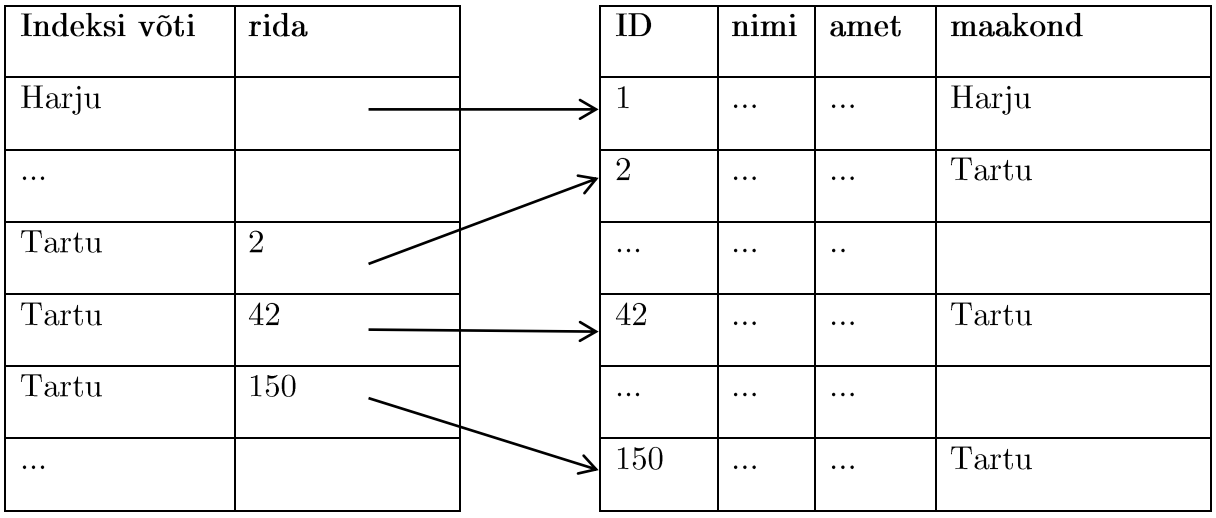
2) sekundaarindeks (ingl *secondary index*) – indeks, mida kasutatakse siis kui tabelis on mitu veergu, mille väärtused on unikaalsed. Üks neist on primaarindeks ja teised on sekundaarsed indeksid. Sekundaarindeksit võib luua veergudele, mille väärtused on kas järjestatud, järjestamata või räsitud. Sekundaarindeks koosneb sarnaselt primaarindeksile võtmeveerust, mis on järjestatud ja viidast mäluplokile või konkreetsele kirjele. Erinevalt primaarindeksist ja kobarindeksist pole oluline, kuidas on andmed tabeli veerus järjestatud, sest sekundaarindeksis leidub kirje iga tabeli kirje kohta;

3) kobarindeks (ingl *clustering index*) – kui kirjed tabelis on järjestatud mingi tabeli veeru järgi, mis ei ole unikaalne, siis sellist väärtust nimetatakse kobarväärtuseks ja sellele väärtusele loodud indeksit nimetatakse kobarindeksiks. Kobarindeks koosneb sarnaselt primaarsele ja sekundaarsele indeksile kahest veerust: kobarväärtusest ja viidast mäluplokile. Kobarindeksis on üks kirje iga unikaalse kirje kohta veerus ning iga viit viitab esimesele mäluplokile, kus antud võtmeväärtusega kirje asub.

Indekseid saab jaotada ka tihedateks (ingl *dense*) või hõredateks (ingl *sparse*). Tihedal indeksil on kirje iga tabeli rea kohta, samas kui hõredal indeksil on kirjed vaid osade kirjete kohta tabelis (reeglina ploki esimesele kirjele). Primaarindeks ja kobarindeks on hõredad indeksid, samas kui sekundaarindeks on tihe indeks.

Võimalik on kategoriseerida indekseid ka ühetasemelisteks ja mitmetasemelisteks. Ühetasemelise indeksi korral paiknevad kirjed ühel tasemel. Mitmetasemeliste indeksite korral jaotatakse indeks mitmeks plokiks, kus kõrgema taseme indeks viitab kirjele esimeses tasemes (baas) ning esimene tase omakorda viitab kirjele tabelis. Indeksite tüüpidest ja nende jaotamisest kategooriatesse saab lähemalt lugeda raamatu Fundamentals of Database Systems (6th ed.) 18. peatükist. [EN10]

Vaatleme alljärgneva näite abil sisend-väljundoperatsioonide arvu indekseeritud ja indekseerimata andmete vahel. Oletame, et meil on tabel TOOTAJA, kus on veerud *ID*, nimi, amet ja maakond. Soovime pärida kõiki töötajaid Tartu maakonnast. Sellisel juhul tuleks luua indeksid veergudele *ID* (peavõti) ja maakond. Tulem on lihtsustatud kujul näha joonisel 1.



Joonis 1. Näites toodud indeksi lihtsustatud struktuur.

Andmebaasist andmete pärimiseks kasutame päringut (*SELECT nimi, amet FROM tootaja WHERE maakond='Tartu'*), mis tagastab meile Tartus asuvate töötajate nimed ja ameti.

Tabelis TOOTAJA on 150 rida. Oletame, et tabel TOOTAJA on indeksita. Sellisel juhul tuleks läbida kogu ridade arv ehk 150 rida, samal ajal kui indeksiga tabelis andmebaasisüsteem teab, et maakonnaga Tartu on seotud kolm rida: rida 2, rida 42 ja rida 150. Andmebaasisüsteem teeb 3 pöördust indeksi poole ja 3 pöördust andmetele, seega kokku 6 sisend-väljundoperatsiooni. Ilma indeksita teeks andmebaasisüsteem 147 sisend-väljundoperatsiooni andmetele, mis ei vasta päringus soovitule.

Vaatleme praktilist näidet (indekseeritud ja indekseerimata andmete sisend- ja väljundoperatsioonide arvu võrdlus) andmebaasihaldussüsteemi PostgreSQL näitel, kus on loodud tabel TOOTAJA ning väljad vastavalt tabelile 1.

Välja nimi	**Andmetüüp**	**Kettaruum**
tootaja_ID (peavõti)	serial	4 baiti
eesnimi	character(50)	50 baiti
perekonnanimi	character(50)	50 baiti
amet	character(50)	50 baiti

Tabel 1. Tabeli „TOOTAJA“ struktuur.

Oletame, et meil on tabelis TOOTAJA 10 miljonit kirjet. Vastavalt eelnevale tabelile saime ühe kirje suuruseks 154 baiti ja ühe mäluploki maht PostgreSQL andmebaasisüsteemis on 8 kilobaiti ehk 8192 baiti[1]. Seega ühte plokki jääks 8192/154 ≈ 53 kirjet ja kokku läheb andmete hoidmiseks vaja 10 000 000/53 ≈ 188 679 plokki. Teeme lineaarse otsingu peavõtme veerule. Peame läbi vaatama 188 679/2 ≈ 94 340 plokki (keskmine juht), et leida konkreetset väärtust või halvimal juhul kõik 188 679 mäluplokki. Kuna peavõtme veerg on järjestatud (andmetüüp *serial*), siis saame otsida konkreetset väärtust läbides vaid $\log_2 94340 \approx 17$ plokki. Indeksi kirjed sisaldavad ainult indekseeritud välja ja viita originaalkirjele, mistõttu läbivaadatavate plokkide arv on kordades väiksem. Vaatame siinkohal teist näidet, kus meil pole väärtused järjestatud. Tabelis 2 on toodud vaadeldava näite tabelistruktuur.

Välja nimi	**Andmetüüp**	**Kettaruum**
amet	character(50)	50 baiti
viit kirjele	oid	4 baiti

Tabel 2. Näites kasutatud lihtsustatud tabelistruktuur.

Oletame, et meil on antud tabelis 10 miljonit kirjet. Ühe kirje maht on 54 baiti ja ühe ploki maht on 8192 baiti. Ühte plokki jääks 8192/54 ≈152 kirjet ja kokku on vaja andmete hoiustamiseks 10 000 000/152 ≈ 65 789 plokki. Kuna meil on eesnime väljale indeks loodud, siis saame otsingut kasutades info kätte keskmiselt $\log_2 65789$ ≤ 17 plokki läbides. Leidmaks kirje tegelikku aadressi, tuleks lugeda üks plokk lisaks ehk kokku 18 plokki. Indekseerimata tabelis tuleks vastava kirje kättesaamiseks läbida halvimal juhul plokkide koguarv ehk 65 789 plokki, siis indekseeritud tabelis tuleks halvimal juhul lugeda 18 plokki ehk läbivaatusi on ligikaudu 3655 korda vähem.

[1] Iga tabel ja indeks hoiustatakse lehtede massiivis (vaikimisi fikseeritud suurusega 8 kB, muudetav).

Indeksid kiirendavad andmete kättesaadavuse kiirust, aga sellegipoolest ei tohiks indeksite kasutamisega liiale minna, sest nende loomine võtab süsteemi ressurssi ning kui uus kirje sisestatakse, uuendatakse või kustutatakse tabelist, siis ka indeks tuleb ümber organiseerida ja see võib väga suurte andmetabelite korral märkimisväärselt aega võtta. Üldiselt võib öelda, et soovitatav on indekseerida veerud, mida kasutatakse *WHERE, DISTINCT, GROUP BY* ja *ORDER BY* lausetes.

Enamik tänapäevastest andmebaasihaldussüsteemidest kasutab mõnda alljärgnevatest indeksitest: B-puu, B^+-puu, paiskindeks, bitiindeks. Vaatleme neid indeksitüüpe lähemalt, aga enne defineerime tähistused, mida kasutame edasistes peatükkides:

1) X on vaadeldud näites defineeritud objektide arv;

2) Y on vaadeldud probleemis defineeritud objektide arv;

3) Z on maksimaalne objektide arv, mis mällu mahub;

4) W on maksimaalne objektide arv ühes mäluplokis;

5) h on puude kõrgus;

6) s on kirje suurus;

7) d on kirjete arv, kus kirjed on kujul $\text{Kirje}_1 ... \text{Kirje}_d$;

8) m on minimaalne kirjete arv puus;

9) M on maksimaalne kirjete arv puus.

1.1.1 B-puu

B-puu (ingl *B-tree*) on andmestruktuur, mis võimaldab andmeid hoiustada ning neid pärida ja hallata, kasutades tasakaalustatud puustruktuuri (vt. joonis 2).

B-puu on m-ndat järku otsingupuu, kus:

1) juurel on vähemalt kaks alamat ja maksimaalselt m alamat;

2) alternatiivselt võib puu sisaldada vaid juurt;

3) igal tipul, v.a juur ja lehed, on $\frac{m}{2}$ kuni m alamat;

4) kõik lehed on samal tasemel;

5) teekond juurest leheni on alati võrdne;

6) igal sisemisel tipul on kuni m - 1 võtmeväärtust ja kuni m viita alamateni;

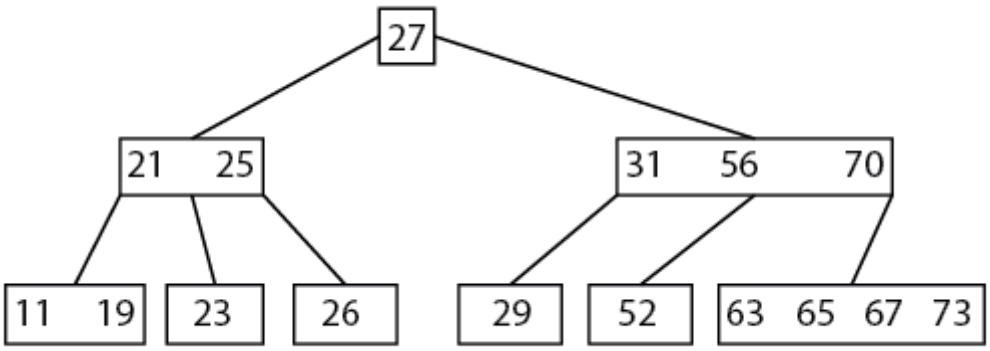
7) võib hoiustada andmeid ja viitu nii lehtedes kui teistes tippudes.

Lähemalt B-puudest saab lugeda raamatutest Fundamentals of Database Systems (6th ed.) ja Introduction to algorithms (3rd ed.), mõlemas peatükk 18. [EN10] [CLRS09]

B-puu funktsioonide keerukused on toodud alljärgnevas tabelis 3.

Funktsioon	**Keerukus**
Tühja B-puu loomine	$O(1)$
Mäluplokkide arv	$O(X/W)$
B-puu otsing (vahemik)	$O(\log_W X + Y/W)$
Kirje lisamine	$O(\log_W X)$
Kirje kustutamine	$O(\log_W X)$

Tabel 3. B-puu funktsioonide keerukused vastavalt raamatule Introduction to algorithms. [CLRS09]



Joonis 2. B-puu struktuur.

Kui võrdleme B-puud tema variatsiooni B$^+$-puuga ning indekseerime sama andmekomplekti mõlema andmestruktuuriga, siis B$^+$-puu võib kasutada B-puust vähem tippe, sest võtmeväärtused pole duplitseeritud. Seoses sellega kasutab B$^+$-puu vähem mäluplokke ja on kompaktsem, samas andmete pärimine on aeglasem. Puude kõrguseid vaadeldes on B-puu samalaadsest B$^+$-puust kõrgem.

1.1.2 B$^+$-puu

B$^+$-puu (ingl *B$^+$-tree*) – variatsioon B-puu struktuurist (vt. joonis 3), kus:

1) juurel on vähemalt kaks alamat ja maksimaalselt m alamat;

2) võtmed ja viidad hoiustatakse sisemistes tippudes (tipud, mis pole lehed);

3) andmed hoiustatakse lehtedes;

4) kõik lehed on samal tasemel;

5) lehed on omavahel ühendatud;

6) igal sisemisel tipul või lehel võib olla vähemalt d võtit;

7) igal sisemisel tipul või lehel võib-olla kuni $2d$ võtit;

8) igal sisemisel tipul võib-olla kuni $2m+1$ viita;

9) igal sisemisel tipul peab olema 1 viit rohkem kui võtmete arv;

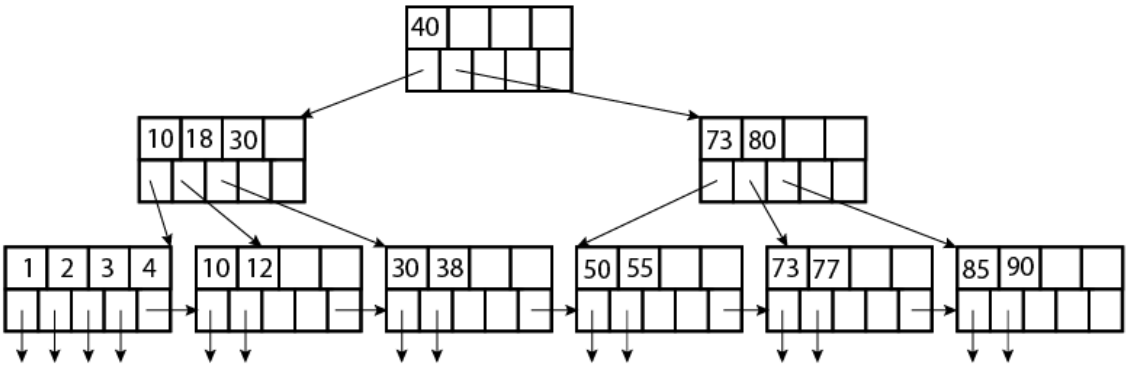
10) igal lehel peab olema sama palju andmeviitu kui on võtmeid;

11) võtmete väärtusi võib duplitseerida.

Lisaks allub B$^+$-puu omadusele, et iga tipu täituvus on minimaalselt 50% ja praktikas 67%.

Selline muudatus struktuuris võimaldab hoida puu kõrguse üsna madala, aga samal ajal kasutab rohkem mäluplokke, sest lubatud on duplitseeritud võtmeväärtused. B$^+$-puu on väga levinud indeksitüüp andmebaasides, sest võimaldab mõistliku ressursikasutusega andmeid pärida ja on isegi väga mahukate andmetega efektiivne. B$^+$-puu indeksit kasutatakse autorile tuntud andmebaasides PostgreSQL, Oracle 11g ja Microsoft SQL Server. Keerukused on üsna analoogsed B-puuga ja siinkohal uuesti välja ei tooda.

