Tietokannan indeksointi: B-puun ja hajautusindeksin tehokkuus

Tuomas Kortelainen

28.4.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Tässä tutkielmassa esitellään tietokannan indeksointimenetelmiä. Indeksointia käytetään nopeuttamaan tietokannan toimintaa, erityisesti hakuja. Tutkielmassa tarkastellaan kahta tietokannan indeksointimenetelmää: B-puuta ja hajautusindeksiä. B-puu on laajalti käytetty puumainen indeksointirakenne. B-puussa indeksoitavien tietueiden avaimet on sijoitettu puun lehtiin. Tässä tutkielmassa tarkastellaan B^+-puuta, joka on B-puun yleinen esitysmuoto. Tämän lisäksi tarkastellaan yhtäaikaisen käytön mahdollistavaa B^{link}-puuta, jota käytetään myös PostgreSQL-tietokannassa. Hajautusindeksi perustuu hajautustauluun ja hajautusfunktioon. Indeksoitavat tietueet hajautetaan hajautustauluun hajautusfunktion avulla. Tässä tutkielmassa tarkastellaan kahta dynaamisen hajautuksen menetelmää, laajentavaa ja lineaarista hajautusta, joilla hallitaan hajautustaulun koon muuttamista. Kumpaakin tässä tutkielmassa esiteltyä indeksointimenetelmää voidaan käyttää PostgreSQL-tietokannassa, jossa kuitenkin B-puu on suositelluin ja tehokkain indeksointimuoto. Tutkielmassa verrataan B-puun ja hajautusindeksin tehokkuutta PostgreSQL-tietokannassa käyttäen apuna tuotantokäytössä olevaa tietokantaa ja siihen kohdistettuja funktioita. Tutkimuksessa huomattiin B-puun ja hajautusindeksin olevan joiltain osin hyvin tasaveroisia. Mutta kuitenkin esimerkiksi merkkijonoihin liittyvissä like-menetelmällä tehdyissä kyselyissä ja numeerisissa arvoalue-hauissa B-puun huomattiin olevan hajautusindeksiä tehokkaampi indeksointimenetelmä.

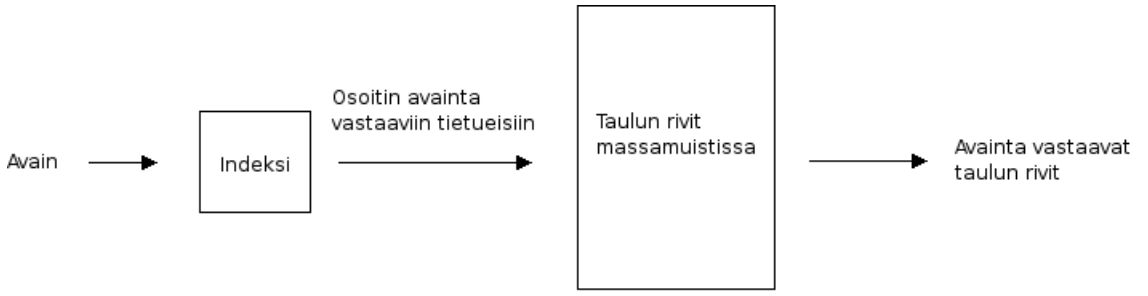
ACM-luokat: (ACM Computing Classification System, 1998 version): G.4, H.2.2, H.2.4, H.3.4

Avainsanat: indeksointi, B-puu, hajautus, tietokanta, PostgreSQL, tehokkuus

Sisällysluettelo

1 Johdanto

Tietokannan *indeksoinnilla* tarkoitetaan menetelmää, jolla tietokannan toimintaa pyritään nopeuttamaan luomalla tietokannan taululle *indeksointirakenne* jonkin taulun sarakkeen tai sarakkeiden perusteella. Indeksi on *tietorakenne*, joka saa tyypillisesti syötteekseen yhden tai useamman tietokannan taulun kentän ja palauttaa kenttiä vastaavat tietueet nopeasti. Taulun saraketta, jonka perusteella taulu on indeksoitu, kutsutaan indeksin *avaimeksi*. Kuvassa 1 on esitetty indeksoinnin perusidea, jossa voidaan avaimen perusteella hakea indeksistä tietueen massamuistiosoitin (Garcia-Molina et al. 2002).



Kuva 1. Indeksoinnin perusidea

Tietokannan indeksoinnin suurimpana etuna pidetään tietokantahakujen nopeutumista. Indeksin rakenteesta riippuen muutosten tekeminen indeksoituun tauluun saattaa hidastua huomattavasti. Tästä johtuen indeksointia tulisi käyttää erityisesti sellaisissa tietokannan tauluissa, joihin kohdistuu paljon hakuja ja vähän muutoksia (Garcia-Molina et al. 2002).

Tässä tutkielmassa esitellään seuraavat kaksi PostgreSQL-tietokannan mahdollistamaa indeksointimenetelmää sekä annetaan esimerkkejä menetelmien käyttämisestä[1]:

1 PostgreSQL-tietokanta mahdollistaa myös R-puun indeksoitaessa moniulotteista tietoa (PostgreSQL, 2007; Kortelainen, 2007). PostgreSQL-tietokannassa voidaan myös käyttää Gist-indeksiä (PostgreSQL, 2007).

- *B-puu* on yleisesti käytetty ja tehokas indeksointimenetelmä. Tämän tutkielman toisessa luvussa esitellään B^+-puun rakenne ja yleisimmät toiminnot. Lisäksi tarkastellaan B^{link}-puuta, joka mahdollistaa useamman prosessin yhtäaikaisen käytön.
- *Hajautusindeksi* on B-puun ohella toinen hyödyllinen ja tärkeä perusindeksointimenetelmä. Tämän tutkielman kolmannessa luvussa esitellään yleisesti hajautusindeksi. Luvussa tarkastellaan myös kahta hajautusindeksin tehokasta toteutustapaa: laajentavaa ja lineaarista hajautusta.

Luvuissa 2 ja 3 esitetään hypoteeseja liittyen tarkasteltujen menetelmien tehokkuuteen. Hypoteesit perustuvat PostgreSQL:n toimittamiin käsityksiin ja tietokantakirjallisuuteen. Luvussa 4 on tarkoitus tarkastella näiden hypoteesien paikkansapitävyyttä vertailemalla B-puun ja hajautusindeksin tehokkuutta PostgreSQL-tietokannassa.

Tutkimuksessa käytetään Fastroi Oy:n Asukastieto-ohjelma Hilkan käyttämää tietokantaa[2]. Testausohjelmissa käytetään myös apuna Hilkka-ohjelmassa valmiina olevia tietokantarajapintoja. Tässä tutkimuksessa keskitytään Hilkka-ohjelman sisältämään Lääkelaitoksen toimittamaan lääkelistaan, jonka kesäkuun 2007 versio sisältää 47 322 lääketietuetta.

Hilkka-ohjelmassa Lääkelaitoksen listaan kohdistuneet tietokantahaut ovat olleet hitaita. Niinpä tässä tutkimuksessa pyritään myös löytämään joitain apukeinoja lääkelistan käytön nopeuttamiseen. Lääkelaitos toimittaa joka kuukausi uuden päivitetyn listan, jonka päivitys Hilkka-ohjelmaan on vienyt tietokoneen nopeudesta riippuen puolesta tunnista tuntiin. Päivitys suoritetaan siten, että tietokannasta poistetaan kaikki käyttämättömät lääkkeet ja tämän jälkeen uudet lääkkeet lisätään tilalle. Tarkoituksena on myös tutkia, millaisia vaikutuksia indeksoinnilla on lääkelistan päivitykseen Hilkka-ohjelmassa.

2 Tutkimukseen on Fastroi Oy:ltä saatu lupa käyttää Asiakastieto-ohjelman Hilkka 2 -versiota.

Luvun 4 tutkimukset on tehty Java-kielisellä ohjelmalla (liite 1), jonka sisältämiä funktioita esitellään myös luvussa 4. Tutkimuksessa suoritettuja tietokantahakuja satunnaisilla syötteillä on pyritty toistamaan riittävän monta kertaa, jotta esimerkiksi kiintolevyn ja tietokannan välimuistin käyttö ei voi vaikuttaa tutkimuksen tulokseen. Koska hauissa käytetyt syötteet generoidaan satunnaisesti, niin mitä enemmän testikierroksia ajetaan, sitä vähemmän testien väillä on satunnaisuudesta johtuvia eroja.

Luvussa 5 tehdään yhteenveto käsitellyistä indeksointirakenteista ja tehdyn tutkimuksen tuloksista. Tutkimuksessa huomattiin B-puun ja hajautusindeksin olevan joiltain osin hyvin tasaveroisia. Tutkimuksessa kävi myös ilmi, että B-puu on monissa tapauksissa hajautusindeksiä tehokkaampi.

2 B-puu

B-puu on nykyään yksi käytetyimmistä indeksointirakenteista (Garcia-Molina et al. 2002; Connolly ja Begg, 2005; Elmasri ja Navathe, 2007). B-puun etuna on erittäin hyvä hakujen nopeutus. Garcia-Molina et al. (2002) mainitsevat B-puulle myös seuraavat ominaisuudet:

- B-puu toteutetaan käyttämällä binääripuumaista rakennetta
- B-puu on tasapainotettu ja se pitää automaattisesti yllä tasojen lukumäärää
- B-puun solmu sisältää tavallisesti useita avaimia
- B-puun solmuissa olevien avainten määrälle on asetettu rajoitus, jonka mukaan solmun tulee olla ainakin puoliksi täynnä
- B-puussa ei tarvita ylivuotosolmuja.

Seuraavaksi tarkastellaan B^+-puuta, joka on yleisin B-puun rakenne, sekä PostgreSQL-tietokannassa käytettyä B^{link}-puuta. Muita tulkintoja B-puulle ovat esittäneet esimerkiksi Comer (1979), Elmasri ja Navathe (2007) sekä Silberschatz et al. (1997).

2.1 B^+-puun rakenne

B^+-puun solmut voidaan jakaa kolmeen luokkaan: *juuri*, *välisolmut* ja *lehdet*. Kuvassa 4 on esimerkki täydellisestä B^+-puusta. B^+-puu on tasapainotettu, eli kaikki polut puun juuresta lehtiin ovat saman pituisia (Garcia-Molina et al. 2002).

Jokaiseen B^+-puuhun liittyy attribuutti *n*, joka ilmoittaa yhteen puun solmuun mahtuvien avainten määrän. Jokaisessa solmussa on myös *n*+1 paikkaa osoittimille. Garcia-Molina et al. (2002) suosittavat valitsemaan attribuutin *n* niin suureksi kuin mahdollista, kuitenkin niin, että yksi täysi solmu mahtuu yhden muistilohkon sisään. Näin puu saadaan toi-

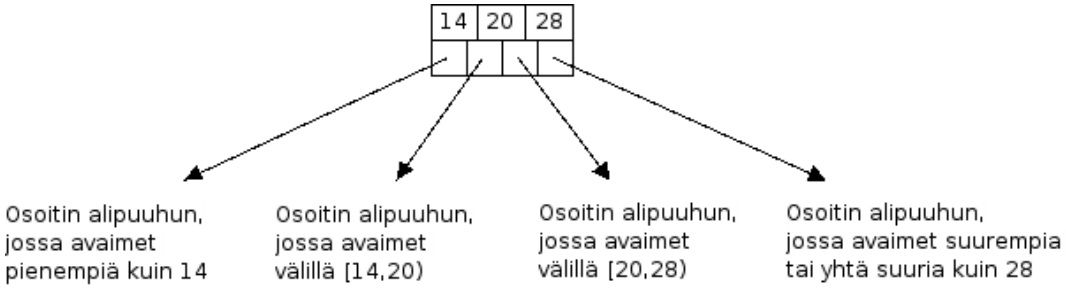
mimaan mahdollisimman tehokkaasti.

Garcia-Molina et al. (2002) esittävät B^+-puun solmuille seuraavia sääntöjä:

- Jokaiseen datatietueessa olevaan tietueeseen liittyy avain. Kaikki nämä avaimet ovat B^+-puun lehdissä nousevassa järjestyksessä vasemmalta oikealle. Jokaiseen lehdessä olevaan avaimeen liittyy osoitin, joka osoittaa datatiedostossa olevaan tietueeseen. Lehtisolmun viimeinen osoitin osoittaa lehden oikeanpuoleiseen naapurilehteen. Puun oikeanpuoleisimman lehtisolmun viimeinen osoitin ei osoita mihinkään.
- Jokaisessa lehtisolmussa pitää olla ainakin $\lfloor (n+1)/2 \rfloor$ osoitinta datatietueisiin.
- Puun välisolmussa voi olla kaikkiaan $n+1$ osoitinta, jotka osoittavat puun seuraavassa kerroksessa oleviin solmuihin. Välisolmuissa tulee ainakin $\lceil (n+1)/2 \rceil$ osoitinta olla käytössä.
- B^+-puun välisolmuista poiketen puun juuressa sallitaan ainoastaan yksi avain ja kaksi osoitinta, jotka osoittavat puun seuraavaan kerrokseen.
- Puun juuren ja välisolmujen avaimille ja osoittimille pätevät seuraavat säännöt: Olkoon solmussa i kappaletta avaimia, K_1, K_2, ..., K_i ja $i+1$ kappaletta osoittimia. Solmun ensimmäisen osoittimen osoittamasta alipuusta löytyvät kaikki avaimet K, joille pätee $K < K_1$. Toisen osoittimen osoittamasta alipuusta löytyvät kaikki avaimet K, joille pätee $K_1 \leq K < K_2$. Solmun $i+1$:n osoittimen osoittamasta alipuusta löytyvät kaikki avaimet K, joille pätee $K_i \leq K$.

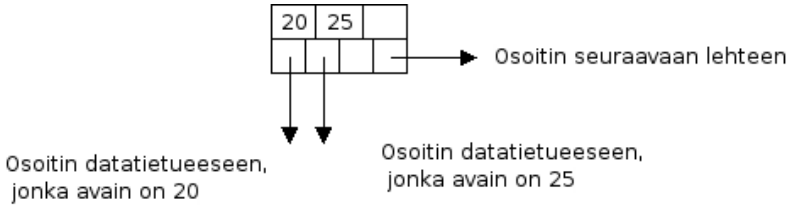
Esimerkki 1. Kuvassa 2 on esitetty B^+-puun solmu, joka voi olla joko puun juuri tai välisolmu. Solmun koko $n = 3$ on valittu esimerkin vuoksi pieneksi. Yksinkertaisuuden vuoksi myös solmun avaimet ovat kokonaislukuja. Solmu on täysi, eli se sisältää n avainta 14, 20 ja 20. Solmun ensimmäinen osoitin osoittaa alipuuhun, jonka kaikki avaimet ovat pienempiä kuin ensimmäinen avain 14. Solmun toinen osoitin osoittaa alipuuhun, jonka

kaikki avaimet ovat väliltä [14,20), ja kolmas osoitin alipuuhun, jonka kaikki avaimet ovat väliltä [20,28). Solmun viimeinen osoitin osoittaa alipuuhun, jonka kaikki avaimet ovat suurempia tai yhtä suuria kuin 28.



Kuva 2. Välisolmun rakenne

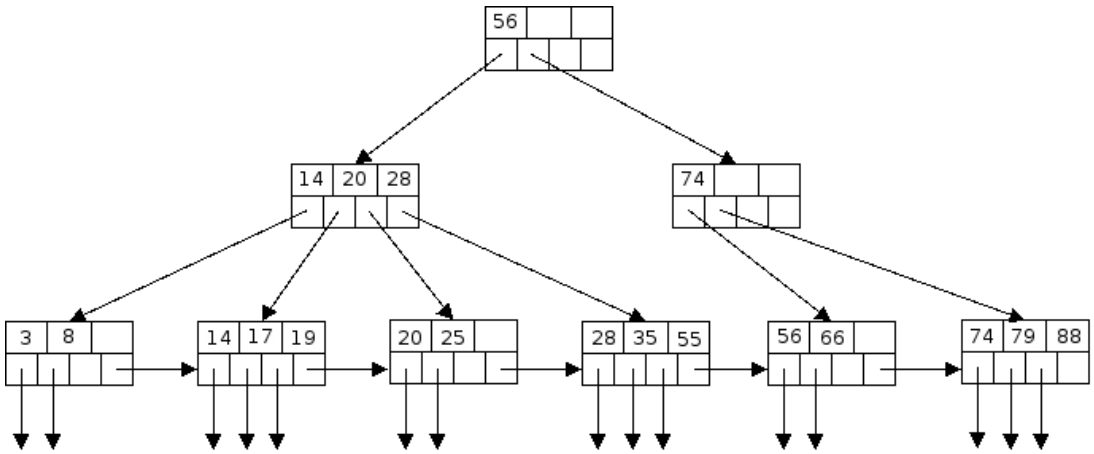
Esimerkki 2. Kuvassa 3 on esitetty B^+-puun lehtisolmu. Lehtisolmussa on kaksi avainta 20 ja 25. Solmun ensimmäinen osoitin osoittaa datatietueeseen, jonka avain on 20. Solmun toinen osoitin osoittaa datatietueeseen, jonka avain on 25. Solmun viimeinen osoitin osoittaa puun seuraavaan lehteen. Jos kyseinen lehti olisi puun oikeanpuoleisin, olisi osoitin tyhjä.



Kuva 3. Lehtisolmun rakenne

Esimerkki 3. Kuvassa 4 on esitetty pieni täydellinen B^+-puu (Garcia-Molina et al. 2002). Puun solmujen koko $n = 3$ ja solmuissa olevat avaimet ovat kokonaislukuja. Yksinkertaisuuden vuoksi puussa on sama avain vain kerran. Puun juuressa on avain 56, joten juuren ensimmäinen osoitin osoittaa alipuuhun, jonka kaikki avaimet ovat pienempiä kuin 56. Juuren toinen osoitin osoittaa taas alipuuhun, jonka avaimet ovat suurempia tai yhtä suu-

ria kuin avain 56. Puussa on kaksi välisolmua. Ensimmäisessä välisolmussa ovat osoittimet puun neljään ensimmäiseen lehteen. Toisessa välisolmussa ovat osoittimet puun kahteen viimeiseen lehteen. Kaikkien datatiedostoissa olevien tietueiden avaimet löytyvät puun lehdistä. Lehdissä olevat avaimet ovat nousevassa järjestyksessä vasemmalta oikealle. Puun lehtien viimeinen osoitin osoittaa puussa seuraavana olevaan lehteen. Tämä mahdollistaa esimerkiksi arvovälikyselyt, joita on käsitelty kohdassa 2.2.



Kuva 4. B⁺-puu

2.2 Tiedon hakeminen B⁺-puusta

Haettaessa B⁺-puusta tietuetta *hakuavaimella K* haku aloitetaan puun juuresta ja edetään lehtiin. Juuressa ja puun välisolmuissa on kussakin korkeintaan n kappaletta hakuavaimia K_1, K_2, ..., K_n. Jos hakuavaimelle K on voimassa $K < K_1$ niin hakua jatketaan solmun ensimmäisestä lapsesta. Jos $K_1 \leq K < K_2$, niin siirrytään tarkastelemaan solmun toista lasta jne.

Kun haussa on edetty lehteen, tiedetään, että haetun tietueen on oltava kyseisessä lehdessä. Jos lehden i:s avain on K, niin lehden i:s osoitin osoittaa haettuun tietueeseen. Jos lehdessä ei ole avainta K, niin tietokannassa ei ole haettua tietuetta (Garcia-Molina et al.

2002).

Esimerkki 4. Haetaan kuvan 4 B^+-puusta tietuetta avaimella 19. Haku aloitetaan puun juuresta ja koska $19 < 56$, niin edetään juuren ensimmäiseen lapseen. Koska $14 \leq 19 < 20$, niin edetään välisolmun toiseen lapseen. Haku on nyt edennyt lehtisolmuun, josta löytyy avain 19. Seuraamalla avaimen osoitinta päästään käsiksi haettuun tietueeseen. Haettaessa samasta puusta tietuetta avaimella 24 päädytään lopulta puun kolmanteen lehteen. Kyseinen lehti ei sisällä avainta 24, joten tästä voidaan päätellä, ettei tietokannasta löydy avainta 24 vastaavaa tietuetta.

Koska B^+-puun hakuavaimet ovat lehdissä nousevassa järjestyksessä ja jokaisessa lehdessä on osoitin seuraavaan lehteen, B^+-puu mahdollistaa tehokkaat *arvovälikyselyt*. Arvovälikyselyllä voidaan hakea kaikki tietueet, joiden hakuavain on tietyllä välillä [*a*,*b*]. Tietokantaan tehty arvovälikysely voi olla esimerkiksi seuraavanlainen:

```
SELECT *
   FROM Asiakas
   WHERE Ika > =18 AND Ika <= 50;
```

Arvovälikysely välillä [*a*,*b*] aloitetaan hakemalla puusta avainta *a*. Tietuetta *a* ei välttämättä puusta löydy, mutta puun lehdistä löydetään kuitenkin paikka, jossa *a*:n tulisi olla. Tästä aloitetaan lehtien sisältämien avaintein selaaminen. Haku lopetetaan, jos lehdestä löytyy avain, joka on suurempi tai yhtä suuri kuin haettu avain *b*. Jos kaikki lehden avaimet ovat pienempiä kuin avain *b*, niin hakua jatketaan puun seuraavasta lehdestä. Seuraavaan lehteen päästään seuraamalla lehtien välillä olevaa osoitinta. Näin jatketaan, kunnes löydetään avain, joka on suurempi tai yhtä suuri kuin *b*, tai kun kaikki puun avaimet on käyty läpi.

B^+-puu mahdollistaa myös arvovälikyselyt, joissa *a* on $-\infty$ ja/tai *b* on ∞ (Garcia-Molina et

al. 2002). Jos *a* on -∞, niin haku aloitetaan ensimmäisen lehden ensimmäisestä avaimesta. B^+-puun ensimmäinen lehti on helppo löytää aloittamalla puun juuresta ja siirtymällä aina solmun ensimmäiseen lapseen. Jos *b* on ∞, haku aloitetaan normaalisti lehden avaimesta *a*. Tämän jälkeen haussa käydään läpi kaikki puun oikeanpuoleiset lehdet.

Esimerkki 5. Haetaan kuvan 4 B^+-puusta arvoja väliltä [10,20]. Haku aloitetaan hakemalla puusta avainta avaimella 10. Haussa päädytään ensimmäiseen lehteen. Lehdestä ei löydy avainta 10 eikä muitakaan avaimia, jotka olisivat suurempia kuin 10. Seuraavaksi siirrytään lehtien välisiä osoittimia pitkin puun toiseen lehteen. Kyseisestä lehdestä löydetään hakuehtoa vastaavat avaimet 14, 17 ja 19. Tämän jälkeen siirrytään kolmanteen lehteen. Tästä lehdestä löydetään avain 20, joka vastaa hakuehtoa. Seuraavana lehdessä on avain 25, joka on suurempi kuin 20, joten haku voidaan lopettaa.

Esimerkki 6. Haetaan kuvan 4 B^+-puusta arvoja väliltä (30,∞). Haku aloitetaan etsimällä puusta avainta 30, jolloin päädytään puun neljänteen lehteen. Kyseisestä lehdestä löytyvät avaimet 35 ja 55, jotka vastaavat hakuehtoa. Hakua jatketaan puun oikeanpuolimmaiseen lehteen lehtien välisiä osoittimia seuraten ja poimien kaikki avaimet.

2.3 Tietueen lisääminen B^+-puuhun

Lisättävää tietuetta vastaavan avaimen lisääminen puuhun aloitetaan etsimällä puusta oikea paikka avaimelle (Garcia-Molina et al. 2002). Tämä voidaan tehdä käyttämällä kohdassa 2.2 esitettyä hakumenetelmää. Menetelmä palauttaa sen puun lehden, jossa lisättävän avaimen tulisi olla.

Jos hakumenetelmän osoittamassa lehdessä on vapaata tilaa, voidaan avain lisätä siihen suoraan. Jos lehtisolmu on täynnä, on kyseinen solmu jaettava eli sen viereen on luotava uusi lehtisolmu. Alkuperäisessä solmussa olevat avaimet sekä uusi lisättävä avain jaetaan

solmujen kesken. Ensimmäiset $\lceil (n+1)/2 \rceil$ avainta pysyvät alkuperäisessä solmussa ja loput $\lfloor (n+1)/2 \rfloor$ avainta siirretään uuteen solmuun.

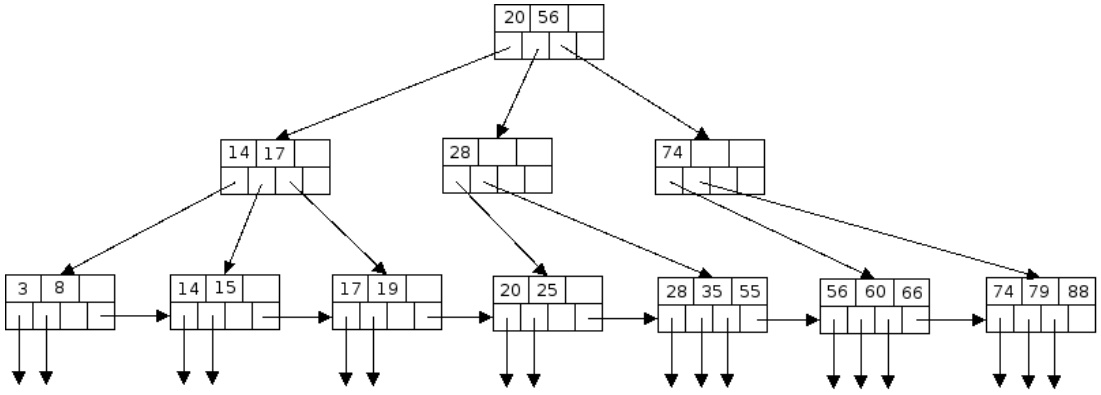
Jos lehtisolmu joudutaan jakamaan, pitää jaetun solmun isäsolmuun lisätä avain ja osoitin juuri luotuun uuteen solmuun. Jos isäsolmussa ei ole tilaa uudelle avaimelle, on se myöskin jaettava. Näin jatketaan rekursiivisesti, kunnes löydetään solmu, jota ei tarvitse jakaa. Jos puun juureen joudutaan lisäämään avain ja juuri on täynnä, on juuri jaettava ja puulle on lisättävä uusi juurisolmu. Näin siis puun korkeus kasvaa yhdellä. Uuteen juureen asetetaan avaimet ja osoittimet alkuperäisestä juuresta muodostuneisiin välisolmuihin. Tämän jälkeen puun uudella juurella on vain kaksi lasta. Tämä on poikkeava tilanne verrattuna puun muihin solmuihin, joiden täytyy olla vähintään puoliksi täysiä.

Puun välisolmun *N* jakaminen aloitetaan luomalla uusi solmu *M* solmun *N* oikealle puolelle. Solmun *N* $n+1$ avaimesta ensimmäiset (pienimmät) $\lceil (n+1)/2 \rceil$ kappaletta pidetään solmussa *N* ja loput $\lfloor (n+1)/2 \rfloor$ avainta siirretään solmuun *M*. Solmun *M* ensimmäistä (pienintä) avainta kutsutaan *vasemmanpuolimmaiseksi* avaimeksi *K*. *K* ilmaisee pienimmän avaimen, joka voidaan löytää solmusta *M*. Avain *K* nostetaan solmun *N* isäsolmuun erottamaan solmuja *N* ja *M*. Samalla isäsolmuun lisätään osoitin uuteen solmuun *M*.

Esimerkki 7. Lisätään kuvan 4 puuhun tietue, jonka avain on 60. Ensimmäisenä puusta haetaan paikka avaimelle, jolloin päädytään puun viidenteen lehteen. Koska lehdessä on tilaa, avain voidaan lisätä siihen ilman lisätoimenpiteitä.

Esimerkki 8. Lisätään seuraavaksi puuhun tietue, jonka avain on 15. Avain pitää lisätä puun toiseen lehteen. Koska kyseinen lehti on täynnä, se joudutaan jakamaan. Olemassa olevaan lehteen jäävät avaimet 14 ja 15. Uuteen lehteen siirtyvät avaimet 17 ja 19. Tämän jälkeen välisolmuun, joka sisältää avaimet 14, 20, 28, täytyy lisätä avain ja osoitin uuteen solmuun. Koska uuden solmun pienin avain on 17, se lisätään välisolmuun erottamaan al-

kuperäistä ja uutta lehteä. Koska välisolmu on täynnä, on sekin jaettava. Alkuperäiseen solmuun jäävät avaimet 14 ja 17, ja uuteen solmuun siirtyvät avaimet 20 ja 28. Tämän jälkeen on vielä juureen lisättävä avain ja osoitin uuteen välisolmuun. Tämä voidaan tehdä nostamalla uuden välisolmun avain 20 puun juureen. Kuvassa 5 on kuvasta 4 saatu puu, johon on lisätty avaimet 60 ja 15.



Kuva 5. Kuvan 4 B⁺-puu avainten 60 ja 15 lisäyksen jälkeen

2.4 Tietueen poistaminen B⁺-puusta

Poistettaessa tietuetta B⁺-puusta tietue on ensin haettava puusta käyttäen kohdassa 2.2 esitettyä menetelmää. Menetelmä palauttaa puun lehden *N*, jossa poistettavan tietueen avaimen tulee olla. Jos lehdessä on kyseinen avain (siis tietokannassa on avainta vastaava tietue), poistetaan itse tietue datatiedostosta sekä avain ja osoitin puun lehdestä. Jos avaimen poistamisen jälkeen lehdessä on ainakin minimimäärä $\lfloor n/2 \rfloor$ avaimia, niin ei ole tarvetta tehdä muuta. Jos avaimen poistamisen jälkeen solmussa on avaimia alle minimimäärän, joudutaan tekemään toinen seuraavista operaatioista (Garcia-Molina et al. 2002):

- Jos jossain solmun *N* naapurisolmussa *M* (*N* ja *M* ovat saman solmun lapsia) on avaimia yli minimimäärän, voidaan yksi solmun *M* avaimista siirtää solmuun *N*. Jos solmut *N* ja *M* eivät ole vierekkäiset, täytyy muutoksia tehdä myös *N*:n ja *M*:n

välisiin solmuihin. Näin siksi, koska avainten pitää olla järjestyksessä lehdissä. Avainten siirtelyn jälkeen solmujen *N* ja *M* isäsolmua pitää myös päivittää vastaamaan lehtien uutta tilannetta.

