SSD-tietoiset hakemistorakenteet

Peitsa Lähteenmäki

Helsinki 9.3.2012

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Peitsa Lähteenmäki

Työn nimi — Arbetets titel — Title

SSD-tietoiset hakemistorakenteet

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaarityö	9.3.2012	10 sivua

Tiivistelmä — Referat — Abstract

Uudet SSD-muistit tarjoavat huomattavasti tavallisia levymuisteja nopeampaa tiedonkäsittelyä. Tämän takia niiden voidaankin olettaa sopivan hyvin käytettäviksi erilaisten hakemistojen alustana. Tässä työssä tarkastellaan tämän oletuksen paikkansapitävyyttä, ja sitä miten hakemistosta saadaan SSD-tietoinen.

ACM Computing Classification System (CCS):
H.3.1 [Information storage and retirieval]: Content Analysis and Indexing — Indexing methods

Avainsanat — Nyckelord — Keywords
SSD-muisti, hakemistot, R-puu, LCR-puu

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Lähes kaikki tietokannat käyttävät osana toimintaansa jonkinlaista hakemistoa. Yleensä niiden avulla pyritään nopeuttamaan yksittäisten arvojen tai arvojoukkojen hakua tietokannasta. Se millaista hakemistoa kannattaa käyttää riippuu paljolti siitä, millaista dataa tietokannasta pyritään sen avulla hakemaan. Yksittäisistä numeroista koostuvan datan hakemistona voidaan käyttää esimerkiksi B-puuta [Cro79]. Se ei kuitenkaan sovellu sellaisenaan tilanteisiin, joissa hakemiston pitäisi pystyä huomioimaan useita arvoja kerrallaan. Esimerkiksi, jos tietokannan arvoina on pisteiden koordinaatteja, ei B-puulla voida luoda tehokkaasti toimivaa hakemistoa niille. Tälläisia tilanteita varten onkin kehitetty muita hakemistorakenteita, yksi niistä on R-puu [Gut84].

Tietokannan käyttötarkoituksesta riippuen siihen tehdään mahdollisesti paljon luku- ja kirjoitusoperaatioita. Jokainen luku kuluttaa yleensä tietyn määrän aikaa (paljoltikaan riippumatta kannan koosta) käytettäessä hakemistoa; olettaen että hakemistorakenne on hyvin tasapainoinen. Operaatioon kuluvaa aikaa voidaan kuitenkin vähentää huomattavasti sijoittamalla hakemisto SSD-muistille (siis tilanteissa, joissa hakemisto on liian suuri mahtuakseen kokonaisuudessaan keskusmuistiin) [EGK10].

Varsinkin, kun SSD-muistien hinnat ovat laskeneet jo useampia vuosia niiden koon samalla kasvaessa, voidaan niille varastoida jo kohtalaisen suuria datakokonaisuuksia ilman kohtuutonta hinnan kasvua. Flash-pohjaisilla SSD-muisteilla on kuitenkin myös heikkoutensa, jotka tulee huomioida, jotta niiden tehokkuus voidaan hyödyntää täysin. Erityisesti satunnaisille epäyhtenäisille muistialueille kohdistuvat kirjoitusoperaatiot kuluttavat verrattaen paljon aikaa [CKZ09].

SSD-muistien toimintaa voidaan yleisellä tasolla tehostaa käyttämällä niihin kohdistuvien eri operaatioiden yhteydessä erilaisia loki järjestelmiä [WuZ94] [KiL99]. Myös hakemistojen toimintaa voidaan nopeuttaa samalla tavalla: lisäämällä hakemistoon eräänlaisena välimuistina toimiva loki, voidaan sen toimintaa nopeuttaa [LLC11] [WCK03]. Lokin avulla voidaan osa hakemistoon tehtävistä muutoksista siirtää myöhempään ajankohtaan, jolloin kaikki muutokset voidaan tehdä samaan aikaan.