Lihtsustatud ülevaate B$^+$-puudest saab raamatu Fundamentals of Database Systems (6th ed.) 18. peatükist ja Spatial databases with application to GIS 6. peatükist. [EN10] [RSV02]

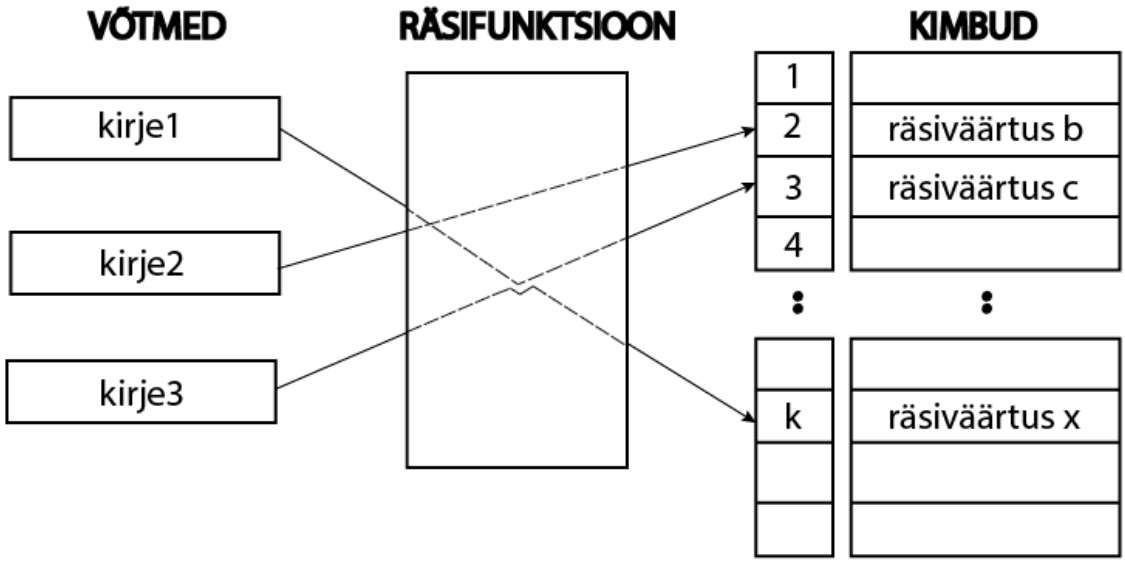


Joonis 3. B$^+$-puu struktuur.

1.1.3 Paiskindeksid

Paisktabeleid (ingl *hash tables*) võib kasutada ka indeksite loomiseks – selliseid indekseid nimetatakse paiskindeksiteks (ingl *hash indexes*). Paisktabelis ei pea andmeid järjestikku kirjutama, need kirjutatakse suvaliselt etteantud mäluploki piires. Mäluploki aadress arvutatakse räsifunktsiooni abil ning andmed räsitakse kimpudesse (vt. joonis 4). Kimp (ingl *bucket*) on andmestruktuur, mis mahutab 1-*n* kirjet ja mis võib olla kuni mäluploki suurune. Räsifunktsioonide probleem seisneb selles, et nad ei garanteeri unikaalseid aadresse, sest andmeid võib olla rohkem kui mäluplokke ja võib esineda olukord, kus kimpudes on erinev arv kirjeid. Probleemiks on ka põrked, st sama aadress on arvutatud mitme kirje jaoks. Ideaalne räsifunktsioon paigutab kirjed ühtlaselt nii, et igas kimbus oleks sama arv väärtuseid ja genereerib aadressid, mis ei korduks. Põrkeid aitab vältida lahtine adresseerimine (otsitakse vaba pesa kogu aadressiruumi piires ja sisestatakse väärtus sinna) ja dünaamiline paiskamine (aadressiruum, kuhu kirjeid salvestatakse, muutub vastavalt sisule). Paiskindeksis räsitakse andmed kimpudesse kujul <*väärtus, viit mäluplokile*>. Paiskindeksid võimaldavad kiiresti andmeid pärida, parimal juhul on otsingu keerukus *O(1)*, halvimal juhul *O(X)*. Keskmine elemendi lisamise ja kustutamise keerukus on *O(1)* ja halvimal juhul *O(X)*.

Täiendavat informatsiooni paiskindeksitest saab alljärgnevatest allikatest: Database systems: a practical approach to design, implementation, and management. 4. ed lisast C, Härmel Nestra loengumaterjalidest paisktabelite kohta ning raamatu Introduction to algorithms 11. peatükist. [CC05] [Nes08] [CLRS09]



Joonis 4. Paiskindeksi struktuur.

1.1.4 Bitiindeksid

Bitiindeks (ingl *bitmap index*) – andmestruktuur, mis hõlbustab päringuid andmetele, indekseerides konkreetses veerus olevad väärtused bitivektoritena, kus „1“ tähendab, et konkreetses reas väärtus eksisteerib ja „0“, kui väärtust ei eksisteeri. Bitiindeks sobib veergudele, kus on suhteliselt vähe unikaalseid väärtuseid ning tabelitele, kus toimub pidev lugemine. Bitiindeksite kohta saab lähemalt lugeda raamatu Database systems: a practical approach to design, implementation, and management. (4. ed) lisast C ja raamatu Fundamentals of Database Systems (6th ed.) 18. peatükist. [CC05] [EN10]

Vaatame lähemalt ühte näidet bitiindeksi kohta. Olgu meil tabel TOOTAJA, kus on veerud ID, nimi, amet, ruum ja maakond (vt. tabel 4).

ID	**nimi**	**amet**	**ruum**	**maakond**
1	Inimese Nimi	tisler	B222	Tartu
2	Inimese Nimi2	korstnapühkija	B333	Põlva
3	Inimese Nimi3	sekretär	B123	Tartu
4	Inimese Nimi4	mehhaanik	B222	Tartu

Tabel 4. Tabeli „TOOTAJA“ struktuur.

Kui luua indeks veerule „ruum“, siis saame alljärgnevas tabelis toodud indeksistruktuuri.

B123	**B222**	**B333**
0	1	0
0	0	1
1	0	0
0	1	0

Tabel 5. Näide bitiindeksist, mis on loodud tabelis 4 esinevale veerule „ruum“.

Võrreldes B^+-puu indeksiga on bitiindeksil mitmeid eeliseid:

1) kasutab vähem mäluplokke, sest on kompaktsemal kujul;

2) mitme bitiindeksi vahel saab teha operatsioone (*WHERE* lauses operaatorid *AND* ja *OR*).

Bitiindeksil on ka negatiivseid külgi:

1) indeksi loomisel peab arvestama, et see ei sobi tabelitele, kus andmeid sisestatakse, uuendatakse või kustutatakse, sest indeksi uuendamine võib kulukas olla;

2) mida rohkem unikaalseid väärtuseid, seda aeglasemaks indeks muutub.

Vaatleme, mis konteksti eespool kirjeldatud indeksid sobivad.

B-puu struktuuri kasutavad indeksid sobivad andmete päringuks, kus *WHERE* lauseosas kasutatakse võrduseid kui ka vahemikke.

N: *SELECT * FROM tootaja WHERE id = 1 OR id = 8;*

N: *SELECT * FROM tootaja WHERE id < 10;*

Paiskindeksid sobivad andmetele, kus *WHERE* lauseosas kasutatakse võrduseid. Bitiindeksid sobivad andmetele, mis on *boolean* tüüpi (1/0, *true/false*, väärtus1/väärtus2).

2 Geoandmebaasid, geoindeksid ja geoandmed

Peatükis 1 käsitleti andmebaase ning andmebaasihaldussüsteeme. Selles peatükis pöörame tähelepanu geoandmetüüpidele ning vaatleme geoandmebaase.

Geoandmebaasides on põhiliseks andmetüübiks geomeetria (ingl *geometry*), mis jaguneb neljaks: punkt (ingl *point*), kõver (ingl *curve*), pind (ingl *surface*), geomeetriakogu (ingl *geometry collection*). Andmetüüp „geomeetria“ on seotud ruumilise referentssüsteemiga, mis määrab koordinaatide lähtepunkti, koordinaattelgede orientatsiooni ja daatumi (geodeetilised parameetrid).

Vaatleme kirjeldatud andmetüüpide kasutusvaldkonda. Punkt on nulldimensionaalne objekt, mis sobib näiteks linnade kirjeldamiseks. Kõver (alamobjektiks joonobjekt) on ühedimensionaalne objekt, mis sobib näiteks jõgede defineerimiseks. Pind (alamobjektiks polügoonid) on kahedimensionaalne objekt kirjeldamaks näiteks riike. Geomeetriakoguna saab defineerida keerukamad ruumiandmed, mille alamobjektideks on punktihulgad ja areaalid.

Eelnevatele lisaks eksisteerib andmetüüp geograafia (ingl *geography*), mis sobib andmete salvestamiseks koos pikkus- ja laiuskraadidega.

Geoandmebaasidest ja andmetüüpidest saab lähemalt lugeda raamatu Spatial databases: a tour 2. peatükist. [SC03]

Ruumiandmed on andmed, mis viitavad konkreetsele geograafilisele asukohale defineerides ka objektide informatsiooni. Geoandmebaas on lihtsaima definitsiooni järgi andmebaas, mis võimaldab kasutajatel ruumiandmeid tsentraalselt kasutada, salvestada, muuta, hallata ning võimaldab samaaegselt paljude kasutajate ligipääsu.

Autori kogemuse järgi sobivad geoandmebaasidena:

1) PostgreSQL – vaikimisi ei toeta PostgreSQL geomeetriatüüpe, kuid need saab lisada tarkvarapaketiga PostGIS;

2) Oracle – ruumiandmete haldamiseks pakub Oracle tasuta Oracle Locatorit ning tasulist Oracle Spatialit. Viimane võimaldab kasutajal kasutada ka keerulisemaid funktsioone, nt kolmemõõtmeliste objektide haldamist;

3) Microsoft SQL Server – alates versioonist 2008 lisati andmetüübid geomeetria ja geograafia;

4) ESRI – pakub nii kasutajapõhiseid andmebaase (Personal GeoDatabase, ESRI GeoDatabase) kui ka ArcSDE laiendit tuntumatele andmebaasidele, nende seas Oracle, SQL Server ja PostgreSQL;

5) MySQL - toetab geomeetria tüüpi alates versioonist 5.0.16. [MyS]

Peatükis 1 käsitleti andmebaaside indekseid, järgnevas peatükis käsitleme geoindekseid, mida kasutatakse selleks, et geoandmeid hõlpsasti leida ja kasutada. Geoandmebaasides hoiustatavad andmed on reeglina üsna mahukad ning ka nendega tehtavad operatsioonid on keerukad (kattuvuspäring, lõikumine, sümmeetriline vahe jt.). Indeksita andmetega opereerimine on ressursimahukas nii protsessoriaja kui ka sisend-väljundoperatsioonide poolest.

2.1 Geoindeksite tüübid

Geoindeksid jagunevad struktuuri poolest kaheks:

1) ruumipõhised struktuurid – andmed paigutatakse mingite kriteeriumite alusel ristkülikukujulistesse (üldjuhul) elementidesse kahedimensionaalsel tasandil;

2) andmepõhised struktuurid – andmed organiseeritakse objektide kogumikuna, kus jaotamiseks kasutatav algoritm arvestab objektide paiknemisega ruumis.

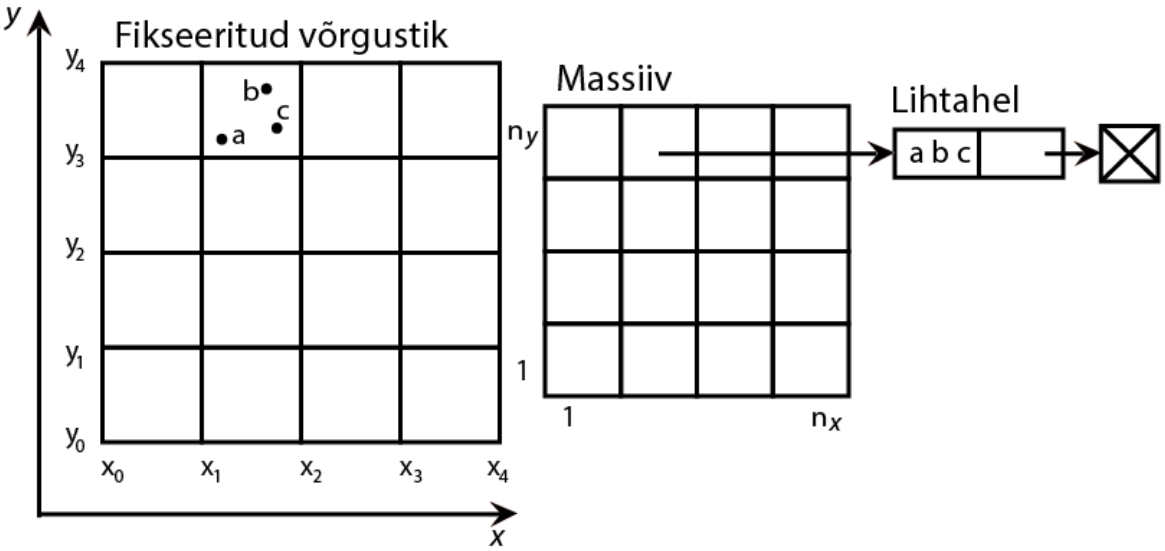
Ruumipõhiste struktuuride hulka kuuluvad erinevate pinnakujudega võrgustikud[2] ja lineaarsed struktuurid ning andmepõhiste struktuuride hulka erinevad puustruktuurid (näiteks R-puu). Rohkem geoindeksite tüüpidest ja geoindeksitest saab lugeda raamatu Spatial databases with application to GIS 6. peatükist. [RSV02]

2.2 Geoindeksid

Peatükis kirjeldatakse geoindeksite struktuure detailsemalt. Vaatleme kõigepealt fikseeritud võrgustikku (ingl *fixed grid*), seejärel adaptiivset võrku ning ruutpuud. Peatüki lõpuosas vaadeldakse R-puud ja tema variatsioone ning antakse ülevaade GiST indeksist.

2.2.1 Fikseeritud võrgustik

Fikseeritud võrgustikus on kogu pind jaotatud üldjuhul kas ruudu- või ristkülikujulisteks elementideks ja struktuur ise moodustab n-dimensionaalse massiivi. Ühes elemendis paiknevad objektid võib hoiustada lihtahelana (ingl *singly linked list*), et vältida nn ületäituvust, kus lahtrisse mittemahtuv objekt paigutatakse konkreetse elemendiga seotud ületäituvuse mäluplokki. Jooniselt 5 on näha, et võrgustikule loodud indeks moodustab kahemõõtmelise massiivi mõõtmetega $n_x \, x \, n_y$. Võrgustikus paiknevad 3 punkti hoiustatakse lihtahelana indeksi massiivis. Igal objektil massiivis on omistatud mäluploki aadress, kus punkt tegelikult asub.



Joonis 5. Näide fikseeritud võrgustikust ja seotud massiivist.

[2] Võrkude all pean silmas seda, et võrku moodustavad elemendid võivad olla mitmesuguse kujuga: kolmnurk, ruut või mõni teine n-küljeline kujund.

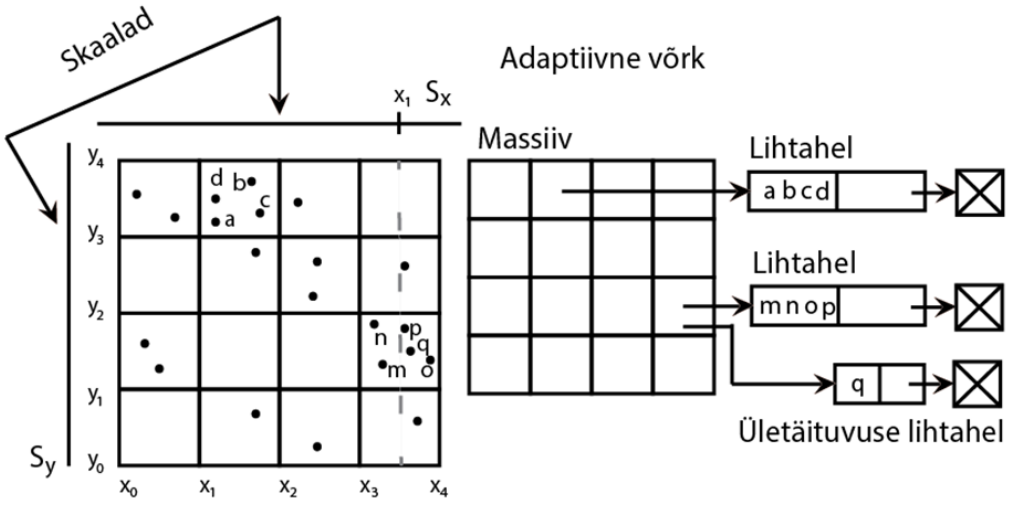
Vaatleme kolme alljärgnevat operatsiooni: punkti sisestamine, punkti pärimine ja aknaga kattuvuspäring.

Punkti sisestamisel sisestame punkti $P(a,b)$, arvutame i ja j, kus i ja j on lahtri koordinaadid, loeme massiivist vastava lehe ja sisestame P. Punkti otsimisel, kui argumendiks on punkt $P(a,b)$, loeme massiivist lehe ning kontrollime lehel olevaid punkte, et teada saada kas nende seas on P. Aknaga kattuvuspäringu rakendamisel vastavalt defineeritud aknale, arvutatakse tema alla jäävad lahtrid ning loetakse iga lahtri lehekülg ning tagastatakse kõik punktid, mis nendesse lahtritesse jäid.