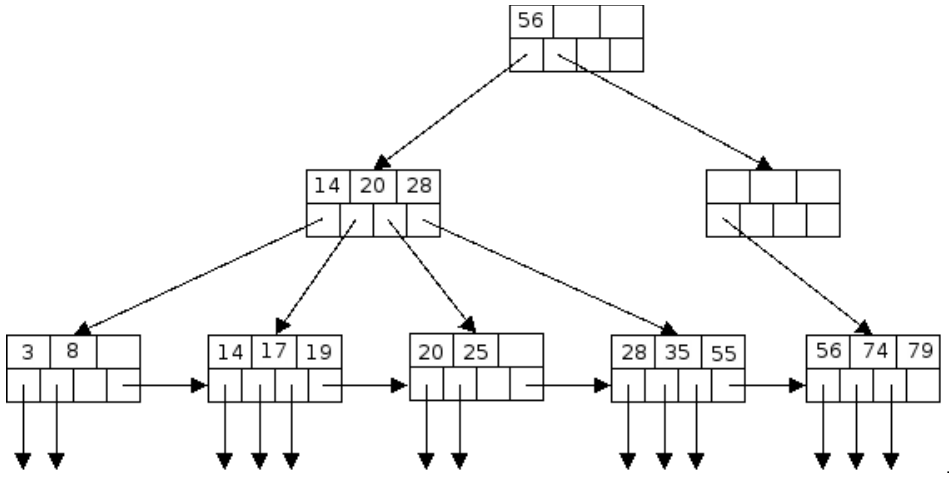
- Jos kaikissa solmun *N* naapureissa on minimimäärä avaimia, ei avainta voi siirtää mistään. Tällöin jokin solmun *N* naapureista *M* pitää yhdistää solmun *N* kanssa. Yhdistämisen jälkeen uusi solmu ei voi olla täynnä, sillä solmu *N* oli vähemmän kuin puoliksi täynnä ja solmu *M* puoliksi täynnä. Yhdistäminen tapahtuu limittämällä solmujen *N* ja *M* avaimet solmuun *N* ja poistamalla solmu *M*. Tämän jälkeen on *N*:n ja *M*:n isäsolmusta poistettava avain ja osoitin solmuun *M*. Jos isäsolmussa on avaimen poistamisen jälkeen avaimia alle minimimäärän, joudutaan poistoalgoritmia soveltamaan isäsolmuun rekursiivisesti.

Garcia-Molina et al. (2002) kertovat B^+-puun toteutuksista, joissa lehteä ei poisteta puusta, vaikka avainten määrä lehdessä alenee alle minimin. Yleensä solmulla on taipumus täyttyä uudelleen. Lisäksi tiedot jakaantuvat yleensä puuhun tasaisesti ja näin puu pysyy tasapainoisena. Tästä on erittäin paljon hyötyä, sillä puusta poistaminen on lisäyksen ohella puun aikaa vievin operaatio, jonka aikavaativuus on tällä järjestelyllä saatu minimoitua. Samoin myös osa puuhun kohdistuneista lisäyksistä nopeutuu, koska puun solmuja joudutaan jakamaan harvemmin.

Esimerkki 9. Poistetaan kuvan 4 B^+-puusta avain 66. Poisto aloitetaan etsimällä kyseinen avain. Avain löytyy puun viidennestä lehdestä. Nyt puusta voidaan poistaa sekä itse avain että avaimen osoittama tietue massamuistista. Poiston jälkeen lehdessä on ainoastaan yksi avain, 56, joten lehti ei ole tarpeeksi täynnä. Tämä tilanne voidaan korjata siirtämällä kuudennesta lehdestä avain 74 viidenteen lehteen. Samalla täytyy myös päivittää lehtien isäsolmua, johon asetetaan kuudennen lehden vasemmanpuolisin avain 79.

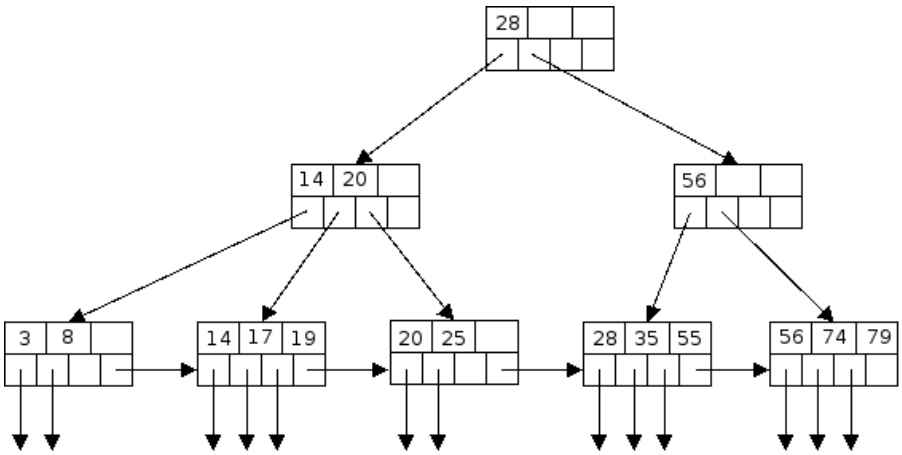
Poistetaan seuraavaksi puun kuudennesta lehdestä löytyvä avain 88. Poiston jälkeen leh-

dessä on vain yksi avain 79, eli lehdessä on nyt avaimia alle sallitun määrän. Tämän takia solmu joudutaan yhdistämään sen naapurisolmuun, joka sisältää avaimet 56 ja 74. Solmujen avaimet limitetään jäljelle jäävään solmuun, joka tulee sisältämään avaimet 56, 74 ja 79. Seuraavaksi on poistettava jäljelle jääneen solmun isäsolmusta avain ja osoitin juuri poistettuun solmuun. Isäsolmussa ei tarvitse olla yhtään lapsia erottavaa avainta, sillä solmulla on nyt vain yksi lapsi. Kuvassa 6 on esitetty B^+-puun tila avainten 66 ja 88 poiston jälkeen.



Kuva 6. B^+-puu avainten 66 ja 88 poiston jälkeen

Vaikka juuri poistetun solmun isäsolmussa ei tarvitse olla yhtään avainta erottamassa solmun lapsia, B^+-puu vaatii, että välisolmun täytyy olla ainakin puoliksi täynnä. Tilanne voidaan korjata siirtämällä solmuun yksi avain toisesta välisolmusta, joka sisältää avaimet 14, 20 ja 28. Mitään avaimista 14, 20 ja 28 ei voida siirtää suoraan tyhjään solmuun, sillä tyhjässä solmussa olevien avainten tulee olla suurempia tai yhtä suuria kuin juuren avain 56. Siksi avain 56 on pudotettava tyhjään solmuun ja täydestä välisolmusta avain 28 on nostettava juureen. Koska vasemmanpuoleisesta välisolmusta poistettiin yksi avain, täytyy myös viimeinen osoitin poistaa. Jotta neljäs lehti ei jäisi ilman osoitinta, asetetaan oikeanpuoleisimman välisolmun ensimmäinen osoitin osoittamaan siihen. Kuvassa 7 on esitetty B^+-puu poistojen jälkeen.



Kuva 7. B-puu poistojen jälkeen

2.5 Lehmanin ja Yaon yhtäaikaisen käytön mahdollistava B^{link}-puu

Edellä esitelty B^+-puu on yksikertainen ja selkeä B-puun rakenne. Se ei kuitenkaan kykene selviämään yhtäaikaisista puuhun kohdistuneista operaatioista, ilman että suuria osia puusta pitäisi lukita vain yhden prosessin käyttöön kerrallaan.

Esimerkki 10. Oletetaan, että jokin prosessi lähtee etsimään kuvan 4 B^+-puusta avainta 19 ja ehtii puun toiselle tasolle löytäen sieltä avaimen ja osoittimen, joiden mukaan avaimen 19 tulisi olla puun toisessa lehdessä vasemmalta luettuna. Samanaikaisesti jokin toinen prosessi lisää puuhun avaimen 15, jolloin puun toinen lehti joudutaan jakamaan (kuva 5). Jos ensimmäinen prosessi ehtii lukemaan toiseksi vasemmanpuoleisimman lehden sisällön vasta avaimen 15 lisäyksen jälkeen, ei hakuprosessi löydä puusta avainta 19, vaikka sellainen puussa onkin.

Edellä annettu esimerkki osoittaa, ettei B^+-puu toimi ilman suuria lukituksia yhtäaikaisessa käytössä. Niinpä Lehman ja Yao (1981) esittelevät yhtäaikaiseen käyttöön suunnitellun B^+-puuta muistuttavan B^{link}-puun, jolla on seuraavat ominaisuudet:

- Jokainen polku puun juuresta lehtiin on yhtä pitkä. Tätä pituutta merkitään kirjaimella h.
- Yhteen solmuun mahtuu korkeintaan $2k$ avainta.
- Avaimet ovat solmun sisällä kasvavassa järjestyksessä vasemmalta oikealle.
- Jokaisella puun solmulla, paitsi juurella ja lehdillä, on vähintään k +1 ja enintään $2k$ +1 lapsisolmua.
- Puun juurella on ainakin kaksi lapsisolmua.
- Solmun i:s osoitin osoittaa alipuuhun, jonka sisältämille avaimille v pätee ehto $K_i < v \leq K_{i+1}$. Solmun ensimmäisen osoittimen osoittamassa alipuussa kaikki avaimet ovat pienempiä tai yhtä suuria kuin avain K_1.
- Kaikki puun data-avaimet on tallennettu puun lehtiin, joissa ovat myös osoitteet massamuistiin.
- Jokaisessa puun solmussa on ”high key”, joka ilmoittaa suurimman avaimen, joka on mahdollista löytää kyseisen solmun muodostamasta alipuusta.
- Jokaisella puun tasolla on solmuissa osoittimet solmun oikeanpuoleiseen naapuriin. Tason oikeanpuoleisimman solmun oikealle soittava osoitin on null.

B^{link}-puussa yhteen solmuun osoittaa siis kaksi osoitinta. Toinen osoitin lähtee isäsolmusta ja toinen vasemmalta puolelta olevasta naapurisolmusta. Kun puuhun lisätään uusi solmu, toinen näistä osoittimista on luotava ensimmäisenä. Lehman ja Yao lisäävät aina oikeaan naapuriin osoittavan osoittimen ensimmäisenä. Solmua jaettaessa vanhan solmun sisältö jaetaan uuden solmun kesken. Jos jokin toinen prosessi on saanut jaettavan solmun osoittimen ennen jakamista ja lukee solmun tiedot jakamisen jälkeen, on haettava avain voinut siirtyä uuteen solmuun. Niinpä jos kyseisen vanhan jaetun solmun ”high key” on suurempi kuin haettava avain, tiedetään, että kyseinen solmu on jaettu. Tämän jälkeen hakua voidaan jatkaa siirtymällä osoitinta pitkin oikeanpuoleiseen naapurisolmuun, josta haettava avain löytyy (jos puussa sellainen on). Ennen kuin uuden solmun isäsolmuun on

asetettu lapsiosoitin uuteen solmuun, uusi solmu ja sen vasemmanpuoleinen naapuri ovat ikään kuin yksi isompi solmu.

Edellä kuvatulla B^{link}-puun rakenteella on saavutettu usean prosessin yhtäaikainen käyttö hyvin vähällä puun lukitsemisella. B^{link}-puussa hakuja tehtäessä ei koskaan tarvitse lukita yhtään solmua ja lisäyksessäkin joudutaan lukitsemaan yhdellä kertaa korkeintaan kolme solmua. Lukitsemisella tarkoitetaan tilannetta, jossa jokin prosessi lukitsee solmun, jolloin mikään muu prosessi ei voi muokata solmun tietoja. Sitä vastoin lukitseminen ei estä muita prosesseja lukemasta solmun tietoja, eli lukitseminen ei aiheuta katkoksia puuhun kohdistuneisiin hakuihin.

Lehman ja Yao esittävät seuraavan hakualgoritmin B^{link}-puulle, jossa ei haun aikana lukita yhtään solmua:

1. Haku aloitetaan puun juuresta, josta edetään lapsiosoittimia pitkin puun lehtiin.
2. Jokaiseen solmuun saavuttaessa tarkastellaan solmun ”high key” -arvoa. Jos haettava avain v on suurempi kuin solmun ”high key”, tarkoittaa tämä sitä, että kyseinen solmu on jaettu ja hakua vastaava avain on siirtynyt uuteen oikeanpuoleiseen solmuun. Tässä tilanteessa on siirryttävä naapuriosoitinta apuna käyttäen oikeanpuoleiseen naapurisolmuun, josta hakua jatketaan.
3. Oikea lapsiosoitin valitaan tarkastelemalla solmussa olevia avaimia ja vertaamalla niitä hakuavaimeen v. Jos solmusta löytyvät avaimet K_i ja K_{i+1} jolle pätee $K_i < v \leq K_{i+1}$, siirrytään i:nnen osoittimen osoittamaan alipuuhun.
4. Lopulta haussa saavutaan puun lehteen. Kuten muihinkin puun solmuihin, myös lehtiin on voinut tulla muutos, joka on edellyttänyt lehden jakamista. Joten jos lehden ”high key” -arvo on suurempi kuin avain v, on lehdissä siirryttävä oikealle.
5. Lopulta päädytään lehtisolmuun, jonka ”high key” on pienempi tai yhtä suuri kuin etsittävä avain v. Tämä tarkoittaa sitä, että v:n tulee löytyä kyseisestä lehdestä.

Lehman ja Yao esittävät seuraavan lisäysalgoritmin B^{link}-puulle:

1. Avaimen *v* lisäys aloitetaan puun juuresta, jota merkitään kirjaimella *A*. Juuresta edetään puun lehteen, johon avain lisätään. Avainta *v* vastaavan tietueen osoitinta merkitään kirjaimella *w*.
2. Niin kauan kuin ei olla puun lehdessä, siirrytään joko solmun *A* lapsiosoitinta pitkin seuraavaan puun kerrokseen tai naapuriosoitinta pitkin *A*:n oikeanpuoleiseen naapuriin. Oikeanpuoleiseen naapuriin on siirryttävä, jos solmun *A* ”high key” on suurempi kuin avain *v*. Siirryttäessä lapsiosoitinta pitkin lapsisolmuun *A*:n osoitin tallennetaan pinoon, johon tallentuu juuresta lehtiin kuljettu reitti.
3. Lopulta saavutaan puun lehteen *A*, joka lukitaan.
4. Niin kauan kuin *A*:n ”high key” on suurempi kuin lisättävä avain *v*, on siirryttävä naapuriosoitinta pitkin oikealle. Siirryttäessä A:n oikeanpuoleiseen naapuriin *A*:n arvoksi asetetaan tämä *A*:n oikeanpuoleinen naapuri ja *A* lukitaan. Tämän jälkeen puretaan lukitus lehdestä, josta naapuriin savuttiin.
5. Jos solmussa *A* on tilaa, avain *v* ja osoitin *w* lisätään siihen. Lisäyksen jälkeen *A*:n lukitus puretaan ja lisäysalgoritmi voidaan lopettaa.
6. Jos solmussa *A* ei ole tilaa, luodaan uusi solmu *B* solmun *A* oikealle puolelle. Tämän jälkeen päivitetään naapuriosoittimet solmuissa *A* ja *B*. Lopuksi solmun *A* sisältämät avaimet sekä avain *v* ja osoitin *w* limitetään solmujen *A* ja *B* kesken.
7. Asetetaan *v*:n arvoksi solmun *A* suurin avain.
8. Asetetaan *w*:n arvoksi solmun *B* osoite.
9. Merkitään solmua *A* kirjaimella *C*.
10. Luetaan pinosta ylemmän tason solmu ja merkitään sitä kirjaimella *A*.
11. Lukitaan solmu *A*.
12. Niin kauan kuin solmun *A* ”high key” on suurempi kuin avain *v*, on siirryttävä naapuriosoitinta pitkin *A*:n oikeaan naapurin. *A*:n arvoksi asetetaan siis *A*:n oi-

keanpuoleinen naapuri. Siirryttäessä oikeanpuoleiseen naapuriin naapuri lukitaan. Tämän jälkeen puretaan lukitus solmusta, josta naapuriin savuttiin.

13. Vapautetaan lukitus solmusta *C*.
14. Siirrytään askeleeseen 5.

Edellä kuvatussa lisäysalgoritmissa jokainen solmu lukitaan siksi aikaa, kun siihen tehdään muutoksia. Siirryttäessä puussa lehdistä juurta kohden muutoksia tehden on korkeintaan kolme solmua kerrallaan lukittuna (Lehman ja Yao, 1981). Lisäksi tämä kolmen solmun lukitseminen tapahtuu suhteellisen harvoin. Tällainen tilanne on ainoastaan silloin, kun pitää seurata naapuriosoitinta lisäysalgoritmin askeleessa 12. Tässä tapauksessa ovat yhtä aikaa lukittuna alkuperäinen jaettu solmu ja kaksi solmua kyseisen solmun ylemmällä tasolla siirryttäessä oikeanpuoleiseen naapuriin.

Edellä kuvatun B^{link}-puun luotettavuus perustuu siihen, että puuhun kohdistunut muutos saadaan aina jäljitettyä naapuriosoittimella. Jaettaessa puun solmua uudet solmut luodaan aina vanhan solmun oikealle puolelle ja ne ovat aina saavutettavissa naapuriosoittimen avulla.

2.6 B-puu PostgreSQL-tietokannassa

PostgreSQL-tietokanta käyttää B-puu-indeksissä Lehmanin ja Yaon *High-concurrency B-trees* -algoritmia (PostgreSQL, 2007; PostgreSQL, 2006). Puuhun kohdistuneissa poistoissa käytetään yksinkertaistettua versiota Laninin ja Sashan (1986) esittämästä *A symmetric concurrent B-tree* -algoritmista. PostgreSQL-tietokannan toteutuksessa näihin algoritmeihin on tehty jonkin verran muutoksia, joita esitellään seuraavaksi (PostgreSQL, 2006).

Lehmanin ja Yaon algoritmissa yhden osoittimen osoittamassa alipuussa olevat avaimet *v*

ovat väliltä $K_i < v \leq K_{i+1}$. Tämä ehto ei kuitenkaan toimi puussa, jonka avaimet eivät ole uniikkeja. Esimerkiksi jos useammassa puun lehdessä on sama avain useampaan kertaan, joudutaan ylemmällä tasolla sallimaan yhtäsuuruuksia. Tästä johtuen ehto on muutettu muotoon $K_i \leq v \leq K_{i+1}$.

Lehmanin ja Yaon algoritmissa oletetaan solmujen koon olevan kiinteä, mutta PostgreSQL-tietokannassa avainten koot vaihtelevat tietotyypin mukaan. Tämän takia B-puun solmuille ei voida asettaa mitään kiinteätä avainten maksimimäärää, vaan solmuun tallennetaan avaimia niin paljon kuin siihen mahtuu. B-puun algoritmi olettaa, että yhteen puun solmuun voidaan tallentaa ainakin kolme avainta (”high key” ja kaksi oikeaa avainta). Tämän lisäksi Lehmanin ja Yaon puusta poiketen B-puun oikeanpuoleisimmalla solmulla ei ole ”high key”-avainta.

Lehmanin ja Yaon puussa ei käytetä solmujen lukitusta puuhun kohdistuneiden hakujen aikana, sillä puun solmujen oletetaan olevan tallennettuna muistiin yksitellen ilman jaettuja muistialueita. PostgreSQL-tietokannassa muistialueet ovat jaettuja, ja tämän takia hakujen yhteydessä on tehtävä lukituksia, jotta voidaan taata, että tietuetta ei ole muokattu haun aikana. Tämä vähentää yhtäaikaisuutta mutta takaa oikean toiminnan. Hyötynä tästä on se, että prosessin vaihtaessa lukulukon kirjoituslukkoon ei solmua tarvitse lukea uudelleen ennen kirjoituslukon asettamista. Solmujen lukulukkoja pidetään yleensä yllä vain sen aikaa, kun solmun tietoja ollaan lukemassa.

PostgreSQL-tietokannan B-puu sisältää Lehmanin ja Yaon kuvaamat oikeanpuoleiseen naapurisolmuun osoittavat osoittimet jokaisessa solmussa. Nämä osoittimet mahdollistavat vaivattoman puun läpikäynnin pienemmästä avaimesta suurempaan. Tässä arvovälihaussa tarvitsee tarkastella vain puun lehtisolmuja eikä korkeampia puun tasoja siis koskaan tarvitse käydä läpi. PostgreSQL-tietokannan B-puussa on myös jokaisessa solmussa osoitin solmun vasempaan naapuriin. Nämä vasempaan naapuriin osoittavat osoittimet

mahdollistavat käänteiseen suuntaan tehtävän puun läpikäynnin. Nämä osoittimet lisäävät myös ylimääräisen toimenpiteen Lehmanin ja Yaon esittämään lisäysalgoritmiin: kun solmua ollaan jakamassa, joudutaan myös tämän solmun aiempi oikeanpuoleinen naapuri lukitsemaan, jotta siihen voidaan asettaa vasemman naapurin osoitin uuteen solmuun. PostgreSQL (2006) ei kerro muista tilanteista, joissa lisäysalgoritmi poikkeaisi Lehmanin ja Yaon esittämästä.

Poistettaessa lehtisolmusta avainta kyseiseen solmuun asennetaan lukko, joka estää kaiken muun käsittelyn kyseisessä solmussa. PostgreSQL-tietokannan B-puun solmua ei poisteta, ennen kuin se on täysin tyhjä. Puoliksi täysien solmujen limitys saattaisi taata paremman tilanhallinnan, mutta avainten siirtäminen vasemmalle ja oikealle on epäkäytännöllistä. Lisäksi puuhun kohdistunut haku, joka etenee päinvastaiseen suuntaan avaimen siirtoon nähden, saattaa hypätä siirrettävän avaimen yli (PostgreSQL, 2006).

PostgreSQL-tietokannan B-puusta ei koskaan poisteta tason oikeanpuoleisinta solmua miltään tasolta, eli tällöin puun juurtakaan ei koskaan poisteta. Tästä seuraa, että puun korkeus ei koskaan voi pienentyä. Suurien poisto-operaatioiden jälkeen voi puun rakenne muuttua niin, että juuren alla on useita yksittäisiä solmutasoja, jotka voivat heikentää puun tehokkuutta. Tämän tilanteen korjaamiseksi käytetään Laninin ja Shashan (1986) esittämää ”fast root” -tasoa, joka on puun matalin yksittäisen solmun sisältävä taso. B-puun metadata-tiedoissa pidetään puun juuriosoitteen lisäksi ”fast root” -tason osoitetta. Kaikki puuhun kohdistuneet operaatiot aloittavat etsintäoperaation puun juuren sijasta ”fast root” -tasolta.

Poistettaessa B-puusta tyhjää solmua on kirjoituslukolla lukittava seuraavat solmut: poistettavan solmun vasemmanpuoleinen naapurisolmu (jos sellainen on), itse poistettava solmu, sen oikeanpuoleinen naapuri (tämä solmu on aina olemassa, sillä tason oikeanpuoleisinta solmua ei koskaan poisteta) ja poistettavan solmun isäsolmu. Tämän jälkeen poiste-

taan itse solmu ja päivitetään naapuriosoittimet. Lopuksi isäsolmusta poistetaan poistettuun solmuun viittaava osoitin ja avain. Kun isäsolmun alta poistetaan viimeinen lapsisolmu, isäsolmu merkitään puolikuolleeksi. Ennen kuin isäsolmu ehditään poistaa, hakualgoritmi hylkää tällaiset puolikuolleet solmut ja siirtyy tarkastelemaan suoraan tämän solmun oikeanpuoleista naapuria.

PostgreSQL tietokannassa B-puu-indeksi voidaan luoda seuraavalla lauseella (PostgreSQL, 2007):

```
CREATE INDEX name ON table USING BTREE (column);
```

Luontilauseessa käytetyillä muuttujilla on seuraavat merkitykset:

- name on indeksiä kuvaava ja yksilöivä nimi
- table on tietokannan taulu, joka indeksoidaan
- column on indeksoitavan taulun sarake, jonka perusteella taulu indeksoidaan
- optio USING BTREE ei ole pakollinen, sillä B-puu on tietokannan oletusindeksointimenetelmä.

Indeksi voidaan poistaa PostgreSQL-tietokannasta seuraavalla komennolla, jossa name on poistettavan indeksin nimi:

```
DROP INDEX name;
```

PostgreSQL-tietokannassa B-puuhun voidaan kohdistaa seuraavanlaisia hakuja:

- *Yhtäsuuruusvertailua* operaatiolla =.
- *Arvoväli kyselyitä* kaikenlaiselle tiedolle, joka voidaan asettaa johonkin järjestyk-

seen. Arvovälikyselyissä voidaan käyttää seuraavia operaatiota: <, <=, =, >=, >.

- B-puuta voidaan käyttää myös *merkkijonohakujen* yhteydessä. Merkkijonoja voidaan hakea seuraavilla operaatioilla: LIKE, ILIKE, ~, ja ~*. Haettaessa merkkijonolausekkeella hakulausekkeen tulee alkaa merkkijonolla, esimerkiksi LIKE 'abc%'. Hakulauseke ei saa alkaa jokerimerkillä, joita ovat: % ja ^. Esimerkiksi lauseketta LIKE '%abc' ei voida käyttää B-puiden yhteydessä.

Esimerkki 11. Oletetaan, että PostgreSQL-tietokannassa on taulu *Asiakas* sisältäen kuvan 8 esittämät arvot.

Nimi	***Ika***	***Ammatti***
Matti	30	Koneistaja
Esko	41	Johtaja
Liisa	52	Opettaja
Maija	24	Leipuri
Hannu	30	Maanviljelijä
Lasse	43	Maalari

Kuva 8. Asiakas-taulu

Indeksoidaan taulu *Asiakas* sarakkeen *Ika* mukaan käyttäen B-puu-indeksiä. Indeksi voidaan luoda seuraavalla komennolla:

```
CREATE INDEX Asiakas_b-puu_indeksi_ika ON Asiakas (Ika);
```

Ika-sarakkeelle luodun indeksin avulla voidaan hakea asiakkaita esimerkiksi seuraavalla yhtäsuuruuskyselyllä:

```
SELECT * FROM Asiakas WHERE Ika=30;
```

PostgreSQL-tietokannan B-puu-indeksi mahdollistaa myös esimerkiksi seuraavanlaiset arvovälikyselyt:

```
SELECT * FROM Asiakas WHERE Ika<35;
SELECT * FROM Asiakas WHERE Ika<=35 AND Ika>=59;
```

Indeksoidaan seuraavaksi kuvan 8 *Asiakas*-taulu ammatin mukaan komennolla:

```
CREATE INDEX Asiakas_b-puu_indeksi_ammatti ON Asiakas (Ammatti);
```

Ammatti-sarakkeelle luodun indeksin avulla voidaan asiakkaita hakea nyt ammatin perusteella esimerkiksi seuraavalla kyselyllä:

```
SELECT * FROM Asiakas WHERE Ammatti like 'Maa%';
```

Indeksit voidaan tarvittaessa poistaa seuraavilla komennoilla:

```
DROP INDEX Asiakas_b-puu_indeksi_ika;
DROP INDEX Asiakas_b-puu_indeksi_ammatti;
```

2.7 B-puun tehokkuushypoteesi

B-puu on nykyisin tehokkain ja käytetyin tietokannan indeksointimenetelmä. Garcia-Molina et al. (2002) kertovat, että yleensä puun korkeudeksi riittää kolme tasoa. Jos puun solmun kooksi asetetaan 255 avainta, mahtuu kolmikerroksiseen puuhun enimmillään 16 600 000 avainta. Tällöin puusta hakemiseen tarvitaan vain neljä *I/O*-operaatiota. Avaimen lisääminen puuhun ja sen poistaminen puusta vievät kumpikin viisi *I/O*-operaatiota. Gar-

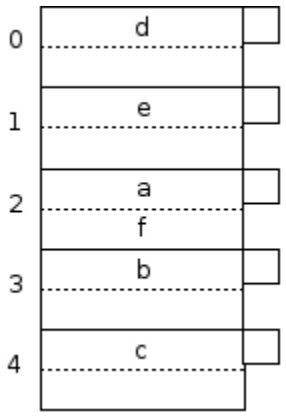
cia-Molina et al. (2002) esittävät myös, että puun ylimpiä kerroksia tai ainakin puun juurta kannattaa säilyttää tietokoneen muistissa. Näin levyoperaatioiden määrä vähenee huomattavasti kohtuullisen pienellä muistivarauksella.

PostgreSQL-tietokannassa B-puu on suositeltavin ja oletuksena käytettävä indeksointimuoto. Verrattuna hajautusindeksiin, joka on PostgreSQL-tietokannassa toinen indeksointimenetelmä, B-puu nopeuttaa myös arvovälikyselyitä kaikenlaiselle tiedolle, joka voidaan asettaa johonkin järjestykseen. PostgreSQL-tietokannassa B-puuta voidaan myös käyttää merkkijonohakujen yhteydessä esimerkiksi like-menetelmällä tehdyissä hauissa; tällöin hakulauseke ei kuitenkaan saa alkaa jokerimerkillä (PostgreSQL, 2007).

3 Hajautusindeksi

Hajautusindeksi (*Hash index*) perustuu *hajautustauluun* ja *hajautusfunktioon* (Garcia-Molina et al. 2002; Connolly ja Begg, 2005). Hajautustaulu koostuu *B*:stä *tietojaksosta*, joihin kuhunkin voidaan tallentaa *n* kappaletta tietueiden avaimia. Indeksiin lisättävät tietueet jaetaan tietojaksoihin hajautusfunktion palauttaman *hajautusavaimen* perusteella. Hajautusfunktio *h* saa syötteenä tietueen avaimen *K* ja palauttaa hajautusavaimen *h*(*K*) väliltä [0, *B*-1]. Hyvä hajautusfunktio palauttaa hajautusavaimia mahdollisimman tasaisesti kyseiseltä väliltä.

Kuvassa 9 on esimerkki hajautustaulusta, jossa on viisi tietojaksoa, joihin jokaiseen mahtuu kaksi avainta (Garcia-Molina et al. 2002). Tauluun on tallennettu avaimet *a*, *b*, *c*, *d*, *e* ja *f*, joille hajautusfunktio *h* (merkin ASCII-arvon modulo 5) antaa seuraavat hajautusavaimet: *h*(*d*) = 0, *h*(*e*) = 1, *h*(*a*) = *h*(*f*) = 2, *h*(*b*) = 3 ja *h*(*c*) = 4.



Kuva 9. Hajautustaulu

3.1 Tietueen hakeminen hajautusindeksistä

Tietueen hakeminen hajautusindeksistä aloitetaan laskemalla hajautusfunktion avulla *ha-*

jautusavain $h(K)$ haettavan tietueen avaimelle K. Seuraavaksi siirrytään hajautusavaimen ilmoittamaan tietojaksoon hajautusindeksissä. Tietojakson sisältö käydään läpi ja etsitään avaimen perusteella oikea tietue. Jos tietojaksoon liittyy ylivuotojaksoja, on nekin käytävä läpi (Garcia-Molina et al. 2002).