Aloitan aiheen tarkastelun esittelemällä lyhyesti R-puu -hakemistorakenteen. Tämän jälkeen käyn läpi muutamia SSD-muisteihin liittyviä seikkoja, joilla on vaikutusta niille toteutettaviin tietokantarakenteisiin. Kappaleessa 4 esittelen yhden nimenomaan SSD-muisteille suunnitellun hakemistorakenteen, joka pohjautuu R-puuhun, ja siihen liittyviä tuloksia.

2 R-puu

Guttmanin alunperin esittelemä R-puu [Gut84] muistuttaa rakenteeltaan ja toiminnaltaan paljon B-puuta. Molemmissa puiden lehdet sisältävät viitteitä hakemiston osoittamiin arvoihin tietokannassa, ja puun muut solmut toimivat reittinä lehtiin. Suurin ero näiden kahden rakenteen välillä onkin tapa jolla puut jakavat viittaamiaan arvoja solmujensa kesken (ts. miten solmujen alipuut muodostetaan). R-puussa jako perustuu suorakulmioihin joiden sisään kaikkien viitattujen arvojen tulee mahtua.

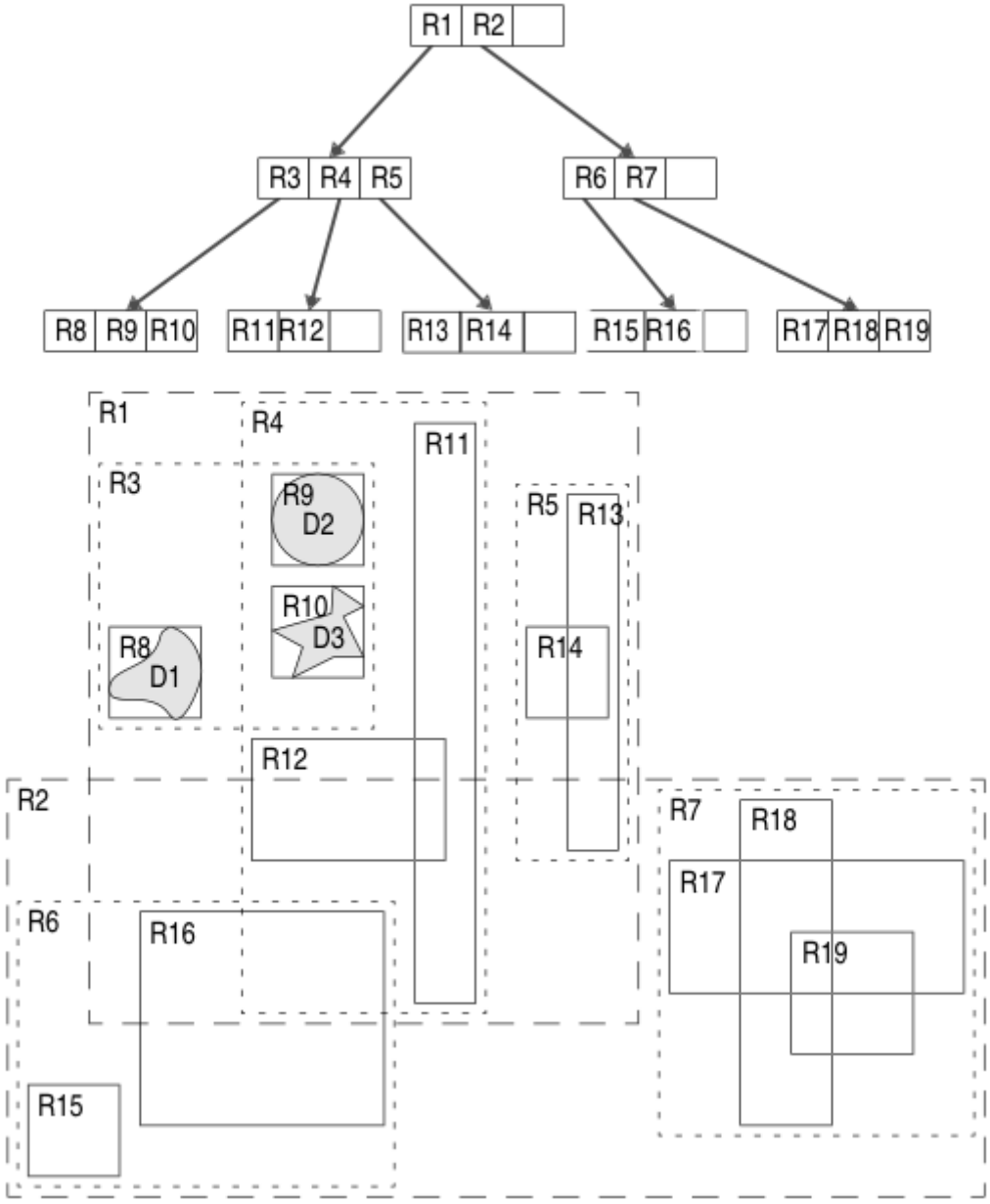
Tarkastellaan tätä tarkemmin esimerkin avulla. Kuvassa 1 on R-puu, jonka juuressa on viitteet kahteen muuhun solmuun. Kuvaa tarkastelemalla havaitaan, että kaikki solmun R1 muodostaman alipuun sisältämät arvot ovat suorakulmion R1 sisäpuolella. Sama pätee myös kaikille muille somuille. Alimman tason solmut (lehdet) sisältävät myös viitattavat arvot (D1, D2 ja D3); tietysti myös muutkin lehdet sisältävät viitattavia arvoja, mutta näitä ei ole vain merkitty kuvioon.