Vaatleme lähemalt fikseeritud võrgustiku operatsioonide keerukusi. Punkti sisestamine on keerukusega *O(1)*. Punkti otsingu keskmine keerukus on *O(X/Z)* ja halvimal juhul *O(X)*. Aknaga kattuvuspäringu keerukus sõltub valitud akna suurusest. Võrgustiku resolutsioon sõltub indekseeritavate punktide arvust, mida tähistame tähega X. Kui mäluploki mahutavus on Z punkti, siis saame luua fikseeritud võrgustiku X/Z elemendiga. Iga võrgu element sisaldab keskmiselt vähem kui X objekti. Element, mis sisaldab rohkem kui X punkti, täitub üle. Ületäituvuse mäluplokk seotakse algse mäluplokiga, kus võrgu elemendis sisalduvaid punkte hoiustati. Kui punktid on jagatud võrgustikus ühtlaselt, siis ületäituvust ei esine ja ületäituvuse jaoks vajaminevaid välisahelaid on vähe. Halvimal juhul langevad kõik punktid sisestamisel ühte võrgu elementi ja struktuur koosneb lineaarsetest lehekülgedest. Päringud oleksid lineaarse keerukusega ja struktuuri efektiivsus kaoks. Pidevad lahtritesse sisestamised võivad tekitada suurel hulgal pikki välisahelaid ja kustutamine võib jätta elemendid tühjaks.

2.2.2 Adaptiivne võrk

Eelmises peatükis kirjeldatud probleemi, kus struktuur koosneb vaid lineaarsetest lehekülgedest üritab lahendada adaptiivne võrk.



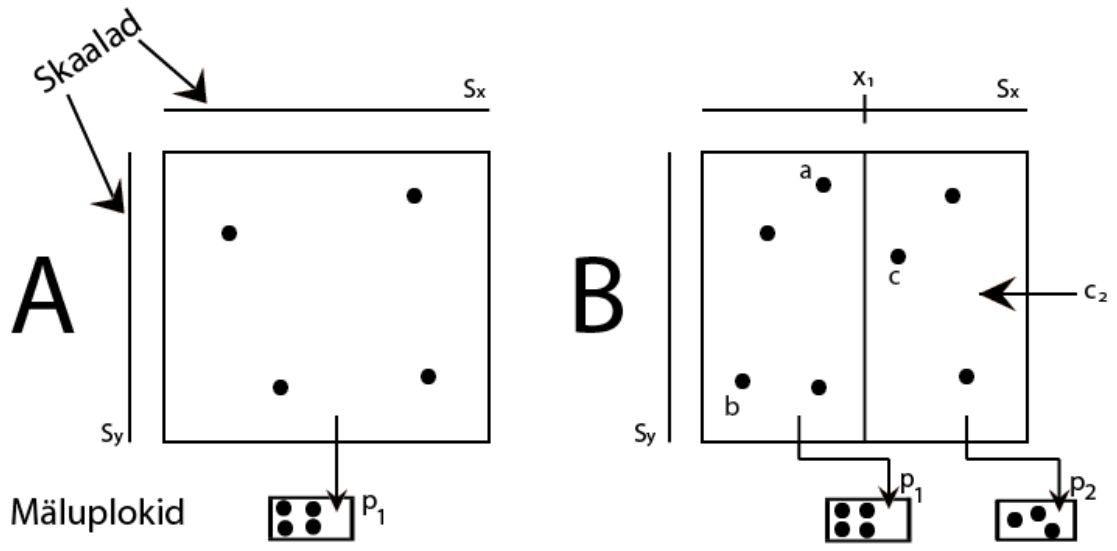
Joonis 6. Adaptiivne võrk ja vastav indeksi massiiv, kus punkte hoiustatakse lihtahelates.

Nii adaptiivses võrgus kui ka fikseeritud võrgustikus on iga mäluplokk seotud vastava elemendiga. Kui viimane täitub üle, siis see poolitatakse ja punktid sisestatakse tekkinud elementidesse (vt. joonis 6). Kui võrku lisatakse uus element ja see ületab võrguelemendi lubatud mahutavuse, siis kogu võrk vastavas reas või veerus poolitatakse. Joonisel 6 sisestatud punkt q poolitab vastava veeru kogu võrgu ulatuses ja poolitamise tagajärjel tekkinud väärtus x_1 sisestatakse skaalasse S_x. Adaptiivne võrk koosneb kolmest andmestruktuurist:

1) massiivist, mis hoiustab võrgu elementidega seotud mäluplokke. Struktuur on sarnane fikseeritud võrgustikule, aga erineb sellepoolest, et kaks külgnevat lahtrit võivad olla seotud ühe mäluplokiga;

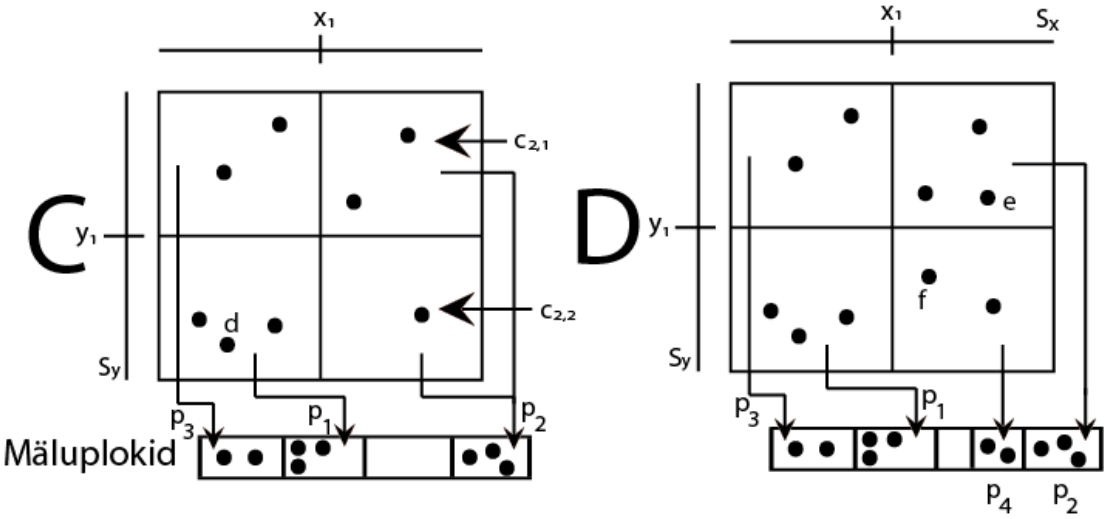
2) kaks skaalat (ingl *scales*) S_x ja S_y on lineaarsed massiivid, kus hoiustatakse koordinaattelgede jagamisel tekkinud väärtuseid ning mis defineerivad otsinguruumi piirid ja adaptiivse võrgu elemendid üheselt.

Vaatleme paremaks arusaamiseks alljärgnevat näidet (vt. joonis 7), kus lehe mahutavus on 4. Alamjoonisel A on struktuuris 4 punkti. Kogu elemendis sisalduv info on seotud vaid ühe mäluplokiga. Alamjoonisel B on lisaks olemasolevale neljale punktile lisatud ka punktid *a*, *b*, ja *c*. Punkti *a* sisestamisel tekib välisahelas ületäituvus ning element jaotatakse kaheks ning vaja on järgmist mäluplokki. Punktid jaotatakse kahe mäluploki vahel, kus esimesele neist jääb 4 punkti ja teisele 3 punkti. Võrgu elementide kaheks jaotumise piir x_1 defineeritakse lineaarses massiivis S_x. Punktid *a* ja *b* sisestatakse leheküljele p_1 ja *c* sisestatakse leheküljele p_2.



Joonis 7. Adaptiivsesse võrgustikku punktide sisestamine ja elementide jaotumine ületäituvusest tingituna.

Joonise 8 alamjoonisel C sisestatakse punkt d. Punkti d peaks sisestama leheküljele p_1, aga see on juba maksimaalselt täitunud. Tekib horisontaalne jaotumine, mille järel hoiustatakse keskpunkt y_1 lineaarses massiivis S_y. Vasakpoolsetes elementides asuvad punktid jaotatakse mäluplokkide p_1 ja p_3 vahel ning säilitatakse seosed varasemate punktide ja mäluplokkide vahel. Element c_2 jaguneb elementideks $c_{2,1}$ ja $c_{2,2}$, mis omakorda viitavad mäluplokile p_2. Alamjoonisel D sisestatakse punktid e ja f. Mäluplokk p_2 täitub üle ning lisatakse mäluplokk p_4. Elementide jaotumist vaja pole, sest kõigis on punktide arv väiksem kui 4. Punktid elemendist $c_{2,1}$ salvestatakse mäluplokki p_2 ja punktid elemendist $c_{2,2}$ salvestatakse mäluplokki p_4.



Joonis 8. Adaptiivse võrgustiku andmestruktuuri muudatused punktide lisamisel.

Kokkuvõtvalt võib öelda, et punkti sisestamisel tuleks vaadelda kolme erinevat juhtumit:

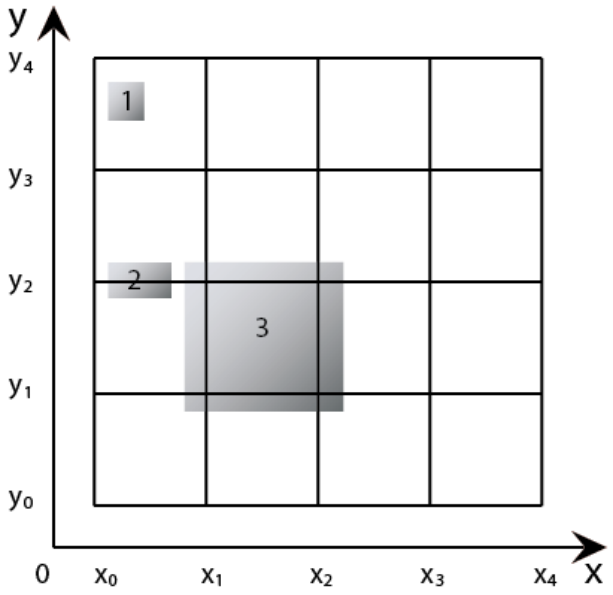
1) ühtegi elementi ei jaotata – kõige lihtsam juhtum, kus punkt sisestatakse elementi, mis pole ületäitunud ja süsteem lisab punkti elemendiga seotud mäluplokki, keerukus *O(1)*;

2) võrguelement jaotatakse ilma seotud massiivi jaotamiseta – sisestatud punkt sisestatakse elementi, millega seotud mäluplokk on täis. See mäluplokk võib seotud olla veel mõne teise võrguelemendiga. Luuakse uus mäluplokk ning seotakse see olemasoleva võrguelemendiga ja punktid koos sisestatava punktiga sisestatakse loodud mäluplokki. Antud näidet kirjeldab joonise 8 näide D;

3) element jaotatakse koos seotud massiiviga – vaadeldud variantidest kõige keerukam. Elemendist $c_{i,j}$ on viidatud mäluplokk p. Suvaline punkt sisestatakse elementi $c_{i,j}$, mis on juba maksimaalselt täitunud. Leidub ka olukord, kus ükski teine element ei viita mäluplokile p. Element poolitub kas x või y teljelt. Element $c_{i,j}$ jaotub elementideks $c_{i,j}$ ja $c_{i+1,j}$. Luuakse uus mäluplokk p', mis seostatakse lahtriga $c_{i+1,j}$. Punktid jaotatakse mäluplokkide p ja p' vahel. Seda olukorda kirjeldab joonisel 8 näide C.

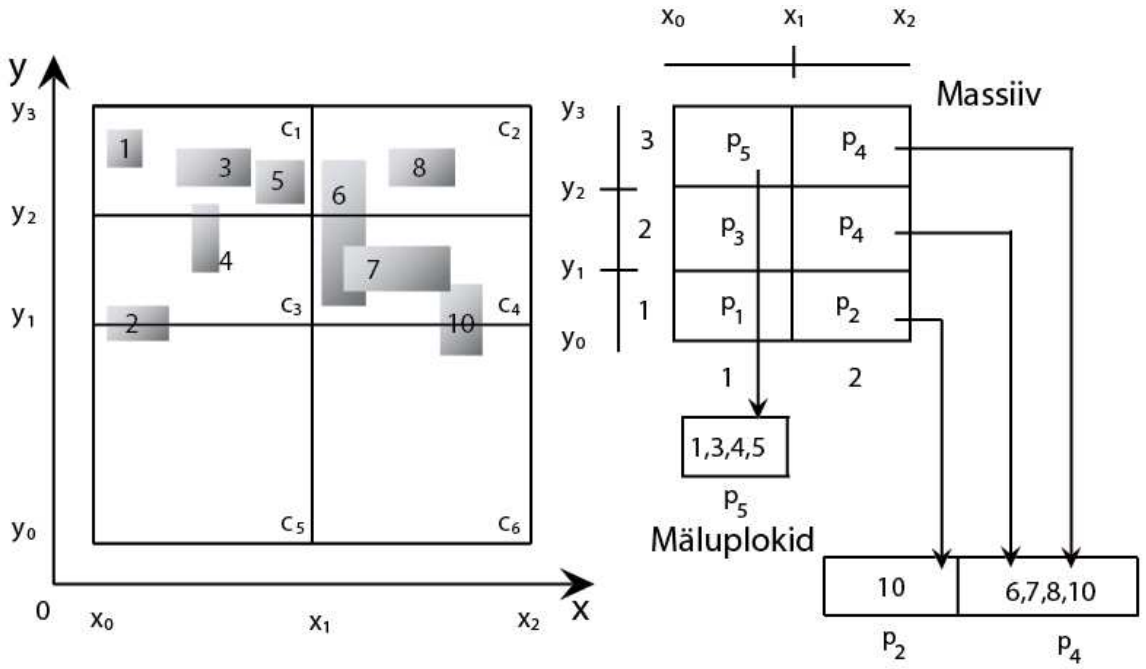
Vaatleme lähemalt adaptiivse võrgustiku keerukusi. Massiivi suurus kirjete ühtlase jaotuse korral on $O(X^d)$, halvimal juhul $O(X^{1+(d-1)/d*w})$. Massiivi laiendamisel kasvab ka ligipääsude arv keerukusega $O(X^{1-1/d})$. Otsing võtab 2 ning sisestamine $O(1)$ kuni $O(X^{d-1})$ operatsiooni. Skaalad on suurusega $O(X)$.

Vaatleme järgnevalt keerulisemat juhtumit, kus indekseerime punktide asemel ristkülikuid. Tekkida võib 3 olukorda (vt. joonis 9) – ristkülik sisaldab elementi, ristkülik lõikub elemendiga või ristkülik on täielikult elemendi sees. Joonistel 10 ja 11 kujutatakse adaptiivset võrgustikku enne ning pärast ristküliku 11 sisestamist. Võrgu elemendi mahutavus näidetes on 4.

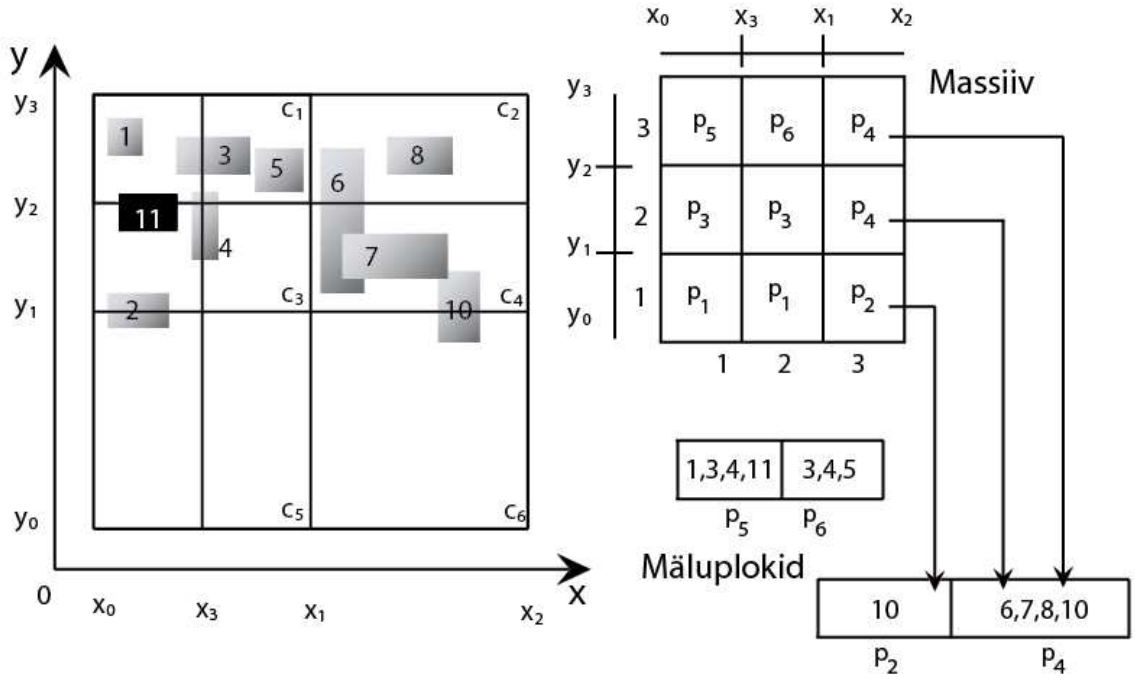


Joonis 9. Adaptiivses võrgustikus kõikvõimalikud ristkülikute paiknemise võimalused.

