Tampereen yliopisto

Joni Koskinen

# HAJAUTUSTAULUJEN TÖRMÄYSTEN RATKAISUTEKNIIKAT

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Huhtikuu 2023

# TIIVISTELMÄ

Joni Koskinen : Hajautustaulujen törmäysten ratkaisutekniikat
Kandidaattitutkielma
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Huhtikuu 2023

---

Hajautustaulu on laajalti käytössä oleva tehokas tietorakenne, joka parhaimmillaan tarjoaa avain-arvo-parien vakioaikaista tallennusta, hakua ja poistoa. Tallennusoperaatiossa hajautusfunktiolla lasketaan tiivistearvo, jonka perusteella avain-arvo-pari tallennetaan tauluun. Mikään hajautusfunktio ei kuitenkaan ole niin hyvä, että jokainen tiivistearvo olisi ainutlaatuinen. Eri parien välillä tapahtuu väistämättä törmäyksiä eli samalle paikalle yritetään lisätä kahta eri paria. Törmäysten välttämiseksi on kehitetty erilaisia ratkaisutekniikoita.

Tässä työssä käydään läpi hajautustaulujen törmäysten ratkaisutekniikoita tavoitteena löytää tehokkaimmat vaihtoehdot tutkittujen ratkaisujen joukosta. Tekniikoista valikoitiin mukaan joitakin yleisesti tunnettuja vanhempia sekä vähemmän tunnettuja uudempia tekniikoita. Työhön valitut tekniikat ovat lineaarinen kokeilu, hyppytekniikat, käkihajautus, ruutuhyppelyhajautus, pariton-parillinen hajautus, erillisketjutus sekä taulukkorakenne. Työssä esitetään tekniikoiden perustoiminta ja käydään läpi kirjallisuuskatsauksen keinoin niiden suoriutuminen verrattuna toisiinsa eri tahojen suorittamissa kokeissa. Lähteinä työssä on käytetty sekä eri tekniikoiden alkuperäisjulkaisuja että yleisesti eri tekniikoita esitteleviä ja vertailevia tutkimuksia. Työ ei ota kantaa rinnakkaisiin hajautustauluihin, vaan keskittyy hajautustauluihin, jotka eivät huomioi rinnakkaisuutta.

Tutkimuksesta ei saatu suoraan selvitettyä parasta ratkaisutekniikkaa. Tehokkuus riippuu monesta tekijästä eikä parhaiten pärjänneiden tekniikoiden välillä ole suoritettu kokeita. Tuloksista voidaan kuitenkin todeta parhaiten pärjänneiksi tekniikoiksi ruutuhyppelyhajautus, pariton-parillinen hajautus, erilllisketjutus sekä taulukkorakenne. Näistä taulukko pärjää tekstisyötteiden osalta parhaiten, mutta ruutuhyppelyn sekä pariton-parillinen hajautuksen ja erillisketjutuksen paremmuudesta ei saatu selkoa. Tässä työssä annetaan ehdotukseksi tehdä jatkotutkimusta keskittyen vain näihin algoritmeihin ja selvittää, missä olosuhteissa mikäkin on tehokkain vaihtoehto.

Avainsanat: hajautustaulu, törmäys, tehokkuus, törmäyksien ratkaisutekniikat

# SISÄLLYSLUETTELO

# 1. JOHDANTO

*Hajautustaulu* (Hashtable) on tehokas tietorakenne, joka pystyy tallentamaan, hakemaan tai poistamaan alkioita parhaimmillaan vakioaikaisella tehokkuudella. Hajautustaulu on tehokkuutensa vuoksi laajalti käytetty tietorakenne ja siksi sitä on myös tutkittu paljon. Hajautustaulutietorakenteita on käytössä useissa ohjelmointikielissä kuten Javassa, Pythonissa ja C++:sa (Tapia-Fernández et al., 2022). Hajautustauluja voidaan nopeutensa vuoksi käyttää erilaisten tietokantojen, kuten osoitetietojen säilyttämiseen.