Jokainen puun solmu vastaa siis tiettyä osaa kaikista hakemiston osoittamista arvoista tietokannassa. Jakamalla kohde arvot tällä tavoin mahdollistetaan niiden nopea löytyminen erilaisten hakujen yhteydessä.

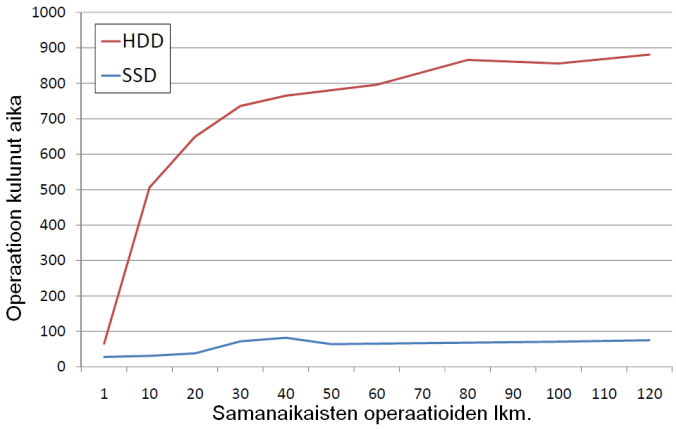
On myös syytä huomata, että samoin kuten B-puussa myös R-puussa voi yhden ainoan arvon lisääminen puuhun aiheuttaa useita muutoksia puun rakenteeseen. Molemmissa rakenteissa lisätty solmu saattaa näet aiheuttaa muutoksia myös solmun vanhempiin, ja nämä muutokset voivat aiheuttaa lisää muutoksia, jne . . . Pahimmassa tapauksessa yksi lisätty solmu voi aiheuttaa muutoksia aina puun juureen asti. Tavallisesti tämä ei muodosta ongelmaa puun käytölle, mutta SSD-muistit muuttavat tilannetta hiukan, palaan tähän vielä kappaleessa 4.