Ristküliku 11 sisestamine tekitab massiivis mäluploki p_5 ületäituvuse (mäluplokis on lubatud 4 objekti asemel 5). Abtsissteljele lisatakse väärtus x_3, mis poolitab võrgu elemendid c_1,c_3 ja c_5 vertikaalselt. Varasema elemendi c_3 osad viitavad mäluplokile p_3 ja varasema elemendi c_5 osad viitavad mäluplokile p_1. Mäluplokis p_5 asuvad objektid on jaotatud mäluplokkide p_5 ja p_6 vahel. Tähelepanu tuleb pöörata ka olukorrale, kus duplitseeritakse ristkülikute 3 ja 4 väärtuseid mäluplokkides p_5 ja p_6, sest antud ristkülikud paiknevad mõlemas mäluplokkidega seotud võrguelemendis. Eelnev kirjeldus kehtib vaadeldud implementatsiooni kohta ja kattuva ristküliku võib sisestada ka elemendiga seotud mäluplokki, kuhu ristkülik suuremas osas jääb.

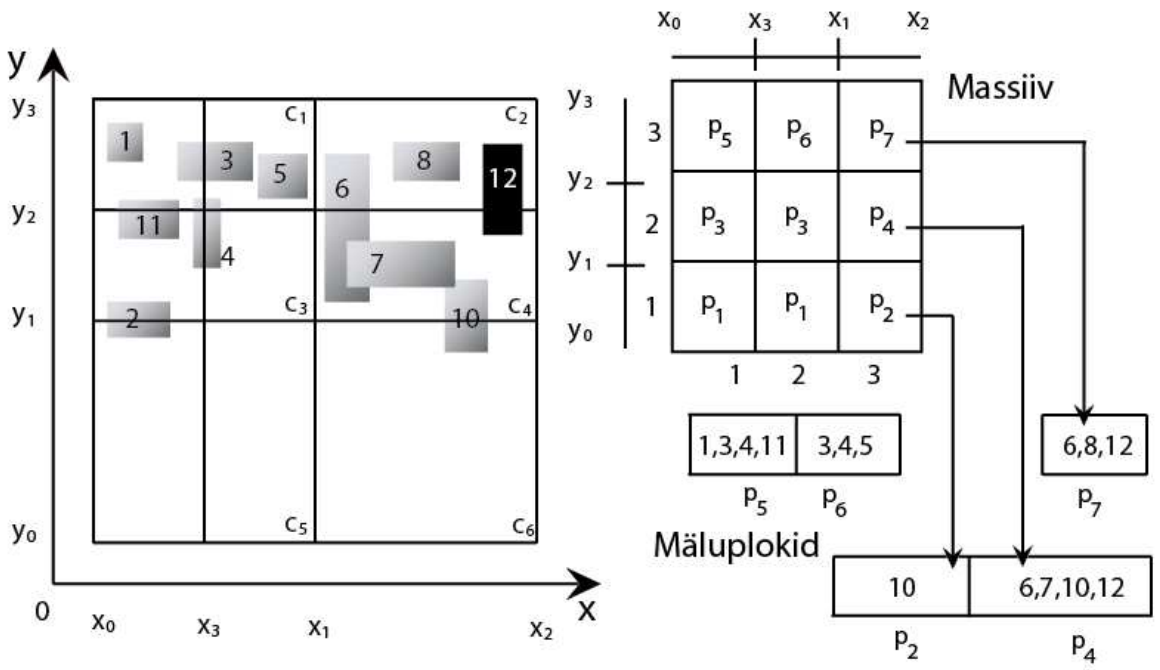


Joonis 10. Adaptiivse võrgustiku seis enne ristkülikute sisestamist.



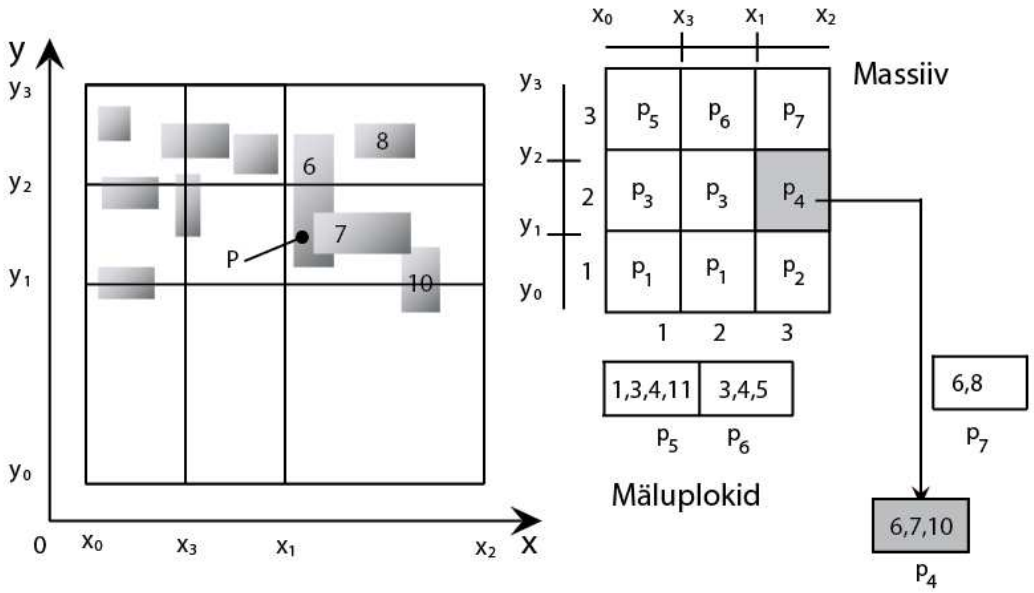
Joonis 11. Adaptiivsesse võrku sisestatakse ristkülik 11.

Vaatleme veel ühte olukorda joonisel 12, kus sisestatakse ristkülik 12. Antud näite korral ei toimu elementide poolitamist. Ristkülik 12 sisestati mäluplokki p_4, kus asuvad ristkülikud 6,7,8 ja 10. Kuna elemendi mahutavuseks on 4, siis tekib lehekülje ületäituvus. Luuakse uus mäluplokk p_7 ning sisestatakse ristkülikud 6,8 ja 12.



Joonis 12. Ristküliku 12 sisestamine adaptiivsesse võrku.

Vaatleme allpool asuva joonise 13 abil punkti päringut adaptiivses võrgustikus. Kuna punkt asub massiivis kohal [3,2], siis viitab viimane mäluplokile p_4. Mäluplokis p_4 on 3 kirjet – 6, 7 ja 10. Vastuseks saame, et punkt asub ristkülikul 6.



Joonis 13. Punktipäring adaptiivses võrgustikus.

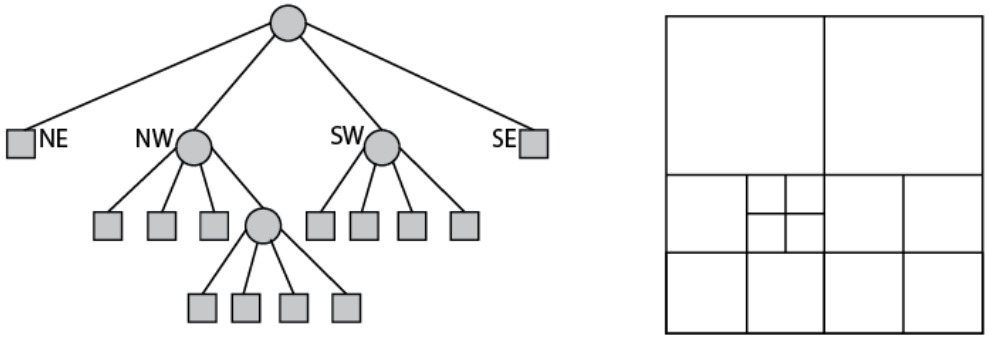
Ka siin on probleem andmete duplitseerimisega mäluplokkide vahel, mis teeb indeksi rakendamise keerukamaks ning päringud võivad vajada täiendavaid sisend-väljundoperatsioone. Kokkuvõtvalt on tegemist üsnagi efektiivse ja paindliku

andmestruktuuriga, mis võimaldab pärida andmeid mõistliku ajaga (punktide korral $n+2$ kettapääsu, kus n tähistab võrguelementide arvu). Lähemalt saab lugeda fikseeritud ja adaptiivsest võrgust raamatu Spatial databases with application to GIS 6. peatükist. [RSV02]

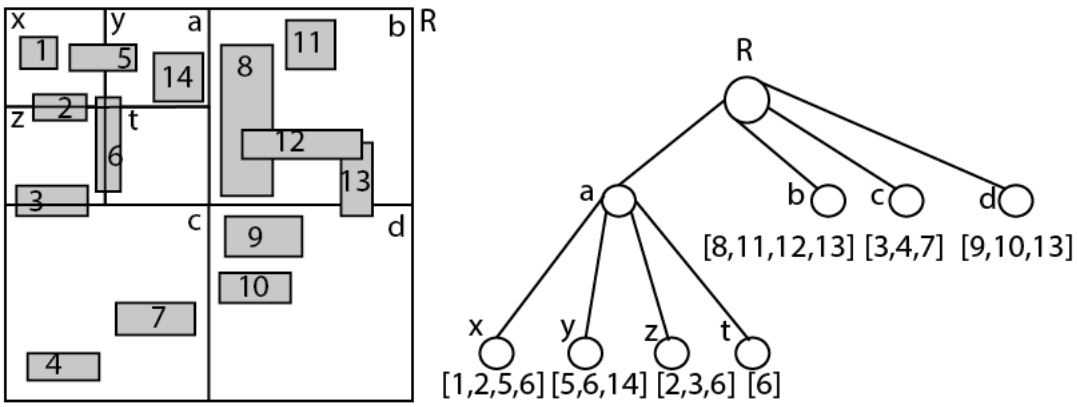
Järgnevates peatükkides vaatleme indekseerimise meetodeid, mis kasutavad B-puu indeksit. Ristkülikud, mida näites indekseeritakse, jaotatakse ruudukujulistesse elementidesse, mis moodustatakse otsinguruumi rekursiivsel „tükeldamisel“. Elemendid ja nende sisu indekseeritakse, kasutades elemendi järku võtmena.

2.2.3 Ruutpuu

Oma lihtsuse tõttu on ruutpuu (ingl *quadtree*) suhteliselt populaarne meetod objektide hoiustamiseks. Vaatleme järgnevates näidetes ristkülikute indekseerimist. Otsinguruum tükeldatakse rekursiivselt ruudukujulisteks elementideks seni kuni ristkülikute arv igas elemendis on väiksem kui mäluploki kogumaht. Juurest lähtuvad 4 tippu märgitakse reeglina lühenditega *NW*, *NE*, *SW*, *SE* (vt. joonis 14), kus lühendid tähistavad ilmakaari. Indeks on kvaternaarse puu kujul (igal sisemisel tipul, mida on 4, on omakorda 4 alamat). Iga leht on seotud mälupiirkonnaga, kus hoiustatakse kirjeid. Nii nagu adaptiivses võrgustikus, võivad ka siin ristkülikud paikneda mitmes mäluplokis. Joonisel 15 vaatleme ruutpuud, kus mäluploki mahutavuseks on 4 kirjet.

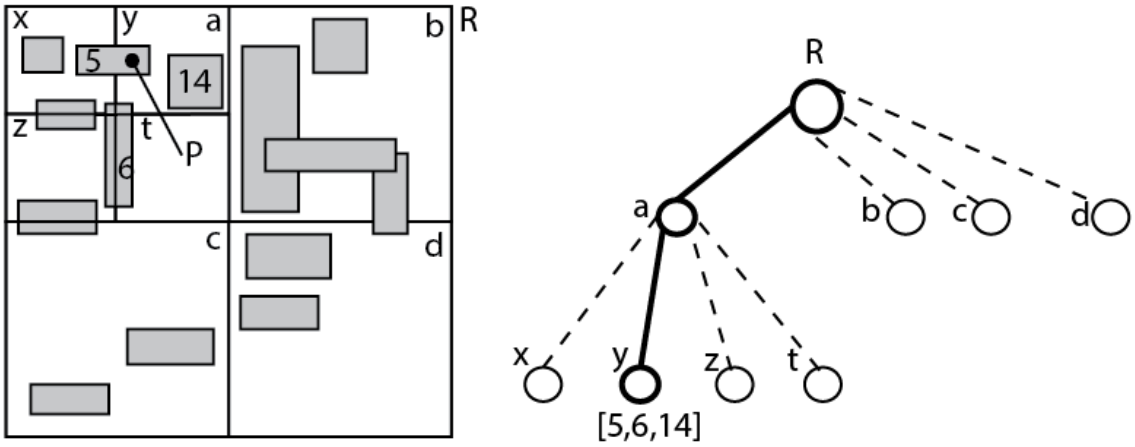


Joonis 14. Ruutpuu ja temale vastav jaotis.



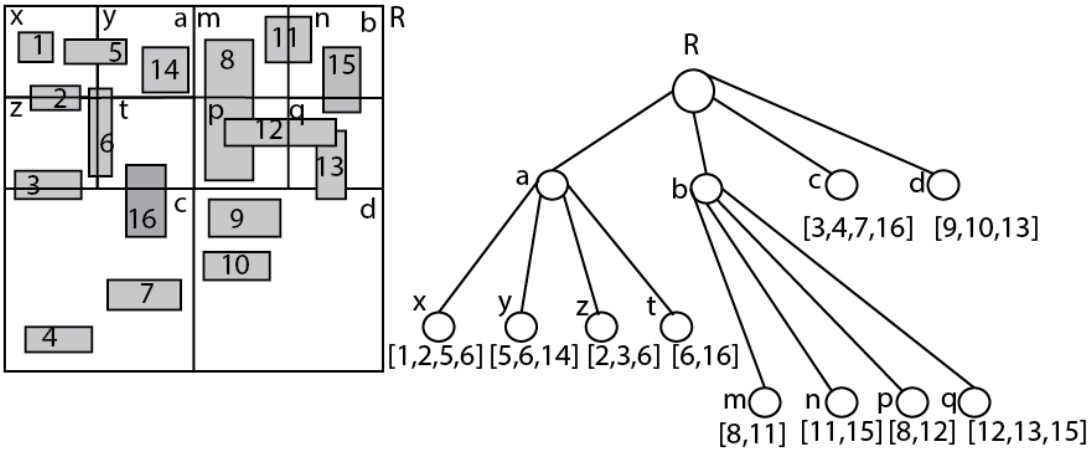
Joonis 15. Ruutpuu, kus lehega seotud mäluploki mahutavus on 4.

Punkti päring on ruutpuus lihtne, sest punktini jõudmiseks on vaja läbida vaid konkreetne tee juurest lehte. Joonis 16 kirjeldab ruutpuus punkti päringut. Otsides punkti P peame läbima ruutpuud juurest. Läbimisel on meil valida 4 tipu vahel. Esimesena valitakse tipp a, sest otsitav punkt asub tipuga seotud elemendis a. Edasi valitakse tippude x, y, z, t vahel. Tipust a valitakse tipp y ning vastuseks on ristkülik 5. Aknaga kattuvuspäringus peame leidma ruudukujulised elemendid, millel on ühisosa aknaga.



Joonis 16. Punktipäring ruutpuus.

Eraldi vaatleme ristkülikute sisestamise operatsiooni. Ristkülik sisestatakse võrgu elementi ning ka siin võib ristkülik paikneda üle mitme elemendi või paikneda täielikult elemendis. Sisestatud ristkülik lisatakse ka kõikidesse elemendiga seotud mäluplokkidesse, kus esineb kattuvus (analoogia adaptiivse võrgustikuga). Kui puu lehega seotud mäluplokk p pole maksimaalselt täitunud, siis lisatakse sinna uus kirje ja kui mäluplokk p on täis, siis jaotatakse võrguelement neljaks alam-elemendiks. Luuakse 3 uut mäluplokki ning seotud kirjed jaotatakse nende 4 mäluploki vahel. Joonis 17 kirjeldab olukorda, kus lisatakse ristkülikud 15 ja 16.



Joonis 17. Objektide 15 ja 16 sisestus ruutpuus.