Hajautustaulu luodaan laskemalla *hajautusfunktiolla* avaimelle tiivistearvo, jonka perusteella määritellään taulun indeksi, jonne avain ja sen sisältämä arvo voidaan tallentaa. Tietoa hakiessa hakuavaimesta lasketaan samaa hajautusfunktiota käyttäen tiivistearvo, jonka perusteella tietosisältö haetaan taulusta.

Hajauttaessa on aina mahdollisuus törmäyksille. Törmäykseksi kutsutaan tapahtumaa, jossa hajautusfunktio laskee saman indeksin kahdelle eri alkiolle. Törmäyksien mahdollisuutta voidaan havainnollistaa syntymäpäiväongelman kautta. Siinä esitetään matemaattisesti, että kun henkilöitä on 23 tai enemmän, on yli 50 % todennäköisyys sille, että kahdella eri henkilöllä on sama syntymäpäivä. Sama teoria pätee myös hajautustauluihin, kun vuoden päivien määrä korvataan hajautustaulun koolla ja ihmisten määrä alkioilla. (Mailund, 2019, luku 2)

Törmäystilanteiden ratkaisua varten on kehitetty erilaisia tekniikoita, joissa määritellään eri tapoja löytää uusi paikka alkiolle. Törmäyksen ratkaisutekniikka määrää sen, kuinka paljon tilaa taulukko vie ja kuinka nopeasti sieltä voi hakea arvoja. Oikean tietorakenteen valinnassa on tärkeää ottaa huomioon, ovatko alkiot numeroita vai merkkijonoja, kuinka nopea tai tehokas tietorakenne on ja kuinka paljon tilaa se käyttää. Taulun täyttöaste vaikuttaa myös tehokkuuteen (Pagh & Rodler, 2004).

Tämän työn tarkoituksena on tutkia törmäystilanteita ratkaisevia tekniikoita kirjallisuuskatsauksen keinoin ja selvittää, onko jokin tekniikoista erityisesti parempi kuin muut. Tarkoitus on samalla selvittää, onko eri tekniikoiden tehokkuuksien määrittelyssä eroavaisuuksia eri tietolähteiden välillä. Tekniikat valikoituivat mukaan perustuen niiden yleisyyteen aihepiirin tutkimuksissa tai niiden tuoreuteen. Valitut tekniikat rajautuvat kuitenkin hajautustauluihin, jotka eivät ota huomioon rinnakkaisuutta.

Luvussa 2 käydään läpi työssä käytettyä tutkimusmenetelmää. Luvussa 3 kerrotaan yleisesti hajautusfunktioista ja törmäyksien ratkaisemisesta, jonka jälkeen luku 4 keskittyy eri ratkaisutekniikoiden esittelemiseen. Luku 5 avaa eri tutkimuksista saatuja tuloksia ja luvussa 6 vertaillaan tuloksia sekä vastataan tutkimuskysymykseen. Luvussa 7 kootaan yhteen koko tutkimus.

# 2. TUTKIMUSMENETELMÄ

Työssä käytettyjä lähteitä etsittiin pääsääntöisesti Andor-, Google Scholar- sekä ProQuest-hakukoneilla. Hakusanoina toimivat parhaiten ”Hash table”, ”Hash table AND Collision resolution” ja ”Hash table applications”. Muita lisähakusanoja olivat ”use cases”, ”NOT parallel” ”fast” ja ”performance”. Lisäksi eri tekniikoiden nimet toimivat hakutermeinä. Haut olivat englanninkielisiä, koska aiheesta ei löydy tieteellisiä lähteitä tai lähteitä ylipäätään suomeksi.

Hauilla sai paljon tuloksia, mutta vain osa soveltui työn tarkoitukseen. Kriteerinä oli, että lähteen tulee olla vertaisarvioitu tai muuten luotettavasta julkaisusta. Jotkin lähteet eivät ole vertaisarvioituja, mutta ne ovat julkaistu tieteellisessä julkaisussa. Aiheesta oli vaikea löytää sopivia vertaisarvioituja julkaisuja, mikä on osasyy sille, että osa käytetyistä lähteistä ovat vertaisarvioimattomia. Työssä käytetään lähteinä esiteltyjen tekniikoiden alkuperäisjulkaisuja, vaikka ne ovatkin vanhoja. Muuten työssä tavoitteena on pitäytyä mahdollisimman tuoreissa lähteissä. Rinnakkaisuuteen liittyvien hajautustaulujen lähteet rajautuivat myös pois.

