TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

PETTERI KOSKINEN

LAISKA EVALUOINTI

Kandidaatintyö

Tarkastaja: Tiina Schafeitel-Tähtinen
Jätetty tarkastettavaksi 30.06.2018

TIIVISTELMÄ

PETTERI KOSKINEN: Laiska evaluointi
Tampereen teknillinen yliopisto
Kandidaatintyö, 22 sivua, 0 liitesivua
Kesäkuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Tiina Schafeitel-Tähtinen

Avainsanat: Laiska evaluointi, Tietokanta, Ohjelmointikieli, Haskell

Tämän työn päätavoitteena on selvittää, miten laiska evaluointi toimii eri sovelluskohteissa sekä mitä hyötyjä ja haittoja sen käytöstä aiheutuu. Tässä työssä keskitytään kahteen sovelluskohteeseen: laiskuus ohjelmointikielessä sekä laiskuus tietokannoissa.

Aineistona tässä tutkimuksessa on aikaisemmat tutkimukset sovelluskohteisiin sekä yleinen kirjallisuus aiheeseen liittyen. Tärkeimpinä lähteinä tässä työssä ovat Faleiro et al. *Lazy evaluation of transactions in database systems* -tutkimus laiskoista tietokannoista sekä Heinrich Apfelmuksen *The Incomplete Guide to Lazy Evaluation (in Haskell)* -artikkeli laiskuuden toiminnasta Haskellissa.

Tuloksena tästä työstä saadaan, että laiska evaluointi sopii joihinkin sovelluskohteisiin hyvin ja toisiin huonosti. Jopa saman sovelluskohteen sisällä löytyy tilanteita, joissa laiska evaluointi ja innokas evaluointi ovat toinen toistaan parempia. Ehkäpä ainoa kohde jossa laiska evaluointi on aina parempi, on loputtomat sarjat. Näiden suoritus ei lopu koskaan innokkaassa evaluoinnissa, joten laiska evaluointi on ainoa vaihtoehto niiden käsittelyyn.

SISÄLLYSLUETTELO

1. JOHDANTO

Nykyaikana tarvitaan jatkuvasti nopeampia ja tehokkaampia tietoteknisiä laitteita. Laitteiden liittäminen internettiin nostaa palveluiden käyttäjämääriä ja tiedonsiirtonopeuden tarvetta. Yksi tapa parantaa suorituskykyä on jättää operaatiot suorittamatta kokonaan. Ongelmana tässä on tietenkin, ettei mitään saavuteta, jos mitään ei tehdä. Täytyy siis löytää operaatioista ne, joiden poisjättämisestä ei ole vaikutusta palvelun suoritukseen. Tämä saavutetaan, kun operaatiot suoritetaan vasta, kun niiden tuloksia todellisuudessa tarvitaan johonkin eli operaatiot suoritetaan käyttäen laiskaa evaluointia.

Tämä tutkimus antaa yleiskuvan laiskan evaluoinnin toiminnasta, sekä sen toteutuksesta ja teoriasta muutamassa käytännön sovelluksessa. Sovelluskohteiden yhteydessä on kerrottu mitä hyötyjä ja haittoja saavutetaan laiskan evaluoinnin käyttämisestä kyseisessä sovelluskohteessa, sekä miten nämä eroavat laiskan evaluoinnin yleisistä hyödyistä ja haitoista.

Laiska evaluointi on laskentapa, jossa laskuja ei lasketa heti, kun ne tulevat vastaan, vaan vasta kun niiden tuloksia tarvitaan johonkin. Laiskan evaluoinnin päätarkoitus on siirtää laskuja laskettavaksi myöhemmäksi. Jos laskujen tuloksia ei koskaan käytetä, ei myöskään laskuja tarvitse koskaan laskea ja näin vältytään ylimääräiseltä työltä. Haittapuolena laiskassa evaluoinnissa on, että jos tuloksia tarvitaan, ei niitä ole valmiiksi laskettuna, vaan ne täytyy laskea.

Tässä työssä tutkitaan, kuinka laiskuutta voidaan hyödyntää tietoteknisissä käytännön sovelluksissa. Tarkoituksena on vertailla mitä etuja ja haittoja saavutetaan laiskuuden hyödyntämisestä, perinteisen innokkaan sijasta. Tämä teksti esittelee vain muutamia sovelluskohteita, painottuen lähinnä tietoteknisiin sovelluksiin. Näissä tietoteknisissä sovelluksissa olevat vertailut pätevät ainakin osittain myös muihinkin kohteisiin. Ohjelmointikielet eivät ole laiskuuden ainoa sovelluskohde, vaikka niitä tässä työssä pääasiassa käsitellään. Esimerkiksi matematiikassa laiskuutta hyödynnetään lambdakalkyylissa (lambda calculus). Tekstin sisältö perustuu aikaisemmin tehtyihin sovelluksiin ja tulosten vertailu suurelta osin alkuperäisten tutkijoiden tekemiin havaintoihin.

Luvussa 2 esitellään laiskan evaluoinnin yleistä teoriaa, sekä mitä hyötyjä ja haittoja laiskan evaluoinnin käytöstä aiheutuu. Tämän jälkeen luvussa 3 kerrotaan laiskan evaluoinnin sovelluskohteista esimerkkien kautta. Luvussa 3 pyritään tuomaan esiin laiskan evaluoinnin teoriaa eri sovelluskohteissa ja miten niissä on ratkaistu mahdollisia ongelmia. Luvussa 4 vertaillaan sovelluskohteittain laiskan evaluoinnin tehokkuutta innokkaaseen. Viimeisenä lukuna on johtopäätökset, jossa esitellään lyhyesti tärkeimmät löydökset vertailuista sekä teoriasta.

2. LAISKUUDEN TEORIA

2.1 Laiska evaluointi yleisesti

Tässä luvussa esitellään laiskuuteen liittyviä erityispiirteitä yleisellä tasolla. Monet tässä luvussa mainitut perusteet liittyvät läheisesti tietotekniikkaan ja erityisesti ohjelmointikieliin, mutta myös yleisempiä laiskuuden piirteitä voidaan niistä löytää.

Innokas evaluointi (eager evaluation) on evaluointitapa (evaluation strategy), jossa muuttujien arvot lasketaan välittömästi, kun ne tulevat ohjelman suorituksessa vastaan. Innokasta evaluointia käyttävissä ohjelmointikielissä muuttujien laskujärjestys on tiedossa jo ohjelmaa kirjoitettaessa. Tämä mahdollistaa suoritusjärjestyksen optimoinnin ohjelman kirjoittajalle. Innokas evaluointi on yleisin tapa laskea muuttujien arvot perinteisissä ohjelmointikielissä. Innokas evaluointi on helppo evaluointitapa toteuttaa, koska se ei vaadi ylimääräistä kirjanpitoa muuttujille, vaan käskyt voidaan kääntää suoraan suoritettaviksi konekäskyiksi.

Laiska evaluointi (lazy evaluation) on erityisesti funktionaalissa ohjelmointikielissä käytetty evaluointitapa laskea muuttujien arvot [1]. Laiskaa suoritusta voidaan kutsua myös tarvepohjaiseksi suoritukseksi (call-by-need), joka kuvaa hyvin tätä evaluointitapaa [2]. Muuttujien arvot lasketaan vasta, kun niiden arvoja tarvitaan [3]. Tästä aiheutuu kuitenkin ylimääräistä kirjanpitoa ja laskentaa päätellä, mihin muuttujiin suoritettavat laskut vaikuttavat [1].

2.2 Laiskuuden yleiset hyödyt

Tässä luvussa käydään läpi laiskuuden yleisiä hyötyjä. Monissa lähteissä mainitaan listattuja hyötyjä, mutta varsinaista selitystä niiden todellisesta toiminnasta on hankala löytää. Kaikki tässä luvussa esitellyt hyödyt pätevät tietoteknisiin ratkaisuihin, mutta myös matematiikassa voidaan nähdä osa hyödyistä.