Ristküliku 15 sisestamine toob kaasa elemendi b jaotumise neljaks elemendiks m, n, p ja q. Ristkülik 15 sisestatakse elementidesse n ja q. Ristküliku 16 sisestus elementidesse c ja t ei too kaasa jaotumist. Siin antud kirjeldus on lihtsustatud ülevaade ruutpuust. Vaatleme lähemalt ruutpuu operatsioonide keerukust. Sisestamise, konkreetse kirje otsingu ja kustutamise keerukus sõltub puu kõrgusest. Halvimal juhul on sisestamine ja kustutamine keerukusega $O(h)$ ja otsing keerukusega $O(2^h+X)$, kus h tähistab puu kõrgust ja X on punktide arv. Tavaliselt on puu kõrgus X punkti jaoks keerukusega $O(\log X)$.

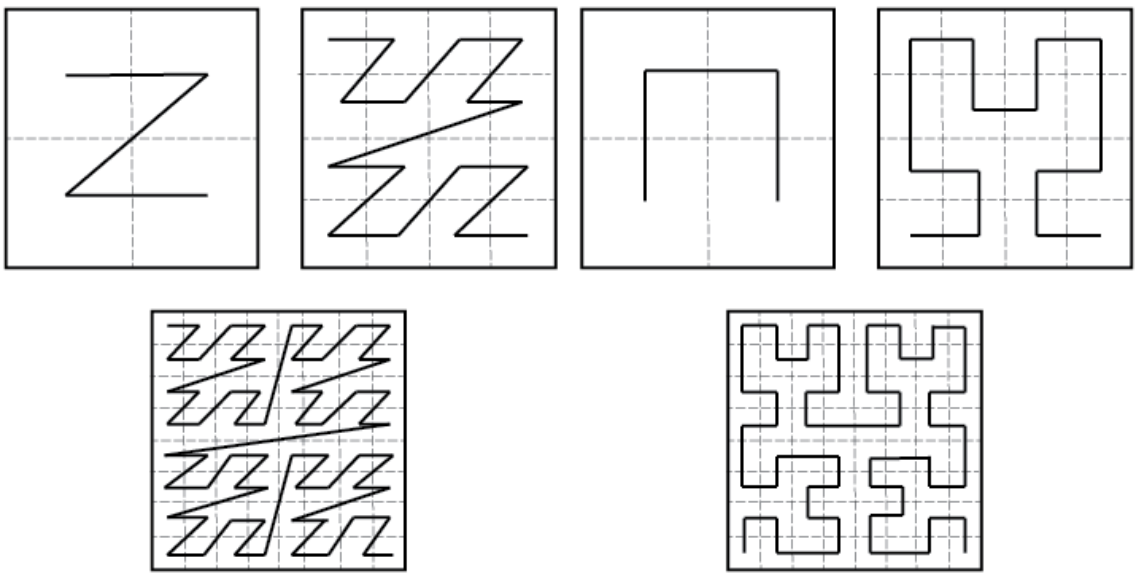
Ruutpuid saab defineerida ka fraktaalsete kõverjoonte, nagu *Z*-kõver (ingl *Z-ordering*) ja Hilberti kõverjoon (ingl *Hilbert curve*), kaudu. Vaatleme lähemalt, miks neid meetodeid kasutatakse. Kuna geoandmete päringud on olemuselt keerukad ja tihtipeale aeganõudvad, siis kasutatakse võimalusel klasterdamist. Klasterdamist saame jaotada kolmeks:

1) sisemine klasterdamine (ingl *internal clustering*) – võimalusel paigutatakse objekt ühte mäluplokki;

2) lokaalne klasterdamine (ingl *local clustering*) – mitme objekti kiireks otsimiseks grupeeritakse objektid (enamasti asukoha järgi) ühte mäluplokki;

3) globaalne klasterdamine (ingl *global clustering*) – mitme objekti kiireks otsimiseks paigutatakse objektid järjestikku mäluplokkidesse.

Probleem seisneb aga selles, et mitmemõõtmelises ruumis pole naturaalset järjekorda. Eelneva probleemi lahendamiseks on vaja meetodit, mis võimaldab kaardistada objektid mitmemõõtmelisest ruumist ühemõõtmelisse (kettapind või mäluplokk on loogiline ühemõõtmeline ruum) ruumi nii, et kaugus ja suhted objektide vahel säiliks.



Joonis 18. *Z*-kõver vasakul ja Hilberti kõverjoon paremal.

Probleemi üritavadki lahendada *Z*-kõver, mille kohta võib kasutada ka terminit Mortoni kood (ingl *Morton code*), ning Hilberti kõver. Ülevaate, kuidas otsinguruum nende kõverate abil täidetakse, annab joonis 18.

Lähemalt saab lugeda ruutpuust raamatu Spatial databases with application to GIS 6. peatükist ja raamatu Spatial databases: a tour 4. peatükist. [RSV02] [SC03]

2.2.4 R-puu

Kui varasemalt vaatlesime ruumipõhiseid andmestruktuure, siis nüüd pöörame tähelepanu andmepõhistele struktuuridele. R-puu (ingl *R-tree*) nagu ka B-puu põhinevad tasakaalustatud puustruktuuril, kus iga leht on ühendatud konkreetse mälu- või kettapiirkonnaga.

R-puu struktuuri kirjeldavad alljärgnevad omadused:

1) tippude, v.a juurtipp, kirjete arv on m kuni M;

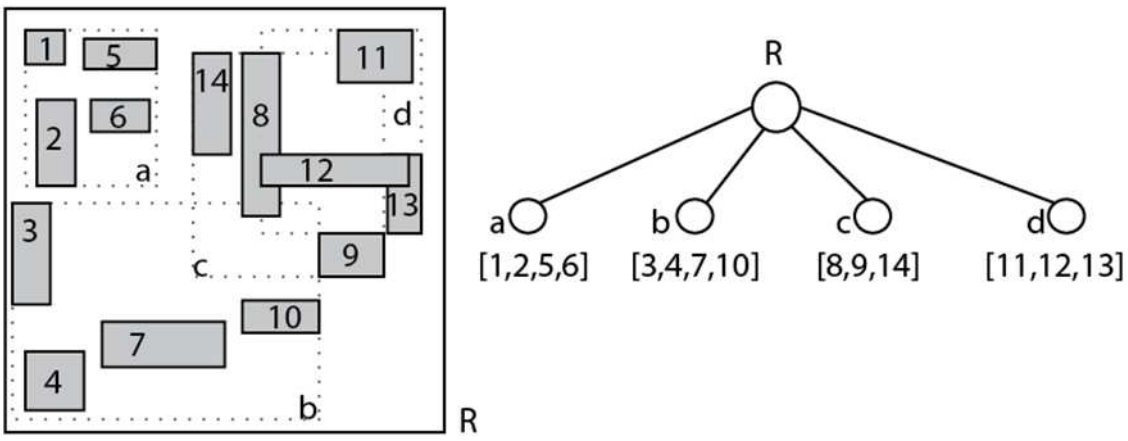
2) tippudes, v.a lehed, on sisestatud objektide kirjed kujul <*tipuga seotud massiiv, viit seotud mäluplokile*>;

3) lehtedes on objektide kirjed kujul <*minimaalne piirdekast, objekti identifikaator*>;

4) juurel on vähemalt 2 alamat, v.a juhul kui juur on leht;

5) kõik lehed on samal tasemel.

Joonisel 19 näeme vasakul ruumi, kus paiknevad massiivid *a*,*b*,*c*,*d* ning paremal R-puud, mille kõrgus on 2 ja igasse mäluplokki saab salvestada 4 kirjet. Kokku on massiivides 14 ristkülikut.



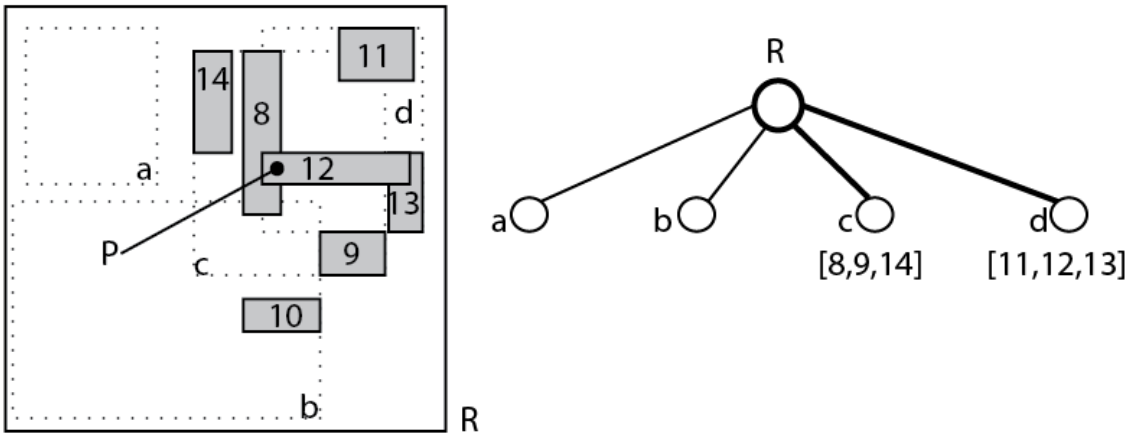
Joonis 19. R-puu jaotis ja struktuur.

R-puu kohandub vastavalt andmete asümmeetrilisusele – kui meil on tegemist piirkonnaga, kus on väga palju objekte, siis genereeritakse vastav arv lehti ning harud on vastavalt kas lühemad või pikemad. Maksimaalne arv kirjeid lehtedes s/W sõltub nii kirje suurusest s kui ka mäluploki mahutavusest W. Maksimaalne kirjete arv sõltub ka tippude liigist. Kui meil on defineeritud m ja M ja R-puu kõrgusega h, siis on võimalik indekseerida minimaalselt m^{h+1} ja maksimaalselt M^{h+1} objekti. R-puu tegelik kõrgus sõltub tippude täituvusprotsendist, aga saame öelda, et see on vähemalt $\log_M X - 1$ ja maksimaalselt $\log_m X - 1$.

Vaatleme R-puu efektiivsust järgneva näitega. Olgu meil mäluploki mahutavuseks 4 kilobaiti, kirje suurus 20 baiti (16 baiti minimaalse piirdekasti defineerimiseks ja 4 baiti objekti identifikaatori kirjelduseks) ja m on 40% mäluploki kogumahust. Sellest järeldame, et maksimaalne kirjete arv on ligikaudu 204 ja minimaalne kirjete arv 81. R-puu kõrgusega 1 võimaldab indekseerida vähemalt 6561 objekti ning kõrgusega 2 vähemalt 531 441 ja maksimaalselt 8 489 664 objekti. Kui meil on objektide kollektsioon miljoni objektiga, siis kulub 3 kettaoperatsiooni puu läbimiseks ja lisaoperatsioon, et leida kirje, kus soovitud objekti hoiustatakse.

Punkti ja aknaga kattuvuspäringud on üsna sarnased. Punkti pärimisel läbitakse juure kõikide alamate massiivid, mis sisaldavad otsitavat punkti P. Kuna ristkülikud võivad üksteisega kattuda, siis võib punkt sattuda ristumispiirkonda ja tuleb läbida ka kõik seotud alampuud. Operatsiooni läbitakse seni, kuni jõutakse lehtedeni. Järgneval sammul läbitakse kõik lehtedes olevad väärtused ja leitakse need, mille piirdekasti (ingl *bounding box*) punkt P jäi.

Alljärgneval joonisel 20 kujutatakse punkti P otsingut, mis jääb objektide 8 ja 12 piirdekasti. Antud näites läbitakse 3 tippu – R, c ja d.



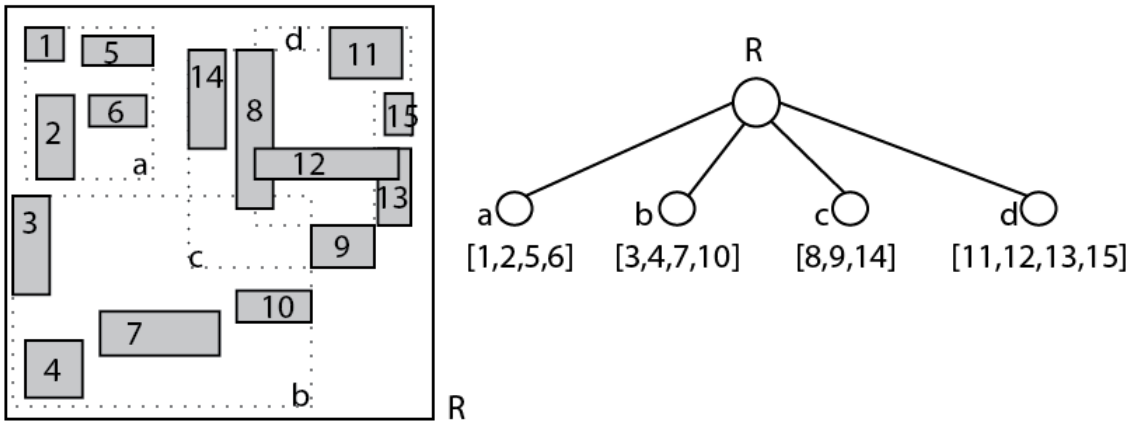
Joonis 20. Punkti P otsing R-puus.

Eespool kirjeldasime, et kattuvuspäring etteantud piirdeaknaga on analoogne. Predikaat „sisaldab punkti P“ (ingl *contains* P) asendatakse predikaadiga „lõikub

piirdeaknaga W“ (ingl *intersects* W). Mida suurem on defineeritud piirdeaken, seda rohkem tippe peab läbima.

Tavajuhul on otsingu keerukus logaritmiline, aga halvimal juhul (eespool mainitud ristumispiirkond) tuleb läbida kogu puu juurest lehtedeni.

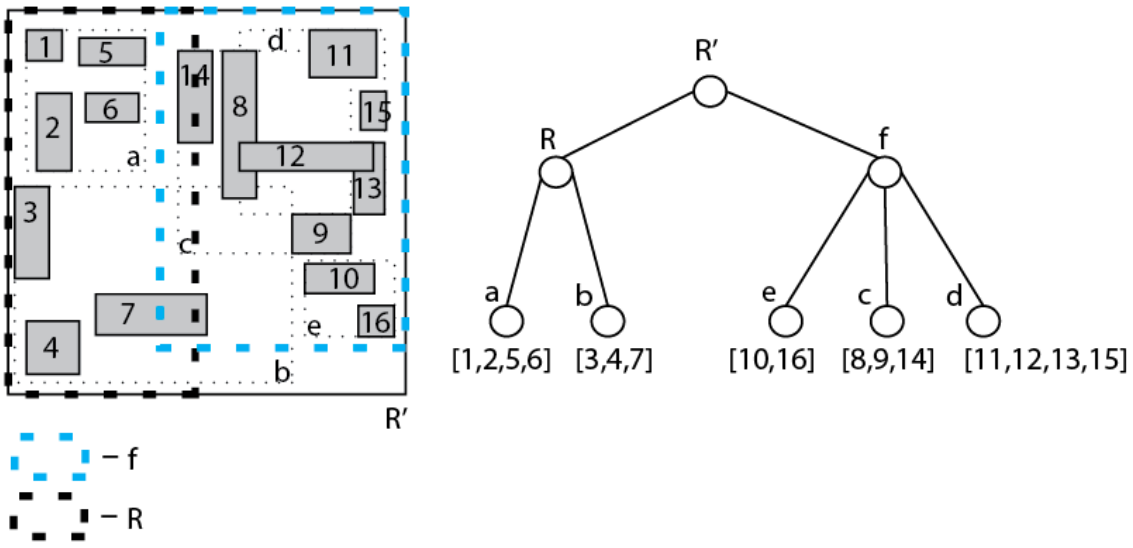
Objekti sisestamisel R-puusse läbitakse puu juurest lehtedeni. Igal tasemel kontrollitakse, kas leidub tipp, mille massiivi objekt jääb ja kui jah, siis läbitakse vastav alampuu, teisel juhul sellist tippu ei leidu. Kordame protsessi senikaua, kuni leiame sobiva lehe. Kui leht *l* pole maksimaalselt täitunud, siis lisame lehega seotud kirje mäluplokki. Lisamise käigus võib tekkida olukord, kus objekti sisestamisel peab suurendama ka massiiviga seotud ristkülikut ning sellega kaasneb eelnevate tippude



Joonis 21. R-puusse sisestatakse ristkülik 15.

kirjete uuendamine. Kui leht *l* on maksimaalselt täitunud, siis toimub poolitamine. Poolitamisel luuakse uus leht *l'* ning $M + 1$ kirjet jaotatakse lehtede *l* ja *l'* vahel. Peale poolitamist uuendatakse kirjed lehele *l* eelnevas tipus k, kuhu lisatakse kirje *l'*. Juhul kui tipp k on maksimaalselt täitunud, siis korratakse eespool kirjeldatud protsessi halvimal juhul juureni, mille poolitamisel kasvab puu kõrgus 1 taseme võrra. Eelneva paremaks kirjeldamiseks vaatame kahte näidet, kus esimeses sisestatakse objekt 15 (vt. joonis 21) ja teises objekt 16 (vt. joonis 22) ning eeldame, et mäluploki mahutavuseks on 4.