Poiketen B-puusta R-puussa voidaan joutua tutkimaan useita solmun alipuita haetun arvon löytämiseksi. B-puu takaa, että etsitty arvo voi olla vain yhdessä solmun alipuussa, mutta R-puu ei näin tee. Tämä johtuu siitä, että R-puun osoittamia arvoja ei jaeta yksikäsitteisesti sen suorakulmioiden kesken, vaan yksi arvo voi olla viitattuna useammassa lehdessä. Tästä johtuen on mahdollista, että tarvitaan useita useita ”ylimääräisiä” lukuja puuhun, oikean arvon löytämiseksi. Käytännössä näin käy kuitenkin harvoin, koska puun rakennetta ylläpidetään siten, että mahdolliset solmujen päällekkäisyydet saadaan minimoitua.

R-puun käyttäminen hakemistona perustuu erityisesti siihen, että puusta voidaan hakea arvoja suorakulmion avulla. Jos haluamme vaikkapa tietää, mitkä kaikki pisteet ovat tietyn etäisyyden päästä jostakin määrittelemästämme pisteestä, voimme suorittaa haun puuhun näiden tietojen muodostaman suorakulmion avulla ja saada vastauksen nopeasti. Tieten-



Kuva 1: Normaali R-puu [Gut84].



Kuva 2: Samanaikaisten tietokantaoperaatioiden suoritusnopeuksien suhtautuminen operaatioiden määrään [EGK10].

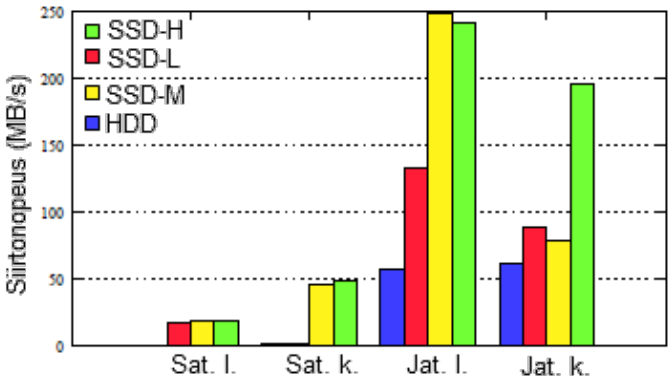
kin myös yksittäisen pisteen haku onnistuu, sillä piste on vain suorakulmio, jonka sivujen pituudet ovat nolla.

Normaalin R-puun toimintaa voidaan tehostaa huomattavasti muuttamalla tapaa, jolla puuhun lisätään solmuja. Parantamalla algoritmia, jolla täyttynyt puun solmu jaetaan kahdeksi uudeksi solmuksi, voidaan puuhun tehtäviä hakuja nopeuttaa. Puuta, jolle on tehty tämä muutos, kutsutaan R*-puuksi [BKS90].

3 SSD-muistien ominaisuudet

SSD-muisteilla tarkoitetaan yleisesti tavallisia kovalevyjä nopeampia muisteja. Erilaisia tekniikoita tälläisten muistien toteuttamiseen on useita, ja muistin ominaisuudet riippuvatkin huomattavasti sen käyttämästä toteutuksesta. Käytännössä SSD-termiä käytetään yleisesti synonyyminä flash-pohjaisille muisteille, ja niin on myös tässä teoksessa. Kaikki maininnat SSD:n toiminnallisuudesta koskevat siis nimenomaan flash-teknologiaan pohjautuvia muisteja, eivätkä esimerkiksi RAM-piirejä käyttäviä muisteja (jotka toimivatkin aivan eri tavalla).