Esimerkki 12. Haetaan kuvan 9 hajautustaulusta tietuetta, jonka avain on *f*. Ensimmäiseksi lasketaan hajautusavain $h(f)$, jonka arvoksi saadaan 2. Seuraavaksi varsinaista avainta haetaan hajautustaulun kolmannesta tietojaksosta, josta kyseinen avain myös löytyy.

Haetaan seuraavaksi kuvan 9 taulusta avainta g, jolle $h(g) = 3$. Avainta on haettava taulun neljännestä tietojaksosta. Koska avainta ei löydy kyseisestä vajaasta tietojaksosta, tiedetään, ettei tietokannassa ole kyseiseen avaimeen liittyvää tietuetta.

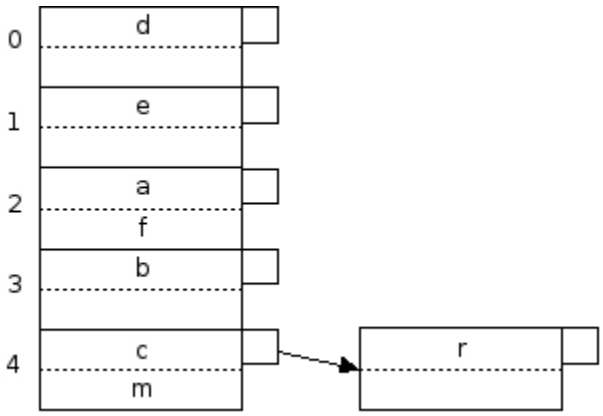
3.2 Tietueen lisääminen hajautusindeksiin

Tietueen lisääminen hajautusindeksiin aloitetaan laskemalla lisättävälle tietueelle *hajautusfunktion* arvo. Mikäli hajautusavaimen osoittamassa tietojaksossa on tilaa, voidaan avain lisätä siihen suoraan. Jos tietojakso on täynnä, joudutaan hajautusindeksiin lisäämään *ylivuotojakso*. Hajautusavaimen osoittamaan tietojaksoon asetetaan osoitin juuri luotuun ylivuotojaksoon. Lisättävä avain voidaan nyt lisätä ylivuotojaksoon (Garcia-Molina et al. 2002).

Esimerkki 13. Lisätään kuvan 9 hajautustauluun tietue, jonka avain on m ja jolle $h(m) = 4$. Koska hajautustaulun viimeisessä tietojaksossa on tilaa, voidaan avain m lisätä siihen suoraan.

Lisätään tämän jälkeen hajautustauluun avain r, jolle $h(r) = 4$. Koska hajautustaulun viimeinen tietojakso on nyt täynnä, joudutaan tauluun lisäämään ylivuotojakso. Seuraavaksi

avain *r* tallennetaan juuri luotuun ylivuotojaksoon. Lopuksi hajautustaulun viimeiseen tietojaksoon asetetaan osoitin uuteen ylivuotojaksoon. Kuvassa 10 on esitetty hajautustaulun tila avainten *m* ja *r* lisäysten jälkeen.



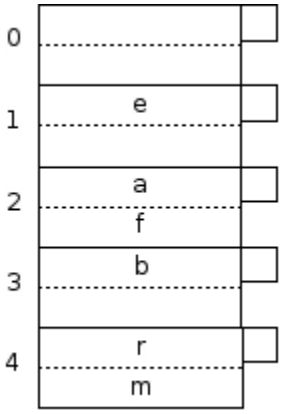
Kuva 10. Hajautustaulu, johon on lisätty avaimet m ja r.

3.3 Tietueen poistaminen hajautusindeksistä

Tietueen poistaminen *hajautusindeksistä* aloitetaan etsimällä poistettava tietue indeksistä käyttäen kohdassa 3.1 esitettyä hakumenetelmää. Jos etsitty avain löytyy indeksistä, se voidaan poistaa. Samalla on myös syytä tarkistaa, voidaanko indeksin ylivuotojaksoja yhdistää ja näin mahdollisesti myös poistaa yksi ylivuotojakso. Garcia-Molina et al. (2002) esittävät myös näkemyksen, jonka mukaan ylivuotojaksojen poistaminen voi johtaa tilanteeseen, jossa ylivuotojaksoja lisätään ja poistetaan lähes jokaisella kerralla, kun indeksiin tehdään muutoksia.

Esimerkki 14. Poistetaan kuvan 10 hajautustaulusta avain *d*, jolle $h(d) = 0$. Avain *d* voidaan poistaa taulun ensimmäisestä tietojaksosta ilman lisätoimenpiteitä, koska kyseen tietojaksoon ei liity ylivuotojaksoja. Poistetaan seuraavaksi kuvan 10 hajautustaulusta avain *c*, jolle $h(c) = 4$. Avaimen poiston jälkeen hajautustaulun viimeinen tietojakso ja siihen lii-

tetty ylivuotojakso ovat kumpikin puoliksi täysiä. Ylivuotojaksossa oleva avain *r* voidaan siirtää varsinaiseen soluun ja tämän jälkeen ylivuotojakso voidaan poistaa. Kuvassa 11 on esitetty hajautustaulu avainten *c* ja *d* poistamisen jälkeen.



Kuva 11. Hajautustaulu avainten c ja d poistamisen jälkeen

3.4 Laajentavan hajautustaulun rakenne

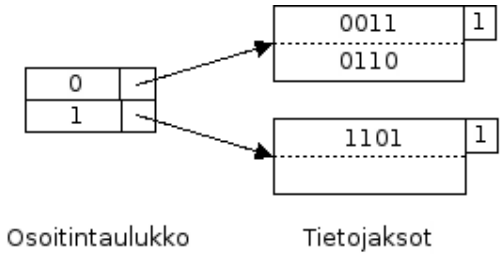
Laajentavan hajautustaulun perusrakenne on samankaltainen kuin kohdassa 3.1 esitellyn yksinkertaisen hajautustaulun rakenne. Laajentavassa hajautustaulussa ei tarvita ylivuotojaksoja, sillä ylivuodon tapahtuessa osoitintaulukkoa kasvatetaan (Garcia Molina et al. 2002; Connolly ja Begg, 2005). Laajentavan hajautustaulun osoitintaulukon koko on aina kahden potenssi 2^i, eli jokaisella kasvatuskerralla osoitintaulukon koko kaksinkertaistuu.

Hajautusfunktion *h* palauttamaa hajautusavainta tarkastellaan *bittiesityksenä*. Hajautusfunktio palauttaa osoitintaulukon koosta riippumatta aina pitkän, esimerkiksi 34-bittisen *hajautusavaimen*. *Osoitintaulukkoon* liittyy muuttuja *i*, joka ilmoittaa, monenko ensimmäisen hajautusavaimen bitin perusteella tietueet on hajautettu.

Indeksin tietojaksoja ei tarvitse olla yhtä monta kuin osoitintaulukon koko 2^i. Osoitintaulukossa voi olla esimerkiksi kaksi osoitinta, jotka osoittavat samaan tietojaksoon. Jokai-

seen tietojaksoon liittyy paikallinen syvyys, joka ilmoittaa, monenko ensimmäisen bitin mukaan kyseisessä tietojaksossa olevat tietueet on hajautettu.

Esimerkki 15. Kuvassa 12 on esitetty yksinkertainen laajentava hajautustaulu, jonka avaimet on yksikertaisuuden vuosi hajautettu käyttäen vain neljän bitin pituista hajautusavainta (Garcia-Molina et al. 2002). Muuttujan *i* arvo on 1, eli osoitintaulukossa on kaksi solua ($2^1 = 2$). Tämä tarkoittaa sitä, että hajautusfunktion palauttamasta hajautusavaimesta tarkastellaan vain ensimmäistä bittiä. Osoitintaulukon ensimmäinen osoitin osoittaa ensimmäiseen tietojaksoon, jossa ovat kaikki tietueet, joiden hajautusavain alkaa bitillä 0. Osoitintaulukon toinen osoitin puolestaan osoittaa tietojaksoon, jossa ovat tietueet, joiden hajautusavaimen ensimmäinen bitti on 1. Kumpaankin tietojaksoon liittyy paikallinen syvyys 1, joka ilmoittaa, että tietueet on indeksoitu ensimmäisen bitin perusteella.



Kuva 12. Laajentava hajautustaulu

3.5 Tietueen lisääminen laajentavaan hajautustauluun

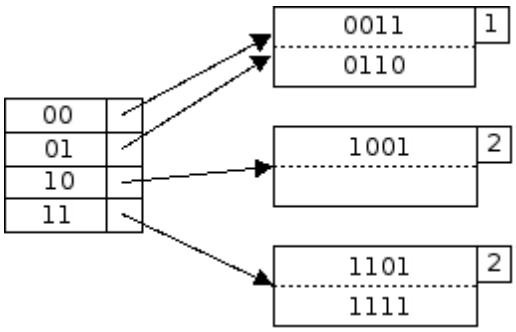
Tietueen lisääminen laajentavaan hajautustauluun aloitetaan laskemalla hajautusfunktiolla *hajautusavain* lisättävälle tietueelle. Hajautusavaimesta poimitaan ensimmäiset *i* bittiä, joiden avulla edetään oikeaan tietojaksoon *B*. Jos tietojaksossa on tilaa, voidaan tietue lisätä siihen ilman lisätoimenpiteitä. Jos tietojaksossa ei ole tilaa, on valittavana kaksi toimenpidettä riippuen tietojakson *B paikallisesta syvyydestä j*, joka ilmoittaa, monenko bitin mukaan tietojaksossa olevat tietueet on indeksoitu (Garcia-Molina et al. 2002). Mahdolliset toimenpiteet ovat:

1. Jos $j < i$:
   - Tietojakso B jaetaan kahtia.
   - Tietojaksossa B olevat tietueet, joiden hajautusavaimen j+1:s bitti on 0, pysyvät alkuperäisessä tietojaksossa. Muut tietueet, eli tietueet, joiden j+1:s bitti on 1, siirretään uuteen tietojaksoon.
   - Tietojakson B paikallista syvyyttä kasvatetaan yhdellä. Uudelle tietojaksolle asetetaan sama paikallinen syvyys.
   - Päivitetään osoitintaulukon osoittimet vastaamaan uutta tilannetta tietojaksoissa. Puolet tietojaksoon B osoittavista osoittimista siirretään osoittamaan uuteen tietojaksoon.
   - Yritetään nyt lisätä tietuetta päivitettyyn hajautustauluun. On mahdollista, että tietue ei vieläkään mahdu haluttuun tietojaksoon, sillä tietojaksoa jaettaessa voivat kaikki tietojaksossa olevat tietueet siirtyä samaan tietojaksoon. Tällöin askel 1 on suoritettava uudelleen. Jos kuitenkin $i = j$, niin on suoritettava toinen askel.

2. Jos $i = j$:
   - Kasvatetaan i:tä yhdellä eli osoitintaulukon koko kaksinkertaistetaan.
   - Jokaista alkuperäisessä osoitintaulukossa olevaa bittijonoa w kohti uuteen osoitintaulukkoon lisätään kaksi bittijonoa $w0$ ja $w1$. Kummankin bittijonon osoitin asetetaan osoittamaan samaan tietojaksoon, johon alkuperäinen bittijono w osoitti.
   - Suoritetaan askel 1, joka voidaan nyt suorittaa, sillä $j < i$.

Esimerkki 16. Lisätään kuvan 12 hajautustauluun tietue, jonka hajautusavain on 1111. Koska $i = 1$, hajautusavaimesta tarkastellaan vain ensimmäistä bittiä. Koska ensimmäinen bitti on 1, tietue lisätään alempaan tietojaksoon. Kyseisessä tietojaksossa on tilaa, joten tietue voidaan lisätä siihen ilman lisätoimenpiteitä.

Lisätään seuraavaksi hajautustauluun tietue, jonka hajautusavain on 1001. Koska $i = 1$, on tietue lisättävä alempaan tietojaksoon. Tietojakso on nyt täynnä, joten hajautustaulua joudutaan laajentamaan. Koska $i = j = 1$, on suoritettava lisäysalgoritmin toinen askel. Osoitintaulukko kaksinkertaistetaan, eli taulukko sisältää nyt neljä osoitinta. Kaksi ensimmäistä osoitinta osoittavat ensimmäiseen tietojaksoon ja osoitintaulukon kaksi viimeistä osoitinta alempaan tietojaksoon. Kummankin tietojakson paikallinen syvyys on 1. Seuraavaksi suoritetaan lisäysalgoritmin ensimmäinen askel. Alempi tietojakso joudutaan jakamaan. Alkuperäiseen tietojaksoon jää tietue, jonka avain on 1001. Uuteen tietojaksoon siirtyvät tietueet, joiden hajautusavaimet ovat 1101 ja 1111. Näiden kahden tietojakson paikallinen syvyys on nyt 2, eli niissä olevat tiedot on hajautettu kahden ensimmäisen bitin perusteella. Lopuksi osoitintaulukon osoittimet 10 ja 11 päivitetään osoittamaan oikeisiin tietojaksoihin. Kuva 13 esittää hajautustaulun tilaa lisäysten jälkeen.

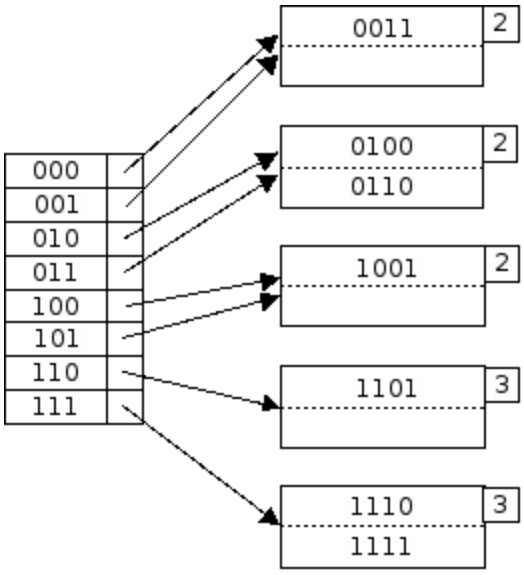


Kuva 13. Hajautustaulu avainten 1111 ja 1001 lisäysten jälkeen

Lisätään seuraavaksi kuvan 13 hajautustauluun tietue, jonka hajautusavain on 0100. Tietue pitää lisätä kuvan 13 hajautustaulun ylimpään tietojaksoon, joka on täynnä. Kyseisen tietojakson paikallinen syvyys on 1, joka on pienempi kuin koko taulun globaalisyvyys 2. Tästä johtuen tietojakso voidaan jakaa kahtia. Ylempään tietojaksoon jää avain 0011, ja juuri luotuun alempaan siirtyvät avaimet 0100 ja 0110. Osoitintaulukon osoitin 01 siirre-

tään osoittamaan juuri luotuun uuteen tietojaksoon.

Lisätään lopuksi hajautustauluun tietue, jonka avain on 1110. Tietue on lisättävä kuvan 13 taulun alimpaan tietojaksoon. Koska tietojakso on täynnä ja sen paikallinen syvyys on sama kuin taulun globaalisyvyys 2, on osoitintaulukkoa laajennettava ja alin tietojakso jaettava kahtia. Kuva 14 esittää hajautustaulun tilaa näiden lisäysten jälkeen.



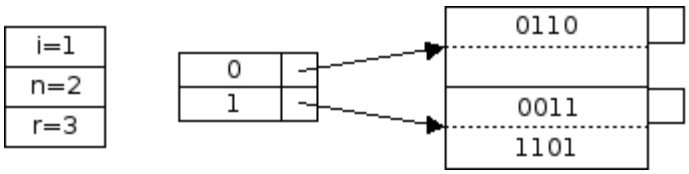
Kuva 14. Hajautustauluun on lisätty avaimet 0100 ja 1110.

3.6 Lineaarisen hajautustaulun rakenne

Laajentavaa hajautustaulua parempana ratkaisuna voidaan pitää *lineaarista* hajautustaulua, jossa on parannettu laajentavan hajautustaulun pahinta ongelmaa, osoitintaulukon päivittämistä. Lineaarinen hajautustaulu kasvattaa osoitintaulukkoa maltillisesti. Lineaarisella hajautuksella on myös seuraavat erot verrattuna laajentavaan hajautukseen (Garcia-Molina et al. 2002):

- Osoitintaulukon kokoa merkitään kirjaimella n. Koko on aina valittu niin, että keskimääräinen tietueitten määrä yhtä tietojaksoa kohden on korkeintaan tietty vakio, esimerkiksi 85 prosenttia tietojakson koosta.
- Yllä mainitusta rajoitteesta johtuen tietojaksoja ei aina voida jakaa, vaan joudutaan käyttämään *ylivuotojaksoja*. Keskimääräinen ylivuotojaksojen määrä yhtä osoitintaulukon osoitinta kohden on kuitenkin paljon pienempi kuin 1.
- Osoitintaulukossa käytettävien hajautusavaimen bittien määrää i kuvaa lauseke $\lceil \log_2 n \rceil$, jossa n on osoitintaulukossa olevien osoittimien määrä. Lineaarisessa hajautustaulussa hajautusavaimesta käytetään i:tä vähiten merkitsevää eli oikeanpuoleisinta bittiä.
- Lisättäessä ja haettaessa tietuetta K sen hajautusavaimen $h(K)$ i *oikeanpuoleisinta* bittiä $a_1, a_2, ..., a_i$ muutetaan *kymmenjärjestelmän* kokonaisluvuksi m. Jos $m < n$, niin osoitintaulukosta löytyy osoitin hajautusavaimesta muodostetulle bittijonolle, ja tietue voidaan lisätä osoittimen osoittamaan tietojaksoon. Jos $n \leq m < 2^i$, niin osoitintaulukosta ei löydy osoitinta bittijonolle. Tällöin tietue lisätään luvun m-2^{i-1} binääriesityksen osoittamaan tietojaksoon. Käytännössä tässä tilanteessa bittijonon ensimmäisen bitin a_1 täytyy olla 1 ja se vaihdetaan nollaksi.
- Lineaarinen hajautustaulu pitää sisällään myös tiedon tietojaksoissa olevien tietueiden määrästä, jota merkitään kirjaimella r.

Esimerkki 17. Kuvassa 15 on esitetty yksinkertainen lineaarinen hajautustaulu samoilla avaimilla kuin esimerkissä 19. Toisin kuin laajentavassa hajautustaulussa hajautusavaimesta tarkastellaan i:tä viimeistä bittiä, joiden perusteella tietueet hajautetaan tietojaksoihin. Parametri r kertoo tietojaksoissa olevan kolme tietuetta, ja parametri n kertoo osoitintaulukossa olevan kaksi osoitinta. Osoitintaulukon koko n riippuu tietojaksoissa olevien avainten määrästä sekä taululle asetetusta vakiosta 85%. Koska yhteen tietojaksoon mahtuu kaksi avainta, niin n:n on toteutettava lauseke $r \leq 1.7n$.



Kuva 15. Lineaarinen hajautustaulu

3.7 Tietueen hakeminen lineaarisesta hajautustaulusta

Tietueen hakeminen aloitetaan laskemalla hajautusfunktion avulla tietueen *hajautusavain*, josta poimitaan *i oikeanpuoleisinta* bittiä. Tästä bittijonosta muodostetaan kymmenjärjestelmän luku *m*. Jos $m < n$, niin tietuetta haetaan osoitintaulukon kyseisen bittijonon osoittamasta tietojaksosta. Jos $m \geq n$, niin kyseisen bittijonon ensimmäinen eli vasemmanpuoleisin bitti muutetaan nollaksi. Tietuetta haetaan kyseisen bittijonon osoittamasta tietojaksosta.

Tietojakson lisäksi on haettaessa tarkastettava kaikki kyseiseen tietojaksoon mahdollisesti liittyvät *ylivuotojaksot*. Jos tietuetta ei löydy kyseisestä tietojaksosta eikä sen ylivuotojaksoista, ei kyseistä tietuetta ole tietokannassa.

Esimerkki 18. Haetaan kuvan 16 esittämästä hajautustaulusta tietuetta, jonka hajautusavain on 1101. Koska nyt $i = 2$, tarkastellaan hajautusavaimen kahta viimeistä bittiä 01. Kyseistä bittijonoa vastaa kymmenjärjestelmän esitys 1. Koska $1 < 3$, voidaan avainta etsiä osoitintaulukon osoittimen 01 osoittamasta tietojaksosta, josta avain myös löytyy.

Haetaan seuraavaksi kuvan 16 hajautustaulusta tietuetta, jonka hajautusavain on 1111. Taas tarkastellaan hajautusavaimen kahta viimeisintä bittiä 11, jonka kymmenjärjestelmän esitys on 3. Koska nyt myös n = 3, bittijonon ensimmäinen bitti on muutettava nollaksi ja tietuetta on etsittävä osoitintaulukon osoittimen 01 osoittamasta tietojaksosta. Kyseisestä tietojaksosta ei löydy kyseistä avainta, eikä tietojaksolla ole ylivuotojaksoja. Tästä tiede-

tään, ettei kyseistä avainta vastaavaa tietuetta löydy tietokannasta.

3.8 Tietueen lisääminen lineaariseen hajautustauluun

Tietueen lisääminen aloitetaan laskemalla lisättävälle tietueelle *hajautusavain*. Hajautusavaimesta poimitaan i oikeanpuoleisinta bittiä, joista muodostetaan kymmenjärjestelmän luku m. Jos $m < n$, voidaan tietue lisätä luvun m binääriesityksen osoittamaan tietojaksoon. Jos $m \geq n$, niin tietue lisätään luvun m-2^{i-1} binääriesityksen osoittamaan tietojaksoon eli tietojaksoon, jonka osoite on saatu muuttamalla bittijonon ensimmäinen bitti (1) nollaksi. Jos tietojakso ja sen mahdolliset ylivuotojaksot, joihin tietuetta ollaan lisäämässä, ovat täynnä, on tietojaksolle luotava uusi ylivuotojakso.

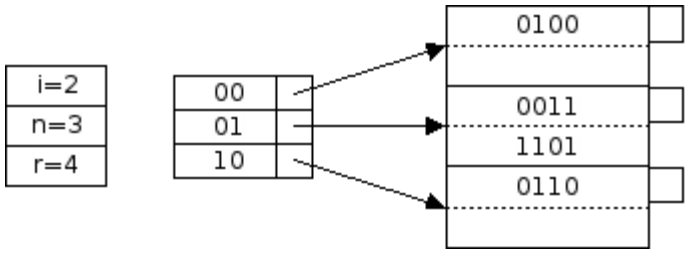
Tietueen lisäämisen jälkeen on tarkasteltava suhdetta r/n. Jos suhde ylittää taululle määrätyn arvon, on hajautustauluun lisättävä uusi tietojakso. Lisäksi osoitintaulukkoon lisätään uusi osoitin osoittamaan lisättyyn tietojaksoon. Lisättävän osoitteen binääriesitys alkaa bitillä 1 ja on muotoa $1a_2...a_i$. Osoitintaulukon osoittimen $0a_2...a_i$ osoittaman tietojakson sisältämät tietueet, joiden hajautusavaimen i:nneksi viimeinen bitti on 1, siirretään uuteen tietojaksoon.

Osoitintaulukossa käytettävien hajautusavaimen bittien määrä i on $\lceil \log_2 n \rceil$. Tästä johtuen jokaisen osoitintaulukkoon tehdyn lisäyksen yhteydessä on tarkastettava, joudutaanko i:tä kasvattamaan yhdellä. Käytännössä i:n kasvattaminen tarkoittaa sitä, että jokaisen osoitintaulukon osoittimen ensimmäiseksi bitiksi lisätään 0. Tämä muutos ei aiheuta muita muutoksia hajautustaulun rakenteeseen.

Esimerkki 19. Lisätään kuvan 15 hajautustauluun tietue, joka hajautusavain on 0100. Koska $i = 1$, hajautusavaimesta tarkastellaan oikeanpuoleisinta bittiä, joka on 0. Tietue lisätään siis ensimmäiseen tietojaksoon. Tietojaksossa on tilaa, joten ylivuotojaksoja ei tarvit-

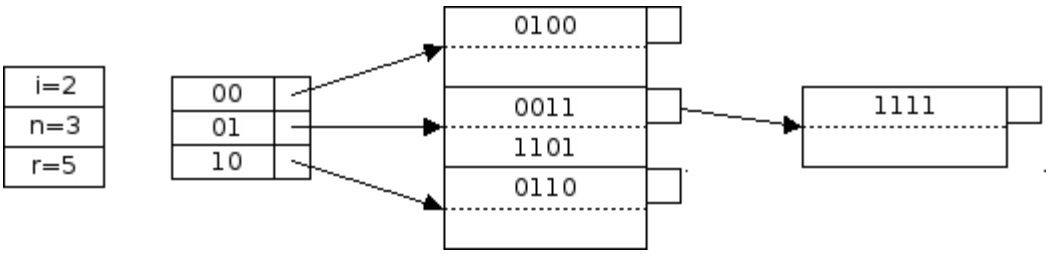
se lisätä.

Nyt tietojaksot sisältävät 4 tietuetta ja osoitintaulukko 2 osoitinta, joten hajautustaululle asetettu ehto $r \leq 1.7n$ ei ole enää voimassa. Tästä johtuen on osoitintaulukon kokoa kasvatettava yhdellä. Koska $\lceil \log_2 3 \rceil = 2$, on myös i:tä kasvatettava yhdellä. Tämä tarkoittaa sitä, että osoitintaulukon kahteen alkuperäiseen osoittimeen 0 ja 1 lisätään alkuun bitti 0, eli uudet osoitteet ovat 00 ja 01. Osoitintaulukkoon lisätään uusi osoitin 10. Osoitteessa 00 olevan tietojakson sisältö jaetaan kyseisen tietojakson ja juuri lisätyn tietojakson kesken. Osoittimen 00 osoittamaan tietojaksoon jää avain 0100, ja tietue 0110 siirtyy uuteen tietojaksoon. Kuvassa 16 on esitetty hajautustaulun tila avaimen 0100 lisäyksen jälkeen.



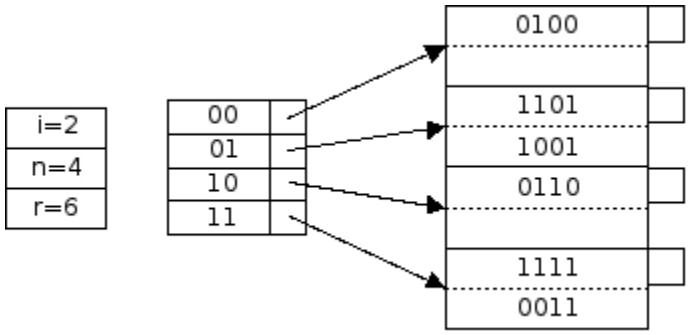
Kuva 16. Hajautustauluun on lisätty avain 0100.

Esimerkki 20. Lisätään kuvan 16 hajautustauluun tietue, jonka hajautusavain on 1111. Koska $i = 2$, tarkastellaan hajautusavaimen kahta viimeisintä bittiä. Bittiesitys 11 vastaa kymmenjärjestelmässä lukua 3, eli merkitään $m = 3$. Koska nyt $m \geq n$, pitää avain 1111 lisätä osoittimen 01 osoittamaan tietojaksoon. Koska tietojaksossa ei ole vapaata tilaa, on tietojaksoon luotava ylivuotojakso, johon avain tallennetaan. Lisäyksen jälkeen tarkastellaan ehtoa $r \leq 1.7n$. Koska ehto on edelleen voimassa, ei osoitintaulukon kokoa eikä hajautusavaimesta tarkasteltavien bittien määrää tarvitse kasvattaa. Kuvassa 17 on esitetty hajautustaulun tila avaimen 1111 lisäyksen jälkeen.



Kuva 17. Hajautustaulu, johon on lisätty avain 1111.

Esimerkki 21. Lisätään lopuksi kuvan 17 tauluun avain 1001. Tietue tulee tallentaa osoittimen 01 osoittamaan tietojaksoon. Kyseisen tietojakson ylivuotojaksossa on tilaa, joten tietue voidaan tallentaa siihen. Nyt $n = 3$ ja $r = 6$, joten ehto $r \leq 1.7n$ ei enää toteudu. Tästä johtuen osoitintaulukkoon on lisättävä uusi osoitin, jonka bittiesitys on 11. Koska nyt $\lceil \log_2 4 \rceil = 2$, ei i:tä tarvitse kasvattaa. Lisättyyn uuteen tietojaksoon siirretään osoittimen 01 osoittaman tietojakson kaikki tietueet, joiden avaimen bittiesitys päättyy bittijonoon 11. Näin uuteen tietojaksoon siirtyvät tietueet, joiden avaimet ovat 0011 ja 1111. Osoittimen 01 osoittamaan tietojaksoon jäävät tietueet 1001 ja 1101. Koska nyt osoittimen 01 osoittama tietojakso sisältää vain kaksi tietuetta, voidaan tietojaksoon liittyvä ylivuotojakso poistaa. Kuva 18 esittää hajautustaulun tilaa tietueen 1001 lisäämisen jälkeen.



Kuva 18. Hajautustaulu, johon on lisätty avain 1001.

3.9 Hajautusindeksi PostgreSQL-tietokannassa

PostgreSQL-tietokannassa hajautusindeksi voidaan luoda seuraavalla lauseella:

```
CREATE INDEX name ON table USING HASH (column);
```

Lauseessa käytetyillä muuttujilla on seuraavat merkitykset:

- name on indeksiä kuvaava ja yksilöivä nimi
- table on tietokannan taulu, joka indeksoidaan
- column on indeksoitavan taulun sarake, jonka perusteella taulu indeksoidaan.

Samoin kuin B-puu voidaan hajautusindeksi poistaa tietokannasta DROP INDEX -lauseella.

Hajautusindeksillä voidaan käsitellä ainoastaan yksinkertaisia *yhtäsuuruusvertailuja*. Tästä johtuen indeksistä haettaessa voidaan käyttää vain = operaatiota.

Esimerkki 22. Oletetaan, että PostgreSQL-tietokannassa on taulu *Asiakas* sisältäen kuvan 8 esittämät rivit.