Ylimääräisen työn välttäminen [4]**.** Ylimääräistä työtä vältetään tapauksessa, jossa kirjoitettavaa arvoa ei koskaan käytetä. Esimerkiksi suuria taulukoita käsiteltäessä tarvitaan vain mahdollisesti muutama yksittäinen arvo ja loput arvoista voidaan jättää kokonaan käsittelemättä. Näitä ylimääräisiä arvoja ei tarvitse edes luoda muistiin missään vaiheessa, jos niiden arvoja ei koskaan käytetä.

Pienemmät kirjoituslatenssit [5]**.** Muuttujiin merkitään mitä operaatioita täytyy suorittaa ennen kuin haluttua muuttujaa voidaan lukea. Ohjelma siis vain palauttaa lupauksen, että muuttujien arvot ovat samat, kuin jos käskyt olisi suoritettu välittömästi. Esimerkiksi

halutaan luoda 1000 satunnaislukua, mutta todellisuudessa ohjelma ei luo näitä 1000 lukua heti. Ohjelma vain palauttaa tiedon, että nämä 1000 lukua ovat saatavilla.

Loputtomien sarjojen käsittelyn mahdollistaminen. Innokkaassa ohjelmointikielessä loputtoman sarjan laskeminen ei päättyisi koskaan, vaan ohjelma vain jatkaisi uusien lukujen laskemista ikuisesti. Laiskassa evaluoinnissa annetaan vain tieto, että kaikki sarjan luvut on mahdollista saada ja ne lasketaan vasta pyydettäessä. [6]

Muistin käytön optimointi [5]**.** Laiskassa ohjelmassa ei tarvitse luoda muuttujia ennen, kuin niitä todellisuudessa käytetään. Muuttujat eivät siis vie muistissa tilaa turhaan, vaan ne luodaan vasta kun niitä oikeasti käytetään johonkin.

Tehokkuus. Jos samaan muuttujaan kohdistuu useita kirjoitusoperaatioita ennen kuin se luetaan, voidaan operaatiot suorittaa hakemalla tarvittavat muuttujat hitaammasta muistista vain kerran. Näin vältytään saman muuttujan siirtelemiseltä järjestelmän eri muistien välillä. Yleisesti ohjelmien suorituksessa kuluu huomattava määrä aikaa tietojen hakemiseen hitaammista muisteista, joten näiden hitaiden muistihakujen vähentäminen parantaa ohjelman suoritustehoa (throughput) [7].

2.3 Laiskuuden yleiset haitat

Hyötyjen lisäksi laiskuus tuo mukanaan myös tiettyjä haittoja. Tässä luvussa esitellään lyhyesti näitä yleisiä laiskuuden haittoja. Myös laiskuuden haitoista on vaikea löytää yksityiskohtaista selitystä, kuinka ne vaikuttavat ohjelman suoritukseen. Lähteet mainitsevat usein näitä haittoja ohimennen, mutta todellista selitystä on hankala löytää.

Korkeammat lukulatenssit [5]. Luettaessa muuttujaa ohjelman täytyy tarkastaa, mitä operaatioita täytyy suorittaa ennen, kuin muuttujaa voidaan lukea. Jos muuttujassa on suorittamattomia operaatioita, nousee lukemiseen kuluva aika. Tämä ei kuitenkaan tarkoita, että operaatiot suoritetaan jokaiselle muuttujalle aina uudestaan, vaan tulos tallennetaan ja niihin liittyvät operaatiot voidaan poistaa suoritettavien operaatioiden listasta. Nämä suoritettavat operaatiot saattavat sivutuotteena laskea myös muita kuin luettavien muuttujien arvoja ja myös ne tallennetaan suoritetuiksi.

Arvaamattomat suoritusajat [5]**.** Innokkaassa ohjelmointikielissä muuttujien lukeminen on vakioaikaista. Muuttujien arvot on tallennettu muistiin, josta ne vain täytyy hakea. Laiskassa ohjelmoinnissa voidaan joutua suorittamaan useita operaatioita ennen kuin haluttua muuttujaa voidaan lukea. Näistä suoritettavista operaatioista voi muodostua hyvinkin pitkä ketju, joka täytyy suorittaa ennen muuttujan lukemista.

Ylimääräinen työ merkitsemisestä. Osa operaatiosta voidaan joutua suorittamaan ainakin osittain, että saadaan pääteltyä mihin muuttujiin operaatio vaikuttaa. Tämä merkitse-

minen kuitenkin kestää usein lyhyemmän aikaa kuin koko operaation suoritus. Jos operaatiossa on määritelty suoraan kirjoitettavat muuttujat, ei ylimääräistä työtä tietenkään tarvitse tehdä.

Suurempi muistinkulutus. Koska operaatioita ei suorita heti kun ne ohjelmassa tulevat vastaan, täytyy muuttujat ja niihin liittyvät operaatiot tallentaa muistiin. Tämä kuluttaa tietenkin enemmän muistia kuin pelkkien muuttujien arvojen muistaminen.

Virheiden paikallistaminen. Koska operaatioita ei suoriteta heti, voivat mahdolliset virheet näkyä vasta pitkän ajan päästä. On myös mahdollista, että jotkin virheet eivät tule koskaan esiin, vaan virheen sisältävä osuus ylikirjoitetaan tai sitä ei koskaan edes luoda.

Näiden haittojen lisäksi laiskassa evaluoinnissa täytyy ratkaista laiskuuden viivästetystä laskennasta johtuva ongelma. Miten varmistetaan, että muuttujien arvot ovat samat kuin laskujen alkuperäisellä suoritushetkellä? Jos muuttujien arvoja muutetaan laskun syötön ja varsinaisen laskennan välissä, saadut tulokset eivät ole samat kuin innokkaalla laskennalla suoritettujen laskujen tulokset. Tämä on ongelma, joka täytyy aina laiskuutta hyödyntävässä sovelluskohteessa ratkaista.

3. LAISKUUDEN SOVELLUKSIA

3.1 Laiska ohjelmointikieli

Tässä luvussa esitellään laiskojen ohjelmointikielien toimintaa Haskellin kautta. Tämän luvun esitellyt teoriat pätevät yleisesti myös moniin muihinkin laiskoihin ohjelmointikieliin, mutta erityisesti funktionaalisten laiskojen kielten toiminta on tämän luvun kannalta olennaista.

Tämä luku sisältää vain tärkeimmät perusteet laiskuuden toiminnasta, eikä yksityiskohtaiseen toteutukseen mennä kovin syvällisesti. Yleisessä teoriaosuudessa esitellyt hyödyt ja haitat pätevät suoraan laiskoihin ohjelmointikieliin, joten niitä ei tässä luvussa enää toisteta uudestaan.

3.1.1 Laiskuuden teoria ohjelmointikielessä

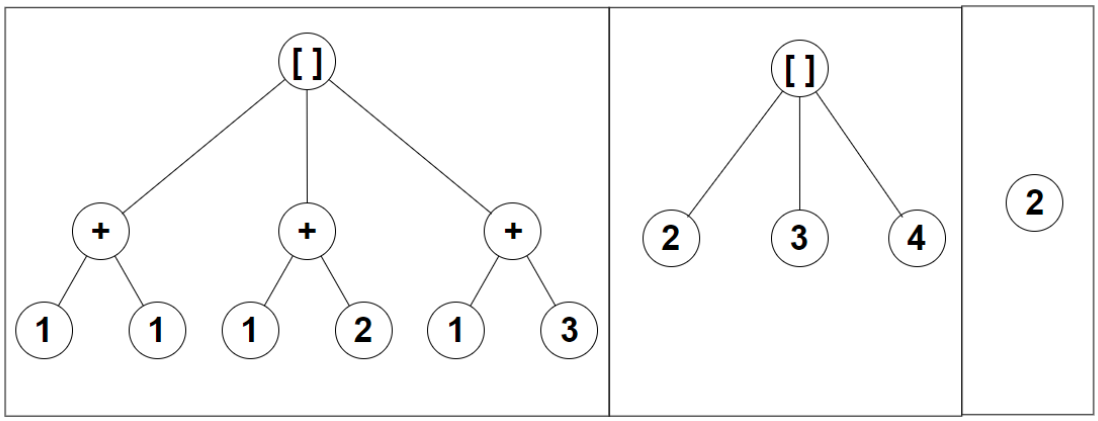
Tässä luvussa käsitellään laiskuuden sisäistä toimintaa Haskell-ohjelmointikielessä. Tämän luvun sisältö perustuu suurelta osin Heinrich Apfelmuksen vuonna 2015 julkaisemaan artikkeliin *The Incomplete Guide to Lazy Evaluation (in Haskell)* [8], sekä Haskellin wikibookkiin [9].