Lisades lehte d objekti 15, tuleb suurendada massiiviga seotud ristkülikut ning uuendada kirjed juures. Lisades lehte b objekti 16 tekib ülevoolavus, sest lehes b on juba 4 kirjet – 3,4,7 ja 10. Leht b poolitatakse ja luuakse uus leht e ning kirjed jaotatakse nende kahe lehe vahel. Lehte b jäävad väärtused 3, 4 ja 7 ning lehte e lisatakse väärtused 10 ja 16. Uuendama peab ka kirjet lehele b eelnevas juures. Kuna juur R sisaldab juba 4 kirjet, siis viimane poolitatakse. Luuakse uus tipp *t*. Tipud a ja b jäävad seotuks tipuga R ning c, e ja d liidetakse tipuga *t*. Viimaseks sammuks luuakse uus juur *R'* ning juure alamateks on tipud R ja *t*.



Joonis 22. Objekti 16 sisestamine R-puusse.

Ristkülikute arv lehes on $O(W)$. Kõrgus h on $O(\log_W X)$. Mäluplokkide arv $O(X/W)$. Sisestamise keerukus halvimal juhul on $O(X)$. Sisestamise ja kustutamise keerukus keskmisel juhul on $O(\log_W X)$, aga kustutamine võib olla lisakeerukusega, sest lisandub otsing kustutavale ristkülikule. Otsing on keskmisel juhul keerukusega $O(\log_W X)$. Võrreldes R$^+$-puuga kasutab R-puu mälupiirkonda ebaefektiivsemalt, sest massiivide ristkülikutel on andmete duplitseerimine lubatud.

Kustutamise algoritm töötab alljärgnevate põhimõtete alusel:

1) otsi leht, mis sisaldab soovitut kirjet;

2) eemalda soovitud kirje lehelt;

3) organiseeri vajadusel puustruktuur ümber, kui tipp sisaldab vähem kui m kirjet (lihtsaim moodus on kustutada tipp ning sisestada ülejäänud $m - 1$ kirjet tagasi).

Lähemalt saab lugeda R-puu indeksist raamatu Spatial databases with application to GIS 6. peatükist. [RSV02]

2.2.5 R$^+$-puu

R$^+$-puu on üks R-puu variatsioonidest, kus kasutatakse kustutamise ja päringu operatsioonideks sama algoritmi, aga optimeeritakse iga massiivi ristküliku piire, pindala ja ületäituvust massiivis. Selline muudatus struktuuris võimaldab meil läbida puu juurest leheni, ignoreerides alampuid, mis vastust ei annaks.

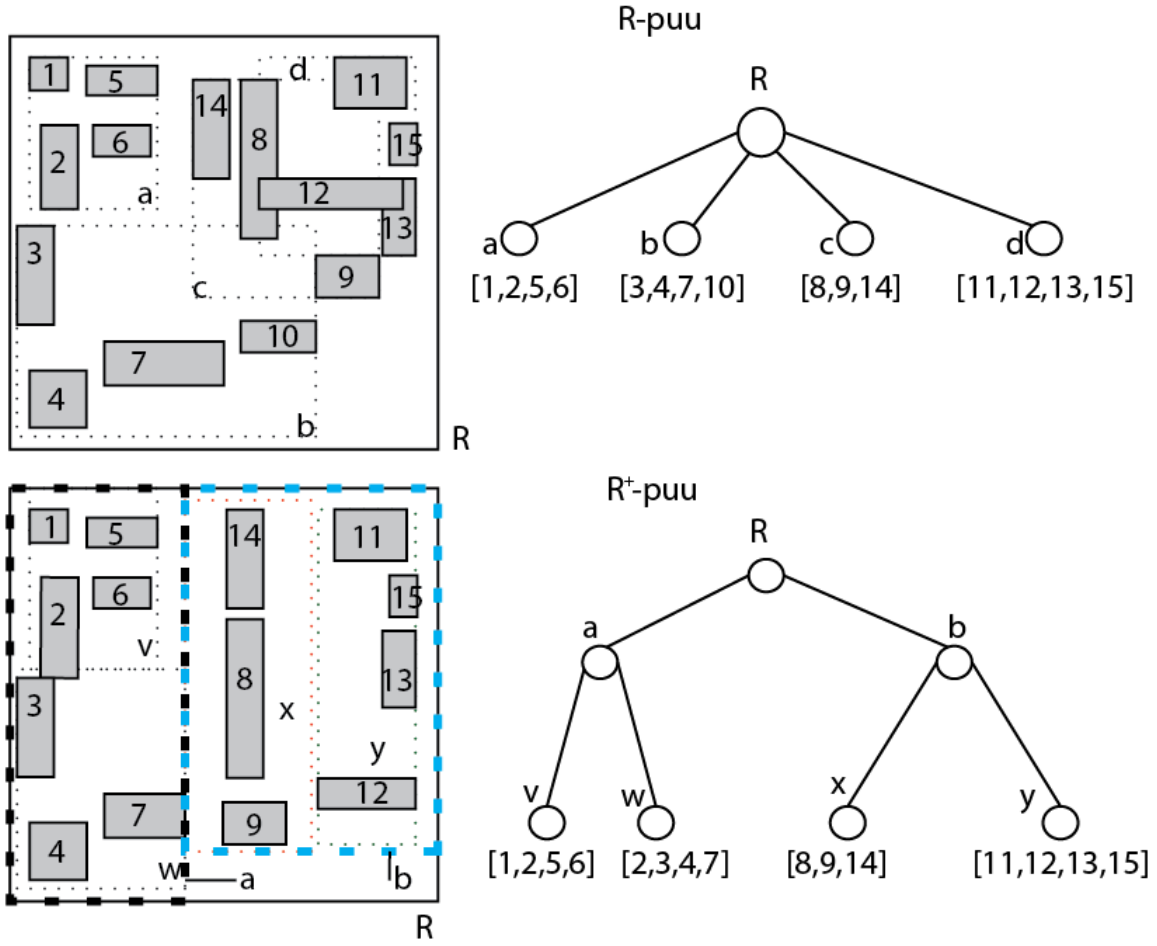
R$^+$-puu on R-puu variant alljärgnevate omadustega:

1) tippude täituvus võib olla vähem kui 50%;

2) kirjed ei kattu tippudes;

3) objekti *ID* võidakse hoiustada mitmes lehega seotud mäluplokis.

Kokkuvõtvalt üritatakse ülekattumist vältida sellega, et objekti võidakse duplitseerida mitmes lehes. Kuna lehtedes lubatakse kirjete duplitseerimist, siis on R⁺-puu samasugusest R-puust mahukam, aga võimaldab efektiivsemat otsingut, sest nn tühi ruum tippude vahel on minimeeritud.



Joonis 23. R-puu massiiv ja struktuur ning võrdluseks R⁺-puu, kus massiivis on näha, et lehtedega seotud ristkülikud ei kattu.

Jooniselt 23 näeme, et objekt 2 on duplitseeritud lehtede *v* ja *w* vahel, kuid nendega seotud ristkülikud ei kattu.

Eespool vaadeldud operatsioonide keerukused on analoogsed R-puuga ja siinkohal uuesti ei korda.

R⁺-puudest saab lähemalt lugeda artiklist The R+-Tree: A dynamic index for multi-dimensional objects. Lisaks R⁺-puule eksisteerib ka teisi R-puu variante nagu R*-puu, millest saab lähemalt lugeda artiklist The R*-tree: an efficient and robust access method for points and rectangles. [SRF87] [BKSS90]

2.2.6 GiST

GiST (ingl *Generalized Search Tree*) on indeksina kasutatav andmestruktuur, mis on paindlik ja võimaldab implementeerida sama koodipõhjaga mitmeid puustruktuure, sh B-puu ning R-puu. GiST indeksil on alljärgnevad omadused:

1) Iga tipp sisaldab kX kuni X indeksikirjet (v.a juur), kus $\frac{2}{X} \leq k \leq \frac{1}{2}$;

2) Juur sisaldab 2 kuni X kirjet;

3) Kirjed on kujul <*väärtus, viit mäluplokile*>;

4) Juurel on vähemalt 2 alamat, v.a kui juur on leht;

5) Kõik lehed on samal tasemel.

GiST andmestruktuuri kasutab tuntud andmebaas PostgreSQL. GiST indeksi keerukus taandub andmestruktuurile, mida andmete indekseerimiseks kasutatakse, vastavalt kas B-puu, R-puu või mõni muu otsingupuu. GiST indeksitest ja andmestruktuurist saab lugeda artiklist Generalized Search Trees for Database Systems. [HNP95]

2.3 Geoandmetega tehtavad operatsioonid

Enne kui läheme jõudluskatsete juurde, peab andma ülevaate operatsioonidest, mida geoandmetega rakendada saab.

Operatsioone[3] jagan viieks:

1) suunaga seotud – kohal, all, põhjas, lõunas, idas, läänes;

2) meetrikaga seotud – kas kaugus on väiksem või suurem kui etteantud ühik;

3) topoloogilised – topoloogiaga seotud suhted objektide vahel (vt. altpoolt täpsemat kirjeldust), olemuselt predikaadid (tagastavad *'TRUE'*, kui tingimus kehtib ja *'FALSE'*, kui tingimus ei kehti);

4) topoloogiaga mitteseotud – pikkus, ümbermõõt, pindala;

5) hulgateoreetilised operatsioonid – ühend, ühisosa, vahe, sümmeetriline vahe.

Topoloogiliste operatsioonide hulka kuuluvad alljärgnevad[4]:

[3] Operatsioonid jagunevad kaheks: unaarsed (üks argument) ja binaarsed (kaks argumenti). Töös vaadeldakse peamiselt binaarseid operatsioone.

[4] Nii topoloogiliste kui ka hulgateoreetiliste operatsioonide korral on aluseks võetud „*Open Geospatial Consortium*“ poolt kirjeldatud standardi „*Simple feature Access*“ dokument. [OGC]

1) Kas objektid on võrdsed? (ingl *Equals*);

2) Kas objektid on mittesidusad? (ingl *Disjoint*);

3) Kas objektid lõikuvad? (ingl *Intersects*);

4) Kas objektid puutuvad? (ingl *Touches*);

5) Kas objektid ristuvad? (ingl *Crosses*);

6) Kas esimene objekt on teise sees? (ingl *Within*);

7) Kas teine objekt on esimese sees? (ingl *Contains*);

8) Kas esimene objekt katab teise? (ingl *Overlaps*);

9) Kas objektid on mingis seoses? (ingl *Relate*).

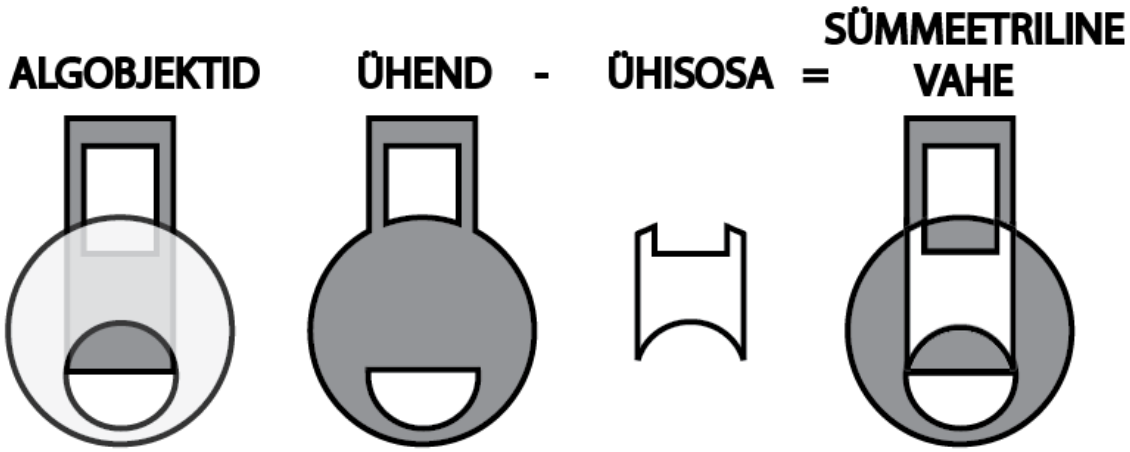
Hulgateoreetilised operatsioonid (vt. joonis 24) on alljärgnevad:

1) ühend – leiab kahe geomeetrilise objekti vahel ühendi (ingl *Union*);

2) ühisosa – leiab kahe geomeetrilise objekti vahel ühisosa (ingl *Intersection*);

3) vahe – leiab kahe geomeetrilise objekti vahel vahe (ingl *Difference*);

4) sümmeetriline vahe – leiab kahe geomeetrilise objekti vahel sümmeetrilise vahe (ingl *Symmetric difference*).



Joonis 24. Hulgateoreetilised operatsioonid geomeetriatega.

3 Testimismetoodika ja testide tulemused

Iga kasutaja või klient, kellega autor on kokku puutunud, arvab, et tema valitud geoandmebaas on funktsiooni X täitmiseks parim. Ka Internetis on arvamusi

erinevaid. Antud töös ei proovita rakendada kõiki võimalikke funktsioone, mida valitud geoandmebaasid (PostgreSQL 9.2.2, Oracle 11g ja SQL Server 2012) toetavad, vaid skoobis on indeksiga ja indeksita kattuvuspäringud. Miks on just valitud kattuvuspäringud ja mitte mingid teised päringud? Enamik geoinfosüsteeme kasutab mingis kontekstis topoloogilisi või hulgateoreetilisi operatsioone, et leida, mis viisil esimene objekt on seoses teisega või vastupidi. Ka Regios kasutatakse kõige enam lihtsaid kattuvuspäringuid ja allpool kirjeldatud PRIA ülesanded on juba spetsiifilisemad. Reeglina on klientide andmemahud miljoni kirje piires, antud töös kasutatud metsaeraldisi on ligikaudu 1.5 miljonit. Kindlasti võib olla kõigi andmebaaside opereerimine erinev kümnete miljonite kirjetega, aga see ei kuulu antud töö skoopi. Praktiline sisend tuli PRIA projektist, kus lisaks kattuvuspäringutele kasutatakse ka väga palju ühisosade pindala leidmist. PRIA kasutab oma töös päringute algandmetena põllumassiive (üle 124 000), maakondi (15) ja katastriüksuseid (üle 600 000). Kuna mul pole luba neid tulemusi avaldada, siis saan vaid anda edasi üldisema kirjelduse. Ühisosade pindalade leidmine oli kordades aeglasem kui tavaline kattuvuspäring, mida oligi ka oodata, sest kontrollitakse, kas põllumassiiv või katastriüksus jääb vastavasse maakonda ja arvutatakse ka ühisosa pindala. Antud töös soovin saada vastuseid alljärgnevatele küsimustele:

1) Kui palju aega võtab kattuvuspäring indeksita?

2) Kui palju aega võtab kattuvuspäring indeksiga?

3) Kui kiire on iga andmebaas konkreetselt valitud kattuvuspäringute sooritamisel?

4) Kui kiire on igapäevases töös kasutatav PostgreSQL konkreetselt valitud kattuvuspäringute sooritamisel?

Nendele küsimustele vastates saan võrrelda tulemusi enda oletustega.

3.1 Andmed ja konfiguratsioon

Andmetena kasutan Maa-ametist allalaetavat omavalitsuste kihti (valdade geomeetriad seisuga 01.01.2013), maakondade kihti (seisuga 01.01.2013) ning Keskkonnateabe Keskusest allalaetavaid metsaeraldisi[5] (metsa-alade geomeetriad seisuga 12.12.2012). Selleks, et rakendatavad päringud ei sõltuks riistvarast, kasutasin kõigi andmebaaside jaoks ühte ja sama *desktop*-tüüpi arvutit, mille spetsifikatsioon on kirjeldatud järgnevates lõikudes. Päringud käivitasin „värske“

[5] Eraldis on pinnalt terviklik metsaosa, mis on oma päritolu, koosseisu, vanuse, rinnaspindala, kõrguse, tagavara ja kasvukohatüübilt kogu ulatuses piisavalt ühetaoline ühesuguste majandamisvõtete rakendamiseks.

masinaga, sest üritasin vältida protsessimisele kuluvat lisaaega. Arvan, et sellise spetsifikatsiooniga masin ei mõjuta oluliselt tulemuste kvaliteeti.