Lisähakusanojen tarkoitus oli tarkentaa aiempia hakuja sekä löytää tietoa erilaisista käyttökohteista, mutta niiden hyöty jäi pieneksi. Omana aiheenaan hajautustaulujen törmäyksistä löytyy vähän lähteitä, mutta ne on mainittu useassa lähteessä ohimennen. Lähteitä löytyi myös jo käsiteltyjen lähteiden lähdeluetteloiden kautta. Tosin eri lähteillä oli jo valmiiksi useita samoja lähteitä käytössä.

# 3. HAJAUTUS YLEISESTI

Hajautustaulun törmäyksien ratkaisemiseksi täytyy valita oikeanlainen hajautusfunktio sekä ratkaisutekniikka. Hyvä hajautusfunktio vaikuttaa hajautustaulun luontinopeuteen sekä vähentää törmäyksien määrää, vaikka se ei niitä kokonaan poista. Hajautusfunktion tulee jakaa avaimia mahdollisimman tasaisesti, jotta taulu pääsee haluttuun vakioaikaisuuteen. Avaimien jakautuminen voi myös epäonnistua, jos data on jo valmiiksi huonosti jakautunutta. Esimerkiksi alkiot voivat luonnollisesti olla painottuneita tiettyihin arvoihin, ne voivat olla vääristyneitä huonon pyöristyksen jäljiltä tai sanat voivat alkaa yleisesti samoilla kirjaimilla, kun kyseessä on tekstisyöte. (OpenDSA, 2021, luku 10)

Hajautustaulun täyttöaste ilmaisee, kuinka moni taulun paikoista on täytetty suhteessa koko tauluun. Täyttöaste ilmaistaan desimaaliarvolla välillä nollasta yhteen. Täyttöasteella on merkittävä rooli ratkaisutekniikoiden tehokkuuksia määritellessä, sillä toisin kuin suljetussa hajautuksessa, avoimessa hajautuksessa tila voi loppua kesken. Siis kun täyttöaste lähestyy yhtä, tapahtuu törmäyksiä useammin ja suoritusaika kasvaa. (Mailund, 2019, luku 3) Suljetussa hajautuksessa täyttöasteella ei ole niin suurta merkitystä, koska linkitetty lista ei ole tilalla rajattu. Toisaalta hakeminen pitkästä listasta ei ole enää tehokasta, joten sekin halutaan pitää lyhyenä.

Törmäyksien ratkomista hankaloittaa alkioiden poistaminen taulusta. Useat menetelmät perustuvat siihen, että paikkoja käydään läpi, kunnes alkio löytyy tai vastaan tulee tyhjä paikka. Jos välistä poistetaan alkio, paikkaa ei voida jättää tyhjäksi vaan se täytyy merkitä käytetyksi. Tämä vie ylimääräistä tilaa taulusta eikä alkioiden poistaminen auta ruuhkautumiseen. (Spraul, 2015, luku 7)

# 4. RATKAISUTEKNIIKAT

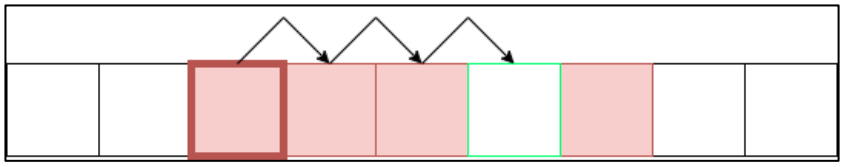
Törmäyksien ratkaisutekniikat voidaan jakaa kahteen kategoriaan eli avoimeen ja suljettuun hajautukseen. Avoimen hajautuksen tekniikat perustuvat tauluun, jossa jokaiseen paikkaan mahtuu yksi alkio. Suljetun hajautuksen tekniikat käyttävät taulua, jossa yhden paikan takaa voi löytyä useampi alkio linkitettynä listana tai taulukko-tietorakenteena. Luvussa esitetyt kuvat ovat itse tehtyjä ellei toisin ole mainittu.