Lähes kaikki flash-pohjaiset SSD-muistit ovat huomattavasti tavallisia levymuisteja nopeampia suorittamaan lukuoperaatioita muistista. Myös kirjoitusnopeus on suurempi, mutta ei yhtä paljon kuin lukunopeus. Erityisesti tilanteissa, joissa tietokantaan kohdistuu useita samanaikaisia operaatioita havaitaan suuria eroja levy- ja SSD-muistien suorituskyvyssä (ks. kuva 2) [EGK10]. Tämä vaikutus korostuu varsinkin hakemistojen osalta. Koska hakemistoa käytetään nimenomaan arvojen löytämiseen, luetaan sitä lähes joka kerralla, kun kannasta haetaan tietoja. Muuttamalla siis käytettyä hakemistoa siten, että se osaa hyödyntää paremmin SSD-muistien ominaisuuksia, voidaan olettaa sen suorituskyvyn myös paranevan.



Kuva 3: Eroja eri SSD-muistien ja levymuistin välillä. Mitattuina satunnaiset ja jatkuvat kirjoitus- ja lukunopeudet [CKZ09].

Luku- ja kirjoitusnopeuksien ero johtuu siitä, että SSD-muisti ei voi suoraan ylikirjoittaa vanhaa dataa, vaan muisti täytyy ensin tyhjentää, ja vasta sen jälkeen sille voidaan kirjoittaa. Näin ollen lähes kaikkien SSD-muistien kirjoitusnopeus on paljon lukunopeutta pienempi. Tämä näkyy erityisesti tilanteissa, joissa muistiin tehdään satunnaisia muutoksia. Kirjoitettaessa pitkiä yhtenäisiä muistialueita ei ero ole yhtä suuri. Chen ja kumppanit [CKZ09] testasivat muutamien erilaisten SSD-muistien ja yhden levymuistiverrokin nopeuksia satunnaisilla ja samalle alueelle kohdistuvilla kirjoituksilla ja luvuilla (ks. kuva 3) saaden juuri tämän suuntaisia tuloksia.

Luku- ja kirjoitusnopeuksien eroa voidaan kuitenkin pienentää. Pitämällä yllä lokia muistiin tehdyistä muutoksista voidaan muistiin tehtäviä muutoksia siirtää myöhemmäksi, jolloin ne voidaan kirjoittaa yhtä aikaa ja näin nopeuttaa tallennusta. Tätä menetelmää voidaan hyödyntää minkä tahansa sovellusalueen yhteydessä — myös tietokantojen [KiL99] [WuZ94].

Eri lokijärjestelmien avulla voidaan myös vähentää SSD-muisiin tallennetun tiedon pirstoutumista. Tämä puolestaan nopeuttaa muistin toimintaa [CKZ09]. Erityisesti lukunopeudet pienenevät muistin pirstoutuessa, syynä tähän on muistiin tallennettujen tietojen todennäköinen hajautuminen eri paikkoihin, jolloin niiden lukemiseen tarvitaan useampia operaatioita.

Suurten lukunopeuksiensa ansiosta SSD-muistit soveltuvat hyvin käytettäviksi tietokantojen kanssa. Pelkän SSD-muistin käyttö, ilman mitään tietokannan SSD-spesifejä optimointejakin, nopeuttaa tavallisten SQL kyselyiden suorittamista [LMP08]. Suunnittelemalla käytetyt tietokantajärjestelmät siten, että ne tukevat SSD-muistien ominaisuuksia voidaan järjestelmän toimintaa nopeuttaa. Käytännössä tämä voidaan tehdä samoin kuten edellä, eli kokoamalla useita yksittäisiä muutoksia suuremmiksi kokonaisuuksiksi.

4 LCR-puu

Tavallista R-puuta voidaan käyttää suoraan sellaisenaan myös SSD-muisteissa. Tämä ei ole kuitenkaan paras mahdollinen järjestely, sillä R-puun päivittäminen (arvon lisääminen tai poistaminen) saattaa aiheuttaa useita muutoksia myös R-puun rakenteeseen muistissa. Tavallisella levymuistilla tämä seikka ei ole huomion arvoinen, koska levymuistien kirjoitus- ja lukunopeudet ovat jokseenkin samaa luokkaa. SSD-muisteilla näin ei kuitenkaan ole, vaan lukunopeudet ovat yleensä paljon kirjoitusnopeuksia suurempia, jolloin ”turhia” kirjoituksia tulisi välttää. R-puusta voidaan kuitenkin muokata jokseenkin helposti nk. LCR-puu, joka pyrkii välttämään kirjoituksia muistiin lisäten kuitenkin samalla tarvittavien lukujen määrää.