Keskeinen käsite laiskassa laskennassa on *reducible expression*, eli lyhyesti redex. Redexit ovat lausekkeita, joita voidaan sieventää ja sieventämällä niistä saadaan uusia lausekkeita. Jos lauseketta ei voi enää sieventää, sanotaan lausekkeen olevan normaalimuodossa. Esimerkiksi lauseke 1+2 voidaan sieventää lisäämällä luvut yhteen ja tulokseksi saadaan 3. Tämä saatu tulos on normaalimuodossa, koska sitä ei voi enää sieventää eteenpäin. [8]

Lausekkeita voidaan laiskassa laskennassa sieventää kahdella eri tapaa: *innermost reduction* ja *outermost reduction.* Innermost reductionissa aloitetaan sievennys sisältä ja edetään ulospäin. Esimerkkinä innermost reductionista otetaan take-funktio, joka ottaa halutun määrän alkioita taulukon alusta:

```
take 1 [(1+1), (1+2), (1+3)]
⇨  take 1[2, 3, 4]
⇨  2
```

Taulukossa on 3 alkiota, joista halutaan vain ensimmäinen. Koska käytössä on innermost reduction, aloitetaan laskeminen sisältäpäin, eli sievennetään kaikki taulukon alkioiden laskutoimitukset. Kun kaikki laskutoimitukset on sievennetty, voidaan suorittaa take-funktio, joka ottaa tässä esimerkissä vain ensimmäisen alkion taulukosta, eli luvun 2. Kuvassa 1 on esitetty saman esimerkin toiminta graafisesti. [8]



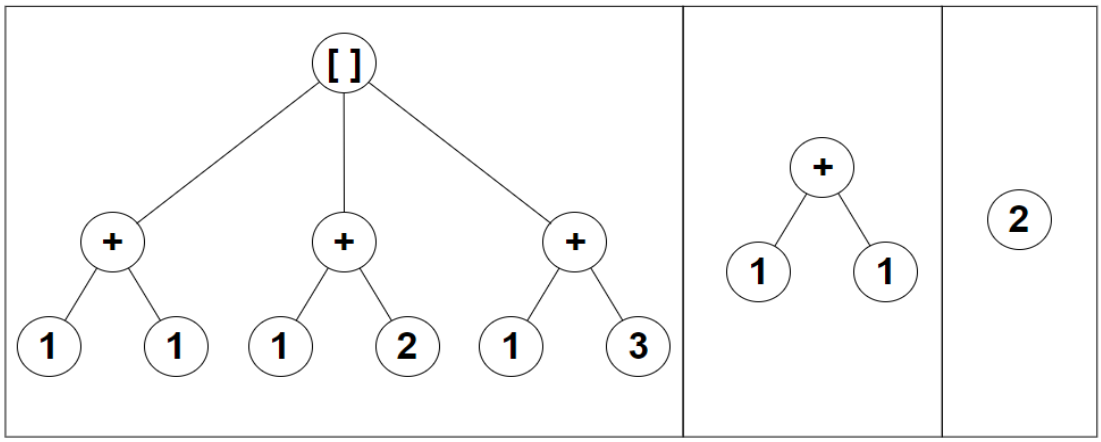
Kuva 1. *Innermost reduction toiminta.*

Vasemman puoleisessa kuvassa on esimerkin alkutilanne. Innermost reduction sieventää kaikki graafin alhaisimmat tasot, ennen kuin se siirtyy ylemmälle tasolle. Keskimmäisessä kuvassa laskutoimitukset on suoritettu ja tuloksena on saatu taulukko, jossa kaikki alkiot ovat normaalimuodossa. Viimeisessä kuvassa take 1 on suoritettu ja tuloksena saatiin sama 2 kuin aikaisemmassakin esimerkissä. Tässä esimerkissä voidaan jo huomata, että innermost reduction tekee ylimääräistä työtä laskemalla taulukosta arvoja, joita ei tarvita tässä esimerkissä lainkaan. Innokkaassa ohjelmoinnissa tämä tilanne ei tule vastaan, koska taulukon arvot on sievennetty jo niiden syöttövaiheessa, eli lähtötilanteena olisi kuvan 1 keskimmäinen graafi. [8]

Ylimääräisen työn välttämiseksi laiskassa ohjelmoinnissa käytetään usein *outermost reductionia*. Outermost reductionissa laskenta aloitetaan ylhäältä ja laskeudutaan alaspäin. Graafissa laskeudutaan alaspäin vain sen verran, että haluttu tulos saadaan laskettua. Seuraavana esimerkkinä sama take-funktion toiminta käyttäen outermost reductionia:

```
take 1 [(1+1), (1+2), (1+3)]
⇨  1+1
⇨  2
```

Outermost reductionissa suoritetaan ensiksi uloin mahdollinen operaation, joka tässä esimerkissä on take 1. Take 1 ottaa taulukon ensimmäisen arvon ja ottaa sen ulos taulukosta. Koska muille taulukon arvoille ei tarvitse tehdä mitään, ei niitä tarvitse myöskään laskea. Lopuksi sievennetään laskutoimitus, josta tulokseksi saadaan edelleen sama luku. Kuvassa 2 on esitetty esimerkin toiminta käyttäen outermost reductionia. [8]



Kuva 2. *Outermost reduction toiminta.*

Ensimmäisessä kuvassa on sama alkutilanne kuin kuvassa 1. Outermost reductionissa suoritetaan ensiksi take 1 -funktio, joka suorittamalla saadaan keskimmäisen kuvan graafi. Kun tämä graafi sievennetään edelleen, saadaan tulokseksi yhä sama luku. Näistä esimerkeistä voidaan nähdä, että outermost reductionissa tehdään vähemmän työtä, jonka takia outermost reductionia usein käytetäänkin laiskassa ohjelmoinnissa. Aina outermost reduction ei kuitenkaan säästä työtä. Esimerkiksi edellisestä take 1 -esimerkistä voidaan esittää versio, jossa taulukon arvot ovat normaalimuodossa, eli niitä ei tarvitse enää sieventää. Lähtötilanteena tässä esimerkissä olisi siis kuvan 1 keskimmäinen kuva. Molemmat sievennystavat tekevät tälläisessä tilanteessa yhtä monta askelta. [8]

Haskellissa, kuten kaikissa muissakin laiskaa laskentaa hyödyntävissä ohjelmointikielissä, täytyy ratkaista laiskasta laskennasta johtuva viivästetyn suorituksen ongelma. Muuttujien arvot täytyvät olla samat kuin operaatioiden syöttöhetkellä, jotta saatu laskennan tulos on oikea. Haskellissa tämä ongelma on ratkaistu tekemällä muuttujista muuttumattomia (immutable). Jokainen muuttuja voidaan määritellä ohjelmassa vain kerran, eikä niitä voida enää myöhemmin muuttaa. Tämä ei kuitenkaan ole ongelma, koska muuttujia voidaan luoda ajon aikaisesti aina lisää. [9]

3.1.2 Laiskuuden optimoinnit ohjelmointikielessä

Osittainen laiskuus. Laiskasta koodista voidaan tehdä innokasta pakottamalla laskut laskettavaksi aina heti, kun ne tulevat ohjelmassa vastaan. Tätä ei tarvitse toteuttaa koko ohjelmalle, vaan voidaan valita operaatioista ne, jotka ovat lukemisen kannalta tärkeitä ja suoritetaan vain nämä tärkeät operaatiot heti. Tämä on erityisen hyödyllistä, jos tiedetään että muuttujien arvot laskettaisiin joka tapauksessa. Nämä laskut voidaan suorittaa ilman laiskuuteen kuuluvaa ylimääräistä merkitsemistä. Tämä tarkoittaa, että laskut lasketaan täysin innokkaasti ja vältytään ylimääräiseltä työltä. [10]

Innokkaasta koodista voidaan myös tehdä laiskaa [4]. Esimerkiksi luonnollisesti innokkaassa C++ -ohjelmointikielen Boost Proto -kirjastosta löytyy laiskat muuttujatyypit [11].