Indeksoidaan *Asiakas*-taulu sarakkeen *Ika* mukaan käyttäen hajautusindeksiä. Indeksi voidaan luoda seuraavalla komennolla:

```
CREATE INDEX Asiakas_hajautus_indeksi ON Asiakas USING HASH (Ika);
```

Indeksin luonnin jälkeen sen avulla voidaan hakea asiakkaita esimerkiksi seuraavasti:

```
SELECT * FROM Asiakas WHERE Ika=30;
```

Koska PostgreSQL-tietokannan hajautusindeksi mahdollistaa pelkästään yhtäsuuruusvertailut, indeksin avulla ei ole mahdollista hakea esimerkiksi seuraavanlaisella haulla:

```
SELECT * FROM Asiakas WHERE Ika>40;
```

Vaikka hajautusindeksi ei mahdollista kyseistä arvovälikyselyä, voi kyselyn kuitenkin tehdä tietokantaan. Tällöin hakua tehtäessä tietokanta ei käytä apunaan indeksiä vaan suorittaa peräkkäisen haun tarkastaen kaikki asiakkaat.

3.10 Hajautusindeksin tehokkuushypoteesi

Garcia-Molina et al. (2002) pitävät kohdissa 3.1, 3.2 ja 3.3 esitettyä hajautusindeksiä varsin tehokkaana. Jos indeksin avaimet mahtuvat tietojaksoihin ja ylivuotojaksoja ei tarvita, on hajautus hyvin tehokas. Indeksistä hakemiseen tarvitaan yksi *I/O*-operaatio. Indeksiin lisääminen ja poistaminen vievät kumpikin kaksi levyoperaatiota.

Jos hajautusindeksi koostuu useista ylivuotojaksoista, indeksin tehokkuus heikkenee. Siksipä ylivuotojaksoja tehokkaampi ratkaisu on käyttää dynaamista hajautusta, jossa tietojaksojen määrää kasvatetaan tallennustilan loppuessa.

Kohdissa 3.4 ja 3.5 esitellyllä *laajentavalla* hajautuksella on muutamia tärkeitä etuja, joista tärkeimpänä Garcia-Molina et al. (2002) mainitsevat sen, ettei tietuetta haettaessa koskaan tarvitse käydä läpi kuin yksi tietojakso. Tämän lisäksi joudutaan aina käymään läpi osoitintaulukko, mutta jos se pidetään muistissa, osoittimien haku ei vaadi yhtään levyoperaatiota.

Laajentavan hajautuksen suurimpana heikkoutena voidaan pitää osoitintaulukon laajenta-

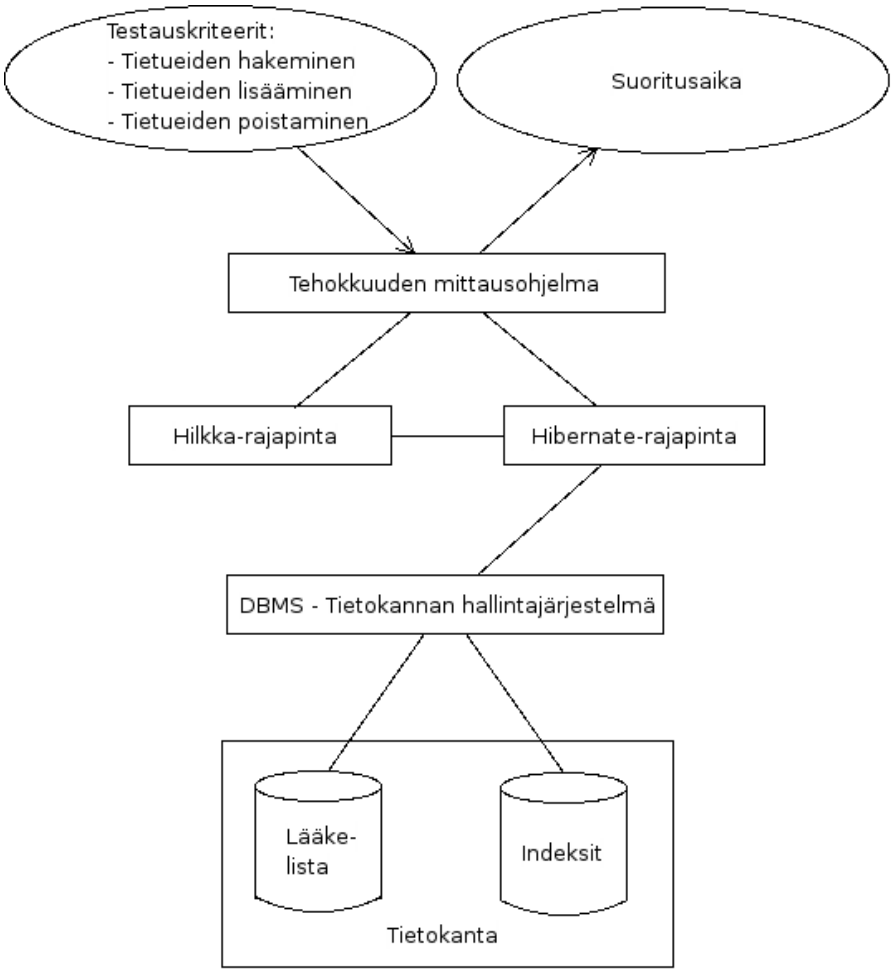
mista. Jos osoitintaulukon koko *i* on iso, on laajennustyö huomattavan suuri. Tällöin päivitys voi viedä paljon aikaa ja aiheuttaa katkon tietokannan käytössä. Toisena heikkoutena voidaan pitää tilannetta, jossa tietojaksot ovat pieniä ja huonon hajautusfunktion valinnan johdosta suurin osa tietueista hajaantuu samalle tietojaksolle. Tällöin osoitintaulukkoa joudutaan laajentamaan jatkuvasti, vaikka muissa tietojaksoissa olisi paljon vapaata tilaa.

PostgreSQL-tietokannassa *hajautusindeksi* on toteutettu käyttämällä lineaarisesti laajentavaa Litwinin *Linear hashing* -algoritmia (PostgreSQL, 2007; Litwin, 1980). Garcia-Molina et al. (2002) pitävät kohdissa 3.6, 3.7 ja 3.8 esitettyä lineaarista hajautustaulukkoa tehokkaimpana hajautusindeksin toteutusmenetelmänä.

PostgreSQL-manuaalin mukaan testaus on osoittanut, että PostgreSQL-tietokannan hajautusindeksi ei suoriudu mistään tilanteesta paremmin kuin B-puu-indeksi (PostgreSQL, 2007). Lisäksi hajautusindeksin koko ja luontiaika ovat paljon epäedullisempia verrattuna B-puuhun. Näistä syistä johtuen hajautusindeksin käyttöä ei suositella PostgreSQL-tietokannassa missään tilanteessa, ja se onkin mukana vain perinteikkäistä syistä. Hajautusindeksin voi aina korvata B-puu-indeksillä, joka on hajautusindeksiä tehokkaampi.

4 B-puun ja hajautusindeksin tehokkuuden vertailu

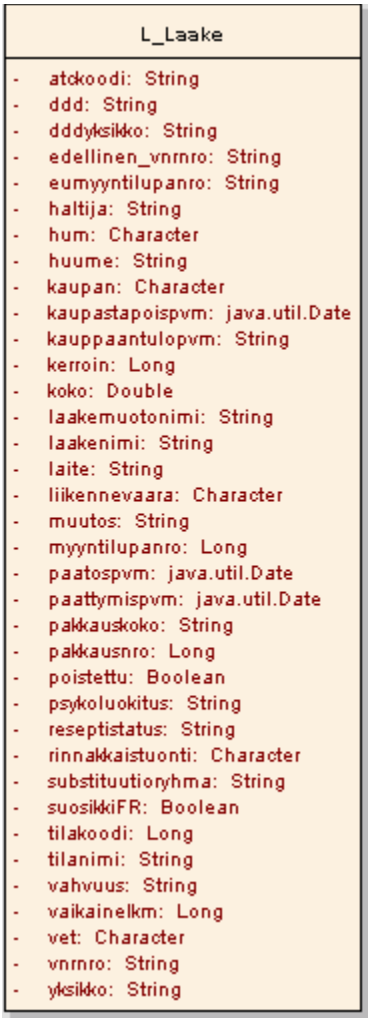
Tässä luvussa pyritään tarkentamaan kohdissa 2.7 ja 3.10 esitettyjä tehokkuushypoteeseja vertailemalla B-puuta ja hajautusindeksiä erilaisissa tilanteissa käyttäen tätä tarkoitusta varten tehtyä ja liitteessä 1 esitettyä Java-kielistä testiohjelmaa. Kuvassa 19 on esitetty testiympäristön rakenne. Testiohjelmalla voidaan ajaa kyselyitä satunnaisilla syötteillä useampia kertoja peräkkäin. Lisäksi kyselyiden kuluttamia aikoja voidaan mitata. Satunnaisten syötteiden generointiin testiohjelmassa käytetään *java.util.Random*-luokan *nextInt*-funktiota, joka palauttaa tasajakaumaa noudattavia kokonaislukuja (Java, 2007).



Kuva 19. Testausympäristön rakenne

Vertailussa käytetään apuna Fastroi Oy:n Asukastieto-ohjelma Hilkka 2:n tietokantaa ja Hilkka-ohjelman sisältämiä tietokannan käsittelyyn liittyviä funktioita. Hilkka-ohjelmassa käytetään PostgreSQL-tietokantaa, joka sisältää Lääkelaitoksen toimittaman lääkelistan. Tehdyissä testeissä käytetään lääkelistan kesäkuun 2007 versiota, joka sisältää yhteensä 47 322 lääketietuetta. Lääkkeiden hakeminen tietokannasta on ollut Hilkka-ohjelmassa hidasta, ja siksi indeksoinnin tutkiminen on varsin tarpeellista ajatellen myös Hilkka-ohjelman jatkokehitystä.

Lääkkeet on tallennettu Hilkan tietokannassa L_Laake-nimiseen tauluun. Kuvassa 20 on esitetty Enterprise Architect -ohjelmalla suunniteltu L_Laake-luokka attribuutteineen. Tietokannassa olevat L_Laake-tietueet eivät ole järjestyksessä minkään sarakkeen mukaan, vaan ne ovat siinä järjestyksessä, jossa Lääkelaitos on ne toimittanut. Missään taulun sarakkeessa ei myöskään ole unique-rajoitetta, joten kahdella eri tietueella voi olla sama arvo samassa tietokantataulun sarakkeessa. Tästä johtuen tässä tutkimuksessa L_Laake-tauluun ei luotu unique-indeksejä.



Kuva 20. Hilkka-ohjelman L_Laake-luokka

Hilkka-ohjelma on toteutettu käyttäen Java Spring -kehitysympäristöä (Java Spring, 2007). Java-koodin ja relaatiotietokannan välinen kytkentä on Hilkka-ohjelmassa hoidettu käyttämällä Hibernatea. Hibernaten avulla relaatiotietokannan taulun rivit saadaan näkymään Java-koodissa puhtaina luokkina (Hibernate, 2007). Suoritetuissa testeissä käytetään apuna myös joitakin Hilkka-ohjelmaan valmiiksi generoituja funktiota, niinpä tutkimus vastaa tältä osin ohjelman todellista käyttöä. Käytettäessä Hiberantea Java-ohjelmassa kirjoitetut tietokantakyselyt eivät ole perinteistä SQL-kieltä, vaan kyselyt kirjoitetaan käyttämällä HQL-kieltä (Hibernate Query Language), joka on kuitenkin hyvin pitkälle

SQL-kielen kaltaista.

Aluksi testataan indeksien tehokkuutta erilaisilla tietokantahauilla. Hauissa tarkastellaan indeksien vaikutusta haettaessa tietueita numeerisen sekä merkkijonotyyppisen tiedon perusteella. Numeeriseksi testisarakkeeksi L_Laake-taulusta on valittu lääkkeen pakkausnumero ja merkkijonotestauksessa käytetään lääkkeen lääkenimeä. Hakujen testaamisen jälkeen on tarkoitus tarkastella indeksoinnin vaikutusta tietokannan nopeuteen, kun tietokantaan lisätään ja tietokannasta poistetaan tietueita. Lopuksi keskitytään indeksien luomiseen ja poistamiseen kuluvaan aikaan. Viimeksi mainitussa osatutkimuksessa ei käytetty kuvassa 19 esitettyä testiympäristöä, vaan testit suoritettiin käyttämällä PostgreSQL-tietokannan SQL-työkalua.

Testiympäristönä tutkimuksissa käytetään Pentium® 4 CPU 3,20 GHz -tasoista tietokonetta, jossa on keskusmuistia 1 GB. Käyttöjärjestelmänä tietokoneessa on Windows XP Professional Versio 2002 Service Pack 2. Tutkimuksessa käytetyn PostgreSQL-tietokannan versio on 8.2 ja käytetyn Java JDK:n versio on 1.6.0

4.1 Satunnaisen pakkausnumeron hakeminen

Ensimmäiseksi B-puun ja hajautusindeksin eroa tutkittiin numeerisen tiedon osalta hakemalla L_Laake-taulusta yksittäisiä tietueita pakkausnumeron perusteella. Jokaisella tietokannan sisältämällä lääketietueella on yksiköllinen pakkausnumero, jonka tietotyyppinä Java-luokassa on Long ja tietokannan taulussa 64-bittinen bigint (PostgreSQL, 2007). Pienin tietokannassa oleva pakkausnumero on 109 ja suurin 84 740. Tässä tutkimuksessa on tarkoituksena hakea lääkkeitä tietokannasta satunnaisella pakkausnumerolla väliltä [0, 100 000). Koska tietokannassa on yhteensä 47 322 lääketietuetta, noin joka toisella haulla löytyy yksi tietue. PostgreSQL-manuaalin mukaan on odotettavaa, että sekä B-puu että hajautusindeksi tulevat nopeuttamaan tutkitun kaltaisia tietokantahakuja. Yleisesti Postg-

reSQL-manuaalista käy ilmi, että hajautusindeksi olisi jonkin verran hitaampi verrattuna B-puuhun (PostgreSQL, 2007).

Kuvissa 21 ja 22 on esitetty tutkimuksessa käytetyt testifunktiot. Kuvan 22 esittämällä *findByPakkausnumero*-funktiolla voidaan hakea tietokannasta kaikki L_Laake-oliot, joilla on funktiolle parametrina annettu pakkausnumero. Varsinainen testi suoritetaan kuvan 21 esittämällä *testPakkausnumero*-funktiolla. Tämä funktio arpoo satunnaisia pakkausnumeroita väliltä [0, 100 000), joiden perusteella L_Laake-olioita haetaan tietokannasta käyttäen *findByPakkausnumero*-funktiota. Viimeksi mainittu funktio käyttää tietokantahaussa seuraavaa HQL-lausetta:

```
FROM L_Laake WHERE pakkausnro = arvottu_numero
```

```
public void testPakkausnumero(int maara){
        Random rand = new Random();

        for (int i=0; i<maara; i++) {
                List<L_Laake> list = findByPakkausnumero(new Long(rand.nextInt(100000)));
        }
}
```

Kuva 21. testPakkausnumero-funktio, jolla testataan L_Laake-olioiden hakemista tietokannasta satunnaisen pakkausnumeron perusteella.

```
@SuppressWarnings("unchecked")
public List<L_Laake> findByPakkausnumero(final Long pakkausnumero) {
        HibernateTemplate hibernateTemplate = getHibernateTemplate();

        List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {

                        String sQuery = "FROM L_Laake WHERE pakkausnro = ?";
                        Query query = session.createQuery(sQuery);
                        query.setLong(0, pakkausnumero);
                        return query.list();
                }
        });
        return list;
}
```

Kuva 22. findByPakkausnumero-funktio. Tällä funktiolla voidaan hakea tietokannasta L_Laake-oliot, joilla on parametrina annettu pakkausnumero.

Tutkimuksessa suoritettujen askelten määrä 2 000 000 on määritetty pienellä alustavalla testillä niin, että ilman indeksointia testiajo kestäisi suurin piirtein 12 tuntia. Testin ajaminen indeksoimattomaan tietokantaan edellä mainitulla askelmäärällä vei aikaa tarkalleen 11 h 41 s.

Seuraavaksi tietokannan L_Laake-taulun pakkausnro-sarake indeksoitiin B-puu-indeksillä. Tämän jälkeen testi ajettiin uudelleen samalla askelmäärällä ja testin ajaminen vei aikaa 41 min 12 s. Testin tulos oli hyvä. Odotettua oli, että B-puu nopeuttaisi hakuja merkittävästi, ja tässä tapauksessa hakujen nopeus kasvoi 16-kertaiseksi, eli niiden kesto pieneni kuuteen sadasosaan indeksoimattomaan tietokantaan verrattuna.

Tämän jälkeen testi ajettiin vielä kerran uudelleen tutkien hajautusindeksin toimintaa. Tämän testin ajaminen vei aikaa 43 min 27 s. Tämä oli lähes sama aika kuin B-puu-indeksillä saavutettu tulos. Verrattuna indeksoimattomaan tietokantaan hakujen nopeus kasvoi 15,2-kertaiseksi.

Testin tuloksena voidaan sanoa, että tulokset olivat odotettuja. B-puu ja hajautusindeksi olivat myös yllättävän tasaisia, mutta B-puu oli kuitenkin hieman parempi. Taulukossa 1 on esitetty yhteenveto testin tuloksista. Tutkimus osoittaa, ettei tällaisen numeerisen tiedon indeksoinnissa ole juuri ollenkaan eroa B-puun ja hajautusindeksin välillä, mutta silti indeksointi kannattaa.

Taulukko 1: L_Laake-olioiden hakeminen tietokannasta satunnaisen pakkausnumeron perusteella.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 2 000 000 kertaa satunnaisen [0, 100 000) pakkausnumeron perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	11 h 41 s	19,8 ms
B-puu	41 min 12 s	1,2 ms
Hajautusindeksi	43 min 27 s	1,3 ms

4.2 Pakkausnumeroiden hakeminen satunnaiselta väliltä

Tässä kohdassa tutkitaan indeksoinnin vaikutusta numeerisen tiedon arvovälikyselyihin. Tarkoituksena on hakea tietokannasta lääketietueita pakkausnumeron perusteella sellaisilla kyselyllä, jossa pakkausnumero on tietyllä satunnaisella välillä.

Koska B-puu mahdollistaa arvovälikyselyt (kohta 2.2), ennen testiä voitiin olettaa, että B-puu tulee nopeuttamaan hakuja ehkäpä jopa enemmän kuin kohdassa 4.1 suoritetussa tutkimuksessa. Näin, koska tällä kertaa tietokantaan suoritetaan vähemmän kyselyitä ja tietokannanhallintajärjestelmä joutuu tekemään suuremman työn, jonka ei indeksoinnin avustamana pitäisi olla kuitenkaan kovin vaativa (Garcia-Molina et al. 2002; Lehman ja Yao, 1981; PostgreSQL, 2006). Koska hajautusindeksi ei mahdollista arvovälikyselyitä (PostgreSQL, 2007), on oletettavaa, että hajautusindeksillä suoritetun haun tulos on lähes

sama kuin ilman indeksointia suoritetun testin tulos.

Testausohjelman funktiot on esitetty kuvissa 23 ja 24. Koska lääkkeiden pakkausnumerot ovat välillä [109, 84 740] tässä testissä pakkausnumeroita haettiin kussakin kyselyssä arvotulta väliltä [a,b]. Ensimmäiseksi kuvan 23 esittämä funktio *testPakkausnumeroVali* arpoo välin aloitusarvon a väliltä [0,100 000). Tämän jälkeen arvotaan välin päätepiste väliltä [a,100 000). Koska testeissä käytetty satunnaislukuja muodostava *getNextInt*-funktio noudattaa tasajakaumaa, alkupisteen keskiarvo on 50 000 ja loppupisteen keskiarvo 75 000. Näin saadun arvovälin keskimääräinen pituus on 25 000 ja välillä on keskimäärin noin 12 000 tietuetta. Arvovälin määrittämisen jälkeen kutsutaan kuvan 24 esittämää *findByPakkausnumeroVali*-funktiota, joka hakee tietokannasta kaikki pakkausnumerot, jotka ovat annetulla välillä käyttäen seuraavaa HQL-lausetta:

```
FROM L_Laake
  WHERE pakkausnro >= alku AND pakkausnro <= loppu
```

```
public void testPakkausnumeroVali(int maara) {
        Random generator = new Random();

        for (int i=0; i<maara; i++) {
                Long alku = new Long(generator.nextInt(100000));
                int jaljella = 100000-alku.intValue();
                Long loppu = new Long(generator.nextInt(jaljella) + alku);

                List<L_Laake> lista = findByPakkausnumeroVali(alku, loppu);
        }
}
```

Kuva 23. testPakkausnumeroVali-funktio. Tällä funktiolla testataan L_Laake-olioiden hakemista satunnaiselta pakkausnumeroväliltä.

```
@SuppressWarnings("unchecked")
public List<L_Laake> findByPakkausnumeroVali(final Long alku, final Long loppu) {
        HibernateTemplate hibernateTemplate = getHibernateTemplate();

        List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {

                        String sQuery = "FROM L_Laake " +
                                        "WHERE pakkausnro >= ? AND pakkausnro <= ?";
                        Query query = session.createQuery(sQuery);
                        query.setLong(0, alku);
                        query.setLong(1, loppu);
                        return query.list();
                }
        });
        return list;
}
```

Kuva 24. findByPakkausnumeroVali-funktio, jolla voidaan hakea tietokannasta kaikki L_Laake-oliot, joiden pakkausnumero on suurempi tai yhtä suuri kuin parametri alku ja pienempi tai yhtä suuri kuin parametri loppu.

Ilman indeksointia pienellä alustavalla testillä selvitettiin, että 12 tunnissa pystyttäisiin suorittamaan noin 20 000 kyselyä. Niinpä ensimmäiseksi testattiin 20 000 kertaa lääketietueiden hakemista satunnaisen pakkausnumerovälin perusteella. Tämän testin ajaminen vei aikaa 14 h 20 min 16 s.

Seuraavaksi tietokantaan luotiin pakkausnumerolle B-puu-indeksi ja sama testi suoritettiin uudelleen. Tämän testin suorittaminen vei aikaa 14 h 22 min 34 s. Testin tulos on yllättävä, sillä se on samaa suuruusluokkaa ja jopa yli kaksi minuuttia (tarkalleen ottaen 1.003 kertaa) hitaampi kuin ilman indeksointia suoritetun testin tulos. Ennen tukimusta oletettiin, että B-puun pitäisi nopeuttaa tietokannan toimintaa tässä tilanteessa merkittävästi.

Tämän osatutkimuksen lopuksi sama testi ajettiin vielä tietokannalle, jossa pakkausnro-

sarake on indeksoitu käyttämällä hajautusindeksiä. Tämän testiajon suorittaminen vei aikaa 14 h 24 min 13 s, mikä on samaa suuruusluokkaa, mutta noin 4 minuuttia pidempi aika kuin ilman indeksointia suoritetun testin tulos. Tarkalleen ottaen kyselyt olivat tässä testissä 1,005 kertaa hitaampia verrattuna indeksoimattomaan tietokantaan. Testin tulos oli odotettu, sillä on tiedossa, että PostgreSQL-tietokannan hajautusindeksi ei tue arvovälikyselyitä.

Kokonaisuutena tämän testin tulos (taulukko 2) ei ollut odotusten mukainen. Ennen tutkimusta oletettiin, että B-puu tulisi toimimaan nopeammin verrattuna hajautusindeksiin ja indeksoimattomaan tietokantaan. Testiajot veivät kuitenkin jokaisessa tapauksessa aikaa lähes yhtä paljon, eikä tämän tutkimuksen perusteella pystytä suosittelemaan tähän tarkoitukseen mitään indeksointimenetelmää.

Taulukko 2: L_Laake-tietueiden hakeminen tietokannasta satunnaisen pakkausnumerovälin perusteella.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 20 000 kertaa. Hakuehtona satunnainen pakkausnumeroväli väliltä [0, 100 000). Tietueita ei ole järjestetty tietokannassa pakkausnumeron perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	14 h 20 min 16 s	2580,8 ms
B-puu	14 h 22 min 34 s	2587,7 ms
Hajautusindeksi	14 h 24 min 13 s	2592,7 ms

Tämän testin epäonnistumisen syynä voi olla levyhakujen raskaus, sillä jokaisella testikierroksella kiintolevyltä joudutaan hakemaan jopa tuhansia tietueita. Testiohjelmassa löytyneet tietueet ladataan myös tietokoneen keskusmuistiin, mikä on raskasta verrattuna itse tietokannanhallintajärjestelmän tekemään työhön. Tätä ongelmaa on tarkoitus tutkia kohdassa 4.6, jossa levyhakuja ja tietueiden palauttamista testiohjelmaan on pyritty helpottamaan luopumalla oliolistan palauttamisesta testiohjelmalle palauttamalla pelkästään

koostetieto haulla löytyvien tietueiden määrästä.

Syynä tässä kohdassa todettuun hakujen hitauteen voi myös olla se, että tietueet on lisätty L_Laake-tauluun satunnaisessa järjestyksessä pakkausnumeron suhteen. Näin ollen haettaessa tietueita peräkkäisten pakkausnumeroiden perusteella joudutaan tietueita lukemaan levyltä satunnaisista paikoista eli klustereista. Testiympäristössä yhden levyn klusterin koko on NTFS-järjestelmän oletusarvo 4 KB (Microsoft, 2003). L_Laake-taulun sisältämien sarakkeiden perusteella voidaan laskea yhden täyden lääketietueen vievän levytilaa 588 B. Tietueen koko voi kuitenkin vaihdella, kun merkkijonotyyppinen tieto on tallennettu varchar-tyyppisiin sarakkeisiin (PostgreSQL, 2007). Näin ollen yhteen klusteriin mahtuu noin kuusi L_Laake-tietuetta. NTFS-tiedostojärjestelmä hakee kiintolevyltä aina yhden kokonaisen klusterin kerrallaan. Jos tietueet eivät ole järjestyksessä klustereiden sisällä, yhtä L_Laake-tietuetta haettaessa haetaan levyltä myös turhaa tietoa. Kohdassa 4.3 on tarkoitus tutkia edellä kuvattua ongelmaa klusteroimalla L_Laake-taulu pakkausnumeron perusteella. Tämä tarkoittaa sitä, että taulun sisältämät tietueet kirjoitetaan kiintolevylle pakkausnumeron mukaan nousevaan järjestykseen.

4.3 Pakkausnumeroiden hakeminen satunnaiselta väliltä järjestetystä taulusta

Kohdassa 4.2 tutkittiin L_Laake-tietueiden hakemista tietokannasta satunnaisen pakkausnumerovälin perusteella. Tämän tyyppisten arvovälikyselyiden huomattiin olevan hitaita, ja tässä kohdassa tullaan tutkimaan, parantaako tietueiden järjestäminen kiintolevyllä kyselyiden nopeutta. Kohdassa 4.2 indeksoinnilla ei todettu olevan nopeuttavaa vaikutusta suoritettuihin kyselyihin. Niinpä on kiintoisaa nähdä, onko indeksoinnilla vaikutusta, jos tietueet on järjestetty.

Tässä kohdassa on tarkoitus suorittaa arvovälikyselyitä tietokantaan, jonka L_Laake-taulu

on järjestetty nousevaan järjestykseen tietokoneen kiintolevylle pakkausnro-sarakkeen perusteella. Näin olleen tietueita ei pitäisi tarvita erityisemmin hakea. Riittää vain ensimmäisen pienimmän pakkausnumeron omaavan tietueen etsiminen ja tietueiden lukeminen kiintolevyltä järjestyksessä eteenpäin. Jos tilanne on näin ihanteellinen, indeksoinnista ei ole muuta hyötyä kuin se, että sen avulla löydetään ensimmäinen tietue nopeasti. Kohdassa 4.1 huomattiin, että sekä B-puu että hajautusindeksi nopeuttavat yksittäisen tietueen löytämistä selvästi. Näin ollen voisi olettaa, että B-puu ja hajautusindeksi nopeuttavat tässäkin kohdassa testattavia hakuja hieman.

Seuraavat testit on tarkoitus suorittaa käyttäen samoja, kuvien 23 ja 24 esittämiä testifunktioita, kuin mitä käytettiin kohdassa 4.2. Myöskin testikierrosten määrä 20 000 tullaan pitämään samana kuin kohdassa 4.2, joten saatuja tuloksia voidaan verrata suoraan keskenään.

PostgreSQL tietokannassa tietueet voidaan järjestää (klusteroida) seuraavalla SQL-lauseella (PostgreSQL, 2007):

```
CREATE TABLE newtable AS
   SELECT * FROM table ORDER BY columnlist;
```

Edellä esitetty SQL-lause luo uuden taulun, jossa entisen taulun sisältö on järjestettynä halutun sarakkeen perusteella. Tämän komennon ajon jälkeen vanha taulu on poistettava DROP-lauseella. Lopuksi uusi taulu voidaan nimetä uudelleen vanhan nimiseksi ALTER-lauseella.

```
CREATE TABLE l_laake_tmp AS
   SELECT * FROM l_laake ORDER BY pakkausnro;

ALTER TABLE l_laake_tmp ADD CONSTRAINT fk5ad2861318cf59ca FOREIGN KEY
   (laakemuotoid) REFERENCES l_laakemuoto (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE l_laake_tmp ADD CONSTRAINT fk5ad2861326c62735 FOREIGN KEY
   (atc_koodiid) REFERENCES l_atc (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE l_laake_tmp ADD CONSTRAINT fk5ad286135653a2a4 FOREIGN KEY
   (maaraamisehtoid) REFERENCES l_maaraamisehto (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE l_laake_tmp ADD CONSTRAINT fk5ad28613dc2715ac FOREIGN KEY
   (sailytysastiaid) REFERENCES l_sailytysastia (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;

ALTER TABLE l_laakkeenaineet DROP CONSTRAINT fkdf157c23371591b6 ;
ALTER TABLE lisalaake DROP CONSTRAINT fkcfb99fbe885b53d ;
ALTER TABLE laakitys DROP CONSTRAINT fk8db12a84cf173a10 ;
ALTER TABLE laakeyliherkkyys DROP CONSTRAINT fk6c68982ecf173a10 ;
ALTER TABLE preventio DROP CONSTRAINT fk459cecdee885b53d ;

DROP TABLE l_laake;
ALTER TABLE l_laake_tmp RENAME TO l_laake;
ALTER TABLE l_laake ADD CONSTRAINT l_laake_pkey PRIMARY KEY (id);
ALTER TABLE l_laake OWNER TO hilkka2user;

ALTER TABLE l_laakkeenaineet ADD CONSTRAINT fkdf157c23371591b6 FOREIGN KEY
   (laakeaineetatl_laakeid) REFERENCES l_laake (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE lisalaake ADD CONSTRAINT fkcfb99fbe885b53d FOREIGN KEY
   (l_laakeid) REFERENCES l_laake (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE laakitys ADD CONSTRAINT fk8db12a84cf173a10 FOREIGN KEY
   (laakeid) REFERENCES l_laake (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE laakeyliherkkyys ADD CONSTRAINT fk6c68982ecf173a10 FOREIGN KEY
   (laakeid) REFERENCES l_laake (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE preventio ADD CONSTRAINT fk459cecdee885b53d FOREIGN KEY
   (l_laakeid) REFERENCES l_laake (id) MATCH SIMPLE
   ON UPDATE NO ACTION ON DELETE NO ACTION;
```

Kuva 25. SQL-lauseet, joilla L_Laake-taulu voidaan järjestää pakkausnumeron perusteella.