Lvin ja kumppaneiden alunperin suunnittelema LCR-puu [LLC11] ei tee mitään suoria muutoksia alkuperäiseen R-puuhun, vaan lisää siihen osan, jonka avulla R-puuhun tehtävät muutokset eivät välttämättä aiheuta suoraan satunnaisia kirjoitusoperaatiota muistiin. Tällä tavalla saadaan yhdistettyä R-puun toiminnallisuus ja SSD-muistien nopeus. LCR-puu ei suoraan vähennä kirjoitusoperaatioita, vaan sen sijaan muuttaa ne satunnaisiin kohtiin muistissa osuvista kirjoituksista (jollaisia syntyisi käytettäessä tavallista R-puuta) pidemmäksi yhtenäiselle muistialueelle kohdistuvaksi operaatioksi.

Käytännössä tämä toteutetaan lisäämällä jokaiseen R-puun solmuun loki, joka pitää yllä tietoa kyseiseen solmuun kohdistuneista operaatioista. Tietenkin haettaessa R-puusta solmua tätyy myös ottaa huomioon solmuun liittyvät lokitiedot. Tämä kasvattaa solmun arvon selvittämiseen tarvittavia lukuoperaatioita, mutta pitämällä yllä lokin lisäksi listaa jokaisen solmun ensimmäisestä lokikirjauksesta keskusmuistissa, tarvitaan ylimääräisiä lukuoperaatioita vain muutamia.

Samaan solmuun kohdistuneet eri muutokset kootaan lisäksi yhteen samalle muistialueelle jonona, jonka pituus vastaa käytetyn muistin sivukokoa. Tarvittaessa edelliset lokikirjaukset kopioidaan sellaisenaan uusien lokikirjauksien yhteyteen, jotta kirjausten peräkkäisyys voidaan varmistaa. Tällöin yhden solmun tilan selvittämiseen tarvitaan mahdollisimman vähän lukuja muistista (ts. yhdellä luvulla levyltä voidaan selvittää useita loki merkintöjä). Tästä menettelystä on hyötyä myös lokitietojen kirjoituksessa. Kun lokitietojonojen pituudet vastaavat käytettävän muistin sivukokoa, voidaan koko lokijono kirjoittaa kerralla talteen.

Lokin kasvaessa tarpeeksi suureksi tarvitaan yhden solmun arvon selvittämiseen suuria määriä lukuja, koska jokainen solmua koskeva lokirjaus on tarkastettava sitä varten. Tässä tilanteessa lokissa olevat muutokset on syytä kirjoittaa alkuperäiseen R-puuhun haun nopeuttamiseksi. Samalla kirjaukset tietysti myös poistetaan lokista. Tämä vie runsaasti

aikaa, mutta on harvinainen operaatio, joten se ei aiheuta huomattavaa tehokkuus häviötä. Käytännössä tämä toteutetaan määrittämällä lokille jokin tietty koko raja, jota suuremmaksi se ei saa kasvaa. Testit ovat osoittaneet, että hyvä arvon lokin koolle on noin 20% puun koosta.

Tarkastellaan seuraavaksi varsinaisen lokin muodostusta LCR-puussa. Kaikki lokin tiedot kootaan yhdelle muistialueelle, jonka leveys vastaa käytetyn SSD-muistin sivukokoa, ja jonka korkeus skaalataan vastaamaan lokille määritettyä maksimikokoa. Lokille varatun muistialueen rivit sisältävät edellä kuvatulla tavalla asemoituja solmujen lokitietoja. Jokainen rivi on siis jaettu osiin sen mukaan kuinka monta yksittäistä lokikirjausta sille mahtuu (joka on siis vakio, koska muistin sivukoko ei muutu). Muistialueen sarakkeilla ei ole käytännön merkitystä.