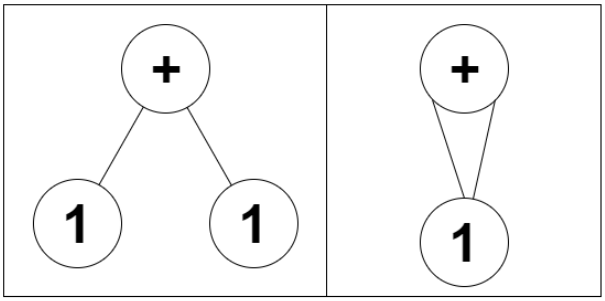
Näiden toteutus on kuitenkin hiukan monimutkaisempaa, eikä tarkka toteutus ole olennaista tämän työn kannalta, joten sen käsittely jätetään tämän työn ulkopuolelle.

Tulosten muistamisella (memoisation) laiskuuden yhteydessä tarkoitetaan jo laskettujen arvojen muistamista. Laskujen tulokset tallennetaan muistiin ja haetaan, jos sama lasku tulee uudestaan vastaan. Tämä pienentää laskuihin kuluvaa aikaa, mutta kasvattaa muistinkäyttöä. Koska sama lasku lasketaan vain kerran, saadaan mahdollisesta eksponentiaalisesta suoritusajasta polynominen, lineaarinen tai jopa vakioaikainen. [12]

Esimerkkinä tulosten muistamisesta voidaan ottaa Fibonaccin luvut. Fibonaccin luvuissa ensimmäiset kaksi lukua ovat 0 ja 1. Tästä seuraavat luvut saadaan laskemalla 2 edellistä lukua yhteen ja näin muodostuu jatkuvasti kasvava ikuinen sarja. Jotta saadaan laskettua joku haluttu luku n, täytyy ensin laskea kaikki sitä edeltävät luvut. Ensimmäisellä kerralla lasketaan kaikki aiemmat luvut ja tallennetaan nämä muistiin. Jos myöhemmin halutaan hakea joku n:ää pienempi luku, voidaan se hakea suoraan muistista, eikä tarvitse laskea edellisiä lukuja uudestaan, eli haku on vakioaikainen. Jos taas halutaan n:ää suurempi luku, haetaan muistista n ja n-1 ja lasketaan lukuja eteenpäin, kunnes saavutetaan haluttu luku. [12]

Jakamisella (sharing) tarkoitetaan samojen lausekkeiden käyttämisestä useassa eri paikassa. Jos tiedetään, että jotkut operaatiot käyttävät samoja lausekkeita keskenään, voi ohjelma sieventää nämä lausekkeet vain kertaalleen ja käyttää tuloksia molemmissa operaatioissa. Tämä on mahdollista, koska muuttujien arvoja ei voi muuttaa laiskassa ohjelmointikielessä. Jakamisella säästetään myös muistia, koska lausekkeet täytyy tallentaa vain kerran. [8]

Yksinkertaisena esimerkkinä jakamisesta lauseke 1+1. Sen sijaan, että luku 1 olisi tallennettu muistiin kahteen eri muistipaikkaan, voidaan tarvittava tila puolittaa ja käyttää vain yksi muistipaikka lukujen tallentamiseen. Kuvassa 3 on esitetty jakaminen graafisesti.



Kuva 3. *Jakaminen*

Kuvassa vasemmalla puolella on graafi ilman jakamista ja oikealla puolella jakamisen kanssa. Näin yksinkertaisessa esimerkissä ei nähdä vielä jakamisen täyttä hyötyä, mutta jo tässä esimerkissä muistinkulutusta on saatu selvästi pienennettyä. Jos pelkkien lukujen

tilalla olisikin suuri joukko suoritettavia operaatioita, saataisiin suoritusaika lähes puolitettua verrattuna ilman jakamista olevaan suoritukseen. Jakaminen ei myöskään ole rajoitettu kahteen, vaan samaa lauseketta voidaan jakaa rajattomasti muiden operaatioiden kanssa. [8]

3.2 Laiska tietokanta

Lazy evaluation of transactions in database systems on vuonna 2014 ACM sigmondin järjestemässä konferenssissä Jose M. Faleiron, Alexander Thomsonin ja Daniel J. Abadin julkaisema tutkimus laiskuuden hyödyntämisestä tietokantajärjestelmässä. Tutkimuksessa esitetään teoreettinen pohja laiskuudelle tietokannoissa sekä toteutetaan toimiva toteutus teorian pohjalta. [13]

Toteutettu järjestelmä on ensimmäinen laiskuutta hyödyntävä järjestelmä, jossa sekä kirjoitus että lukuoperaatiot ovat toteutettu laiskasti, moniajo ei kärsi, eikä ole tarvetta kommunikoida suoritettavia operaatioita koko järjestelmän tasolla. Laiskuutta hyödyntäviä tietokantajärjestelmiä on siis aikaisemminkin toteutettu, mutta niissä on ollut huomattavia ongelmia suorituskyvyn kanssa. [13]

Tietokantojen termeistä tärkeimmät Faleiro et al. tutkimuksen kannalta ovat sokeat kirjoitukset (blind writes), transaktio (transaction) ja transaktioiden ACID-ominaisuudet (ACID properties). Näiden termien lisäksi tietokantoihin liittyy paljon muitakin termejä, mutta selkeyden vuoksi nämä termit on yksinkertaistettu ymmärrettävämpään ja yksinkertaisempaan muotoon.

Transaktio on tietokantatapahtuman yksikkö, jota käsitellään kokonaisuutena. Se saattaa sisältää useita luku- ja kirjoitusoperaatioita, mutta päätös siihen sitoutumisesta (commit) tai palautuksesta (rollback) tehdään koko transaktiolle kokonaisuutena. Transaktiolla pitää olla ACID-ominaisuudet, että sitä voidaan pitää transaktiona. [14]

Tietokantojen **ACID-ominaisuudet** ovat kirjainlyhenne sanoista atomisuus (atomicity), eheys (consistency), eristyvyys (isolation) ja pysyvyys (durability). Atomisuudella tarkoitetaan, että transaktio suoritetaan joko kokonaan tai sillä ei ole mitään vaikutusta tietokantaan. Virhetilanteen sattuessa tehdään palautus kaikille transaktion tapahtumille ja palataan takaisin transaktiota edeltävään tilaan. Eheydellä tarkoitetaan, että transaktion aiheuttamat muutokset, eivät riko tietokantaan asetettuja eheysrajoitteita (constraint). Esimerkiksi ihmisen ikä ei voi olla negatiivinen. Eristyvyydellä tarkoitetaan, että transaktio suoritaan kuin se olisi ainoa tapahtuma koko järjestelmässä. Mahdolliset muut käynnissä olevat transaktiot eivät vaikuta toisiinsa mitenkään. Transaktiot eivät pääse käsiksi muiden transaktioiden keskeneräiseen dataan. Pysyvyydellä tarkoitetaan, että muutokset tietokantaan ovat pysyviä. Esimerkiksi levyrikon sattuessa voidaan palauttaa tietokanta levyrikkoa edeltävään tilaan käyttämällä transaktiolokia ja varmuuskopioita. [15]

Sokea kirjoitus on tapahtuma, jossa muuttujan arvo asetetaan välittämättä sen nykyisestä arvosta [13]. Esimerkiksi puhelinnumeron muutos on esimerkki sokeasta kirjoituksesta. Vanha puhelinnumero ei vaikuta uuteen numeroon mitenkään ja numero voidaan kirjoittaa suoraan vanhan arvon päälle.