Ennen varsinaisien testien ajamista tietokannan sisältämät L_Laake-tietueet järjestettiin kuvan 25 esittämillä SQL-lauseilla. Ensimmäiseksi luodaan uusi pakkausnumeron perusteella järjestetty L_Laake_tmp-taulu. Hilkka-ohjelmassa viisi muuta tietokannan taulua liittyy L_Laake-tauluun. Niinpä L_Laake-taulua ei voida poistaa, ennen kuin näistä viittaavista tauluista on poistettu L_Laake-taulua koskevat CONSTRAINT-määritteet. Näiden määritteiden poiston jälkeen vanha L_Laake-taulu voidaan poistaa. Uudelle L_Laake_tmp-taululle luodaan myös samat CONSTRAINT-määritteet, jotka vanhalla L_Laake-taululla oli. Lopuksi uusi L_Laake_tmp-taulu nimetään uudelleen ja L_Laake-tauluun viittaaviin tauluihin lisätään poistetut CONSTRAINT-määritteet uudelleen.

L_Laake-taulun uudelleen järjestämisen jälkeen aloitettiin varsinainen testaus. Ensimmäiseksi testiohjelma ajettiin käyttäen tietokantaa, jossa ei ollut indeksointia L_Laake-taululle. Testikierroksia ajettiin 20 000 kappaletta, ja testin suorittaminen vei aikaa 14 h 43 min 20 s. Tämä on yllättävän pitkä aika, sillä kohdassa 4.2 saman testin suorittaminen kesti vain 14 h 20 min 16 s järjestämättömällä tietokannalla.

Seuraavaksi tietokannan L_Laake-taulu indeksoitiin pakkausnumeron perusteella käyttäen B-puu-indeksiä. Testiohjelman ajaminen vei tällä kertaa aikaa 14 h 43 min 21 s. Tämä aika on vain yhden sekunnin pidempi kuin indeksoimattomalla tietokannalla saavutettu aika. Tästä voidaan päätellä, ettei B-puulla ole minkäänlaista vaikutusta tämän tyyppisiin hakuihin.

Lopuksi testiohjelma ajettiin tietokannalle, joka oli indeksoitu pakkausnumeron perusteella käyttäen hajautusindeksiä. Tämän testiajon kokonaisajaksi saatiin 15 h 7 min 25 s. Tulos on 1,03 kertaa hitaampi verrattuna ilman indeksointia saavutettuun tulokseen. Järjestämättömän tietokannan tapauksessa hajautusindeksin ero muihin tutkittuihin tapauksiin ei ollut näin suuri.

Tulokset tässä kohdassa suoritetuista testeistä on esitetty taulukossa 3. Nämä tulokset eivät olleet odotettuja, sillä tietueiden järjestämisen toivottiin nopeuttavan tietokannan toimintaa. Tällä kertaa kyselyt olivat kuitenkin noin 20 minuuttia hitaampia kuin järjestämättömän tietokannan tapauksessa. Syytä tähän eroon on vaikea löytää. Lisäksi tässä kohdassa huomattiin, että indeksoinnilla ei ole vaikutusta tämän tyyppisiin arvovälikyselyihin, vaikka tietokannan taulun sisältämät tietueet olisivat järjestetty haussa käytettävän sarakkeen perusteella. Tästä voidaan päätellä, ettei PostgreSQL-tietokannan B-puu-indeksi toimi arvovälikyselyiden yhteydessä eikä indeksointia kannata käyttää.

Taulukko 3: L_Laake-tietueiden hakeminen tietokannasta satunnaisen pakkausnumerovälin perusteella. Tietueet on järjestetty levylle pakkausnumeron perusteella.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 20 000 kertaa. Hakuehtona satunnainen pakkausnumeroväli väliltä [0, 100 000). Tietueet on järjestetty tietokannassa pakkausnumeron perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	14 h 43 min 20 s	2650,0 ms
B-puu	14 h 43 min 21 s	2650,1 ms
Hajautusindeksi	15 h 7 min 25 s	2722,3 ms

4.4 Arvovälin pituuden vaikutus arvovälikyselyn nopeuteen Hilkka-ohjelman oletustietokannalla

Kohdissa 4.2 ja 4.3 tehdyissä tutkimuksissa indeksoinnin vaikutuksesta arvovälikyselyihin ei saatu aikaan toivottuja tuloksia. Yksi syy testien epäonnistumiseen voi olla tietokannan kyselyiden optimoija. PostgreSQL-tietokannassa on kyselyiden optimoija, joka optimoi kyselyiden suoritustapaa niin, että kysely pystytään suorittamaan mahdollisimman nopeasti (PostgreSQL 2007). Tietokannan tauluun luotu indeksi ei vielä takaa sitä, että tietokannanhallintajärjestelmä käyttäisi indeksiä kyselyä suoritettaessa. Voi olla niin,

että kyselyn optimoija katsoo paremmaksi suorittaa kyselyn selaamalla kaikki tietueet läpi käyttämättä indeksointia. Arvovälikyselyiden yhteydessä yksi syy indeksien käyttämättömyyteen voi olla arvovälin pituus. Kun arvoväli on pitkä - jolloin suuri osa taulun sisältämistä tietueista kuuluu kyselyn tulosjoukkoon - voi optimoija katsoa parhaaksi selata kaikki tietueet läpi käyttämättä indeksiä. Lyhyellä arvovälillä indeksoinnista olisi todennäköisesti enemmän hyötyä, ja silloin optimoija saattaisi käyttää indeksiä.

Kohdissa 4.2 ja 4.3 suoritetuissa testeissä arvoväli arvottiin satunnaisesti, mutta kuitenkin niin, että keskimääräinen arvovälin pituus oli 25 000. Pakkausnumerot jakaantuivat välille [109, 84 740], joten keskimäärin yhdessä tulosjoukossa on tietueita noin yksi neljäsosa koko taulun sisällöstä eli verrattain paljon. Tästä herää kysymys, käyttikö optimoija indeksiä edellä tehdyssä testeissä ollenkaan suuren arvovälin pituuden vuoksi. Siksipä tässä kohdassa tullaan tutkimaan arvovälin pituuden vaikutusta B-puu-indeksin toimintaan. Lisäksi tutkitaan, vaikuttaako tietueiden järjestäminen hakujen nopeuteen ja kannattaako yhtäaikainen taulun järjestäminen ja indeksointi. Hajautusindeksin tutkiminen jätettiin tästä kohdasta kokonaan pois, koska sen ei ole todettu nopeuttavan arvovälikyselyitä ollenkaan.

Tämän kohdan testit on tarkoitus suorittaa käyttämällä kuvassa 26 esitettyä *testKiinteaArvovali*-funktiota, joka tekee tietokantaan kiinteän mittaisia arvovälikyselyitä. Edellä mainittu funktio käyttää tietokantahauissa kuvassa 24 esitettyä *findByPakkausnumeroVali*-funktiota. Arvovälin alkupiste arvotaan ja loppupiste saadaan lisäämällä alkupisteeseen arvovälin pituus. Testejä on tarkoitus ajaa käyttäen neljää eripituista arvoväliä. Näitä arvovälejä ovat 10, 100, 1 000, 10 000. Kunkin arvovälin kohdalla testin pituudeksi on suunniteltu noin 3 tuntia.

```
public void testKiinteaArvovali(int maara, int pituus) {
        Random generator = new Random();

        for (int i=0; i<maara; i++) {
                Long alku = new Long(generator.nextInt(100000));
                if ((alku+pituus) > 100000) {
                        alku=100000L-pituus;
                }
                Long loppu = new Long(alku + pituus);

                List<L_Laake> lista = findByPakkausnumeroVali(alku, loppu);
        }
}
```

Kuva 26. testKiinteaArvovali-funktio jolla voidaan testata arvovälikyselyitä käyttäen kiinteän pituista arvoväliä.

Testit aloitettiin käyttämällä Hilkka-ohjelman oletustietokantaa, jonka L_Laake-taulua ei ollut indeksoitu eikä järjestetty. Tällöin pienellä alustavalla testillä selvitettiin, että kolmessa tunnissa 10:n ja 100:n pituisia arvovälikyselyitä voidaan suorittaa noin 500 000 kappaletta, 1 000:n pituisia noin 100 000 kappaletta ja 10 000:n pituisia noin 10 000 kappaletta. Varsinaiset testit suoritettiin käyttäen edellä mainittuja arvoja ja näiden testien tulokset on esitetty taulukossa 4.

Taulukko 4: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä indeksoimatonta ja järjestämätöntä Hilkka-ohjelman oletustietokantaa.

Tutkittu asia	Arvovälikyselyiden suorittaminen Hilkka-ohjelman indeksoimattomaan ja järjestämättömään oletustietokantaan.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	500 000	2 h 59 min 58 s	21,6 ms
100	500 000	4 h 21 s	28,8 ms
1 000	100 000	2 h 49 min 8 s	101,5 ms
10 000	10 000	2 h 30 min 28 s	902,8 ms

Tämän jälkeen L_Laake-taulun pakkausnumero-sarake indeksoitiin B-puu-indeksillä ja edellä kuvatut testit ajettiin uudelleen. Tämän testin tulokset on esitetty taulukossa 5. Tuloksista huomataan B-puun nopeuttavan lyhyen mittaisia arvovälikyselyitä huomattavasti enemmän kuin pidempiä. 10:n pituisella arvovälillä haku nopeutui 9,4-kertaisesti, 100:n pituisella 3,0-kertaisesti ja 1 000:n pituisella 1,2-kertaisesti. 10 000 pituisella arvovälillä B-puun ei juuri enää havaittu nopeuttavan kyselyitä. Näyttää siis siltä, että tutkitun kokoista tietokantaa käytettäessä kyselyiden optimoija ei käytä B-puuta, jos arvovälin pituus ylittää 1 000, eli tulosjoukossa on tietueita noin 10% kaikista tietueista.

Taulukko 5: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä järjestämätöntä Hilkka-ohjelman oletustietokantaa, joka on indeksoitu B-puu-indeksillä.

Tutkittu asia	Arvovälikyselyiden suorittaminen Hilkka-ohjelman järjestämättömään oletustietokantaan käyttäen B-puu-indeksiä.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	500 000	18 min 53 s	2,3 ms
100	500 000	1 h 19 min 13 s	9,5 ms
1 000	100 000	2 h 16 min 5 s	81,7 ms
10 000	10 000	2 h 26 min 10 s	877,0 ms

Seuraavaksi sama testi suoritettiin edellä kuvatuilla arvoilla tietokannalle, jonka L-Laake-taulu oli järjestetty pakkausnumeron perusteella kuvassa 25 esitetyllä menetelmällä. Tässä testissä L_Laake-taulussa ei käytetty indeksejä. Testin tulokset on esitetty taulukossa 6. Verrattaessa tuloksia ilman järjestämistä saatuihin tuloksiin (taulukko 4) huomataan, ettei tietokannan järjestämisellä ole vaikutusta arvovälikyselyiden nopeuteen. Tähän voi olla syynä se, että tietueet mahtuvat käyttöjärjestelmän tai tietokannanhallintajärjestelmän ylläpitämään välimuistiin. Tietueiden ollessa välimuistissa ei tietueiden järjestyksellä kiintolevyllä ole mitään merkitystä.

Taulukko 6: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä indeksoimatonta Hilkka-ohjelman oletustietokantaa, jonka L_Laake-taulu on järjestetty.

Tutkittu asia	Arvovälikyselyiden suorittaminen Hilkka-ohjelman indeksoimattomaan oletustietokantaan, jonka L_Laake-taulu on järjestetty pakkausnumeron perusteella.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	500 000	3 h 2 s	21,6 ms
100	500 000	3 h 59 min 45 s	28,8 ms
1 000	100 000	2 h 47 min 27 s	100,5 ms
10 000	10 000	2 h 28 min 26 s	890,8 ms

Lopuksi tutkittiin, tuottaisiko B-puu-indeksi ja taulun järjestäminen yhdessä paremman tuloksen. Niinpä Hilkka-ohjelman oletustietokanta järjestettiin kuvan 25 esittämällä järjestysmenetelmällä ja tietokantaan luotiin B-puu-indeksi. Arvovälien pituudet ja testikierrosten määrät pidettiin samoina ja edellä kuvattu testi ajettiin uudelleen. Tämän testin tulokset on esitetty taulukossa 7. Tälläkään kertaa tietokannan järjestämisestä ei ollut hyötyä, sillä tulokset ovat hyvin lähellä pelkällä B-puulla saavutettuja tuloksia (taulukko 5).

Taulukko 7: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä Hilkka-ohjelman oletustietokantaa, jonka L_Laake-taulu on järjestetty ja indeksoitu B-puu-indeksillä.

Tutkittu asia	Arvovälikyselyiden suorittaminen Hilkka-ohjelman oletustietokantaan, jonka L_Laake-taulu on järjestetty ja indeksoitu B-puu-indeksillä pakkausnumeron perusteella.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	500 000	18 min 50 s	2,3 ms
100	500 000	1 h 18 min 49 s	9,5 ms
1 000	100 000	2 h 15 min 15 s	81,2 ms
10 000	10 000	2 h 23 min 57 s	863,7 ms

Edellä suoritetuista testeistä kävi ilmi, että tämän kokoisella tietokannalla tietokannan taulua ei kannata järjestää nopeampien arvovälikyselyiden toivossa. Kyselyt eivät nopeutuneet arvovälin pituudesta riippumatta. Järjestäminen ei myöskään nopeuttanut B-puulla indeksoidun kannan toimintaa. Testeissä huomattiin B-puu-indeksin nopeuttavan arvovälikyselyitä tilanteissa, joissa arvovälin pituutena oli 10 ja 100. Näissä tilanteissa kyselyn tulosjoukossa oli vain pieni osa taulun sisältämistä tietueista. Pidemmillä arvoväleillä B-puu-indeksistä oli vähemmän apua. 1 000:n pituisella arvovälillä voidaan B-puun sanoa auttavan vain hieman, jolloin kyselyt nopeutuivat 1,2-kertaisesti, mutta pisimmällä tutkitulla arvovälillä B-puusta ei ollut mitään apua. Siispä jos arovälikyselyiden tuottama tulosjoukko ei ole kovin iso verrattuna tietueiden kokonaismäärään, B-puu-indeksiä kannattaa käyttää arvovälikyselyiden yhteydessä.

Yksi selitys edellä havaitulle voi olla tämän kohdan alussa kuvattu kyselyiden optimoija, joka optimoi kyselyitä niin, että pidemmillä arvoväleillä tauluun tehdään täydellinen läpikäynti käyttämättä indeksiä. Koska Hilkka-ohjelman oletustietokanta ei sisällä kovin paljon lääketietueita, voi olla myös mahdollista, että läpikäynnin yhteydessä tietueet tallentuvat käyttöjärjestelmän tai tietokannan ylläpitämään välimuistiin. Näin ollen tietueiden se-

laaminen on nopeaa eikä indeksiä siten tarvita. Tätä teoriaa tukee myös se, että tietokoneen välimuistin käyttöä tarkasteltiin tämän kohdan testejä tehtäessä CachemanXP-ohjelmalla (CachemanXP, 2008). Tällöin poikkeuksetta huomattiin, että testejä ajettaessa tietokoneen välimuistia käytettiin 10 MB[3]. Niinpä kohdassa 4.5 näitä testejä haluttiin suorittaa uudelleen käyttämällä isompaa tietokantaa, joka ei ehkä mahtuisikaan keskusmuistiin.

4.5 Arvovälin pituuden vaikutus arvovälikyselyn nopeuteen isolla tietokannalla

Kohdassa 4.4 tutkittiin B-puu-indeksin vaikutusta arvovälikyselyihin hakemalla tietueita eripituisilla arvoväleillä Hilkka-ohjelman oletustietokannasta, joka sisältää 47 322 lääketietuetta. Tällöin huomattiin, että pitkien arvovälikyselyiden yhteydessä B-puu-indeksi menettää tehokkuuteensa. Lisäksi huomattiin, ettei tietokannan taulun järjestämisestä ole hyötyä. Tässä kohdassa samat testit on tarkoitus ajaa uudelleen käyttäen kohdassa 4.11 syntynyttä isompaa tietokantaa, joka sisältää 2 047 322 lääketietuetta[4]. Koska tietokanta on nyt isompi, ei kyselyiden optimoija todennäköisesti selaa enää jokaisella hakukerralla kaikkia tietueita läpi. Ja koska tietokannan taulu on isompi, se tuskin enää mahtuu välimuistiin. Niinpä on kiintoisaa nähdä, poikkeavatko tämän kohdan tulokset kohdan 4.4 tuloksista.

Tämän kohdan tutkimukset on tarkoitus suorittaa käyttäen samoja kuvien 24 ja 26 esittämiä testifunktioita kuin kohdassa 4.4. Arvovälien pituudet oli myös tarkoitus pitää samana. Pienellä alustavalla testillä kuitenkin huomattiin, että 10 000:n pituisella arvovälillä tietueita löytyy niin paljon, etteivät ne mahdu testiympäristössä käytettyyn kekomuistiin, jota oli varattu 530 MB. Muistin määrää ei enää ryhdytty suurentamaan, koska sen jälkeen tämän kohdan tulokset eivät enää olisi vertailukelpoisia muiden kohtien kanssa.

3 Tutkimuksessa ei ole tarkasteltu yksityiskohtaisemmin Hibernaten välimuistin käyttöä (JavaBeat, 2008).

4 Kohdan 4.11 tutkimus on suoritettu todellisuudessa ennen tämän kohdan tutkimusta.

Niinpä 10 000:n pituinen arvoväli korvattiin 5 000:n pituisella. Alustavalla testillä myös havaittiin 200 hyväksi testikierrosten määräksi kaikilla arvoväleillä.

Testit aloitettiin käyttämällä kohdassa 4.11 syntynyttä tietokantaa jota ei ollut indeksoitu eikä järjestetty. Tämän testin tulokset on esitetty taulukossa 8. Tuloksista huomataan, että 10:n, 100:n ja 1 000:n pituisilla arvoväleillä hakuajat ovat hyvin lähellä toisiaan. 5 000:n pituisella arvovälillä tulos on jo yli puolet hitaampi verrattuna lyhyempiin arvoväleihin.

Taulukko 8: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä indeksoimatonta ja järjestämätöntä kohdassa 4.11 syntynyttä isoa tietokantaa.

Tutkittu asia	Arvovälikyselyiden suorittaminen indeksoimattomaan ja järjestämättömään tietokantaan, joka sisältää 2 047 322 L_Laake-tietuetta.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	200	2 h 17 min 44 s	41,3 s
100	200	2 h 9 min 55 s	39,0 s
1 000	200	2 h 40 min 30 s	48,2 s
5 000	200	5 h 58 min 25 s	107,5 s

Seuraavaksi tietokanta indeksoitiin B-puu-indeksillä ja testit ajettiin uudelleen. Näiden testien tulokset on esitetty taulukossa 9. Tällä kertaa lyhyimmällä arvovälillä indeksi nopeutti kyselyitä 26,5-kertaisesti. Kohdassa 4.4 pienemmällä tietokannalla kyselyt nopeutuivat 9,5-kertaisesti. 100:n mittaisella arvovälillä haut nopeutuivat 3,6-kertaisesti, kohdassa 4.4 tämä nopeutus oli 3,0-kertainen. 1 000:n pituisella arvovälillä tulokset paranivat 1,5-kertaisesti ja 5 000:n pituisella 1,7-kertaisesti. Viimeksi mainitut tulokset ovat hieman parempia verrattuna kohdassa 4.4 havaittuihin tuloksiin. Kokonaisuutena näyttää siis siltä, että B-puusta on sitä enemmän hyötyä mitä suurempi tietokanta on ja mitä lyhyemmät arvovälit ovat toisin sanoen mitä vähemmän arvovälikysely tuottaa tuloksia. Käytettäessä tämän kokoista tietokantaa tietokannanhallintajärjestelmä siis käyttää B-puu-indeksiä

apuna kaikissa tutkituissa tilanteissa. Se miksi pidemmillä arvoväleillä haut eivät nopeutuneet niin paljon, selittyy levyhakujen raskaudella. Koska tietueita on paljon ja ne eivät ole järjestyksessä levyllä, joudutaan levyhakuja suorittamaan paljon.

Taulukko 9: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä järjestämätöntä kohdassa 4.11 syntynyttä isoa tietokantaa, joka on indeksoitu B-puu-indeksillä.

Tutkittu asia	Arvovälikyselyiden suorittaminen B-puu-indeksiä käyttäen järjestämättömään tietokantaan, joka sisältää 2 047 322 L_Laake-tietuetta.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	200	5 min 12 s	1,6 s
100	200	36 min 15 s	10,9 s
1 000	200	1 h 45 min 29 s	31,6 s
5 000	200	3 h 27 min 29 s	62,2 s

Tämän jälkeen kohdassa 4.11 syntyneen tietokannan L_Lake-taulu järjestettiin pakkausnumeron perusteella kuvassa 25 esitetyillä SQL-lauseilla. Nyt tietokannassa ei käytetty indeksointia ja edellä kuvatut testit ajettiin uudelleen. Näin suoritetun testin tulokset on esitetty taulukossa 10. Kohdassa 4.4 Hilkka-ohjelman oletustietokannalla suoritetuissa testeissä taulun järjestämisen ei todettu nopeuttavan kyselyitä. Sitä vastoin tämän kohdan testeissä kyselyt nopeutuivat hieman alle 2-kertaisesti riippuen arvovälin pituudesta. Paras nopeutus havaittiin 1 000:n pituisella arvovälillä, jolloin haut nopeutuivat indeksoimattomaan ja järjestämättömään tietokantaan verrattuna 2,2-kertaisesti. Saman pituisella arovälillä B-puu-indeksi nopeutti kyselyitä vain 1,5-kertaisesti. Muilla testatuilla arvoväleillä B-puu oli nopeampi verrattuna järjestämiseen. Verrattuna järjestämättömään tietokantaan tietokannanhallintajärjestämä joutuu edelleenkin käymään kaikki taulun tietueet läpi, mutta koska tietueet ovat levyllä järjestettynä, selaamineen onnistuu nopeammin ja haut nopeutuvat.

Taulukko 10: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä indeksoimatonta kohdassa 4.11 syntynyttä isoa tietokantaa, jonka L_Laake-taulu on järjestetty.

Tutkittu asia	Arvovälikyselyiden suorittaminen indeksoimattomaan tietokantaan, joka sisältää 2 047 322 L_Laake-tietuetta ja jonka L_Laake-taulu on järjestetty pakkausnumeron perusteella.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	200	1 h 18 min 11 s	23,5 s
100	200	1 h 9 min 5 s	20,7 s
1 000	200	1 h 13 min 10 s	22,0 s
5 000	200	4 h 40 min	84,0 s

Lopuksi kaksi edellä suoritettua testitilannetta yhdistettiin järjestämällä tietokannan taulu sekä indeksoimalla se pakkausnumeron perusteella käyttäen B-puu-indeksiä. Testit ajettiin uudelleen ja näin saadut tulokset on esitetty taulukossa 11. Lyhyellä 10:n pituisella arvovälillä järjestys ja B-puu yhdessä saivat aikaan todella suuren nopeutuksen: haut nopeutuivat 688,7-kertaisesti verrattuna järjestämättömään ja indeksoimattomaan tietokantaan. Ilman indeksointia ja järjestämistä yksi kysely vei aikaa keskimäärin 41,3 s. Tässä kohdassa järjestettynä ja indeksoituna aikaa kului vain 0,06 s. Seuraavaksi pidemmällä, 100:n pituisella arvovälillä, B-puun ja järjestämisen yhteisvaikutus oli myös erittäin hyvä: kyselyt nopeutuivat 89,6-kertaisesti. 1 000:n pituisella arvovälillä haut nopeutuivat 11,3- ja 5 000:n pituisella arvovälillä 5,1-kertaisesti. Syynä näinkin hyviin tuloksiin on se, että B-puu-indeksin avulla tietokannanhallintajärjestelmä pystyy löytämään nopeasti haluttujen tietueiden levyosoitteet, ja koska tietueet ovat järjestettynä levyllä, voidaan ne lukea sieltä nopeasti yhdellä kertaa.

Taulukko 11: Arvovälikyselyiden nopeuden riippuminen arvovälin pituudesta käytettäessä kohdassa 4.11 syntynyttä isoa tietokantaa, jonka L_Laake-taulu on järjestetty ja indeksoitu B-puu-indeksillä.

Tutkittu asia	Arvovälikyselyiden suorittaminen tietokantaan, joka sisältää 2 047 322 L_Laake-tietuetta ja jonka L_Laake-taulu on järjestetty ja indeksoitu B-puu-indeksillä pakkausnumeron perusteella.		
Arvovälin pituus	***Testikierrosten määrä***	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
10	200	12 s	0,06 s
100	200	1 min 27 s	0,44 s
1 000	200	14 min 9 s	4,2 s
5 000	200	1 h 10 min 50 s	21,3 s

Tässä kohdassa tutkittiin arvovälin pituuden vaikutusta arvovälikyselyiden nopeuteen käyttäen isoa kohdassa 4.11 syntynyttä tietokantaa, joka sisältää 2 047 322 lääketietuetta. Testeissä huomattiin B-puun nopeuttavan kaikkia tutkittuja tilanteita toisin kuin kohdassa 4.4. Parhaiten B-puun huomattiin nopeuttavan arvovälikyselyitä, joiden tulosjoukko ei ole kovin suuri. Niinpä tämän kokoisessa tietokannassa kannattaa käyttää B-puu-indeksiä, jos tietokantaan aiotaan suorittaa arvovälikyselyitä numeeriseen sarakkeeseen.

Tämän kokoisella tietokannalla huomattiin myös tietueiden järjestämisestä olevan hyötyä toisin kuin kohdassa 4.4. Järjestämisen tuloksena arvovälikyselyt nopeutuivat arvovälin pituudesta riippuen noin 1,3 - 2,2-kertaisesti. Todellinen yllätys kuitenkin koettiin, kun tietokanta yhtä aikaa järjestettiin ja indeksoitiin B-puu-indeksillä. Tällöin lyhyet 10:n pituiset arvovälikyselyt nopeutuivat jopa 688,7-kertaisesti. Kyselyt nopeutuivat hyvin myös pidemmillä arvovälikyselyillä. Niinpä voidaan sanoa, että tavoiteltaessa mahdollisimman tehokkaita arvovälikyselyitä B-puu-indeksin käyttäminen ei pelkästään riitä, vaan tietokannan taulu on myös järjestettävä haettavan avaimen perusteella, jos suinkin mahdollista.

4.6 Pakkausnumeroiden määrän hakeminen satunnaiselta väliltä

Koska kohdissa 4.2 ja 4.3 tehdyt tutkimukset indeksoinnin vaikutuksesta pitkiin arvoväli-kyselyihin eivät tuottaneet odotettua tulosta, haluttiin tutkia samaa asiaa hieman toisella tavalla. Tarkoituksena on antaa tietokannanhallintajärjestelmän laskea hakuehtoja vastaavien tietueiden määrät Hilkka-ohjelman oletustietokannasta, ja vain tämä määrä palautetaan testaavaan ohjelmaan jokaisella testikierroksella. Tämä testaus ei kuitenkaan vastaa Hilkka-ohjelman todellista käyttötilannetta.

Tässä tutkimuksessa käytetyt testifunktiot on esitetty kuvissa 27 ja 28. Kuvan 27 *testiPakkausnumeroValiCount*-funktio toimii samalla tavalla kuin edellisessä luvussa tarkasteltu kuvan 23 *testPakkausnumeroVali*-funktio, mutta se kutsuu kuvan 28 esittämää *findByPakkausnumeroValiCount*-funktiota. Viimeksi mainittu funktio hakee tietokannan sisältämien L_Laake-tietueiden määrän seuraavalla HQL-lauseella:

```
SELECT count(laake)
  FROM L_Laake as laake
  WHERE laake.pakkausnro >= alku AND laake.pakkausnro <= loppu
```

```
public void testPakkausnumeroValiCount(int maara) {
        Random generator = new Random();

        for (int i=0; i<maara; i++) {
                Long alku = new Long(generator.nextInt(100000));
                int jaljella = 100000-alku.intValue();
                Long loppu = new Long(generator.nextInt(jaljella)) + alku;

                List<Integer> lista = findByPakkausnumeroValiCount(alku, loppu);
        }
}
```

Kuva 27. Funktio testPakkausnumeroValiCount. Tällä funktiolla testataan sellaisten L_Laake-olioiden määrän laskemista tietokannasta, joiden pakkausnumero on satunnaisella välillä.

```
@SuppressWarnings("unchecked")
public List<Integer> findByPakkausnumeroValiCount(final Long alku, final Long loppu) {
        HibernateTemplate hibernateTemplate = getHibernateTemplate();

        List <Integer> list = hibernateTemplate.executeFind(new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {

                        String sQuery = "SELECT count(laake) FROM L_Laake as laake " +
                                "WHERE laake.pakkausnro >= ? AND  laake.pakkausnro <= ?";
                        Query query = session.createQuery(sQuery);
                        query.setLong(0, alku);
                        query.setLong(1, loppu);
                        return query.list();
                }
        });
        return list;
}
```

Kuva 28. findByPakkausnumeroValiCount-funktio, jolla voidaan hakea tietokannasta kaikkien sellaisten L_Laake-olioiden määrä, joiden pakkausnumero on suurempi tai yhtä suuri kuin parametri alku ja pienempi tai yhtä suuri kuin parametri loppu.

Pienellä alustavalla testillä selvitettiin, että tietokannanhallintajärjestelmä näyttäisi pystyvän tekemään 12 tunnin aikana noin 2 000 000 hakua ilman indeksointia. Niinpä tällä kertaa testikierrosten määräksi asetettiin edellä mainittu.

Testaus aloitettiin ajamalla testiohjelma käyttäen tietokantaa, jossa ei ollut indeksointia L_Laake-taulun pakkausnro-sarakkeelle. Tämän testauksen kokonaiskesto oli 12 h 4 min 37 s.

Tämän jälkeen tietokantaan luotiin pakkausnro-sarakkeelle B-puu-indeksi. Testi ajettiin uudelleen ja testin ajaminen vei aikaa 5 h 55 min 58 s. Tällä kertaa B-puu-indeksiä käytettäessä hakujen nopeus kasvoi kaksinkertaiseksi eli kesto puolittui.