Kuva 4 esittää tilanteen, jossa LCR-puuhun tehdään neljä muutosta, jotka kohdistuvat solmuun R9. Ensimmäinen taulukko näyttää tilanteen lokissa ennen muutoksia ja jälkimmäinen tilanteen niiden jälkeen. Punaiset solut ovat vanhentuneita lokirjauksia, eli kirjauksia joiden jälkeen samaan solmuun on tehty lisää muutoksia. Vihreät solut sisältävät tallennettuja voimassa olevia lokikirjauksia. Valkeat solmut ovat tyhjiä kohtia lokissa, joihin uusia merkintöjä voidaan tehdä.

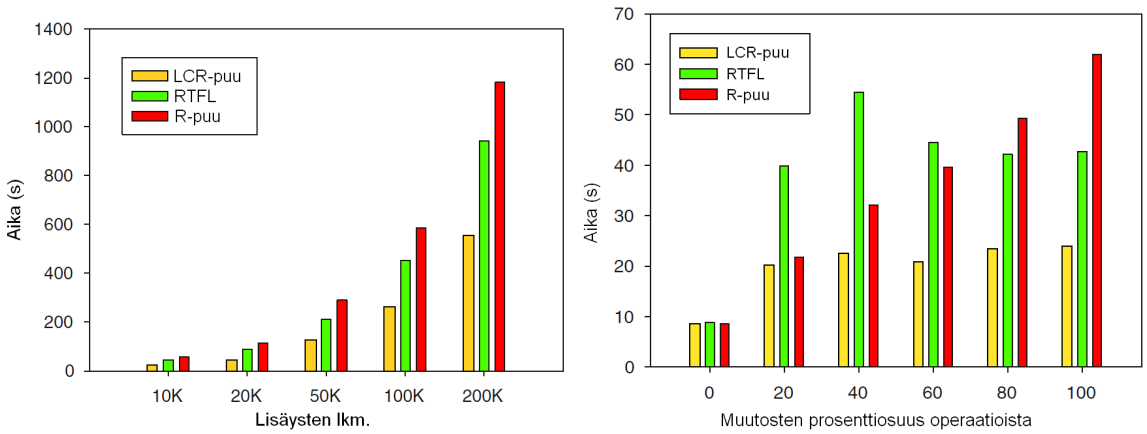
R1	R1	R2	R2	R1	R1	R1	R1
R2	R2	R2	R3	R3	R3	R3	R9
R8	R8	R8	R8	R8			

⇓

R1	R1	R2	R2	R1	R1	R1	R1
R2	R2	R2	R3	R3	R3	R3	R9
R8	R8	R8	R8	R8			
R9	R9	R9	R9				

Kuva 4: Esimerkki LCR-puun lokista.

Muutokset solmuun R9 eivät siis aiheuta muita muutoksia puun muihin solmuihin. Ainoa lokissa oleva edellinen merkintä koskien solmua R9 (rivillä 2) on myös siirretty uuteen paikkaansa muiden saman solmun lokikirjausten yhteyteen (riville 4). Lokikirjaukset on myös siirretty kokonaan tyhjälle riville siksi, että ne mahtuisivat samalla sivulle. Edellisellä sivulla on tilaa vain kolmelle kirjaukselle, joten sinne kirjoitettaessa vuotaisi yksi lokikirjaus yli seuraavalle riville, mikä tarkoittaisi, että solmun R9 lokikirjausten selvittä-



(a) LCR-puun lisäysnopeus verrokkien kanssa

(b) LCR-puun muutosnopeus verrokkien kanssa

Kuva 5: LCR-puun suhtautuminen verrokkeihin [LLC11]

miseen tarvittaisiin kaksi lukua.