3.2.1 Laiskuuden hyödyt ja haitat tietokannoissa

Tekstissään Faleiro et al. esittelevät useita hyötyjä laiskuudesta hyödyntämisestä tietokannoissa, mutta pääosin ne seuraavat laiskuuden yleisiä periaatteita:

- Ylimääräistä työtä vältetään, jos muuttujaa ei koskaan lueta (avoiding unnecessary work) [13].
- Muistiin haetaan arvot vain kertaalleen, jos useita operaatioita on ketjutettuna (improved overall cache/buffer pool locality) [13].
- Kirjoitusoperaatiot ovat nopeampia, koska tarvitaan vain tieto, voidaanko transaktio suorittaa (reduced transaction execution latency) [13].

Näiden yleisesti pätevien hyötyjen lisäksi teksti esittelee myös uusia hyötyjä, jotka pätevät erityisesti tietokannoille:

Kuorman tasoitus (temporal load balancing). Koska transaktiota ei suoriteta kokonaan heti, saadaan kerralla suoritettavaa työmäärää pienennettyä. Erityisesti kiireisistä ajoista työmäärän siirtäminen hiljaisempiin aikoihin auttaa tasaamaan työmäärän järjestelmässä paremmin. [13] Toisaalta tämä saattaa myös kasata suoritettavia transaktioita yhteen, jotka voidaan joutua suorittamaan kiireisenä aikana lukutapahtuman yhteydessä.

Parempi rinnakkaistettavuus (reduced contention footprint). Transaktioista suoritetaan heti vain se osuus, joka estää muita transaktioita suorittamasta samanaikaisesti ja loput suoritetaan myöhemmin. Laiskasti suoritettavasta osuudesta voidaan siirtää osa suoritettavaksi heti paremman rinnakaistettavuuden saavuttamiseksi. [13]

Hyötyjen lisäksi tekstissä esitellyt laiskuuden haitat tietokannoissa seuraavat hyvin läheisesti yleisiä haittoja. Erityisesti tietokantoihin vaikuttavia haittoja ei tuoda esille, mutta pieniä eroavaisuuksia niistä löytyy yleisiin verrattuna:

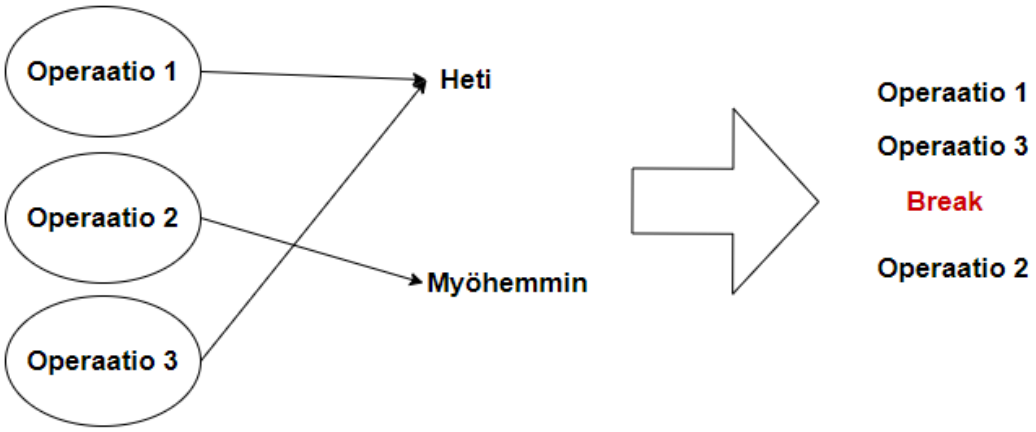
- Ennen muuttujan lukemista täytyy suorittaa transaktiot, jotka kohdistuvat muuttujaan (higher read latencies) [13].
- Että voidaan lukea jokin muuttuja, voidaan joutua suorittamaan pitkä ketju transaktioita muihin muuttujiin, ennen kuin voidaan suorittaa transaktiot, jotka kohdistuvat luettavaan muuttujaan (dependencies between deferred transactions) [13].
- Muuttujiin täytyy merkitä, mitkä transaktiot niihin vaikuttavat. Lisäksi voidaan joutua päättelemään mihin muuttujiin transaktio kohdistuu, jos sitä ei ole suoraan annettu transaktiossa (overhead of determining the write set of a transaction) [13].

Laiskuuden hyödyntämisestä tietokannoissa eroaa yleisistä haitoista lähinnä käytettyjen termien erilaisuudella. Sen sijaan, että laiskuus vaikuttaisi yksittäisiin muuttujiin, puhutaan tietokantojen yhteydessä kokonaisista transaktioista.

3.2.2 Laiskan tietokannan teoria ja toteutus

Lazy evaluation of transactions in database systems -tutkimuksessa toteutettu laiska tietokanta näyttää ulospäin samalta kuin mikä tahansa muukin tietokanta. Erona perinteiseen on, että vaikka tietokanta on sitoutunut transaktioon, ei kaikkea transaktioon liittyvää työtä ole todellisuudessa vielä suoritettu. Transaktioista suoritetaan heti mahdollisimman vähän ja loput suoritetaan laiskasti vasta silloin, kun jotakin kohdemuuttujaa yritetään lukea. Laiskasta suorittamisesta huolimatta transaktioiden tulokset ovat samat kuin innokkaassa järjestelmässä ja transaktiot säilyttävät täydet ACID-ominaisuudet. Rajoituksena laiskuuden hyödyntämiselle tietokannoissa on, että tietokannan täytyy olla deterministinen. Deterministisessä tietokannassa transaktiot voidaan keskeyttää vain determinististä syistä. Transaktioista voidaan päätellä suoraan, aiheutuuko niiden suorittamisessa keskeytystä esimerkiksi eheyden rikkoutumisen takia. Epädeterministisissä keskeytyksissä, esimerkiksi lukkiintumisessa, transaktioita ajetaan uudestaan, kunnes ne on saatu tallennettua tietokantaan. [13]