Lopuksi testiohjelma ajettiin käyttäen tietokantaa, jonka lääketaulu oli indeksoitu pakkausnumeron mukaan hajautusindeksillä. Testiajon kokonaiskesto oli 11 h 54 min 20 s. Tämä tulos on 1,01 kertaa nopeampi kuin ilman indeksointia saatu tulos. Tulos oli aivan odotettu, sillä PostgreSQL-manuaalin mukaan hajautusindeksin ei pitänytkään nopeuttaa arvovälikyselyitä.

Tämän osatutkimuksen tulokset on esitetty taulukossa 12. Tulokset olivat kohtuullisen odotettuja. Tutkimus osoitti, että B-puu nopeuttaa arvovälikyselyitä noin puolella. Lisäksi huomattiin, ettei hajautusindeksi nopeuta arvovälikyselyitä. Hieman yllättävää oli kuitenkin se, että B-puu kasvattaa yksittäisten kyselyiden nopeutta noin 16-kertaisesti (kohta 4.1), mutta arvovälikyselyillä nopeuden kasvu oli vain kaksinkertainen. B-puun rakenteen perusteella olisi voinut olettaa, että nopeutuminen olisi voinut olla arvovälikyselyillä jopa merkittävämpi kuin yksittäisillä kyselyillä, sillä jokaisessa yksittäisessä kyselyssä B-puussa joudutaan siirtymään juuresta lehtiin samalla suorittaen puun korkeuden verran vertailuja. B-puun rakenteen huomioiden pitäisi olla mahdollista, että arvovälikyselyissä voitaisiin seurata valmiiksi rakennettua osoitinketjua tekemättä mitään muita vertailuja. Jokaisessa lehdessä tulisi korkeintaan tarkistaa, onko lehden arvo suurempi kuin alueen päätepiste. B-puu-indeksin tapauksessa yksi kysely vei aikaa 10,7 ms, ja koska kiintolevyjen hakuajat ovat noin 10 ms, vaikuttaa siltä, että indeksi oli testien suorituksen hetkellä tietokoneen keskusmuistissa.

Taulukko 12: L_Laake-olioiden määrän hakeminen tietokannasta satunnaisen pakkausnumeroväli n perusteella.

Tutkittu asia	L_Laake-tietueiden määrän hakeminen tietokannasta 2 000 000 kertaa. Hakuehtona satunnainen pakkausnumeroväli väliltä [0, 100 000).	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	12 h 4 min 37 s	21,7 ms
B-puu	5 h 55 m 58 s	10,7 ms
Hajautusindeksi	11 h 54 m 20 s	21,4 ms

Tämän kohdan tutkimuksen tulos antaa kuitenkin selvyyden sille, että jos tietokannan tauluun suoritetaan koostearvon palauttavia arvovälikyselyitä numeeriseen sarakkeeseen, on taulu syytä indeksoida B-puu-indeksillä. Indeksin tuoma etu ei kuitenkaan ole niin suuri kuin yksittäisillä kyselyillä. Hajautusindeksiä numeeriseen taulun sarakkeeseen ei arvovälikyselyitä varten kannata luoda.

4.7 Lääkenimen hakeminen like merkkijono% -menetelmällä

Merkkijonojen indeksoinnin tutkimiseen soveltuu erittäin hyvin Hilkka-ohjelman oletustietokannan L_Laake-taulun laakenimi-sarake. Juuri tätä saraketta käytetään Hilkka-ohjelmassa haettaessa lääkkeitä tietokannasta lääkenimen perusteella. Tässä kohdassa indeksoinnin tehokkuutta on tarkoitus testata tilanteessa, jossa like-kyselyissä käytetään jokerimerkkiä % merkkijonon lopussa. PostgreSQL-manuaalin perusteella tällaisessa tilanteessa B-puun pitäisi nopeuttaa hakuja. Manuaali ei kuitenkaan lupaa hajautusindeksin toimivan tässä tilanteessa.

Tämä tutkimus on tarkoitus tehdä käyttämällä kuvissa 29, 30 ja 31 esitettyjä testifunktioita. Kuvan 29 *testLaakenimiLikeP*-funktiossa haettavat merkkijonot pyritään generoimaan mahdollisimman satunnaisesti, kutenkin niin, että niitä vastaavia lääkenimiä löytyy tietokannasta. Niinpä ensiksi kaikki tietokannan sisältämät lääkenimet haetaan String-tyyppi-

siä olioita sisältävään listaan kuvan 30 esittämällä *findLaakenimet*-funktiolla. Tämän jälkeen testiä ajettaessa listasta otetaan aina yksi satunnainen lääkenimi, josta otetaan satunnainen määrä merkkejä jonon alusta. Jonon loppu korvataan jokerimerkillä %. Näin saatu hakusana annetaan syötteeksi kuvan 31 esittämälle *findByLaakenimiLike*-funktiolle, joka hakee lääkenimeä vastaavat L_Laake-oliot tietokannasta seuraavalla HQL-lauseella:

```
FROM L_Laake
   WHERE laakenimi like kriteeri
```

```
public void testLaakenimiLikeP(int maara) {
        Random generator = new Random();

        List<String> laakenimiList = findLaakenimet();
        int listSize = laakenimiList.size();

        for (int i=0; i<maara; i++) {
                String laakenimi = laakenimiList.get(generator.nextInt(listSize));
                int nimiLength = laakenimi.length();
                String subNimi = laakenimi.substring(0,generator.nextInt(nimiLength)+1);
                List<L_Laake> lista = findByLaakenimiLike(subNimi + "%");
        }
}
```

Kuva 29. testLaakenimiLikeP-funktio. Tällä funktiolla testataan lääketietueiden hakua like merkkijono% -menetelmällä.

```
@SuppressWarnings("unchecked")
public List<String> findLaakenimet() {
	HibernateTemplate hibernateTemplate = getHibernateTemplate();

	List <String> list = hibernateTemplate.executeFind(new HibernateCallback() {
		public Object doInHibernate(Session session) throws HibernateException {

			String sQuery = "SELECT DISTINCT laakenimi FROM L_Laake";
			Query query = session.createQuery(sQuery);
			return query.list();
		}
	});
	return list;
}
```

Kuva 30. findLaakenimet-funktio. Tämän funktion avulla voidaan hakea kaikki tietokannan sisältämät lääkenimet.

```
@SuppressWarnings("unchecked")
public List<L_Laake> findByLaakenimiLikeP(final String laakenimi) {
	HibernateTemplate hibernateTemplate = getHibernateTemplate();

	List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
		public Object doInHibernate(Session session) throws HibernateException {

			String sQuery = "FROM L_Laake WHERE laakenimi like ?";

			Query query = session.createQuery(sQuery);
			query.setString(0, laakenimi);
			return query.list();
		}
	});
	return list;
}
```

Kuva 31. findByLaakenimiLikeP-funktio, jolla lääketietueita voidaan hakea tietokannasta lääkenimen perusteella like-menetelmällä.

Tutkimus aloitettiin ajamalla testi käyttäen tietokantaa, jossa ei ole indeksointia L_Laake-taulun laakenimi-sarakkeelle. Ensiksi alustavalla testillä selvitettiin, että 12 tunnissa testi-

ohjelma pystyy hakemaan lääkkeitä noin 500 000 kertaa. Niinpä testikierrosten määräksi valittiin edellä mainittu. Varsinaisessa testissä lääkkeiden hakeminen ilman indeksien käyttöä kesti 14 h 13 min 44 s.

Tämän jälkeen tietokantaan luotiin B-puu-indeksi L_Laake-taulun laakenimi-sarakkeelle. Indeksin luonnin jälkeen testiohjelma ajettiin uudelleen. Tällä kertaa testiajon suorittaminen vei aikaa 10 h 27 min 47 s. Tämä tulos oli odotetun mukainen, sillä PostgreSQL-manuaalin mukaan B-puun tulee toimia tällaisissa tilanteissa. Hakujen nopeus kasvoi 1,4-kertaiseksi verrattuna indeksoimattomaan tietokantaan, mitä voidaan pitää tyydyttävänä tuloksena.

Lopuksi sama testiohjelma ajettiin käyttäen tietokantaa, jossa laakenimi-sarake oli indeksoitu hajautusindeksillä. Testin suorittaminen vei aikaa 14 h 10 min 11 s, joka on lähes sama aika kuin indeksoimattomalla tietokannalla saavutettu tulos. Tästä voidaan päätellä, että PostgreSQL-tietokannan hajautusindeksi ei tue merkkijonojen hakua like merkkijono% -menetelmällä.

Tässä kohdassa suoritettujen testien tulokset on esitetty taulukossa 13. Tulokset olivat juuri sen kaltaiset kuin oli odotettua. PostgreSQL-tietokannan B-puu nopeuttaa tämän kaltaisten kyselyiden suoritusta noin kolmasosalla. Hajautusindeksillä ei huomattu olevan mitään vaikutusta tietokantaan tehtyjen hakujen nopeuteen. Siispä tällaisia hakuja tehtäessä tietokannan taulun sarake kannattaa indeksoida käyttäen B-puu-indeksiä.

Taulukko 13: Lääketietueiden hakeminen tietokannasta like merkkijono% -menetelmällä.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 500 000 kertaa like merkkijono% -menetelmällä satunnaisesti generoidun lääkenimen perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	14 h 13 min 44 s	102,4 ms
B-puu	10 h 27 min 47 s	75,3 ms
Hajautusindeksi	14 h 10 min 11 s	102,0 ms

4.8 Lääkenimen hakeminen like %merkkijono% -menetelmällä

Toisena merkkijonojen indeksointiin liittyvänä tilanteena haluttiin tutkia tilannetta, jossa tietueita haetaan like-menetelmää käyttäen merkkijonolla, jonka alussa sekä lopussa on jokerimerkki %. PostgreSQL-manuaali lupaa B-puiden toimivan vain sellaisissa tilanteissa, joissa merkkijonon alussa ei ole jokerimerkkiä. Hajautusindeksin ei ole luvattu toimivan ollenkaan merkkijonojen hauissa, joissa käytetään like-menetelmää. Siispä oletetaan, ettei indeksoinnilla ole mitään vaikutusta tällä hakumenetelmällä tehtyihin testeihin. Tämän menetelmän tutkiminen on kuitenkin kiintoisaa, sillä tällä hetkellä Hilkka-ohjelmassa lääketietueita haetaan käyttäen juuri tätä menetelmää.

Tämä tutkimus on tarkoitus suorittaa kuvien 30, 31 ja 32 esittämillä testifunktiolla. Kuvan 32 *testLaakenimiLikePP*-funktio toimii pääpiirteittäin samalla tavalla kuin kuvan 29 esittämä funktio kohdan 4.7 tutkimuksessa. Erona kuitenkin on se, että satunnaisesta lääkenimestä otetaan keskeltä satunnainen merkkijono. Näin saadun hakusanan alkuun ja loppuun lisätään jokerimerkki %. Tämä hakusana annetaan kuvan 31 esittämälle *findLaakenimiLike*-funktiolle, joka hakee tietokannasta hakusanaa vastaavat L_Laake-oliot käyttäen like-menetelmää.

```
public void testLaakenimiLikePP(int maara) {
	Random generator = new Random();

	List<String> laakenimiList = findLaakenimet();
	int listSize = laakenimiList.size();

	for (int i=0; i<maara; i++) {
		String laakenimi = laakenimiList.get(generator.nextInt(listSize));
		int nimiLength = laakenimi.length();
		int nimiAlku = generator.nextInt(nimiLength-1);
		int nimiLoppu = nimiAlku + generator.nextInt(nimiLength-nimiAlku -1) +1;
		String subNimi = laakenimi.substring(nimiAlku, nimiLoppu);
		List<L_Laake> lista = findByLaakenimiLike("%" + subNimi + "%");
	}
}
```

Kuva 32. testLaakenimiLikePP-funktio. Tällä funktiolla testataan lääkenimen hakua like %merkkijono% -menetelmällä.

Ilman indeksointia alustavalla testillä huomattiin, että hakusanan sekä alussa että lopussa oleva jokerimerkki raskauttaa hakua huomattavasti. Huomattiin, että 12 tunnissa testiohjelma pystyy suorittamaan noin 30 000 hakua Hilkka-ohjelman oletustietokantaan. Niinpä aluksi testiohjelma asetettiin suorittamaan hakua 30 000 kertaa indeksoimattomaan tietokantaan. Tämän testiajon kokonaiskesto oli 14 h 51 min 46 s.

Tämän jälkeen laakenimi-sarakkeelle luotiin B-puu-indeksi ja testiohjelma ajettiin uudelleen. Tämän testiajon kokonaisaika oli 15 h 9 min 36 s. Tulos on 1,02 kertaa hitaampi kuin ilman indeksointia suoritetun testauksen tulos, mutta suuruusluokka on kuitenkin sama.

Lopuksi testaus suoritettiin tietokantaan, jossa lääkenimi-sarake on indeksoitu hajautusindeksillä. Tämä testiajo vei aikaa 15 h 2 min 49 s. Tämäkin tulos on samaa suuruusluokkaa edellisten testien kanssa. Tarkalleen ottaen tulos on 1,01 kertaa hitaampi verrattuna ilman indeksointia saavutettuun tulokseen.

Tämän testauksen tulokset on esitetty taulukossa 14. Tulokset olivat odotettuja, sillä PostgreSQL-manuaali ei luvannut indeksoinnin auttavan tällaisissa tilanteissa, joissa hakusana alkaa jokerimerkillä. Tutkimuksessa paras tulos saatiin tietokannalla, jossa ei ollut käytetty mitään indeksointia. Erot olivat kuitenkin pieniä, ja ne voivat selittyä satunnaisten hakusanojen erilaisuudella. Tämä tutkimus osoitti myös tällaisen kyselyiden olevan noin 17 kertaa raskaampia kuin merkkijono%-menetelmällä tehtävät kyselyt.

Taulukko 14: Lääketietueiden hakeminen tietokannasta like %merkkijono% -menetelmällä.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 30 000 kertaa like %merkkijono% -menetelmällä satunnaisesti generoidun lääkenimen perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	14 h 51 min 46 s	1783,5 ms
B-puu	15 h 9 min 36 s	1819,2 ms
Hajautusindeksi	15 h 2 min 49 s	1805,6 ms

4.9 Lääkenimen hakeminen like-menetelmällä ilman jokerimerkkejä

Yhdeksi tutkimuksen kohteeksi otettiin myös merkkijonojen hakeminen like-menetelmällä ilman jokerimerkkejä. Tätä menetelmää ei kuitenkaan käytetä Hilkka-ohjelmassa, mutta on kuitenkin kiinnostavaa nähdä, toimiiko like-menetelmä nopeammin kuin kohdassa 4.10 tutkittava yhtäsuuruus-menetelmä ja kummallako hakumenetelmällä indeksoinnista on enemmän hyötyä. PostgreSQL-manuaali lupaa vain B-puu-indeksin toimivan like-menetelmää käytettäessä.

Tutkimus suoritettiin kuvan 33 esittämällä *testLaakenimiLike*-testifunktiolla, joka hakee

tietokannasta lääkkeitä satunnaisella lääkenimellä. Tietueiden haussa käytetään myös kuvan 30 esittämää *findLaakenimet*-funktiota sekä kuvan 31 esittämää *findByLaakenimiLike*-funktiota. Testikierrosten määräksi valittiin 2 000 000 kierrosta, mikä on sama määrä mitä käytettiin luvussa 4.1 tutkittaessa yksittäisten lääketietueiden hakua pakkausnumeron perusteella. Näin saadaan selville myös se, onko merkkijonon perusteella hakeminen työläämpää kuin numeerisen tiedon perusteella hakeminen.

```
public void testLaakenimiLike(int maara) {
	Random generator = new Random();

	List<String> laakenimiList = findLaakenimet();
	int listSize = laakenimiList.size();

	for (int i=0; i<maara; i++) {
		String laakenimi = laakenimiList.get(generator.nextInt(listSize));

		List<L_Laake> lista = findByLaakenimiLike(laakenimi);
	}
}
```

Kuva 33. testLaakenimiLike-funktio. Tällä funktiolla testataan lääkenimen hakua like-menetelmällä ilman jokerimerkkejä.

Ensimmäiseksi testi suoritettiin Hilkka-ohjelman oletustietokannalla, jossa lääkenimeä ei ollut indeksoitu millään indeksillä. Tämän testin suorittaminen vei aikaa 17 h 18 min 5 s. Tämä on huomattavasti pidempi aika kuin kohdassa 4.1 2 000 000 lääketietueen hakemiseen satunnaisen pakkausnumeron perusteella kulunut aika 11 h 41 s. Tästä käy ilmi, että merkkijonon perusteella hakeminen on ainakin ilman indeksointia vaativampaa. Tässä tilanteessa eroa tasoittaa hieman se, että kohdassa 4.1 yhdellä pakkausnumerolla löytyi aina korkeintaan yksi lääketietue. Tässä kohdassa suoritetussa testissä lääketietueita voi löytyä samalla lääkenimellä useampiakin. Yleensä samalla lääkenimellä löytyy kuitenkin korkeintaan vain muutamia lääketietueita, joten ero ei tässä tilanteessa ole kuitenkaan kovin suuri.

Seuraavaksi testi suoritettiin tietokantaan, joka oli indeksoitu käyttämällä B-puu-indeksiä. Tämän testin suorittaminen kesti 2 h 2 min 58 s. Indeksointi siis nopeutti hakua 8,4-kertaisesti. Nopeutus ei ollut aivan niin suuri kuin kohdassa 4.1 B-puulla saavutettu 16-kertainen nopeutus. Tätä voidaan kuitenkin pitää hyvänä tuloksena.

Lopuksi testi ajettiin tietokannalle, jossa käytettiin hajautusindeksiä. Tämän testiajon kokonaisaika oli 17 h 12 min 9 s. Aika on 1,003 kertaa eli vajaat 6 minuuttia parempi kuin ilman indeksointia suoritetun testin tulos. Suuruusluokka tulosten välillä on kuitenkin sama. Tästä voidaan päätellä, ettei hajautusindeksi nopeuta tietokantahakuja käytettäessä like-menetelmää.

Tässä kohdassa tehdyn tutkimuksen tulokset on esitetty taulukossa 15. Tulokset olivat hyvin pitkälle odotetun kaltaisia. B-puu nopeutti tietokannan toimintaa merkittävästi, sillä nopeus kasvoi jopa 8,4-kertaiseksi. Hajautusindeksin ei todettu vaikuttavan hakujen toimintaan tässä tapauksessa. Lopputuloksena voidaan sanoa, että jos tietokantaan tehdään hakuja like-menetelmällä ilman jokerimerkkejä, tietokannan taulu kannattaa indeksoida B-puu-indeksillä.

Taulukko 15: L_Laake-olioiden hakeminen tietokannasta like-menetelmällä ilman jokerimerkkejä.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 2 000 000 kertaa like-menetelmällä ilman jokerimerkkejä satunnaisesti generoidun lääkenimen perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	17 h 18 min 5 s	31,1 ms
B-puu	2 h 2 min 58 s	3,7 ms
Hajautusindeksi	17 h 12 min 9 s	31,0 ms

4.10 Lääkenimen hakeminen yhtäsuuruus-menetelmällä

Vertailun vuoksi merkkijonoihin liittyvää hakua tutkittiin vielä lopuksi Hilkka-ohjelman oletustietokannassa käyttäen yhtäsuuruus-menetelmää. Tämä menetelmä vastaa tuloksiltaan kohdassa 4.9 tutkittua like-menetelmää, sillä kummatkin hakumenetelmät tuottavat saman tuloksen. Tässä kohdassa nähdään, kumpiko näistä menetelmistä on tehokkaampi ja kumpaanko indeksointi vaikuttaa enemmän. PostgreSQL-manuaalin mukaan sekä B-puun että hajautusindeksin pitäisi nopeuttaa tällaisia yksinkertaisia yhtäsuuruusvertailuita, joten tässä tilanteessa on odotettua, että yhtäsuuruus-menetelmä on ainakin hajautusindeksiä käytettäessä paljon nopeampi verrattuna like-menetelmään.

Tutkimus suoritettiin kuvien 30, 34 ja 35 funktiolla. Kuvan 34 *testLaakenimiYhtasuuruus*-funktio hakee tietokannan sisältämät lääkenimet kuvan 30 esittämällä *findLaakenimet*-funktiolla. Näistä lääkenimistä valitaan jokaisella testikierroksella satunnaisesti yksi ja se ohjataan kuvan 35 *findByLaakenimiYhtasuuruus*-funktiolle, joka hakee lääkenimeä vastaavat tietueet tietokannasta käyttäen seuraavaa HQL-lausetta:

```
FROM L_Laake
  WHERE laakenimi = kriteeri
```

```
public void testLaakenimiYhtasuuruus(int maara) {
        Random generator = new Random();

        List<String> laakenimiList = findLaakenimet();
        int listSize = laakenimiList.size();

        for (int i=0; i<maara; i++) {
                String laakenimi = laakenimiList.get(generator.nextInt(listSize));
                List<L_Laake> lista = findByLaakenimiYhtasuuruus(laakenimi);
        }
}
```

Kuva 34. testLaakenimiYhtasuuruus-funktio. Tällä funktiolla testataan lääketietueiden hakua lääkenimen perusteella yhtäsuuruus-menetelmällä.

```
@SuppressWarnings("unchecked")
public List<L_Laake> findByLaakenimiYhtasuuruus(final String laakenimi) {
        HibernateTemplate hibernateTemplate = getHibernateTemplate();

        List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {

                        String sQuery = "FROM L_Laake WHERE laakenimi = ?";

                        Query query = session.createQuery(sQuery);
                        query.setString(0, laakenimi);
                        return query.list();
                }
        });
        return list;
}
```

Kuva 35. findByLaakenimiYhtasuuruus-funktio. Tällä funktiolla haetaan tietokannasta L_Laake-olioita käyttäen yhtäsuuruus-menetelmää.

Tässä tutkimuksessa edellä mainittuja hakuja suoritettiin 2 000 000 kertaa, mikä on sama määrä kuin kohdassa 4.9 suoritetussa like-menetelmän tutkimuksessa. Näin näiden kohtien mittaustulokset ovat keskenään täysin vertailukelpoisia.

Ilman indeksointia suoritetun tutkimuksen kokonaisaika oli 16 h 49 min 51 s, joka on 1,03 kertaa parempi aika kuin like merkkijono -menetelmällä saatu tulos kohdassa 4.9. Aikojen suuruusluokka on kutenkin sama.

Tämän jälkeen tietokantaan luotiin B-puu-indeksi ja testi ajettiin uudelleen. Tämän testiajon kokonaisajaksi saatiin 2 h 3 min 41 s, mikä on 8,2 kertaa nopeampi verrattuna ilman indeksointia tehtyihin hakuihin. Kohdassa 4.9 tietokannasta pystyttiin hakemaan like-menetelmää käyttäen 2 000 000 lääketietuetta ajassa 2 h 2 min 58 s, mistä voidaan päätellä B-puun nopeuttavan yhtäsuuruus-menetelmällä tehtyjä kyselyitä suurin piirtein saman verran kuin like-menetelmällä tehtyjä. Niinpä tämän tutkimuksen perusteella on aivan sa-

ma, käytetäänkö B-puu-indeksin käytön yhteydessä like- vai yhtäsuuruus-menetelmää.

Lopuksi testiohjelma ajettiin käyttäen tietokantaa, jossa laakenimi-sarake oli indeksoitu hajautusindeksillä. Tämän testiajon kesto oli 2 h 3 min 54 s, joka on lähes sama aika kuin B-puuta käyttäen saatu tulos. Verrattuna kohdassa 4.9 suoritettuun testiin, jossa hajautusindeksi ei nopeuttanut like-menetelmällä tehtyjä hakuja lainkaan, yhtäsuuruus-menetelmällä tehdyillä hauilla nopeus kasvoi 8,2-kertaiseksi. Siispä jos tietokannassa halutaan käyttää hajautusindeksiä, kannattaa merkkijonojen haut suorittaa yhtäsuuruus-menetelmällä like-menetelmän sijaan.

Tämän testin tulokset on esitetty taulukossa 16. Tulokset olivat odotettuja, sillä sekä B-puu että hajautusindeksi nopeuttivat tietokantahakuja 8,2-kertaisesti. Pieni yllätys oli se, kuinka samanlaiset tulokset näiden indeksointimenetelmien välillä olivatkaan. 2 000 000:n satunnaisen lääketietueen haussa aikaeroa oli vain 13 s. Myös kohdassa 4.1 huomattiin B-puun ja hajautusindeksin välillä olevan vain hyvin pieni ero. Näyttää siis siltä, että riippumatta tietotyypistä yhtäsuuruus-menetelmällä tehdyissä hauissa B-puu ja hajautusindeksi ovat hyvin tasaveroisia. Tässä tutkimuksessa huomattiin myös se, että B-puu nopeuttaa like- ja yhtäsuuruus-menetelmillä tehtyjä merkkijonohakuja suurin piirtein yhtä paljon. Hajautusindeksin huomattiin toimivan vain yhtäsuuruus-menetelmää käytettäessä.

Taulukko 16: Lääketietueiden hakeminen tietokannasta lääkenimen perusteella yhtäsuuruus-menetelmällä.

Tutkittu asia	L_Laake-tietueiden hakeminen tietokannasta 2 000 000 kertaa yhtäsuuruus-menetelmällä satunnaisesti generoidun lääkenimen perusteella.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	16 h 49 min 51 s	30,3 ms
B-puu	2 h 3 min 41 s	3,7 ms
Hajautusindeksi	2 h 3 min 54 s	3,7 ms

4.11 Indeksoinnin vaikutus tietueiden lisäämiseen

Tässä kohdassa haluttiin tutkia, kuinka indeksointi vaikuttaa tietueiden lisäämiseen tietokantaan. Kirjallisuudessa (Garcia-Molina et al. 2002) indeksoinnin varjopuolena on pidetty tietokannan päivityksen hidastumista. Jos indeksointia ei käytetä, tietuetta lisättäessä tarvitaan vain kirjoittaa tietue tietokantaan. Käytettäessä indeksointia tarvitaan tietueen tietojen tallennuksen lisäksi päivittää indeksiin tieto uudesta lisätystä tietueesta. Tässä tutkimuksessa on tarkoitus luoda tietokannan L_Laake-tauluun kaksi indeksiä, toinen pakkausnro-sarakkeelle ja toinen laakenimi-sarakkeelle. Tällä tavoin tietokannanhallintajärjestelmä joutuu päivittämään kahta indeksiä, ja näin tässä tutkimuksessa todennäköisesti saadaan selvempi ero aikaan. PostgreSQL-manuaali (PostgreSQL, 2007) ei ota kantaa tietokannan hidastumiseen tietueiden lisäämisen yhteydessä. Manuaali ei myöskään kerro, vaikuttaako B-puu vai hajautusindeksi enemmän hidastavasti. Manuaalissa on kuitenkin sanottu B-puun toimivan jokaisessa tilanteessa paremmin kuin hajautusindeksin, joten oletus tähän testiin on se, että tietokannan tulisi toimia nopeammin käytettäessä B-puu-indeksiä.

Tässä tutkimuksessa käytetty *testLaakeLisaaminen*-funktio on esitetty kuvassa 36. Funktio käyttää apuna Hilkka-ohjelman L*_LaakeManeger*-luokkaa. Tämä luokka sisältää kaksi tässä testissä käytettyä funktiota, *loadAll* ja *save*. *loadAll*-funktion kutsuminen palauttaa listassa kaikki tietokannan sisältämät L_Laake-oliot. Funktion *save* avulla voidaan tallentaa yksi L_Laake-olio tietokantaan. Lisäksi t*estLaakeLisaaminen*-funktio käyttää apuna kuvassa 30 esitettyä *findLaakenimet*-funktiota.

testLaakeLisaaminen-funktiossa on tavoiteltu sitä, että tietokantaan lisättävät lääkkeet olisivat mahdollisimman samankaltaisia kuin Lääkelaitoksen oikeat lääkkeet. Niinpä testissä haetaan ensin kaikki Hilkka-ohjelman oletustietokannassa olevat 47 322 lääketietuetta

listLaakkeet-nimiseen listaan käyttäen *L_LaakeManager*-luokan *loadAll*-funktiota. Tämän jälkeen haetaan tietokannan sisältämät lääkenimet *listLaakenimiet*-nimiseen listaan. Seuraavaksi siirrytään silmukkaan, jossa *listLaakkeet*-listalta otetaan satunnainen tietokannassa oleva lääketietue. Tämän jälkeen luodaan uusi L_Laake-olio *uusiLaake*. Vanhasta lääkkeestä tehdään kopio *SpringFrameWork*:n tarjoaman *BeanUtils*-luokan *CopyProperties*-funktiolla. Uusi lääketietue sisältää nyt samat tiedot kuin vanha tietue. *CopyProperties* kopioi myös olion viitteet tietokantaan. Nämä viitteet voidaan tuhota asettamalla uudelle lääkkeelle id:ksi null, jonka jälkeen *uusiLaake* on jälleen täysin puhdas tietokantaan kuulumaton olio. Uudelle lääkkeelle pyritään asettamaan mahdollisimman satunnaiset ja kuitenkin oikeita tilanteita kuvaavat pakkausnumero ja lääkenimi. Niinpä uudelle lääkkeelle arvotaan uusi pakkausnumero väliltä [0, 100 000). Lääkenimen asettamisessa käytetään jo kohdassa 4.8 esiteltyä menetelmää, jossa tietokannan sisältämistä lääkenimistä otetaan satunnainen osa. Tässä tapauksessa listasta *vanhatNimetList* otetaan kaksi lääkenimeä, joista kummastakin otetaan keskeltä satunnainen merkkijono. Uudelle lääkkeelle asetetaan näiden merkkijonojen yhdiste, kuitenkin vain 64 merkin pituinen, sillä tietokannassa lääkenimen maksimipituus on 64 merkkiä. Lopuksi näin generoitu uusi *L_Laake*-olio tallennetaan tietokantaan käyttämällä L*_LaakeManager*-luokan *save*-funktiota.

```
public void testLaakeLisaaminen(int maara) {
	Random generator = new Random();

	List<L_Laake> laakeList = l_LaakeManager.loadAll();
	List<String> laakenimiList = findLaakenimet();
	int laakeListSize = laakeList.size();
	int nimiListSize = laakenimiList.size();

	for (int i=0; i<maara; i++) {
		L_Laake laake = laakeList.get(generator.nextInt(laakeListSize));
		L_Laake uusiLaake = new L_Laake();
		BeanUtils.copyProperties(laake, uusiLaake);
		uusiLaake.setId(null); // Nyt tämä todellakin on uusi lääke, ei viitettä kantaan

		String laakenimi1 = laakenimiList.get(generator.nextInt(nimiListSize));
		int nimiLength1 = laakenimi1.length();
		int alku1 = generator.nextInt(nimiLength1-1);
		int loppu1 = alku1 + generator.nextInt(nimiLength1-alku1 -1) +1;
		String subNimi1 = laakenimi1.substring(alku1, loppu1);

		String laakenimi2 = laakenimiList.get(generator.nextInt(nimiListSize));
		int nimiLength2 = laakenimi2.length();
		int alku2 = generator.nextInt(nimiLength2-1);
		int loppu2 = alku2 + generator.nextInt(nimiLength2-alku2 -1) +1;
		String subNimi2 = laakenimi2.substring(alku2,loppu2);

		String uusiLaakenimi = subNimi1 + subNimi2;
		if (uusiLaakenimi.length() > 64) {
			uusiLaakenimi = uusiLaakenimi.substring(0, 64);
		}

		uusiLaake.setLaakenimi(uusiLaakenimi);
		uusiLaake.setPakkausnro(new Long(generator.nextInt(100000)));

		l_LaakeManager.save(uusiLaake);
	}
}
```

Kuva 36. testLaakeLisaaminen-funktio. Tällä funktiolla testataan lääkkeiden lisäämistä tietokantaan.