Testandmete töötlemiseks, andmebaasidega ühendumiseks ning andmebaasi andmete importimiseks kasutasin alljärgnevaid tarkvarasid:

1) MapInfo Professional – tarkvara .tab failide töötlemiseks ja konvertimiseks;
2) VisualGIS – AS Regio poolt loodud tarkvara .shp ja .tab failide haldamiseks;
3) Oracle SQL Developer – klientrakendus Oracle andmebaasiga ühendumiseks ja päringute tegemiseks;
4) PgAdmin – klientrakendus PostgreSQL andmebaasiga ühendumiseks ja päringute tegemiseks;
5) Toad for SQL Server – klientrakendus Microsoft SQL Server andmebaasiga ühendumiseks ja päringute tegemiseks;
6) ogr2ogr – käsurealt käivitatav rakendus, mis võimaldab geoandmeid konvertida erinevatesse formaatidesse ning importida neid ka otse andmebaasi;
7) Notepad++ – võimekas tekstitoimeti testandmete salvestamiseks ja töötlemiseks.

Andmebaaside versioonid:

1) PostgreSQL 9.2.2, PostGIS 2.0.1 r9979 (x64);
2) Oracle Database 11g Release 2 (11.2.0.1.0) (x64);
3) Microsoft SQL Server 2012 – 11.0.2100.60 (x64).

Testmasina konfiguratsioon:

1) Protsessor: Intel Core i5 2500 @ 3.30GHz;
2) Emaplaat: Intel DQ67SW;
3) Mälu: 16GB DDR3 @ 1333MHz;
4) Graafika: Intel HD 2500;
5) Kõvakettad: 2x1TB @ RAID1;
6) Operatsioonisüsteem: Windows 8 Pro 64bit.

PostgreSQL ja Oracle kasutavad indeksi loomisel R-puud ja SQL Server kasutab B-puu põhist indeksit. Oracle andmebaasi programmeerimiseks on kasutatud

programmeerimiskeeli assembler, C ja C++. SQL Serveri arendamisel on kasutatud programmeerimiskeelt C++ ning PostgreSQL on täielikult arendatud keeles C. Oracle kasutab geoandmete töötlemiseks Oracle Spatialit (kinnine lähtekood) ja PostgreSQL GEOS (ingl *Geometry Engine – Open Source*) paketti. SQL Server kasutab Microsofti arendatud geopaketti. Antud töö raames ei lahata, miks tulevad geoandmetega opereerides ajalised erinevused valitud andmebaaside vahel.

3.2 Andmete import/eksport ja jõudluskatsete disain

Importimiseks ja eksportimiseks kasutatud käsurea utiliit ogr2ogr kasutab GDAL (ingl *Geospatial Data Abstraction Library*) teeki, mida kasutavad ka teised sarnased tööriistad. Metsaeraldiste importimiseks ja eksportimiseks vajalikud käsud on loetletud lisas 1. Lisas 1 toodud käsud olid sarnased valdade importimiseks ja eksportimiseks ning eraldi ei vaadelda. Andmete importimise järel oli igas andmebaasis 226 valda, 15 maakonda ja 1 460 964 metsaeraldist.

PostgreSQL ja SQL Server lõid automaatselt ID veerule indeksi, Oracle andmebaasis tuli luua see käsitsi. Indeksite loomiseks Oracle 11g andmebaasis kasutasin alljärgnevaid käske:

1) *CREATE UNIQUE INDEX metsaeraldised_ogr_fid_idx ON metsaeraldised(ogr_fid);*

2) *CREATE UNIQUE INDEX vallad_ogr_fid_idx ON vallad(ogr_fid).*

PostgreSQL andmebaasis loodi andmete importimisel automaatselt ka geoindeks, võimalikuks põhjuseks parem ühilduvus GDAL teegiga. Enne kui vaatleme lähemalt rakendatud päringuid, tuleks anda kiire ülevaade, kuidas geoindekseid kõigis kolmes andmebaasis luua ja eemaldada (vt. tabel 6 ja tabel 7).

Andmebaas	**Süntaks indeksi loomiseks**
PostgreSQL	*CREATE INDEX indeksinimi ON tabelinimi USING gist (geomeetriaveerg);*
Oracle	*CREATE INDEX indeksinimi ON tabelinimi(geomeetriaveerg) INDEXTYPE IS MDSYS.SPATIAL_INDEX;*
SQL Server	*CREATE SPATIAL INDEX [indeksinimi] ON [dbo].[tabelinimi] ([geomeetriaveerg]) USING GEOMETRY_GRID WITH(BOUNDING_BOX=(370000,6300000,780000,6800000),GRIDS=(LEVEL_1=MEDIUM,LEVEL_2=MEDIUM,LEVEL_3= MEDIUM,LEVEL_4= MEDIUM), CELLS_PER_OBJECT=256, PAD_INDEX=OFF,FILLFACTOR=100,ALLOW_ROW_LOCKS=ON, ALLOW_PAGE_LOCKS=ON) ON [PRIMARY];*

Tabel 6. Indeksite loomine andmebaasides.

Andmebaas	Süntaks indeksi eemaldamiseks
PostgreSQL	*DROP INDEX indeksinimi; (vajadusel määrata ka skeem: skeem.indeksinimi)*
Oracle	*DROP INDEX indeksinimi;*
SQL Server	*DROP INDEX indeksinimi ON tabelinimi;*

Tabel 7. Indeksi kustutamine andmebaasides.

Kõige mugavam on indekseid luua ja eemaldada läbi graafilise liidese ning kasutusel olevad klientrakendused võimaldasid seda lihtsasti ka teha. SQL Serveri geomeetriaindeks vajab lisaks tiheduse määramisele ka piiride määramist (*BOUNDING_BOX* väärtused tähistavad Eesti minimaalseid ja maksimaalseid x ja y koordinaate).

Testimiseks valisin kattuvuspäringud metsaeraldiste ja valdade või maakondade vahel. Tegin kattuvuspäringud, kus pärisin esiteks Tartu valla kõiki metsaeraldisi, teiseks Tartu maakonna kõiki metsaeraldisi ja lõpuks 8 maakonna (Järva, Hiiu, Saare, Jõgeva, Lääne, Põlva, Harju, Tartu) kõiki metsaeraldisi. Päringute rakendamisel vaatlesin olukorda, kus esimesel juhul on andmed indekseeritud ja teisel juhul indekseerimata. Eraldi uurisin PostgreSQL kiirust 4 päringu sooritamisel. Eelnevalt kirjeldatud stsenaariumite võrdlemine peaks andma hea ülevaate, kui palju erineb kiirus indekseerimata ja indekseeritud andmete vahel ja millised erinevused on andmebaaside endi vahel.

3.3 Tulemused ja tulemuste analüüs

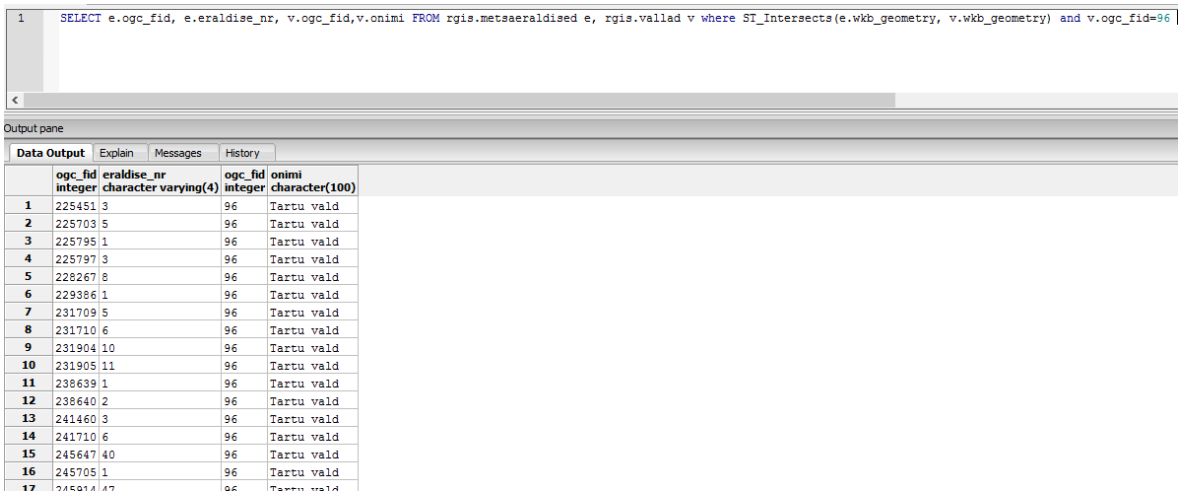
Kattuvuspäringu „Anna mulle Tartu valla kõik metsaeraldised“ rakendamine indekseeritud ja indekseerimata andmetega tõi päris hästi välja erinevuse kiiruses ja ka erinevuse andmebaaside endi vahel. Vaadeldud andmebaasidest oli kiireim PostgreSQL, järgnes SQL Server ja viimaseks jäi Oracle. Kattuvuspäringuid kordasin kõikidel juhtudel 10 korda ning võtsin nende koguaja keskmise ja esitasin ka standardhälbe.

Kattuvuspäringute süntaks kõigi kolme andmebaasi jaoks on toodud tabelis 8.

Andmebaas	Kattuvuspäringu süntaks	Tagastatud ridade arv
PostgreSQL	*SELECT e.ogc_fid, e.eraldise_nr, v.ogc_fid,v.onimi FROM rgis.vallad v, rgis.metsaeraldised e where*	7585

	ST_Intersects(e.wkb_geometry, v.wkb_geometry) and v.ogc_fid=96;	
Oracle indeksiga	*SELECT /*+ordered*/ e.ogr_fid,e."eraldise_nr",v.ogr_fid,v.onimi FROM vallad v, metsaeraldised e where ST_Intersects(e.ora_geometry, v.ora_geometry)='TRUE' and v.ogr_fid=94;*	7587
Oracle indeksita	*SELECT /*+ordered no_index(e metsaeraldised_gidx) no_index(v vallad_gidx) */ e.ogr_fid,e."eraldise_nr",v.ogr_fid,v.onimi FROM vallad v, metsaeraldised e where ST_Intersects(e.ora_geometry, v.ora_geometry)='TRUE' and v.ogr_fid=94;*	
SQL Server	*SELECT e.ogr_fid, e.eraldise_nr, v.ogr_fid,v.onimi FROM rgis.dbo.vallad v, rgis.dbo.metsaeraldised e WHERE (e.ogr_geometry.STIntersects(v.ogr_geometry) = 1) and v.ogr_fid=94;*	7586

Tabel 8. Kattuvuspäring „Anna mulle kõik Tartu valla metsaeraldised“.



	ogc_fid integer	eraldise_nr character varying(4)	ogc_fid integer	onimi character(100)
1	225451	3	96	Tartu vald
2	225703	5	96	Tartu vald
3	225795	1	96	Tartu vald
4	225797	3	96	Tartu vald
5	228267	8	96	Tartu vald
6	229386	1	96	Tartu vald
7	231709	5	96	Tartu vald
8	231710	6	96	Tartu vald
9	231904	10	96	Tartu vald
10	231905	11	96	Tartu vald
11	238639	1	96	Tartu vald
12	238640	2	96	Tartu vald
13	241460	3	96	Tartu vald
14	241710	6	96	Tartu vald
15	245647	40	96	Tartu vald
16	245705	1	96	Tartu vald

Joonis 25. Näide PostgreSQL haldusvahendist PgAdmin, kus on näha ka päringuga tagastatud andmed.

Päring tagastab tabeli (vt. joonis 25), kus on veerud metsaeraldise *ID*, metsaeraldise number, valla *ID* ja valla nimi ning tabeli iga rida tähistab metsaeraldist, mis Tartu valda kuulub. Iga andmebaas kasutab funktsiooni *ST_Intersects* natuke erinevalt ja erinevad on ka tolerantsid ning sellest tulenevalt on ka iga andmebaasi tagastatud metsaeraldiste arv erinev (vastavalt 7585, 7587 ja 7586). Tolerants antud kontekstis

määrab kattuvuse alampiiri, st mis piirist loetakse veel objektid kattuvaks ja millest mitte. Oracle tolerants on 0.005 ja see tähendab, et kui kattuvus kahe objekti vahel on väiksem kui 5m, siis seda eiratakse. Märkuseks veel, et PostgreSQL andmebaasis olid *ID* väärtused 2 võrra nihkes ning seepärast oli Tartu vald *ID* väärtusega 96.

Tabelis 9 on toodud indeksiga ja indeksita päringute kümne korra keskmine ja standardhälve.

Andmebaas	**Indeksiga (10 korra keskmine)**	**Standardhälve**	**Indeksita (10 korra keskmine)**	**Standardhälve**	**Vahe**
PostgreSQL	1.6 s	7.97	114 s	0.57	71x
Oracle	3.2 s	0.01	1962 s	7.54	613x
SQL Server	2.0 s	0.03	371 s	0.39	186x

Tabel 9. Kattuvuspäringu „Anna mulle kõik Tartu valla metsaeraldised“ tulemused.

PostgreSQL oli antud päringu tulemuste tagastamisel kõige kiirem. Indeksita päringutes oli PostgreSQL üle 3 korra kiirem kui SQL Server ja 17 korda kiirem kui Oracle 11g. Indeksiga päringutes olid vahed väiksemad – PostgreSQL oli üle 1.2 korra kiirem kui SQL Server ja üle 2 korra kiirem kui Oracle 11g. Oracle andmebaas nõudis füüsilise indeksi olemasolu ka indeksita kattuvuspäringute rakendamisel erinevalt andmebaasidest PostgreSQL ja SQL Server, kus oli võimalik indeksid *DROP INDEX* käsuga eemaldada ja jooksutada päringut kasutamata indeksit. Oracle 11g indeksite kasutust saab juhtida spetsiaalse süntaksi (*index, no_index*) abil, mida piiritletakse plokk-kommentaari tähistega /* ja */. Kõige paremini iseloomustab geoindeksite efektiivsust aegade võrdlus indeksita ja indeksiga kattuvuspäringute rakendamisel. Indeksit kasutades tagastas PostgreSQL Tartu valla metsaeraldised 71 korda, SQL Server 186 korda ja Oracle 11g märkimisväärt 613 korda kiiremini.

Kattuvuspäring „Anna mulle kõik Tartu maakonda kuuluvad metsaeraldised“ täiendas varasemaid tulemusi. Kattuvuspäringu süntaks kõigi kolme andmebaasi jaoks on toodud tabelis 10.

Andmebaas	**Kattuvuspäringu süntaks**	**Tagastatud ridade arv**
PostgreSQL	*SELECT e.ogc_fid, e.eraldise_nr, m.ogc_fid,m.mnimi FROM rgis.maakonnad m, rgis.metsaeraldised e where*	79568

	ST_Intersects(e.wkb_geometry, m.wkb_geometry) and m.ogc_fid=8;	
Oracle (indeksiga)	*SELECT /*+ordered*/ e.ogr_fid,e."eraldise_nr",m.ogr_fid,m."mnimi" FROM maakonnad m, metsaeraldised e where ST_Intersects(e.ora_geometry, m.ora_geometry)='TRUE' and m.ogr_fid=8;*	79569
Oracle (indeksita)	*SELECT /*+ordered no_index(e metsaeraldised_gidx) no_index(m maakonnad_gidx) */ e.ogr_fid,e."eraldise_nr",m.ogr_fid,m."mnimi" FROM maakonnad m, metsaeraldised e where ST_Intersects(e.ora_geometry, m.ora_geometry)='TRUE' and m.ogr_fid=8;*	
SQL Server	*SELECT e.ogr_fid, e.eraldise_nr, m.ogr_fid,m.mnimi FROM rgis.dbo.metsaeraldised e, rgis.dbo.maakonnad m WHERE (e.ogr_geometry.STIntersects(m.ogr_geometry) = 1) and m.ogr_fid=8;*	79568

Tabel 10. Kattuvuspäring „Anna mulle Tartu maakonna kõik metsaeraldised“.

Ajalised tulemused on esindatud tabelis 11 kümne korra keskmisena ning eraldi on välja toodud ka standardhälve.

Andmebaas	**Indeksiga (10 korra keskmine)**	**Standardhälve**	**Indeksita (10 korra keskmine)**	**Standardhälve**	**Vahe**
PostgreSQL	15.6 s	0.26	20 s	0.04	1.3x
Oracle	580 s	23.55	15690 s	318.99	27.0x
SQL Server	161 s	3.93	4203 s	102.85	26.1x

Tabel 11. Kattuvuspäringu „Anna mulle Tartu maakonna kõik metsaeraldised“ tulemused.

PostgreSQL oli jätkuvalt kiire, sest indeksiga ja indeksita päringutes oli vahe vähem kui 5 sekundit. PostgreSQL oli indeksiga päringutes SQL Serverist ligikaudu 10.3 korda ja Oraclest 37.2 korda kiirem. Indeksita päringutes olid vahed suuremad – PostgreSQL oli üle 210 korra kiirem kui SQL Server ja üle 784 korra kiirem kui

Oracle. Kiiruse vahe tuleb hästi välja indeksiga ja indeksita päringutes – PostgreSQL tagastas indeksit kasutades tulemuse 1.3 korda, SQL Server 26.1 korda ja Oracle 27.0 korda kiiremini. PostgreSQL oli indeksita päringutes, võrreldes teiste andmebaasidega, üle 200 korra kiirem ning seetõttu proovisin mitmeid erinevaid stsenaariume veel, mis kinnitasid, et saadud ajalised tulemused pidasid paika[6]. Tulemused on toodud tabelis 12.