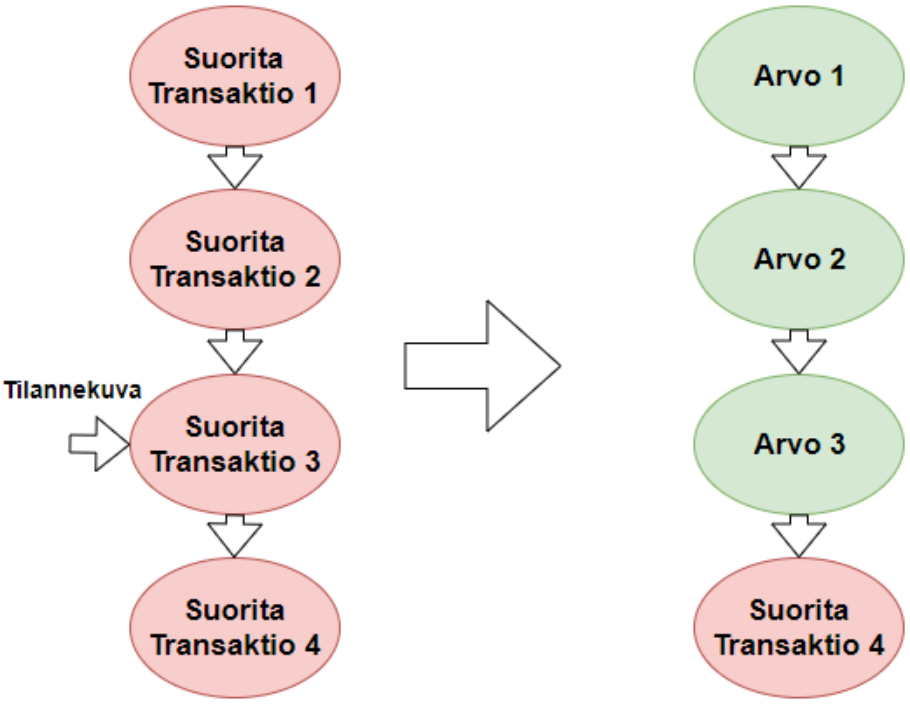
Täysin laiskassa tietokantajärjestelmässä transaktiot vain kirjoitettaisiin lokiin ja suoritettaisiin kuin mitä tahansa muuttujaa haluttaisiin lukea tietokannasta. Tämänkaltaisessa järjestelmässä ei saavuteta laiskuuden käytöstä juurikaan mitään hyötyä, vaan operaatiot vain kasaantuisivat yhteen. Jotta laiskuuden käytöstä saavutettaisiin jotain hyötyä, täytyy selvittää mihin muuttujiin transaktio vaikuttaa ja varmistaa ettei sen suoritus aiheuta keskeytystä. Järjestelmä ei ole siis täysin laiska, vaan osa työstä tehdään heti, kun transaktio saadaan. Järjestelmä on jaettu kahteen osaan: innokkaasti suoritettavaan heti-osaan (now-phase) ja laiskasti suoritettavaan myöhemmin-osaan (later-phase). Tämä jako voidaan tehdä joko manuaalisesti tai automaattisesti. Manuaalisessa toteutuksessa luodaan jokaiselle transaktiolle oma tiedostonsa, johon kirjoitetaan heti-osaan kuuluvat operaatiot ja myöhemmin-osaan kuuluvat operaatiot. Nämä erotetaan toisistaan paluu-operaatiolla, joka keskeyttää operaatioiden suorituksen ja palauttaa suorituksen takaisin järjestelmälle. [13] Manuaalinen jako voi olla aiheellinen, jos erilaisia transaktioita järjestelmässä on vähän. Jokaista transaktiota varten joudutaan luomaan oma tiedostonsa, joten tästä voi muodostua suuri työmäärä, jos kyselyitä on paljon. Automaattinen jako toimii muuten samoin, mutta ei tarvitse luoda erillistä tiedostoa tätä varten vaan tutkitaan mikä on viimeinen operaatio transaktiossa, joka voi aiheuttaa keskeytyksen tai vaikuttaa kirjoitettaviin muuttujiin. Tämä ja sitä edeltävät operaatiot suoritetaan innokkaasti ja palautetaan suoritus takaisin järjestelmälle. [13] Kuvassa 4 on esitetty heti- ja myöhemmin-osiin jako, sekä niiden suoritus transaktiossa.



Kuva 4. *Jako heti- ja myöhemmin-osiin.*

Operaatiot jaetaan innokkaasti suoritettavaan heti-osaan ja laiskasti suoritettavaan myöhemmin-osaan. Transaktiosta suoritetaan heti-osaan kuuluvat operaatiot ja tämän jälkeen palautetaan suoritus takaisin järjestelmälle. Kun myöhemmin halutaan lukea jokin transaktion kohdemuuttujista, jatketaan suoritusta pisteestä, josta ohjelman suoritus aikaisemmin palautettiin. [13]

Riippumatta siitä, suoritetaanko operaatioiden jako heti- ja myöhemmin-osiin automaattisesti vai manuaalisesti, ovat niiden sisältämät operaatiot samoja. Heti-osassa suoritetaan operaatioista ne, jotka voivat aiheuttaa keskeytyksen transaktiossa tai vaikuttavat transaktiossa kirjoitettavien sekä luettavien muuttujien joukkoon. Deterministisessä tietokannassa keskeytyksen välttämiseksi riittää tarkastaa, että transaktio ei riko tietokannan eheyttä. Eheyden varmistamiseksi tarkistetaan kaikki käyttäjän asettamat eheysrajoitteet, kuten suomalaisen postinumeron täytyy olla 5 numeroa pitkä, sekä tietokannan asettamat valmiit rajoitteet, kuten pääavaimena käytetyn henkilötunnuksen täytyy olla yksilöllinen. Jos transaktiossa ei ole annettu suoraan muuttujia, joihin kirjoitus kohdistuu, joudutaan päättely tekemään heti-vaiheessa. Transaktiosta siis suoritetaan vain sen verran, että saadaan selville nämä kohdemuuttujat ja varsinainen suoritus jätetään myöhemmäksi. Näihin kohdemuuttujiin kirjoitetaan mitä transaktioita tulee suorittaa, ennen kuin muuttujaa voidaan lukea. Suoritettavien transaktioiden lisäksi tarvitaan tieto muista tietokannan muuttujien tiloista transaktion suoritushetkellä, että transaktion tulos on sama, kuin se olisi innokkaasti suoritettuna. Tätä varten joudutaan päättelemään, mitä muuttujia transaktiossa luetaan ja otetaan näistä tilannekuva (snapshot). Tilannekuvaa varten ei tarvitse vielä suorittaa lukumuuttujien transaktioita, vaan näistä muuttujista tallennetaan vain niiden vaatimat transaktiot. [13] Kuvassa 5 on esitetty tilannekuvan toiminta transaktioiden yhteydessä.



Kuva 5. *Tilannekuvan toiminta.*

Kuvassa vasemmalla puolella on kuvattu tilanne ennen transaktion suoritusta. Jokin transaktio tarvitsee muuttujan X arvoa voidakseen suorittaa oman operaationsa. Muuttujalla X oli transaktion suoritushetkellä 3 transaktiota, jotka täytyy suorittaa ennen kuin muuttujaa X voidaan lukea. Suoritettava transaktio otti tilannekuvan muuttujasta X ja tämän jälkeen transaktio 4 lisättiin vielä suoritettavien transaktioiden listaan. Kuvan oikealla puolella on tilanne, kun tämä haluttu transaktio on suoritettu. Transaktiot 1, 2 ja 3 ovat suoritettu ja niiden tulokset tallennettu muiden luettavaksi, mutta transaktion 4 arvoa ei ole vielä laskettu, koska suoritettu transaktio ei sitä tarvinnut. Todellisuudessa muistiin on tallennettu transaktion suorituksen jälkeen vain arvo 3, paitsi jos arvoista 1 tai 2 on otettu tilannekuva jonkin muun transaktion toimesta. Kun kaikki heti-osaan kuuluvat operaatiot on suoritettu, voidaan transaktioon sitoutua. Tämä tarkoittaa, että transaktio on pysyvästi tallennettu tietokantaan, vaikkei varsinaista työtä ole vielä tehty. Varsinainen työ tehdään myöhemmin-osassa, kun jotakin transaktion kohdemuuttujista halutaan lukea. Transaktiot suoritetaan ja niiden tulokset tallennetaan muuttujiin muiden transaktioiden saataville. Samoja transaktioita ei siis tarvitse suorittaa useampaan kertaan, vaan tulokset tallennetaan muuttujiin suoritettavien transaktioiden tilalle. [13]

Varsinaista toteutusta varten järjestelmä jaetaan kahteen osaan: merkitsemiskerrokseen (stickification layer) ja laskentakerrokseen (substantiation layer). Merkitsemiskerroksen tehtävänä on hoitaa kaikki heti-osaan kuuluvat tehtävät, valvoa transaktioiden määrää, sekä antaa transaktiot hoidettavaksi laskentakerrokselle. Laskentakerroksen tehtävänä on ainoastaan suorittaa transaktiot loppuun, eli hoitaa kaikki myöhemmin-osaan kuuluvat tehtävät. Transaktioiden määrää valvotaan, etteivät muuttujien lukutapahtumien kestot

olisi liian pitkiä. Jokaisella muuttujalla on laskuri suorittamattomien transaktioiden määrästä. Jos tämän laskurin arvo ylittää käyttäjän asettaman luvun, suoritetaan kaikki transaktiot jonosta ja nollataan laskuri. [13] Tässä siis poiketaan laiskuudesta, eli työtä suoritetaan vaikkei todellista tarvetta suoritukselle olisi.