Aluksi testaus suoritettiin tietokantaan, jonka L_Laake-taulua ei ollut indeksoitu millään indeksillä. Testikierrosten määräksi asetettiin 2 000 000. Tämän testin ajaminen vei aikaa

12 h 19 min 49 s.

Tämän jälkeen sama testaus suoritettiin tietokantaan, jonka L_Laake-taulu oli indeksoitu B-puu-indeksillä sekä pakkausnumeron että lääkenimen perusteella. Tämän testin ajaminen vei aikaa, 15 h 17 min 52 s, mikä on 1,2 kertaa pidempi aika kuin ilman indeksointia saatu tulos.

Lopuksi testaus suoritettiin tietokantaan, jonka L_Laake-taulu oli indeksoitu hajautusindeksillä sekä pakkausnumeron että lääkenimen perusteella. Testin suoritus vei aikaa 16 h 9 min 35 s. Tämä tulos on 1,3 kertaa hitaampi verrattuna ilman indeksointia saavutettuun tulokseen.

Tässä kohdassa tehdyn tutkimuksen tulokset on esitetty taulukossa 17. Tässä tutkimuksessa huomattiin, että tutkitut indeksit hidastavat tietokannan tauluun tehtäviä tietueiden lisäyksiä noin 30%. Tutkimuksessa huomattiin myös B-puun toimivan lisäysten yhteydessä hieman hajautusindeksiä nopeammin.

Taulukko 17: Indeksionnin vaikutus tietueiden lisäämiseen.

Tutkittu asia	2 000 000 L_Laake-tietueen lisääminen tietokantaan, jossa L_Laake-taulun pakkausnumero- ja laakenimi-sarakkeet on yhtä aikaa indeksoitu.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	12 h 19 min 49 s	22,2 ms
B-puu	15 h 17 min 52 s	27,5 ms
Hajautusindeksi	16 h 9 min 35 s	29,1 ms

4.12 Indeksoinnin vaikutus tietueiden poistamiseen

Tässä kohdassa on tarkoitus tutkia indeksoinnin vaikutusta tietokannan tauluun kohdistuviin tietueiden poistoihin. Kuten kohdassa 4.11 tässäkin osatutkimuksessa L_Laake-taulu on tarkoitus indeksoida sekä pakkausnumeron että lääkenimen perusteella mahdollisimman suuren eron saavuttamiseksi. Tässä tutkimuksessa tietokantaan ei tule poistojen välissä ollenkaan lisäyksiä, joten jos tietokannanhallintajärjestelmä ei ajoittain päivitä indeksin rakennetta, tulee indeksi jossain vaiheessa niin hajanaiseksi, että indeksin rakennetta on päivitettävä. Siispä todennäköisesti indeksointi tulee hidastamaan tietokannan toimintaa jonkin verran.

Tämä tutkimus on tarkoitus suorittaa käyttämällä kuvissa 37 ja 38 esitettyjä testifunktioita. Tarkoituksena on poistaa tietokannasta satunnaisia L_Laake-olioita. Niinpä kuvan 37 *testLaakePoistaminen*-funktio hakee tietokannan kaikkien lääkkeiden id:t Long-tyyppisiä olioita sisältävään *idList*-nimiseen listaan käyttäen kuvassa 38 esitettyä *findLaakeIds*-funktiota. Tämän jälkeen tullaan silmukkaan, jossa *idList*-listalta otetaan satunnainen id. Tämän id:n perusteella tietokannasta poistetaan id:tä vastaava lääketietue käyttäen L*_LaakeManager*-luokan *delete*-funktiota.

```
public void testLaakePoistaminen(int maara){
        Random generator = new Random();

        List<Long> idList = findLakkeIds();
        int idListSize = idList.size();

        for (int i=0; i<maara; i++) {
                Long id = idList.get(generator.nextInt(idListSize));
                l_LaakeManager.delete(id);
                idList.remove(id);
                idListSize--;
        }
}
```

Kuva 37. testLaakePoistaminen-funktiolla testataan lääketietueiden poistamista.

```
@SuppressWarnings("unchecked")
public List<Long> findLakkeIds() {
	HibernateTemplate hibernateTemplate = getHibernateTemplate();

	List <Long> list = hibernateTemplate.executeFind(new HibernateCallback() {
		public Object doInHibernate(Session session) throws HibernateException {

			String sQuery = "SELECT id FROM L_Laake";
			Query query = session.createQuery(sQuery);
			return query.list();
		}
	});
	return list;
}
```

Kuva 38. findLaakeIds-funktiolla voidaan tietokannasta hakea kaikkien L_Laake-olioiden id:t Long-tyyppisiä olioita sisältävään listaan.

Tätä tutkimusta ajettaessa käytetään kohdassa 4.11 syntynyttä tietokantaa, jossa on 2 000 000 lisäyksen jälkeen yhteensä 2 047 322 lääketietuetta. Tutkimusta tehtäessä huomattiin, että lääketietueen poistaminen tietokannasta on vaativampaa kuin tietueen lisääminen. Kohdassa 4.11 tietokantaan onnistuttiin lisäämään 2 000 000 L_Laake-tietuetta ajassa 12 h 19 min 49 s käyttämättä indeksointia. Pienellä alustavalla testillä selvitettiin, ettei reilussa 12 tunnissa pystytä poistamaan samaa määrää. Niinpä tämän kohdan testikierrosten lukumääräksi asetettiin 500 000. Ensimmäiseksi testiohjelma ajettiin tietokannalle, jonka L_Laake-taulua ei ollut indeksoitu millään indeksillä. Tämän testin kokonaisajaksi saatiin 8 h 51 min 23 s.

Seuraavaksi testi ajettiin käyttäen tietokantaa, jonka L_Laake-taulun pakkausnro- ja laakenimi-sarake olivat indeksoitu käyttämällä B-puu-indeksiä. Tämän testiajon tulokseksi saatiin 8 h 44 min 57 s. Tulos on samaa suuruusluokkaa indeksoimattoman tietokannan testin kanssa, jopa noin 5 minuuttia ja 1,01 kertaa nopeampi. Tästä voidaan päätellä, että B-puu ei hidasta tietueiden poistamista PostgreSQL-tietokannasta.

Lopuksi testi suoritettiin käyttäen hajautusindeksiä L_Laake-taulun pakkausnero- ja laakenimi-sarakkeissa. Tämän testin ajaksi saatiin 8 h 58 min, mikä on 1,01 kertaa hitaampi tulos verrattuna indeksoimattomaan tietokantaan. Saavutettu tulos on kuitenkin hyvin lähellä muiden testien tuloksia, joten hajautusindeksilläkään ei voida sanoa olevan hidastavaa vaikutusta tietueiden poistoon.

Tämän testin tulokset on esitetty taulukossa 18. Vastoin odotuksia tässä tutkimuksessa huomattiin, ettei tutkituilla indekseillä ole hidastavaa vaikutusta tietueiden poistoon tietokannasta. B-puu-indeksi tuotti tällä kertaa jopa nopeimman tuloksen. Myöskään ei ole mahdollista, että tietokanta olisi käyttänyt indeksejä hyödykseen poistojen yhteydessä. Testiohjelmassa ei käytetty lääkenimeä eikä pakkausnumeroa, vaan tietueet poistettiin tietokannasta pelkästään niiden *id*-numeroiden perusteella. Tässä testissä myös huomattiin tietueiden poistamisen olevan työläämpi operaatio kuin niiden lisääminen tietokantaan.

Taulukko 18: Indeksionnin vaikutus tietueiden poistamiseen tietokannasta.

Tutkittu asia	500 000 L_Laake-tietueen poistaminen tietokannasta, jossa L_Laake-taulun pakkausnro- ja laakenimi-sarakkeet on yhtä aikaa indeksoitu.	
Käytetty indeksi	***Kokonaisaika***	***Yhden askeleen keskimääräinen aika***
Ei indeksiä	8 h 51 min 23 s	63,8 ms
B-puu	8 h 44 min 57 s	63,0 ms
Hajautusindeksi	8 h 58 min	64,6 ms

4.13 Indeksien luominen ja poistaminen Hilkka-ohjelman oletustietokannassa

Tässä kohdassa tutkitaan B-puun ja hajautusindeksin luomiseen ja poistamiseen kuluvaa aikaa käyttäen Hilkka-ohjelman oletustietokantaa, joka sisältää 47 322 lääketietuetta. Tar-

koitus on vertailla, kummanko indeksin luominen ja poistaminen on nopeampaa samantyyppiselle sarakkeelle. Lisäksi nähdään, onko numeerisen ja merkkijonotyppisen sarakkeen indeksointiaikojen välillä eroja. PostgreSQL-manuaalin mukaan B-puun luontiajan pitäisi olla lyhyempi kuin hajautusindeksin.

Aiemmin tässä luvussa on indeksoinnin tehokkuutta testattu käyttäen Java-kielellä kirjoitettuja testausfunktiota, joissa tietokantaoperaatiot on suoritettu käyttäen Hibernaten HQL-kieltä. Koska HQL-kieli on nimensä mukaisesti vain kyselykieli, se ei mahdollista indeksien luomista ja poistamista (Hibernate, 2007). Tästä johtuen tässä kohdassa esitetyt testit on jouduttu tekemään PostgreSQL-tietokannan tarjoamalla PgAdmin III -ohjelmalla, jolla tietokantaan voidaan ajaa SQL-kielisiä komentoja. Edellä mainittu ohjelma kertoo myös komennon suorituksessa kuluneen ajan, joten se sopii hyvin näiden testien ajamiseen.

Tämän kohdan testeissä on tarkoitus lisätä ja poistaa indeksejä useita kertoja peräkkäin. Niinpä testit on koottu kuvien 39, 40, 41 ja 42 esittämiin SQL-funktioihin, jotka saavat syötteenään ajettavien testikierrosten määrän.

Testaus aloitettiin ennalta raskaimmaksi oletetusta merkkijonon indeksoimisesta hajautusindeksillä. Tämä testi suoritettiin kuvan 39 esittämällä *lisaaJaPoistaHajautusLaakenimi*-funktiolla. Alustavissa testeissä huomattiin, että 12 tunnissa indeksiä voidaan lisätä ja poistaa noin 30 000 kertaa. Ensimmäistä kertaa testiä ajettaessa kuitenkin huomattiin, että jokainen testausfunktion luoma indeksi vie 16,5 MB levytilaa ja varattu levytila ei vapaudu indeksiä poistettaessa. Kaikki indeksitiedostot poistuvat vasta funktion suorituksen jälkeen yhdellä kertaa. Niinpä testin suorittaminen 30 000 kertaa veisi levytilaa 495 GB. Koska testikoneella ei ole näin isoa kiintolevyä, jouduttiin testikierrosten määrä rajoittamaan 1 000 kierrokseen. Tällöin levytilaa tarvitaan 16,5 GB. Testin kokonaisajaksi saatiin 27 min 9 s. Testi käynnistettiin seuraavalla SQL-lauseella:

SELECT lisaaJaPoistaHajautusLaakenimi(1000);

```
create or replace function lisaaJaPoistaHajautusLaakenimi(maara integer) returns text AS $$
DECLARE
        i integer;
        BEGIN
                i := 1;
                WHILE i <= maara LOOP
                        CREATE INDEX laakenimi_hajautus
                                ON L_Laake using HASH (laakenimi);
                        DROP INDEX laakenimi_hajautus;
                        i := i + 1;
                END LOOP;
                RETURN 'Testi suoritettu';
        END;
$$ LANGUAGE plpgsql;
```

Kuva 39. lisaaJaPoistaHajautusLaakenimi-funktio. Tällä funktiolla voidaan testata hajautusindeksin luomiseen ja poistamiseen kuluvaa aikaa indeksoitaessa merkkijonotyyppistä laakenimi-saraketta.

Seuraavaksi siirryttiin testaamaan hajautusindeksin luomista ja poistamista numeeriseen pakkausnro-sarakkeeseen. Testi suoritettiin kuvan 40 esittämällä *lisaaJaPoistaHajautusPakkausnro*-funktiolla. Testikierrosten määrä pidettiin 1 000 kierroksessa. Tämän testin suorittaminen vei aikaa 4 min 47 s. Tästä huomataan, että hajautusindeksin luominen ja poistaminen merkkijonotyyppiseen sarakkeeseen on 5,7 kertaa hitaampaa numeeriseen sarakkeeseen verrattuna. Tällä kertaa yksi indeksi vei levytilaa 2,3 MB. Tämä on 7,2 kertaa pienempi tilantarve kuin indeksoitaessa merkkijonotyypistä saraketta.

```
create or replace function lisaaJaPoistaHajautusPakkausnro(maara integer) returns text AS $$
DECLARE
        i integer;
        BEGIN
                i := 1;
                WHILE i <= maara LOOP
                        CREATE INDEX pakkausnumero_hajautus
                                ON L_Laake using HASH (pakkausnro);
                        DROP INDEX pakkausnumero_hajautus;
                        i := i + 1;
                END LOOP;
                RETURN 'Testi suoritettu';
        END;
$$ LANGUAGE plpgsql;
```

Kuva 40. lisaaJaPoistaHajautusPakkausnro-funktio. Tällä funktiolla voidaan testata hajautusindeksin luomiseen ja poistamiseen kuluvaa aikaa indeksoitaessa numeerista pakkausnro-saraketta.

Hajautusindeksin tutkimisen jälkeen siirryttiin testaamaan B-puu-indeksiä. Testikierrosten määrä 1 000 pidettiin ennallaan, jotta tuloksia voidaan vertailla suoraan keskenään. Ensiksi testattiin B-puun luomista ja poistamista pakkausnro-sarakkeeseen kuvan 41 esittämällä *lisaaJaPoistaB_puuLaakenimi*-funktiolla. Tämän testin kokonaisajaksi saatiin 5 min 37 s. Tästä voidaan päätellä, että indeksoitaessa merkkijonoja hajautusindeksin luominen ja poistaminen on 4,8 kertaa hitaampaa kuin B-puun. Lisäksi huomattiin yhden B-puu-indeksin vievän levytilaa 1,8 MB. Tämä on 9,2 kertaa pienempi tilantarve verrattuna hajautusindeksin viemään levytilaan indeksoitaessa merkkijonotyyppistä saraketta.

```
create or replace function lisaaJaPoistaB_puuLaakenimi(maara integer) returns text AS $$
DECLARE
        i integer;
        BEGIN
                i := 1;
                WHILE i <= maara LOOP
                        CREATE INDEX laakenimi_b_puu
                                ON L_Laake using BTREE (laakenimi);
                        DROP INDEX laakenimi_b_puu;
                        i := i + 1;
                END LOOP;
                RETURN 'Testi suoritettu';
        END;
$$ LANGUAGE plpgsql;
```

Kuva 41. lisaaJaPoistaB_puuLaakenimi-funktio. Tällä funktiolla voidaan testata B-puu-indeksin luomiseen ja poistamiseen kuluvaa aikaa indeksoitaessa merkkijonotyyppistä laakenimi-saraketta.

Lopuksi tutkittiin B-puu-indeksin luomista ja poistamista pakkausnro-sarakkeelle. Tämä testi suoritettiin kuvan 42 esittämällä *lisaaJaPoistaB_puuPakkausnro*-funktiolla. Tämäkin testi ajettiin suorittaen 1 000 testikierrosta ja kokonaisajaksi saatiin 3 min 38 s. Tästä huomataan, että B-puun luominen ja poistaminen merkkijonotyyppiseen sarakkeeseen on 1,5 kertaa hitaampaa verrattuna numeeriseen sarakkeeseen. Lisäksi hajautusindeksin luominen ja poistaminen on 1,3 kertaa hitaampaa verrattuna B-puuhun indeksoitaessa numeerista pakkausnumeroa. Tässä tapauksessa yksi indeksi vei levytilaa 1,1 MB, mikä on 2,1 kertaa vähemmän kuin hajautusindeksiä käytettäessä. Lisäksi B-puun huomataan vievän 1,6 kertaa enemmän levytilaa indeksoitaessa merkkijonotyyppistä tietoa verrattuna numeeriseen tietoon.

```
create or replace function lisaaJaPoistaB_puuPakkausnumero(maara integer) returns text AS $$
DECLARE
        i integer;
        BEGIN
                i := 1;
                WHILE i <= maara LOOP
                        CREATE INDEX pakkausnumero_b_puu
                                ON L_Laake using BTREE (pakkausnro);
                        DROP INDEX pakkausnumero_b_puu;
                        i := i + 1;
                END LOOP;
                RETURN 'Testi suoritettu';
        END;
$$ LANGUAGE plpgsql;
```

Kuva 42. lisaaJaPoistaB_puuPakkausnro-funktio. Tällä funktiolla voidaan testata B-puu-indeksin luomiseen ja poistamiseen kuluvaa aikaa indeksoitaessa numeerista pakkausnro-saraketta.

Tämän kohdan Hilkka-ohjelman oletustietokannalla suoritetun testin tulokset on esitetty taulukossa 19. Tuloksista huomataan, että hajautusindeksin luominen ja poistaminen merkkijonotyyppiseen sarakkeeseen on muihin testattuihin tilanteisiin verrattuna hidasta. Lisäksi hajautusindeksin tapauksessa ero indeksoitaessa merkkijonoja ja indeksoitaessa numeerista tietoa on suuri. B-puullakin merkkijonojen indeksisointi oli hitaampaa kuin numeerisen tiedon indeksointi, mutta ero ei ollut niin suuri. Lisäksi huomattiin, että B-puun luominen on hajautusindeksin luomiseen verrattuna huomattavasti nopeampaa indeksoitaessa sekä merkkijonotyypistä että numeerista saraketta.

Taulukko 19: Indeksien lisääminen ja poistaminen Hilkka-ohjelman oletustietokantaan.

Tutkittu asia	Indeksien lisääminen ja poistaminen 1 000 kertaa Hilkka-ohjelman oletustietokantaan, joka sisältää 47 322 L_Laake-tietuetta.		
Käytetty indeksi	***Indeksoitu sarake***	***Kokonaisaika***	***Yhden askeleen keskim. aika***
Hajautusindeksi	laakenimi	27 min 9 s	1629 ms
	pakkausnro	4 min 47 s	287 ms
B-puu	laakenimi	5 min 37 s	337 ms
	pakkausnro	3 min 38 s	218 ms

Tämän kohdan tutkimuksissa kiinnitettiin myös huomiota indeksien kuluttamaan levytilaan. Tämän kokoiseen tietokantaan luotujen indeksien levytilantarpeet on esitetty taulukossa 20. Nämä tulokset ovat hyvin samankaltaisia kuin indeksien luonti- ja poistoajat. Huomataan, että merkkijonotyyppiseen laakenimi-sarakkeeseen luotu hajautusindeksi kuluttaa levytilaa huomattavan paljon enemmän verrattuna muihin testattuihin tilanteisiin. Yleisesti ottaen voidaan sanoa hajautusindeksin kuluttavan levytilaa enemmän kuin B-puu. Lisäksi merkkijonotyyppisen sarakkeen indeksoiminen näyttäisi vievän enemmän tilaa kuin numeerisen sarakkeen indeksoiminen.

Taulukko 20: Indeksien kuluttama levytila Hilkka-ohjelman oletustietokannassa.

Tutkittu asia	Yhden indeksin kuluttama levytila Hilkka-ohjelman oletustietokannassa, joka sisältää 47 322 L_Laake-tietuetta.		
Käytetty indeksi	***Indeksoitu sarake***	***Levytila kilotavuina***	***Levytila megatavuina***
Hajautusindeksi	laakenimi	16 464 kB	16,5 MB
	pakkausnro	2 344 kB	2,3 MB
B-puu	laakenimi	1 760 kB	1,8 MB
	pakkausnro	1 072 kB	1,1 MB

Tässä kohdassa mitattujen indeksien luontiaikojen ja levytilantarpeiden perusteella voidaan sanoa B-puun olevan hajautusindeksiä parempi. Kuitenkin tässä kohdassa käytetyssä

tietokannassa hitaimmankin luonti- ja poisto-operaation yhteenlaskettu aika on alle 2 sekuntia. Todellisessa käytössä indeksejä luodaan ja poistetaan harvoin, joten siinä mielessä tämän kohdan tuloksia ei voida pitää kovin ratkaisevana tekijänä indeksityypin valinnassa. Seuraavaksi kohdassa 4.14 tarkastellaan tilannetta isommalla tietokannalla, jossa indeksien välillä saattaa olla suurempia eroja.

4.14 Indeksien luominen ja poistaminen kohdassa 4.11 syntyneessä tietokannassa

Tässä kohdassa on tarkoitus tutkia samoja asioita kuin luvussa 4.13 käyttäen tietokantaa, jossa on enemmän lääketietueita. Tietokantana tämän kohdan testeissä käytetään kohdassa 4.11 syntynyttä tietokantaa, joka sisältää 2 047 322 L_Laake-tietuetta. Tutkimuksissa on tarkoitus käyttää samoja testifunktioita kuin kohdassa 4.13 eli kuvien 39, 40, 41 ja 42 esittämiä SQL-funktioita.

Tämä osatutkimus aloitettiin jälleen oletetusti vaikeimmasta tapauksesta eli hajautusindeksin lisäämisestä laakenimi-sarakkeelle. Alustavalla testillä huomattiin, että 12 tunnissa tietokantaan pystytään luomaan ja poistamaan hajautusindeksi noin 20 kertaa. Lisäksi huomattiin yhden indeksin kuluttavan levytilaa 526,8 MB, eli 20 kierroksen testissä levytilaa tarvitaan 19,5 GB. Niinpä testikierrosten määräksi valittiin edellä mainittu. Tämän testin ajaminen vei aikaa 16 h 40 min 44 s. Tässä testissä käytettiin kuvan 39 esittämää testifunktiota, joka käynnistettiin seuraavalla SQL-lauseella:

```
SELECT lisaaJaPoistaHajautusLaakenimi(20);
```

Seuraavaksi tutkittiin hajautusindeksin luomista ja poistamista indeksoitaessa numeerista pakkausnro-saraketta. Testi suoritettiin kuvan 40 esittämällä *lisaaJaPoistaHajautusPakkausnro*-funktiolla ja testikierrosten määrä pidettiin 20 kierroksessa. Tämän testin suorit-

taminen vei aikaa 1 h 24 min 8 s. Tästä huomataan hajautusindeksin luomisen olevan merkkijonosarakkeeseen 11,9 kertaa hitaampaa kuin numeeriseen sarakkeeseen. Pienemmällä tietokannalla kohdassa 4.13 testattuna tämä sama ero oli vain 5,7-kertainen. Näyttää siis siltä, että hajautusindeksin luominen ja poistaminen käy hitaammaksi tietokannan koon kasvaessa. Lisäksi tässä tapauksessa yksi indeksi vei levytilaa 83,4 MB. Tämä on 6,3 kertaa pienempi tilantarve kuin indeksoitaessa merkkijonotyyppistä tietoa hajautusindeksillä tämän kokoisessa tietokannassa.

Tämän jälkeen siirryttiin testaamaan B-puu-indeksin luomista ja poistamista laakenimi-sarakkeeseen kuvan 41 esittämällä, *lisaaJaPoistaB_puuLaakenimi*-funktiolla. Testikierrosten määrä pidettiin 20:ssä ja testin kokonaisajaksi saatiin 27 min 48 s. Tästä voidaan päätellä, että indeksoitaessa merkkijonotyyppistä tietoa hajautusindeksin luominen ja poistaminen on 36 kertaa hitaampaa verrattuna B-puuhun. Pienellä tietokannalla kohdassa 4.13 testattuna tämä ero oli vain 4,8-kertainen. Tässä tapauksessa huomattiin yhden indeksin vievän levytilaa 59,6 MB. Tämä on 8,8 kertaa pienempi määrä kuin hajautusindeksin tapauksessa indeksoitaessa merkkijonotyyppistä saraketta.

Lopuksi tutkittiin B-puu-indeksin luomista ja poistamista pakkausnro-sarakkeelle. Tämä testi suoritettiin kuvan 42 esittämällä *lisaaJaPoistaB_puuPakkausnro*-funktiolla. Testikierroksia ajettiin 20 ja kokonaisajaksi saatiin 27 min 23 s. Tämä tulos on vain 1,02 kertaa parempi verrattuna B-puun luomiseen laakenimi-sarakkeelle. Näyttää siis siltä, että B-puun tapauksessa tietotyypillä ei ole merkitystä luonti- ja poistoaikaan. Pienemmällä kannalla kohdassa 4.13 testattuna tämä ero oli suurempi, 1,5-kertainen. Tästä voidaan päätellä, että B-puu toimii hyvin myös suurillakin tietokannoilla. Indeksoitaessa numeerista pakkausnumeroa hajautusindeksi on 3,1 kertaa hitaampi verrattuna B-puuhun. Pienemmällä tietokannalla testattuna kohdassa 4.13 tämä ero oli 1,3-kertainen. Tästä huomataan jälleen, että mitä suurempi kanta, sitä suurempi ero indeksien välillä on. Tässä tapauksessa yksi indeksi vei levytilaa 44,9 MB, mikä on 1,9 kertaa vähemmän kuin hajautusindek-

sin tapauksessa. Lisäksi tämä on 1,3 kertaa pienempi tilantarve verrattuna merkkijonotyyppisen tiedon indeksointiin B-puu-indeksillä.

Tässä kohdassa suoritettujen testien tulokset on esitetty taulukossa 21. Tuloksista huomataan, että hajautusindeksin luominen merkkijonotyyppiseen sarakkeeseen on suurellakin tietokannalla muihin testattuihin tilanteisiin verrattuna hidasta. Huomattiin, että hajautusindeksin tapauksessa ero indeksoitaessa merkkijonoja ja numeerista tietoa on suuri. B-puun tapauksessa merkkijonojen ja numeeristen tietojen indeksisointi oli hyvin lähellä toisiaan, pienemmällä kannalla tämä ero oli suurempi. Lisäksi huomattiin, että B-puun luominen on hajautusindeksin luomiseen verrattuna huomattavasti nopeampaa indeksoitaessa sekä merkkijonotyyppistä että numeerista saraketta. Tämän kokoisella tietokannalla hajautusindeksin ja B-puun välinen ero on jo huomattava, joten B-puun käyttö on suositeltavampaa.

Taulukko 21: Indeksien lisääminen ja poistaminen kohdassa 4.11 syntyneeseen tietokantaan.

Tutkittu asia	Indeksien lisääminen ja poistaminen 20 kertaa kohdassa 4.11 syntyneeseen tietokantaan, joka sisältää 2 047 322 L_Laake-tietuetta.		
Käytetty indeksi	***Indeksoitu sarake***	***Kokonaisaika***	***Yhden askeleen keskim. aika***
Hajautusindeksi	laakenimi	16 h 40 min 44 s	3 002 200 ms (3002,2 s)
	pakkausnro	1 h 24 min 8 s	252 400 ms (252,4 s)
B-puu	laakenimi	27 min 48 s	83 400 ms (83,4 s)
	pakkausnro	27 min 23 s	82 150 ms (82,15 s)

Taulukossa 22 on esitetty tämän kohdan testeissä luotujen indeksien tilantarpeet. Nämä tulokset ovat hyvin samankaltaisia kuin kohdassa 4.13. Merkkijonotyyppiseen laakenimisarakkeeseen luotu hajautusindeksi kuluttaa levytilaa yli kuusi kertaa enemmän kaikkiin muihin tutkittuihin tilanteisiin verrattuna. B-puun huomattiin vievän levytilaa hajautusindeksiä vähemmän. Lisäksi B-puun kohdalla indeksoitavalla tietotyypillä ei ole kovin

suurta merkitystä indeksin kokoon kiintolevyllä.

Taulukko 22: Indeksien kuluttama levytila kohdassa 4.11 syntyneessä tietokannassa.

Tutkittu asia	Yhden indeksin kuluttama levytila kohdassa 4.11 syntyneessä tietokannassa, joka sisältää 2 047 322 L_Laake-tietuetta.		
Käytetty indeksi	***Indeksoitu sarake***	***Levytila kilotavuina***	***Levytila megatavuina***
Hajautusindeksi	laakenimi	526 800 kB	526,8 MB
	pakkausnro	83 416 kB	83,4 MB
B-puu	laakenimi	59 576 kB	59,6 MB
	pakkausnro	44 928 kB	44,9 MB

5 Yhteenveto

Tämän tutkielman alussa perehdyttiin tietokannan indeksointiin ja sen tarpeellisuuteen yleisellä tasolla. Tämän jälkeen tutkielmassa tarkasteltiin kahta seuraavaa tietokannan indeksointimenetelmää:

- B-puut: B-puut ovat yksi käytetyimmistä indeksointirakenteista. Tämän tutkielman toisessa luvussa tarkasteltiin B^+-puun rakennetta. Lisäksi tarkasteltiin tietueiden hakemista, lisäämistä ja poistamista B^+-puusta. Tämän lisäksi tarkasteltiin B^{link}-puuta, joka mahdollistaa yhtäaikaisen puun käytön.
- Hajautusindeksi: Hajautusindeksi on B-puun ohella toinen käytetyimmistä indeksointimenetelmistä. Tämän tutkielman kolmannessa luvussa tarkasteltiin aluksi yleisesti hajautuksen periaatteita ja sen tehokkuutta. Tämän jälkeen tarkasteltiin kahta dynaamisen hajautuksen menetelmää: laajentavaa ja lineaarista hajautusta.

Kummankin indeksointimenetelmän kohdalla tarkasteltiin myös niiden käyttämistä PostgreSQL-tietokannassa. Tämän lisäksi annettiin myös esimerkkejä indeksien toiminnasta sekä esitettiin hypoteeseja liittyen indeksien tehokkuuteen.