Kaiken kaikkiaan nämä lisäykset R-puuhun nopeuttavat sen toimintaa huomattavasti. Verrattaessa tavallista R-puuta, LCR-puuta ja erästä toista lokin ylläpitoon pohjautuvaa R-puu varianttia (RTFL [WCK03]) havaitaan LCR-puun olevan niitä nopeampi. Kuva 5 näyttää näiden nopeus eroja solmujen lisäyksen ja muutosten suhteen. Tuloksista havaitaan LCR-puun skaalatutuvan verrokkeja paremmin lisäysnopeuden osalta. Myös muutosten osuuden lisääminen ei hidasta puun toimintaa juuri ollenkaan, toisin kuin verrokkien tapauksessa.

5 Yhteenveto

Olen tarkastellut tässä työssä sitä, millaisia mahdollisuuksia uudet SSD-muistit tarjoavat tietokantojen hakemistojärjestelmille. Niiden tarjoamat edut tavallisiin levymuisteihin verrattuna ovat huomattavat, ja tästä johtuen ne soveltuvatkin hyvin hakemistoille. Erityisesti SSD-muistien tarjoama mahdollisuus suorittaa nopeasti useita satunnaisia lukuoperaatioita mahdollistaa hakemiston tehokkaan käytön.

Vaikka suurinta osaa hakemistojen toiminnallisuudesta ei tarvitsekaan muuttaa millään tavoin, jotta ne voisivat hyötyä SSD-muistien eduista, niin suunnittelemalla hakemistot erityisesti niille sopiviksi voidaan saavuutta parempia tuloksia. Voidaankin siis todeta, että hakemiston suunnittelu erityisesti SSD-muistia varten on järkevää, kuten esimerkiksi LCR-puu osoittaa.

Toisaalta SSD-muistien verrattaen hidas kirjoitusnopeus saattaa muodostua ongelmaksi. Tähänkin voidaan kuitenkin varautua hakemiston suunnittelun yhteydessä siten, ettei ha-

kemistoon tehtävät kirjoituksia vaativat muutokset aiheuta huomattavia pullonkauloja hakemiston toimintaan.
Kaiken kaikkiaan SSD-muistit tarjoavat paljon mahdollisuuksia hakemistojen tehokkuuden parantamiseen. Tästä hyötyminen ei myöskään ole kovin monimutkaista hakemistojen toiminnan kannalta ja onkin siksi kohtalaisen helposti toteutettavissa.

Lähteet

AGS09 D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao ja S. Singh. Lazy-Adaptive Tree: An optimized index structure for flash devices. *Proceedings of the VLDB*, 2009.

BKS90 N. Beckmann, H.-P. Kriegel, R. Schneider ja B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. *In Proceedings of the SIGMOD*, 1990.

Cro79 D. Cromer. The Ubiquitous B-Tree. *ACM Computing Surveys* 11,2(1979).

CKZ09 F. Chen, D. A. Koufaty ja X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, 2009.

EGK10 T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert ja M. Thoma. On the impact of flash SSDs on spatial indexing. *DaMoN*, 2010.

Gut84 A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984.

KiL99 H.-J. Kim ja S.-G. Lee. A new flash memory management for flash storage system. *Proceedings of the Computer Software and Applications Conference*. 1999

LLC11 Y. Lv, J. Li, B. Cui ja X. Chen. Log-Compact R-Tree: An Efficient Spatial Index for SSD. *DASFAA Workshops*, 2011.

LMP08 S.-W Lee, B. Moon, C. Park, J.-M Kim ja S.-W Kim. A Case for Flash Memory SSD in Enterprise database applications. *Proceedings of the 2008 ACM SIGMOD*, 2008.

WCK03 C.-H. Wu, L.-P. Chang ja T.-W. Kuo. An efficient R-Tree implementation over flash-memory storage systems. *ACM GIS*, 2003.

WuZ94 M. Wu ja W. Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, 1994.