Faleiro et al. toteuttivat transaktioiden operaatioiden jaon heti- ja myöhemmin-osiin tässä työssä manuaalisesti. Jokaista transaktiota varten luotiin oma tiedostonsa C++-kielellä, joissa ensin suoritettiin kaikki heti-osan operaatiot ja tämän jälkeen kutsuttiin EndNowPhase-metodia, joka palautti suorituksen takaisin kutsuvalle ohjelmalle. Kun transaktiot suoritettiin loppuun, jatkettiin suoritusta EndNowPhase-metodia seuraavasta operaatiosta. [13]

4. EVALUOINTITAPOJEN VERTAILU

4.1 Ohjelmointikieli

Tässä luvussa vertaillaan laiskan Haskellin ja innokkaan C++:san tehokkuutta toisiinsa. Testit ovat Alexandra Băicoianun, Raluca Pândarun ja Anca Vasilescun vuonna 2013 julkaisemasta tutkimuksesta *Upon the performance of a haskell parallel implementation.* He vertasivat tutkimuksessaan laiskuutta hyödyntävän Haskellin tehokkuutta kolmeen eri C++:salla toteutettuun ohjelmaan. Näistä 3 vertailuohjelmasta 1 on toteutettu ilman rinnakkaisuutta ja loput 2 on toteutettu käyttäen C++ rinnakkaisuuskirjastoja PPL (parallel patterns library) ja OpenMP (Open multiprosessing). Testiympäristössään Băicoianu et al. käyttivät kaksiytimistä Intel Pentium T4300 -prosessoria 2.10 GHz kellotaajuudella [16]

Testikohteena Băicoianu et al. toteuttamassa tutkimuksessa oli reunantunnistus kuvasta (edge detection). Reunantunnistuksella tarkoitetaan eri kohteiden tunnistusta kuvasta, eli esimerkiksi ihmisen erottamista taustalla olevasta seinästä. Koneellisesti tämä tunnistus tehdään muuttamalla kuva harmaasävykuvaksi, suodattamalla kuva ja lopuksi vertailemalla yksittäisten pikseleiden värejä niiden ympäristössä oleviin pikseleihin. Kuvan suodatuksella tasoitetaan kuvasta pieniä alueita pois, ettei niitä virheellisesti tunnisteta reunantunnistuksessa. Băicoianu et al. suorittivat testit käyttäen 3 eri kokoista suodatinta: 3x3, 5x5 ja 7x7. Suodattimen koko määrää kuinka laajalta alueelta yksittäisen pikselin ympäristössä tasoitus tehdään. Esimerkiksi 3x3 kokoisessa suodattimessa jokaisen pikselin ympäristöstä otetaan huomioon vain pikselin viereiset 8 muuta pikseliä. [16]

Băicoianu et al. testikohteena oli 3008x2000 -pikselin kokoinen kuva, johon testien algoritmit ajettiin. He suorittivat jokaisen testin 11 kertaa ja tulokset on saatu näiden ajojen keskiarvoista. Taulukossa 1 on esitetty Băicoianu et al. saamat tulokset testien suoritusten keskiarvoista. [16]

Taulukko 1. *Kuvankäsittelyn tulokset. (perustuen lähteeseen [16])*

Algoritmi	3x3 Suodatin (ms)	5x5 Suodatin (ms)	7x7 Suodatin (ms)
Haskell	226	457	4999
C++	555	732	995
C++ PPL	438	513	626
C++ OpenMP	295	385	533

Taulukossa on merkitty vihreällä värillä nopein ja punaisella hitain algoritmi jokaisesta suodatinkoosta. Taulukosta nähdään, että Haskell on nopeampi kuin nopeinkaan C++-algoritmi, kun käytössä on pieni suodatin. Vastaavasti hitaimmaksi algoritmiksi jäi C++-algoritmi, joka ei käytä rinnakkaisuutta, mutta myös PPL toteutus on hidas verrattuna Haskell- ja OpenMP-toteutuksiin. Keskimmäisessä suodattimessa OpenMP on nopein ja ero muiden suodattimien välillä on kutistunut. Suurinta suodatinta testattaessa OpenMP on edelleen nopein ja PPL on hyvin lähellä nopeudessa. Haskell jää selvästi jälkeen, jopa ilman rinnakkaisuutta olevasta C++-toteutuksesta. [16]

Băicoianu et al. testien päätarkoituksena oli verrata funktionaalista Haskellia ja imperatiivista C++:saa toisiinsa, mutta samalla voidaan saada joitakin viitteitä tehokkuudesta laiskuuden hyödyntämisestä ohjelmointikielissä. C++ on erityisesti tehokkuuteen keskittyvä ohjelmointikieli, mutta laiskuutta hyödyntävä Haskell on kuitenkin tässä sovelluskohteessa nopeampi pienellä suodatinkoolla. Nämä suoritetut testit eivät ole yleispäteviä, eikä edes tässä sovelluskohteessa voida sanoa, mikä käytetyistä algoritmeista on paras joka tilanteessa. [16]

4.2 Tietokanta

Tässä luvussa vertaillaan lyhyesti laiskaa ja innokasta tietokantaa erilaisissa testeissä. Vertailut perustuvat Faleiro et al. tekemiin testeihin *Lazy evaluation of transactions in database systems* -tutkimuksessa ja niiden tulokset vain esitellään tässä lyhyesti. Näiden vertailujen tarkoituksena on näyttää, millaiseen sovelluskohteeseen laiska tietokanta soveltuu hyvin ja millaiseen se sopii huonosti. Lisäksi nämä testit todistavat luvun 3.3.2 väitteet todeksi laiskuuden hyödyistä ja haitoista tietokannoissa.

Näissä testeissä Faleiro et al. testasivat suoritustehoa, läpäisylatenssia (end-to-end latency), kuorman tasoitusta ja sokeita kirjoituksia. Näiden testien lisäksi he suorittivat yleisiä TPC-C (transaction processing council) suorituskykytestejä (benchmark). Faleiro et al. suorittivat jokaisen testin 10 kertaa ja tulokset on esitetty näiden ajojen keskiarvoista. [13]

Testiympäristönä Faleiro et al. käyttivät Linux 3.9.2 käyttöjärjestelmää. Prosessoriksi he valitsivat 10-ytimisen Intel Xeon E7-8850 -prosessorin, josta 8 ydintä varattiin tietokannan pyörittämiseen. Lopuista 2 ytimestä toinen varattiin tietokantakyselyiden syöttöön ja toinen tehokkuuden mittaamiseen. Laiskassa järjestelmässä tietokantaa varten varatuista 8:sta ytimestä 1 varattiin merkitsemiskerrokselle ja laskentakerrokselle loput 7. Vertailukohteena oleva innokas järjestelmä Faleiro et al. toteuttivat muokkaamalla laiskan järjestelmän samanaikaisuudenhallintajärjestelmää ja siinä kaikki 8 ydintä varattiin transaktioiden suorittamiseen. [13] Faleiro et al. toteuttama innokas järjestelmä ei siis ole optimoitu innokkaaseen laskentaan, eikä valittu tietokantateknologia ole paras mahdollinen innokkaita transaktioita varten. Testeissä saadut tulokset ovat lähinnä suuntaa antavia,

mutta niistä saa viitteitä millaiseen sovelluskohteeseen laiska tietokantajärjestelmä sopisi hyvin.