Luvussa 4 vertailtiin B-puun ja hajautusindeksin välistä tehokkuutta PostgreSQL-tietokannassa yrittäen löytää näyttöä kohdissa 2.7 ja 3.10 esitettyihin tehokkuushypoteeseihin. Tässä luvussa esitetyissä testeissä käytettiin apuna Fastroi Oy:n Hilkka-ohjelman tietokantaa. Tutkimuksen tarkoituksena oli myös löytää ratkaisuja, Hilkka-ohjelman lääkelistan käytön nopeuttamiseen.

Kohdissa 4.1 - 4.12 vertailtiin B-puuta ja hajautusindeksiä erilaisilla tietokantahauilla, tietueiden lisäyksillä sekä poistoilla. Taulukoissa 23 ja 24 on esitetty näissä testeissä esiintulleet indeksien aiheuttamat muutokset suoritusaikaan indeksoimattomaan tietokantaan

verrattuna. Positiivinen muutos tarkoittaa tilannetta, jossa indeksi nopeutti testin suorittamista. Esimerkiksi kohdassa 4.1 B-puuta käytettäessä huomattiin hakujen nopeuden kasvavan 16,03-kertaiseksi verrattuna indeksoimattomaan tietokantaan. Vastaavasti negatiivisen tuloksen kohdalla indeksin todettiin hidastavan tietokannan toimintaa. Esimerkiksi kohdassa 4.2 B-puun huomattiin hidastavan hakuja 1,003-kertaisesti.

Taulukko 23: Kohdissa 4.1 - 4.3 ja 4.6 - 4.12 suoritettujen testien tulokset verrattuna indeksoimattomaan tietokantaan.

Testi	***B-puu***	***Hajautusindeksi***
4.1 Satunnaisen pakkausnumeron hakeminen	+16,03	+15,2
4.2 Pakkausnumeroiden hakeminen satunnaiselta väliltä	-1,003	-1,005
4.3 Pakkausnumeroiden hakeminen satunnaiselta väliltä. Taulu järjestetty pakkausnumeron perusteella.	-1	-1,03
4.6 Pakkausnumeroiden määrän hakeminen satunnaiselta väliltä	+2,0	+1,01
4.7 Lääkenimen hakeminen like merkkijono% -menetelmällä	+1,4	+1,0
4.8 Lääkenimen hakeminen like %merkkijono% -menetelmällä	-1,02	-1,01
4.9 Lääkenimen hakeminen like-menetelmällä ilman jokerimerkkejä	+8,4	+1,003
4.10 Lääkenimen hakeminen yhtäsuuruus-menetelmällä	+8,2	+8,2
4.11 Indeksoinnin vaikutus tietueiden lisäämiseen	-1,2	-1,3
4.12 Indeksoinnin vaikutus tietueiden poistamiseen	+1,01	-1,01

Kohdissa 4.1 - 4.10 indeksien tehokkuutta vertailtiin tietokantahauilla, joissa indeksien huomattiin toimivan hyvin pitkälle niin kuin PostgreSQL-manuaalissa on kerrottu. Tutkimuksessa huomattiin B-puun ja hajautusindeksin välillä olevan suuriakin eroja. Esimerkiksi numeeristen arvovälikyselyiden ja merkkijonotyyppisten like-menetelmillä tehtyjen

kyselyiden kohdalla hajautusindeksillä ei huomattu olevan mitään vaikutusta. Sen sijaan suurimmassa osassa tutkituista tilanteista B-puu nopeutti tietokantahakuja huomattavasti. Tutkittaessa yhtäsuuruus-menetelmällä tehtyjä hakuja sekä numeeriseen että merkkijonotyyppiseen sarakkeeseen huomattiin B-puun ja hajautusindeksin olevan hyvin tasaveroisia.

Taulukko 24: Kohdissa 4.4 ja 4.5 suoritettujen testien tulokset koskien B-puu-indeksin ja tietokannan taulun järjestämisen vaikutusta arvovälikyselyihin eripituisilla arvoväleillä verrattuna indeksoimattomaan ja järjestämättömään tietokantaan.

Testi	*Arvovälin pituus*	*B-puu*	*Järjestetty*	*B-puu ja järjestetty*
4.4 Arvovälin pituuden vaikutus arvovälikyselyn nopeuteen Hilkka-ohjelman oletustietokannalla	10	+9,5	+1,0	+9,6
	100	+3,0	+1,0	+3,0
	1 000	+1,2	+1,0	+1,3
	10 000	+1,0	+1,0	+1,0
4.5 Arvovälin pituuden vaikutus arvovälikyselyn nopeuteen isolla tietokannalla	10	+26,5	+1,8	+688,7
	100	+3,6	+1,9	+89,6
	1 000	+1,5	+2,2	+11,3
	5 000	+1,7	+1,3	+5,1

Kohdissa 4.2 ja 4.3 tutkittiin numeeriseen sarakkeeseen suoritettuja arvovälikyselyitä. Näissä testeissä indeksoinnin ja taulun järjestämisen ei havaittu nopeuttavan arvovälikyselyitä. Niinpä kohdissa 4.4 ja 4.5 haluttiin tarkastella tarkemmin B-puun toimintaa eripituisten arvovälikyselyiden yhteydessä erikokoisilla järjestämättömillä ja järjestetyillä tietokannoilla. Tulokset näistä testeistä on esitetty taulukossa 24. Näissä testeissä huomattiin B-puu-indeksin toimivan arvovälikyselyissä tietokannan koosta riippumatta sitä paremmin, mitä pienempi kyselyn tuottama tulosjoukko on taulun kokoon verrattuna. Lisäksi huomattiin, ettei tietokannan taulun järjestämisestä ole hyötyä Hilkka-ohjelman oletustietokannalla. Järjestämisestä todettiin kuitenkin olevan hyötyä isommalla tietokannalla, joka sisältää 2 047 322 lääketietuetta. Lisäksi isomman tietokannan tapauksessa arvoväliky-

selyiden huomattiin nopeutuvan todella paljon, jos tietokannan taulu yhtä aikaa sekä järjestetään että indeksoidaan B-puu-indeksillä haettavan avaimen perusteella.

Kohdissa 4.11 ja 4.12 tutkittiin indeksien vaikutusta tietokannan tauluun kohdistuneisiin tietueiden lisäyksiin ja poistoihin. Tässä tutkimuksessa huomattiin sekä B-puun että hajautusindeksin hidastavan tietueiden lisäyksiä hieman. Ero indeksien välillä ei ollut kovin suuri, mutta hajautusindeksi hidasti tietokannan toimintaa hieman enemmän. Kohdassa 4.12 huomattiin, ettei tutkituilla indekseillä ole vaikutusta tietokannan tauluun kohdistuviin tietueiden poistoihin.

Kohdissa 4.13 ja 4.14 tutkittiin indeksien luomiseen ja poistamiseen kuluvaa aikaa. Kohdassa 4.13 käytettiin tietokantana Hilkka-ohjelman oletustietokantaa, joka sisältää 47 322 lääketietuetta. Tämän kokoisella tietokannalla indeksien luomisen ja poistamisen todettiin olevan nopeaa indeksistä ja tietotyypistä riippumatta. Kohdassa 4.14 indeksien luomista ja poistamista tutkittiin käyttäen kohdassa 4.11 syntynyttä suurempaa tietokantaa, joka sisältää 2 047 322 lääketietuetta. Edellä mainitun kokoisella tietokannalla havaittiin jo merkittäviä eroja indeksien ja indeksoitavien tietotyyppien välillä.

Tutkittaessa indeksien luomiseen ja poistamiseen kuluvaa aikaa huomattiin merkkijonotyyppisen lääkeniminen olevan hitaampi indeksoitava kuin numeerisen pakkausnumeron. Taulukossa 25 on esitetty lääkenimen ja pakkausnumeron indeksoimiseen ja indeksin poistamiseen kuluneen ajan suhteet riippuen tietokannan koosta sekä käytettävästä indeksistä. Näistä tuloksista voidaan huomata tietotyypin vaikuttavan verrattain paljon hajautusindeksin luonti- ja poistoaikoihin, lisäksi tietotyypeistä johtuva ero näyttäisi kasvavan tietokannan koon kasvaessa. Sitä vastoin B-puun kohdalla tietotyypillä ei näytä olevan suurta merkitystä luonnin ja poiston nopeuteen.

Taulukko 25: Lääkenimen ja pakkausnumeron indeksoimiseen ja indeksin poistamiseen kuluneen ajan suhteet riippuen tietokannan koosta sekä käytettävästä indeksistä.

Testi	*Indeksi*	*Lääkenimi / Pakkausnumero*
4.13 Indeksien luominen ja poistaminen Hilkka-ohjelman oletustietokannassa	B-puu	1,5
	Hajautusindeksi	5,7
4.14 Indeksien luominen ja poistaminen kohdassa 4.11 syntyneessä tietokannassa	B-puu	1,0
	Hajautusindeksi	11,9

Lisäksi kohdissa 4.13 ja 4.14 huomattiin hajautusindeksin luomisen ja poistamisen olevan B-puuta hitaampaa kaikissa testatuissa tilanteissa. Taulukossa 26 on esitetty hajautusindeksin ja B-puun luonti- ja poistoaikojen suhteet riippuen tietokannan koosta sekä indeksoitavasta sarakkeesta. Näistä tuloksista huomataan, että indeksoitaessa numeerista pakkausnumeroa hajautusindeksin ja B-puun ero ei ole niin suuri kuin merkkijonotyyppistä lääkenimeä indeksoitaessa. Lisäksi huomataan, että hajautusindeksin ja B-puun välinen ero kasvaa tietokannan koon kasvaessa.

Taulukko 26: Hajautusindeksin ja B-puun luonti- ja poistoaikojen suhde riippuen tietokannan koosta sekä indeksoitavasta sarakkeesta.

Testi	*Sarake*	*Hajautusindeksi / B-puu*
4.13 Indeksien luominen ja poistaminen Hilkka-ohjelman oletustietokannassa	Pakkausnumero	1,3
	Lääkenimi	4,8
4.14 Indeksien luominen ja poistaminen kohdassa 4.11 syntyneessä tietokannassa	Pakkausnumero	3,1
	Lääkenimi	36,0

Luvussa 4 tehdyissä testeissä huomattiin tutkittujen indeksien toimivan suurelta osin PostgreSQL-manuaalin mukaisesti. Ainoastaan kohdissa 4.2 ja 4.3 tutkittujen arvoväliky-selyiden kohdalla B-puu ei tuottanut PostgreSQL-manuaalin lupaamaa tulosta. Suurimmassa osassa testeistä B-puu oli tehokkaampi kuin hajautusindeksi. Osassa testeistä B-

puu ja hajautusindeksi olivat hyvin tasaveroisia, mutta yhdessäkään testissä hajautusindeksi ei osoittautunut B-puuta paremmaksi. Tämän lisäksi kohtien 4.13 ja 4.14 testeissä hajautusindeksin huomattiin vievän selvästi enemmän levytilaa kuin B-puun. Niinpä tämän tutkimuksen tuloksena voidaan pitää sitä, että PostgreSQL-tietokannassa hajautusindeksiä ei kannata käyttää vaan kannattaa käyttää B-puu-indeksiä.

Tutkimuksessa löydettiin myös joitain ratkaisuja Hilkka-ohjelman lääkelistan toiminnan nopeuttamiseen. Normaalissa Hilkka-ohjelman käytössä lääketietueita haetaan lääkenimen ja pakkausnumeron perusteella. Lääkenimihaut tehdään käyttäen kodassa 4.8 tutkittua like %merkkijono% -menetelmää. Tässä tutkimuksessa ei ikävä kyllä havaittu indeksoinnin vaikuttavan tämän tyyppisiin merkkijonohakuihin. Sen sijaan kohdassa 4.1 todettiin indeksoinnin parantavan yksittäisten lääketietueiden hakemista pakkausnumeron perusteella merkittävästi. Hilkka-ohjelmassa L_Laake-taulua ei päivitetä normaalin käytön aikana, joten pakkausnumeron indeksoimisesta ei voi olla haittaa ohjelman toiminnalle. Näin ollen tulevissa Hilkka-ohjelman versioissa lääkelista tullaan todennäköisesti indeksoimaan pakkausnumeron perusteella B-puu-indeksillä.

Hilkka-ohjelman lääketaulu voidaan päivittää kerran kuukaudessa ilmestyvällä päivitysohjelmalla, joka sisältää Lääkelaitoksen uusimman lääkelistan. Päivitys suoritetaan niin, että tietokannasta poistetaan kaikki sellaiset irralliset L_Laake-tietueet, jotka eivät liity asiakkaitten lääkityksiin. Tietokantaan jäävät vanhat lääkkeet merkitään käytöstä poistetuiksi. Tämän jälkeen tietokantaan syötetään lääkelistan uudet lääkkeet. Jos lisättävä uusi lääketietue on jo tietokannassa olevissa vanhoissa lääkkeissä, uutta tietuetta ei lisätä, vaan vanha lääketietue otetaan takaisin käyttöön.

Päivityksen yhteydessä uutta lääketietuetta lisättäessä tietokannasta tarkistetaan, onko lisättävä tietue vanhoissa lääkkeissä. Tämä tarkastelu hoidetaan kahdessa vaiheessa. Ennen uusien lääkkeiden lisäyksen aloittamista tietokannan sisältämien vanhojen lääkkeiden

pakkausnumerot haetaan listaan. Uutta lääkettä lisättäessä tarkastetaan ensin, sisältääkö vanhojen pakkausnumeroiden lista lisättävän uuden lääkkeen pakkausnumeron. Jos pakkausnumero ei sisälly listaan, uusi lääketietue lisätään suoraan, jolloin tietokantaan ei kohdistu yhtään kyselyä. Jos vanhojen pakkausnumeroiden lista sisältää lisättävän lääkkeen pakkausnumeron, joudutaan tietokantaan suorittamaan haku, joka tarkistaa, löytyykö tietokannasta juuri lisättävän kaltaista lääketietuetta. Tässä kyselyssä vertailut tehdään puhtaina yhtäsuuruusvertailuina ilman jokerimerkkejä. Pakkausnumero-sarakkeelle luotu B-puu-indeksi todennäköisesti nopeuttaa tätä kyselyä, ainakin jos pakkausnumerovertailu tehdään ensimmäisenä HQL-lauseessa.

Hilkka -ohjelmaa kehitettäessä on kuitenkin käynyt ilmi, että poiston jälkeen tietokantaan jäävien lääkkeiden määrä on varsin pieni verrattuna koko lääkelistan kokoon. Niinpä tätä edellä kuvattua raskasta hakua suoritetaan päivityksen aikana suhteellisen harvoin. Päivityksessä suurin osa ajasta kuluu vanhojen lääkkeiden poistoon ja uusien tietueiden kirjoittamiseen tietokantaan. Tässä tutkimuksessa indeksoinnin ei todettu vaikuttavan ensin mainittuun. Viimeksi mainitun toimenpiteen todettiin kuitenkin hidastuvan indeksejä käytettäessä. Niinpä onkin syytä tutkia, onko päivityksen ajaminen kokonaisuudessaan nopeampaa, jos pakkausnumerosarake on indeksoitu B-puulla, vai onko indeksi syytä poistaa päivityksen ajaksi ja luoda se uudelleen päivityksen jälkeen.

Viitteet

CachemanXP (2008) *Outertech - computer memory and cache optimization, tuneup and internet utility software* http://www.outertech.com/index.php?_charisma_page=product&id=7 (14.2.2008)

Comer, D. (1979) *The Ubiquitous B-Tree,* Computing Surveys, Vol 11, No 2, June 1979.

Connolly, T., Begg, C. (2005) *Database Systems, A Practical Approach to Design, Implementing and Management*. Addison-Wesley, Harlow, England.

Elmasri, R., Navathe, S.B. (2007) *Database Systems*. Addison-Wesley, Boston.

Garcia-Molina, H., Ullman, J.D., Widom, J. (2002) *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey.

Hibernate (2007) *Hibernate documentation*. http://www.hibernate.org/5.html (25.11.2007).

Java (2007) *Random (Java Platform SE 6).* http://java.sun.com/javase/6/docs/api/java/util/Random.html (16.12.2007).

JavaBeat (2008) *Introduction to Hibernate Caching.* http://www.javabeat.net/articles/37-introduction-to-hibernate-caching-1.html (21.4.2008).

Java Spring (2007) *Spring Framework Documentation.* http://www.springframework.org/documentation (25.11.2007).

Kortelainen, T. (2007) *Tietokannan indeksointi.* Kandidaatin tutkielma, Joensuun yliopisto, tietojenkäsittelytiede.

Lanin V., Shasha D.: (1986) *A Symmetric Concurrent B-Tree Algorithm*, Proceedings of 1986 Fall Join Computer Conference, Pages 380-389.

Lehman, P.L., Yao, S.B.: (1981) E*fficient Locking For Concurrent Operations On B-Trees*, ACM Transactions Of Database Systems, vol. 6, No. 4, December. 1981, Pages. 650-670.

Litwin, W. (1980) *Linear hashing : A new tool for file and table addressing*, I. N. R. I. A. 78 150 Le Chesnay, France.

Microsoft (2002) *The Default Cluster Size for the NTFS and FAT File Systems.* http://support.microsoft.com/kb/314878 (30.12.2007).

PostgreSQL (2006) Lähdekoodi: pgsql/src/backend/access/nbtree/README versio 1.14

PostgreSQL (2007) *PostgreSQL Manuals versiot 8.1 ja 8.2*. http://www.postgresql.org/docs/manuals/ (10.3.2007).

Silberschatz, A., F., Korth, H., Sudarshan, S. (1997) *Database System Concepts,* McGraw-Hill, New York.

Liite 1: Luvussa 4 käytetty testiohjelma

Luokka IndeksiTesti, joka sisältää kohdissa 4.1 - 4.12 käytetyt testifunktiot:

```
package fi.fastroi.hilkka2.indeksitesti;

import java.util.List;
import java.util.Random;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.springframework.beans.BeanUtils;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.HibernateTemplate;

import fi.fastroi.hilkka2.dao.impl.generated.laakelaitos.L_LaakeDAOImplBase;
import fi.fastroi.hilkka2.model.laakelaitos.L_Atc;
import fi.fastroi.hilkka2.model.laakelaitos.L_Laake;
import fi.fastroi.hilkka2.service.laakelaitos.L_LaakeManager;

public class IndeksiTesti extends L_LaakeDAOImplBase{
	protected L_LaakeManager l_LaakeManager = null;

	public void init(org.springframework.context.ApplicationContext ctx){
		org.springframework.context.ApplicationContext applicationContext;
		applicationContext = ctx;

		l_LaakeManager = (L_LaakeManager)
applicationContext.getBean(org.springframework.util.StringUtils.uncapitalize("L_LaakeManager"),
L_LaakeManager.class);
	}

	public void testPakkausnumero(int maara){
		Random rand = new Random();

		for (int i=0; i<maara; i++) {
			List<L_Laake> list = findByPakkausnumero(new Long(rand.nextInt(100000)));
		}
	}

	public void testPakkausnumeroVali(int maara) {
		Random generator = new Random();

		for (int i=0; i<maara; i++) {
			Long alku = new Long(generator.nextInt(100000));
			int jaljella = 100000-alku.intValue();
			Long loppu = new Long(generator.nextInt(jaljella) + alku);
```

```
                List<L_Laake> lista = findByPakkausnumeroVali(alku, loppu);
        }
}

public void testKiinteaArvovali(int maara, int pituus) {
        Random generator = new Random();

        for (int i=0; i<maara; i++) {
                Long alku = new Long(generator.nextInt(100000));
                if ((alku+pituus) > 100000) {
                        alku=100000L-pituus;
                }
                Long loppu = new Long(alku + pituus);

                List<L_Laake> lista = findByPakkausnumeroVali(alku, loppu);
        }
}

public void testPakkausnumeroValiCount(int maara) {
        Random generator = new Random();

        for (int i=0; i<maara; i++) {
                Long alku = new Long(generator.nextInt(100000));
                int jaljella = 100000-alku.intValue();
                Long loppu = new Long(generator.nextInt(jaljella)) + alku;

                List<Integer> lista = findByPakkausnumeroValiCount(alku, loppu);
        }
}

public void testLaakenimiLikeP(int maara) {
        Random generator = new Random();

        List<String> laakenimiList = findLaakenimet();
        int listSize = laakenimiList.size();

        for (int i=0; i<maara; i++) {
                String laakenimi = laakenimiList.get(generator.nextInt(listSize));
                int nimiLength = laakenimi.length();
                String subNimi = laakenimi.substring(0,generator.nextInt(nimiLength)+1);
                List<L_Laake> lista = findByLaakenimiLike(subNimi + "%");
        }
}

public void testLaakenimiLikePP(int maara) {
        Random generator = new Random();

        List<String> laakenimiList = findLaakenimet();
        int listSize = laakenimiList.size();
```

```
		for (int i=0; i<maara; i++) {
			String laakenimi = laakenimiList.get(generator.nextInt(listSize));
			int nimiLength = laakenimi.length();
			int nimiAlku = generator.nextInt(nimiLength-1);
			int nimiLoppu = nimiAlku + generator.nextInt(nimiLength-nimiAlku -1) +1;
			String subNimi = laakenimi.substring(nimiAlku, nimiLoppu);
			List<L_Laake> lista = findByLaakenimiLike("%" + subNimi + "%");
		}
	}

	public void testLaakenimiLike(int maara) {
		Random generator = new Random();

		List<String> laakenimiList = findLaakenimet();
		int listSize = laakenimiList.size();

		for (int i=0; i<maara; i++) {
			String laakenimi = laakenimiList.get(generator.nextInt(listSize));

			List<L_Laake> lista = findByLaakenimiLike(laakenimi);
		}
	}

	public void testLaakenimiYhtasuuruus(int maara) {
		Random generator = new Random();

		List<String> laakenimiList = findLaakenimet();
		int listSize = laakenimiList.size();

		for (int i=0; i<maara; i++) {
			String laakenimi = laakenimiList.get(generator.nextInt(listSize));
			List<L_Laake> lista = findByLaakenimiYhtasuuruus(laakenimi);
		}
	}

	public void testLaakeLisaaminen(int maara) {
		Random generator = new Random();

		List<L_Laake> laakeList = l_LaakeManager.loadAll();
		List<String> laakenimiList = findLaakenimet();
		int laakeListSize = laakeList.size();
		int nimiListSize = laakenimiList.size();

		for (int i=0; i<maara; i++) {
			L_Laake laake = laakeList.get(generator.nextInt(laakeListSize));
			L_Laake uusiLaake = new L_Laake();
			BeanUtils.copyProperties(laake, uusiLaake);
			uusiLaake.setId(null); // Nyt tämä todellakin on uusi lääke, ei viitettä kantaan

			String laakenimi1 = laakenimiList.get(generator.nextInt(nimiListSize));
			int nimiLength1 = laakenimi1.length();
```

```
                int alku1 = generator.nextInt(nimiLength1-1);
                int loppu1 = alku1 + generator.nextInt(nimiLength1-alku1 -1) +1;
                String subNimi1 = laakenimi1.substring(alku1, loppu1);

                String laakenimi2 = laakenimiList.get(generator.nextInt(nimiListSize));
                int nimiLength2 = laakenimi2.length();
                int alku2 = generator.nextInt(nimiLength2-1);
                int loppu2 = alku2 + generator.nextInt(nimiLength2-alku2 -1) +1;
                String subNimi2 = laakenimi2.substring(alku2,loppu2);

                String uusiLaakenimi = subNimi1 + subNimi2;
                if (uusiLaakenimi.length() > 64) {
                        uusiLaakenimi = uusiLaakenimi.substring(0, 64);
                }

                uusiLaake.setLaakenimi(uusiLaakenimi);
                uusiLaake.setPakkausnro(new Long(generator.nextInt(100000)));

                l_LaakeManager.save(uusiLaake);
        }
}

public void testLaakePoistaminen(int maara){
        Random generator = new Random();

        List<Long> idList = findLakkeIds();
        int idListSize = idList.size();

        for (int i=0; i<maara; i++) {
                Long id = idList.get(generator.nextInt(idListSize));
                l_LaakeManager.delete(id);
                idList.remove(id);
                idListSize--;
        }
}

@SuppressWarnings("unchecked")
public List<L_Laake> findByPakkausnumero(final Long pakkausnumero) {
        HibernateTemplate hibernateTemplate = getHibernateTemplate();

        List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {

                        String sQuery = "FROM L_Laake WHERE pakkausnro = ?";
                        Query query = session.createQuery(sQuery);
                        query.setLong(0, pakkausnumero);
                        return query.list();
                }
        });
        return list;
}
```

```
@SuppressWarnings("unchecked")
public List<L_Laake> findByPakkausnumeroVali(final Long alku, final Long loppu) {
	HibernateTemplate hibernateTemplate = getHibernateTemplate();

	List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
		public Object doInHibernate(Session session) throws HibernateException {

			String sQuery = "FROM L_Laake " +
					"WHERE pakkausnro >= ? AND pakkausnro <= ?";
			Query query = session.createQuery(sQuery);
			query.setLong(0, alku);
			query.setLong(1, loppu);
			return query.list();
		}
	});
	return list;
}

@SuppressWarnings("unchecked")
public List<Integer> findByPakkausnumeroValiCount(final Long alku, final Long loppu) {
	HibernateTemplate hibernateTemplate = getHibernateTemplate();

	List <Integer> list = hibernateTemplate.executeFind(new HibernateCallback() {
		public Object doInHibernate(Session session) throws HibernateException {

			String sQuery = "SELECT count(laake) FROM L_Laake as laake " +
				"WHERE laake.pakkausnro >= ? AND laake.pakkausnro <= ?";
			Query query = session.createQuery(sQuery);
			query.setLong(0, alku);
			query.setLong(1, loppu);
			return query.list();
		}
	});
	return list;
}

@SuppressWarnings("unchecked")
public List<String> findLaakenimet() {
	HibernateTemplate hibernateTemplate = getHibernateTemplate();

	List <String> list = hibernateTemplate.executeFind(new HibernateCallback() {
		public Object doInHibernate(Session session) throws HibernateException {

			String sQuery = "SELECT DISTINCT laakenimi FROM L_Laake";
			Query query = session.createQuery(sQuery);
			return query.list();
		}
	});
	return list;
}
```

```
	@SuppressWarnings("unchecked")
	public List<L_Laake> findByLaakenimiLike(final String laakenimi) {
		HibernateTemplate hibernateTemplate = getHibernateTemplate();

		List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
			public Object doInHibernate(Session session) throws HibernateException {

				String sQuery = "FROM L_Laake WHERE laakenimi like ?";

				Query query = session.createQuery(sQuery);
				query.setString(0, laakenimi);
				return query.list();
			}
		});
		return list;
	}

	@SuppressWarnings("unchecked")
	public List<L_Laake> findByLaakenimiYhtasuuruus(final String laakenimi) {
		HibernateTemplate hibernateTemplate = getHibernateTemplate();

		List <L_Laake> list = hibernateTemplate.executeFind(new HibernateCallback() {
			public Object doInHibernate(Session session) throws HibernateException {

				String sQuery = "FROM L_Laake WHERE laakenimi = ?";

				Query query = session.createQuery(sQuery);
				query.setString(0, laakenimi);
				return query.list();
			}
		});
		return list;
	}

	@SuppressWarnings("unchecked")
	public List<Long> findLakkeIds() {
		HibernateTemplate hibernateTemplate = getHibernateTemplate();

		List <Long> list = hibernateTemplate.executeFind(new HibernateCallback() {
			public Object doInHibernate(Session session) throws HibernateException {

				String sQuery = "SELECT id FROM L_Laake";
				Query query = session.createQuery(sQuery);
				return query.list();
			}
		});
		return list;
	}
}
```

TestRunner-luokka, jolla IndeksiTesti-luokan testejä voidaan ajaa:

```
package fi.fastroi.hilkka2.indeksitesti;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.SimpleTimeZone;

import org.hibernate.SessionFactory;


public class TestRunner {

	protected IndeksiTesti indeksiTesti;

	public static void main(String[] args) {
		TestRunner testRunner = new TestRunner();
		testRunner.test();

	}

	public void test(){

		org.springframework.context.ApplicationContext applicationContext = getAppCtx();

		SessionFactory sessionFactory =


((SessionFactory)applicationContext.getBean(org.springframework.util.StringUtils.uncapitalize("SessionFa
ctory"),SessionFactory.class));

		indeksiTesti = new IndeksiTesti();
		indeksiTesti.setSessionFactory(sessionFactory);

		indeksiTesti.init(applicationContext);

		Date alkuAika = new Date();

		//Näistä valitaan kommentoimalla haluttu testi

		//Kohta 4.1
		indeksiTesti.testPakkausnumero(2000000);

		//Kohdat 4.2 ja 4.3
		indeksiTesti.testPakkausnumeroVali(20000);

		//Kohta 4.4
		indeksiTesti.testKiinteaArvovali(500000,10);
		indeksiTesti.testKiinteaArvovali(500000,100);
		indeksiTesti.testKiinteaArvovali(100000,1000);
```

```
                indeksiTesti.testKiinteaArvovali(10000,10000);

                //Kohta 4.5
                indeksiTesti.testKiinteaArvovali(200,10);
                indeksiTesti.testKiinteaArvovali(200,100);
                indeksiTesti.testKiinteaArvovali(200,1000);
                indeksiTesti.testKiinteaArvovali(200,5000);

                //Kohta 4.6
                indeksiTesti.testPakkausnumeroValiCount(2000000);

                //Kohta 4.7
                indeksiTesti.testLaakenimiLikeP(500000);

                //Kohta 4.8
                indeksiTesti.testLaakenimiLikePP(30000);

                //Kohta 4.9
                indeksiTesti.testLaakenimiLike(2000000);

                //Kohta 4.10
                indeksiTesti.testLaakenimiYhtasuuruus(2000000);

                //Kohta 4.11
                indeksiTesti.testLaakeLisaaminen(2000000);

                //Kohta 4.12
                indeksiTesti.testLaakePoistaminen(500000);

                Date loppuAika = new Date();
                java.text.DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
                dateFormat.setTimeZone(new SimpleTimeZone(0,"London time"));
                Long kulunutAika = alkuAika.getTime()-loppuAika.getTime();

                System.out.println("Kulunut aika: " + dateFormat.format(kulunutAika));

        }

        public org.springframework.context.ApplicationContext getAppCtx() {

                org.springframework.context.support.FileSystemXmlApplicationContext fac = new
org.springframework.context.support.FileSystemXmlApplicationContext("/WEB-INF/applicationContext-
*.xml");
                org.springframework.context.ApplicationContext applicationContext = fac;

                return applicationContext;
        }
}
```