Koostatud päringud olid alljärgnevad:

1) **„Anna mulle kõik Jõgeva maakonna metsaeraldised“** - *SELECT e.ogc_fid, e.eraldise_nr, m.ogc_fid,m.mnimi FROM rgis.maakonnad m, rgis.metsaeraldised e where ST_Intersects(e.wkb_geometry, m.wkb_geometry) and m.ogc_fid=4;*

2) **„Anna mulle kõik Tartu maakonna metsaeraldised“** - *SELECT e.ogc_fid, e.eraldise_nr, m.ogc_fid,m.mnimi FROM rgis.maakonnad m, rgis.metsaeraldised e where ST_Intersects(e.wkb_geometry, m.wkb_geometry) and m.ogc_fid=8;*

3) **„Anna mulle kõik Tartu ja Jõgeva maakonna metsaeraldised“** - *SELECT e.ogc_fid, e.eraldise_nr, m.ogc_fid,m.mnimi FROM rgis.maakonnad m, rgis.metsaeraldised e where ST_Intersects(e.wkb_geometry, m.wkb_geometry) and m.ogc_fid in(4,8);*

4) **„Anna mulle kõik Tartu ja Jõgeva maakonna metsaeraldised“** - *SELECT e.ogc_fid, e.eraldise_nr, m.ogc_fid,m.mnimi FROM rgis.maakonnad m, rgis.metsaeraldised e where ST_Intersects(e.wkb_geometry, m.wkb_geometry) and (m.ogc_fid=4 or m.ogc_fid=8).*

Stsenaaruumi number	**Tulemus (10 korra keskmine)**
1	Aeg: 11.6 s. Ridade arv: 91005.
2	Aeg: 20.6 s. Ridade arv: 79568.
3	Aeg: 733 s. Ridade arv: 170573.
4	Aeg: 732 s. Ridade arv: 170573.

Tabel 12. PostgreSQL stsenaariumite tulemused.

Kahes esimeses stsenaariumis tehakse täielik tabeli läbimine vaid metsaeraldiste tabelis, kahes viimases tehakse täielik tabeli läbimine mõlemas tabelis (metsaeraldised, maakonnad). Arvasin esialgu, et PostgreSQL käitub operaatoriga

[6] Päringud ei kasutanud geoindeksit, indeks oli vaid *ID* (peavõtme) veerul.

IN ja *OR* erinevalt, aga nagu tulemustes näha, siis vahe oli üsna minimaalne. Võimalik, et PostgreSQL kiirus tuleneb GEOS paketi eripärast. Isiklikult arvan, et PostgreSQL loob koopiaid objektidest mälus ning oskuslikult on ära kasutatud C-keele paindlikku mäluhaldust.

Et tulemusi veelgi ilmestada, koostasin päringud, kus operatsioonide arv oli üle 11 miljoni (8 korda läbiti metsaeraldiste tabelit). Kattuvuspäringu „Anna mulle 8 maakonna kõik metsaeraldised“ süntaksid on toodud tabelis 13.

Andmebaas	**Kattuvuspäringu süntaks**	**Tagastatud ridade arv**
PostgreSQL	*SELECT e.ogc_fid, e.eraldise_nr, m.ogc_fid,m.mnimi FROM rgis.maakonnad m, rgis.metsaeraldised e where ST_Intersects(e.wkb_geometry, m.wkb_geometry) and m.ogc_fid in(1,2,3,4,5,6,7,8);*	652683
Oracle (indeksiga)	*SELECT /*+ordered*/ e.ogr_fid,e."eraldise_nr",m.ogr_fid,m."mnimi" FROM maakonnad m, metsaeraldised e where ST_Intersects(e.ora_geometry, m.ora_geometry)='TRUE' and m.ogr_fid in(1,2,3,4,5,6,7,8);*	-
Oracle (indeksita)	*SELECT /*+ordered no_index(e metsaeraldised_gidx) no_index(m maakonnad_gidx) */ e.ogr_fid,e."eraldise_nr",m.ogr_fid,m."mnimi" FROM maakonnad m, metsaeraldised e where ST_Intersects(e.ora_geometry, m.ora_geometry)='TRUE' and m.ogr_fid in(1,2,3,4,5,6,7,8);*	
SQL Server	*SELECT e.ogr_fid, e.eraldise_nr, m.ogr_fid,m.mnimi FROM rgis.dbo.maakonnad m, rgis.dbo.metsaeraldised e WHERE (e.ogr_geometry.STIntersects(m.ogr_geometry) = 1) and m.ogr_fid in(1,2,3,4,5,6,7,8)*	652683

Tabel 13. Kattuvuspäring „Anna mulle 8 maakonna kõik metsaeraldised“.

Antud päringu tulemused on toodud tabelis 14.

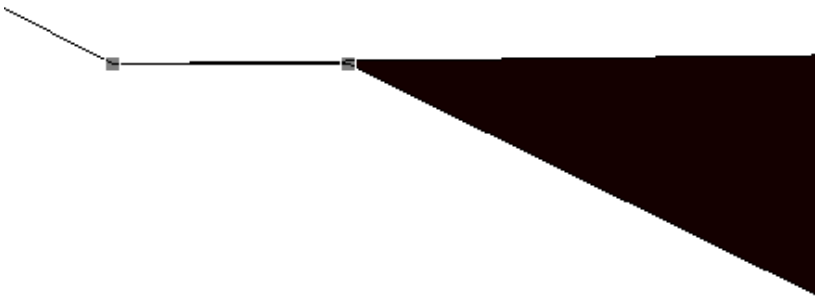
Andmebaas	Indeksiga (10 korra keskmine)	Standardhälve	Indeksita (10 korra keskmine)	Standardhälve	Vahe
PostgreSQL	511 s	4.88	9055 s	100.04	17.7x
Oracle	-	-	-	-	-
SQL Server	4646 s	103.18	81456 s	177.51	17.5x

Tabel 14. Kattuvuspäringu „Anna mulle 8 maakonna kõik metsaeraldised“ tulemused.

Indeksita päringutes oli PostgreSQL 9 korda kiirem kui SQL Server. Indeksiga päringutes oli PostgreSQL üle 9.1 korra kiirem kui SQL Server 2012. Indeksit kasutades tagastas PostgreSQL 8 maakonna metsaeraldised 17.7 korda ja SQL Server 17.5 korda kiiremini.

Oracle andmebaasi tulemused puuduvad, sest korduvalt käivitatud päring töötas 3-7 päeva ning tagastas kas veateate indeksist või sai Oracle andmebaasile eraldatud mälu (8 - 12GB) otsa. Oracle indeksi veateade nägi välja alljärgnev: ORA-29903: error in executing ODCIIndexFetch() routine ORA-13236: internal error in R-tree processing. Veateade mälu puudulikkusest oli järgnev: ORA-04030: out of process memory when trying to allocate 16396 bytes (koh-kghu call ,pmuccst: adt/record).

Probleemi põhjuseks on metsaeraldiste tabelis geomeetriad, milles esinesid vead (iseendaga lõikumine, korduvad koordinaadid vms.). Oracle andmebaasi enda valideerimise funktsioon leidis kümmekond sellist geomeetriat ja need said parandatud, kuid hoolimata sellest veateated ei kadunud. Eemaldasin andmebaasist kõik päringutega seotud tabelid (k.a seotud süsteemsed tabelid) ning laadisin need uuesti puhastatud andmebaasi ning lõin ka kõik vajalikud indeksid. Probleemid ei kadunud ja järgmiseks otsustasin hakata geomeetriaid ükshaaval parandama. Vigade otsimiseks kasutasin AS Regio arendatud tarkvara VisualGIS, millega on lihtne topoloogilisi vigu leida ja parandada.



Joonis 26. Üks näide geomeetrias esinevast veast, mille põhjuseks on siinkohal kiil.

Sellised kiilud (vt. joonis 26) tekitavad operatsioonides lõputut tsüklit, sest geomeetriat läbitakse päripäeva ning kui on jõutud kiiluni, siis sisenetakse kiilu ja algoritm jääbki kiilu pendeldama. Eelnev tõstab andmebaasihaldussüsteemi mälukasutust ja lubatava mäluhulga lõppedes jookseb haldussüsteem kokku. Vigade otsimine (umbes kuu jooksul) muutus liiga ajakulukaks, sest lõpuks tuli hakata päringuid sooritama ühe metsaeraldise kaupa (leidmaks kõikvõimalikke vigaseid geomeetriaid) ja metsaeraldisi oli üle 1.4 miljoni. PRIA projekti esindajalt sain teada, et nemad on saatnud vigased geomeetriad otse Oracle korporatsioonile parandamiseks, kuid ka see variant ei sobinud ajaliselt. Lõpuks võtsin vastu otsuse, et ei suuda Oracle tulemusi antud töös esitada.

Kokkuvõtvalt võib testimistulemuste põhjal öelda, et antud kattuvuspäringute sooritamisel oli PostgreSQL kõige efektiivsem, mis ei tähenda, et mõnes teises olukorras oleks ta sama kiire. Nende tulemuste alusel ei saa ka öelda, et SQL Server 2012 või Oracle 11g on aeglased, vaid töö raames valitud stsenaariumites ei suutnud nad PostgreSQL kiirusele samaga vastata. PostgreSQL ja SQL Server 2012 olid tolereerivamad vigaste geomeetriate suhtes ja suutsid alati mõistliku aja jooksul tulemused tagastada. Täpsemalt aga ei oska öelda, kas nad jätsid vigased geomeetriad vahele tulemusest või üritasid ikkagi vigase objektiga tulemust anda. Eelneva ning GEOS paketi detailsema uurimise jätan edaspidiseks.

Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli uurida andmebaasi indeksite ja geoindeksite andmestruktuure ja need esitleda emakeeles. Ühelt poolt tegin seda leiduvate materjalide vähesuse tõttu ja teiselt poolt, et tutvustada inimestele, kuidas indeksid andmebaasihaldussüsteemides töötavad. Kolmandaks põhjuseks tooksin enda huvi ja neljandaks soovi luua õppematerjal antud teemast huvituvale inimesele.

Antud töö esimeses osas vaadeldi andmebaase ja indekseid. Pikemalt keskenduti iga indeksitüübi andmestruktuurile. Eraldi peatuti ka tööpõhimõtetel ja toodi juhiseid indeksite kasutamiseks.

Teises osas vaadeldi lähemalt geoandmebaase ja geoindekseid. Antud peatükis kirjeldati põhjalikult ka iga indeksitüübi andmestruktuure ja viimaste paremaks mõistmiseks lisati ilmekad illustratsioonid.

Bakalaureusetöö kolmandas peatükis võrreldi kolme andmebaasihaldussüsteemi opereerimist kattuvuspäringute sooritamisel ja indeksite mõju tulemustele. Võrreldi ka ajalist erinevust indeksiga ja indeksita päringute vahel, tehti põhjalik analüüs ning võeti tulemused kokku.

Püstitatud eesmärgid said bakalaureusetöö lõpuks täidetud. Sain isiklikult palju uusi teadmisi juurde ning mõistan andmebaaside käitumist paremini. Enda jaoks huvitavaid tulemusi sain just jõudluskatsetest indekseeritud ja indekseerimata andmete vahel. Rahule jäin ka sisuga, sest tööd saab kasutada õppematerjalina ka täiesti teemavõõras inimene, sest usun, et see on piisava detailsusega, ent arusaadav.

Antud bakalaureusetöö edasiarendamiseks on võimalik uurida PostgreSQL, PostGIS ja GEOS paketi lähtekoodi ning leida põhjus tulemustes kajastunud kiirusele. Eesmärgiks lähitulevikus on parandada metsaeraldiste vead Oracle andmebaasis ning leida töös kajastamata jäänud tulemused.

Geodatabases and indices

Bachelor Thesis (6 EAP)

Ragnar Ziugand

Summary

The goal of the thesis was to examine data structures of indices and geoindices and to present it in my mother tongue. On one hand, I did it because of the scarcity of material and on the other hand to give people better understanding how indices work in database management systems.

The first chapter of the thesis focused on databases and indices. Data structures of each index type were described in more detail. In addition the operating principles of indices were introduced and tips how to use indices correctly were given.

The second chapter focused on geodatabases and geoindices. Data structures of these index types were introduced and lots of expressive illustrations were included.

The third chapter compared three database management systems on spatial queries and described how indices influenced these results. The comparison between the time and speed of indexed data and non-indexed data was presented. Results were analyzed and summarized.

For future work it is possible to research the source code of PostgreSQL, PostGIS and GEOS package and to find the cause of the speed reflected in the results.

Kasutatud kirjandus

[CMR11] Coronel, Carlos, Steven Morris, and Peter Rob. Database systems: design, implementation, and management 9th ed.. Boston, Mass.: Course Technology, 2009.

[CC05] Connolly, Thomas M., and Carolyn E. Begg. Database systems: a practical approach to design, implementation, and management. 4. ed. Boston, Mass. Addison-Wesley, 2005.

[EN10] Ramez Elmasri and Shamkant Navathe. Fundamentals of Database Systems (6th ed.). Addison-Wesley Publishing Company, 2010.

[Nes08] Härmel Nestra loengumaterjalid paisktabelitest. http://www.cs.ut.ee/~nestra/mat/inf/p/aa/08s/s/paisk.pdf (15.07.2013)

[CLRS09] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. 3rd ed. Cambridge, Mass.: MIT Press, 2009.

[MyS] MySQL documentation, Spatial Extensions. http://dev.mysql.com/doc/refman/5.0/en/spatial-extensions.html (15.07.2013)

[RSV02] Rigaux, Philippe, Michel O. Scholl, and Agnes Voisard. Spatial databases with application to GIS. San Francisco: Morgan Kaufmann Publishers, 2002.

[SC03] Shekhar, Shashi, and Sanjay Chawla. Spatial databases: a tour. Upper Saddle River, N.J.: Prentice Hall, 2003.

[BKSS90] Beckmann, N.; Kriegel, H. P.; Schneider, R.; Seeger, B. (1990). "The R*-tree: an efficient and robust access method for points and rectangles".Proceedings of the 1990 ACM SIGMOD international conference on Management of data - SIGMOD '90.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In VLDB, 1987.

[OGC] Open Geospatial Consortium, Simple Feature Access - Part 1: Common Architecture. http://www.opengeospatial.org/standards/sfa (15.07.2013)

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. Generalized Search Trees for Database Systems. Proc. 21st Int'l Conf. on Very Large Data Bases, Zürich, September 1995.

Lisad

Lisa 1. *Metsaeraldiste import ja eksport programmiga ogr2ogr.*

Metsaeraldiste importimiseks vajalikud käsud nägid kolme andmebaasi jaoks välja alljärgnevad:

1) *ogr2ogr -f "PostgreSQL" -a_srs "EPSG:3301" PG:"host=localhost port=5432 dbname=rgis user=kasutaja password=parool" "C:\TEMP\eraldised_ERA_RMK.tab" -T_SRS EPSG:3301 -nln rgis.metsaeraldised -append –progress;*

2) *ogr2ogr -append -update -f OCI OCI:kasutaja/parool@orcl -lco DIM=2 -lco SRID=3301 -nln metsaeraldised "C:\TEMP\eraldised_ERA_RMK.tab";*

3) *ogr2ogr -append -update -f "MSSQLSpatial" "MSSQL:server=localhost;database=rgis;trusted_connection=yes;" -lco DIM=2 -lco SRID=3301 -nln metsaeraldised "C:\TEMP\eraldised_ERA_RMK.tab" –progress.*

Metsaeraldiste eksportimiseks vajalikud käsud kolme andmebaasi jaoks olid alljärgnevad:

1) *ogr2ogr -f "Mapinfo File" "C:\TEMP\eraldised_ERA_RMK.tab" PG:"host=localhost port=5432 dbname=rgis user=kasutaja password=parool" -sql "select * from rgis.eraldised" -a_srs EPSG:3301 -nln eraldised -nlt MULTIPOLYGON POLYGON*

2) *ogr2ogr -f "Mapinfo File" "C:\TEMP\eraldised_ERA_RMK.tab" OCI:kasutaja/parool@orcl -sql "select * from eraldised" -a_srs EPSG:3301 -nln eraldised -nlt MULTIPOLYGON POLYGON*

3) *ogr2ogr -f "Mapinfo File" "C:\TEMP\eraldised_ERA_RMK.tab" "MSSQL:server=localhost;database=rgis;trusted_connection=yes;" -sql "select * from dbo.eraldised" -a_srs EPSG:3301 -nln eraldised -nlt MULTIPOLYGON POLYGON.*

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Ragnar Ziugand (sünnikuupäev: 13.06.1989)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Geoandmebaasid ja indeksid“, mille juhendaja on Sven Laur,

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace'i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **15.08.2013**