Faleiro et al. suorittamista suoritustehotesteistä voidaan havaita, että laiskan järjestelmän suoritusteho paranee suurilla transaktioiden ketjujen pituuksilla ja transaktioissa kirjoitettavien muuttujien päällekkäisyyksillä. Tämä johtuu Faleiro et al. mukaan useiden transaktioiden käyttämistä yhteisistä muuttujista, jotka täytyy tuoda vain kerran hitaammasta muistista välimuistiin. Näihin muuttujiin voidaan suorittaa useita muutoksia samalla kertaa, ennen kuin ne täytyy tallentaa takaisin hitaampaan muistiin pysyvää tallennusta varten. [13]

Läpäisylatenssissa Faleiro et al. toteuttama järjestelmä häviää aina innokkaalle järjestelmälle. Tämä johtuu Faleiro et al. mukaan laiskan järjestelmän suorittamattomista transaktioista muuttujissa, jotka täytyy suorittaa ennen kuin muuttujaa voidaan lukea. Vaikka muuttujassa ei olisi suorittamattomia transaktioita lukuhetkellä, kuluu hiukan aikaa joka tapauksessa muuttujan suorittamattomien transaktioiden tarkastamiseen. [13]

Faleiro et al. toteuttama järjestelmä pystyy tasoittamaan kuormaa kiireisestä ajasta myöhemmin suoritettavaksi. Ruuhkaisessa järjestelmässä saapuville transaktioille hoidetaan vain merkitseminen ja varsinainen laskenta suoritetaan vähemmän kiireisenä aikana. Innokkaassa järjestelmässä tämä ei olisi mahdollista, vaan transaktiot epäonnistuisivat kokonaisuutena. [13]

Tietokannassa tapahtuvat sokeat kirjoitukset parantavat laiskan järjestelmän suorituskykyä. Faleiro et al. mukaan laiskassa järjestelmässä ei tarvitse suorittaa transaktioita lainkaan, jos niiden kohdemuuttujat ylikirjoitetaan ilman niiden lukemista. Innokas järjestelmä ei puolestaan hyödy sokeista kirjoituksista lainkaan, koska muuttujiin kohdistuvat transaktiot ovat jo suoritettu. [13]

Viimeisenä testikokonaisuutena Faleiro et al. testasivat yleisiä TPC-C suorituskykytestejä. Näiden testien tarkoituksena oli verrata yleisesti käytetyllä suorituskykytestillä laiskaa ja innokasta tietokantaa. Tärkeimpänä tuloksena tästä testikokonaisuudesta on laiskan järjestelmän parempi rinnakkaistettavuus. Faleiro et al. mukaan laiskassa järjestelmässä voidaan siirtää osa laiskasti suoritettavasta laskennasta heti-osaan. Nämä heti-osaan siirrettävät operaatiot valitaan niin, että ne sisältävät osuuden, joka estää muita transaktioita suorittamasta samanaikaisesti. Tälläiset operaatiot jotka estävät rinnakkaisen suorituksen ovat usein ne, joiden kirjoitusoperaatiot kohdistuvat usein käytettyihin muuttujiin. Samaa muuttujaa ei voi kirjoittaa kuin yksi operaatio kerrallaan ja muut operaatiot odottavat vuoroaan. [13]

5. YHTEENVETO

Laiskan evaluoinnin päätarkoitus on välttää ylimääräistä työtä suorittamalla operaatiot vasta, kun niiden tuloksia todellisuudessa tarvitaan. Laiska evaluointi mahdollistaa loputtomien sarjojen käyttämisen, joka ei olisi mahdollista ilman laiskuutta. Laiskuutta voidaan hyödyntää useissa sovelluskohteissa, mutta ei voida sanoa, että se olisi yleisesti parempi tai huonompi kuin innokas evaluointi. Haittapuolena laiskan evaluoinnin käytössä on pääasiassa ylimääräinen kirjanpito sekä hitaammat lukuoperaatiot.

Ohjelmointikielissä lausekkeet voidaan esittää graafeina ja niitä voidaan sieventää kahdella eri tavalla. Yleensä laiskat ohjelmointikielet hyödyntävät *outermost reductionia*, jossa sievennys aloitetaan ylhäältä ja yritetään saada mahdollisimman vähän suoritettua tuloksen saamiseksi. Laiskoissa ohjelmointikielissä samoja lausekkeita voidaan käyttää useassa paikassa, koska muuttujia ei voi muuttaa jälkikäteen. Testien tulosten perusteella laiska ohjelmointikieli sopii kuvankäsittelyssä tilanteeseen, jossa suodattimen koko on pieni.

Tietokannoissa laiskuutta voidaan hyödyntää transaktioiden viivästettyyn suoritukseen. Transaktioita ei suoriteta täysin laiskasti, vaan ne jaetaan innokkaaseen heti-osaan ja laiskaan myöhemmin-osaan. Heti-osassa selvitetään mihin muuttujiin transaktiot vaikuttavat ja niihin merkitään transaktio suoritettavaksi. Myöhemmin osassa suoritetaan varsinainen laskenta, joka tulee vastaan vasta muuttujia luettaessa. Laiska tietokanta sopii järjestelmiin, jossa transaktiot usein käsittelevät samoja muuttujia, eikä lukunopeus ole kriittistä.

Laiska evaluointi on vain yksi työkalu, jota voidaan hyödyntää. Ei voida yleisesti sanoa, että se olisi hyvä tai edes hyödyllinen jokaisessa sovelluskohteessa. Laiskan evaluoinnin käyttäminen on vaihtokauppaa eri hyötyjen ja haittojen kanssa. Näiden tietojen perusteella voidaan arvioida kannattaako laiskaa evaluointia käyttää halutussa sovelluskohteessa.

LÄHTEET

[1] P. Hudak, Conception, evolution, and application of functional programming languages, ACM Computing Surveys (CSUR), Vol. 21, Iss. 3, 1989, pp. 359─411.

[2] F. Sinot, Complete Laziness: a Natural Semantics, Electronic Notes in Theoretical Computer Science, Vol. 204, 2008, pp. 129─145.

[3] lazy evaluation, in: A Dictionary of Computer Science, 7th ed., Oxford University Press. 2016.

[4] B. Lampson, Lazy and Speculative Execution in Computer Systems in: Anonymous (ed.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 1─2.

[5] Lazy Evaluation, C2 Wiki, 2009, Saatavissa: http://wiki.c2.com/?LazyEvaluation.

[6] V. Torra, Scala: From a Functional Programming Perspective: An Introduction to the Programming Language, Springer International Publishing, Cham, 2016.

[7] L. Azriel, A. Mendelson, U. Weiser, Peripheral Memory: A Technique for Fighting Memory Bandwidth Bottleneck, IEEE Computer Architecture Letters, Vol. 14, Iss. 1, 2015, pp. 54─57.

[8] Heinrich Apfelmus, The Incomplete Guide to Lazy Evaluation (in Haskell), Hack.hands(), 2014, Saatavissa: https://hackhands.com/lazy-evaluation-works-haskell/.

[9] Haskell Wikibook, Saatavissa: https://en.wikibooks.org/wiki/Haskell.

[10] G. Tremblay, G. R. Gao, The Impact of Laziness on Parallelism and the Limits of Strictness Analysis, 1995, pp. 119─133.

[11] Boost Proto, Boost Proto Users' Guide, 2008, Saatavissa: http://www.boost.org/doc/libs/1_65_0/doc/html/proto/users_guide.html.

[12] S. Abdallah, Memoisation: Purely, Left-recursively, and with (Continuation Passing) Style, 2017.

[13] J. Faleiro, A. Thomson, D. Abadi, Lazy evaluation of transactions in database systems, Proceedings of the 2014 ACM SIGMOD International Conference on management of data, ACM, pp. 15─26.

[14] transaction, in: A Dictionary of Computer Science, 7th ed., Oxford University Press. 2016.

[15] ACID Properties, Microsoft Developer Network, 1998, Saatavissa: https://msdn.microsoft.com/en-us/library/aa480356.aspx.

[16] A. Baicoianu, R. Pândaru, A. Vasilescu, Upon the performance of a Haskell parallel implementation, Bulletin of the Transilvania University of Brasov.Mathematics, Informatics, Physics.Series III, Vol. 6, Iss. 2, 2013, pp. 61.