## 4.1 Avoimen hajautuksen tekniikat

Avoimen hajautuksen tekniikoita on kehitetty enemmän kuin suljetun hajautuksen tekniikoita. Suuresta määrästä johtuen tässä työssä esitellään vain osa näistä. Tekniikat valikoituivat niiden tunnettavuuden, tehokkuuden ja tuoreuden perusteella. Joitakin uudempia tekniikoita jäi työssä käsittelemättä, koska niille ei ole tehty muiden tekniikoiden kanssa verrattavaa suorituskyvyn arviointia.

### 4.1.1 Lineaarinen kokeilu

*Lineaarisessa kokeilussa* (Linear probing) törmäyksen sattuessa paikkoja ryhdytään käymään lävitse niin kauan kunnes vapaa paikka löytyy tai koko taulu on käyty lävitse. Sama toistetaan avainta haettaessa, kunnes avain löytyy tai löytyy tyhjä paikka, joka on merkki siitä, että arvoa ei ole taulussa. Korkealla täyttöasteella tekniikan tehokkuus vähenee, koska arvot pakkaantuvat helposti peräkkäin ja vapaita paikkoja ei välttämättä löydy heti. Lineaarisen kokeilun asymptoottinen tehokkuus on O(n), koska täyden taulukon kanssa algoritmi käy jokaisen paikan lävitse (Yusuf et al., 2021b).



***Kuva 1.*** *Lineaarinen kokeilu tilanteessa, jossa laskettu paikka on varattu.*

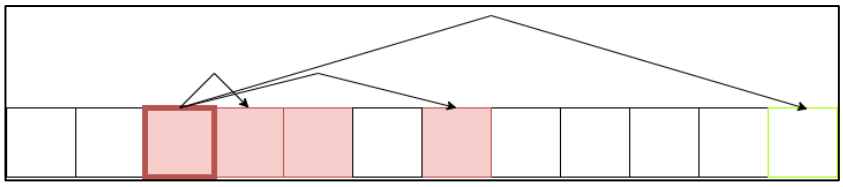
Kuvassa 1 punainen väri tarkoittaa varattua paikkaa, valkoinen vapaata, punainen reunus alkuperäistä laskettua paikkaa ja vihreä reunus merkitsee paikkaa, jolle alkio tallennetaan. Tätä merkitsemistyyliä käytetään myös muissa luvun kuvissa.

### 4.1.2 Hyppytekniikat

*Neliöllisessä kokeilussa* (Quadratic probing) sekä muissa hyppytekniikoissa törmäyksen sattuessa paikkoja ei käydä läpi peräkkäin vaan määrätyn etäisyyden mukaisesti. Neliöllisessä kokeilussa hyppyetäisyys määräytyy peräkkäisten numeroiden toisen asteen polynomien arvoista eli kokeilujärjestys on kaavan 1 mukainen:

$$i + 1^2, i + 2^2, \dots, i + n^2, \quad (1)$$

jossa i on avaimelle laskettu indeksi (Yusuf et al. 2021a).



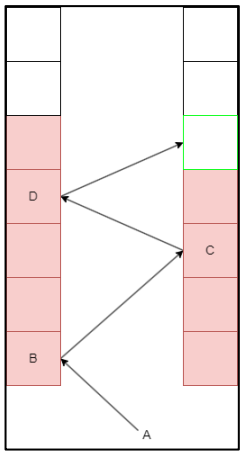
***Kuva 2.*** *Neliöllinen kokeilu tilanteessa, jossa laskettu paikka on varattu. Paikkoja kokeillaan kaavan 1 mukaisessa järjestyksessä.*

*Kaksoishajautuksessa* (Double Hashing) ensimmäinen hajautusfunktio laskee paikan avain-arvo-parille. Jos paikka on varattu, paikkoja käydään läpi toisen hajautusfunktion laskemassa järjestyksessä vapaan paikan löytymiseen saakka. (Mailund, 2019, luku 3). Hyppytekniikoiden tarkoitus on välttää arvojen kasaantuminen taulun alkuun, mutta toisaalta riskinä on, että taulu kasvaa suureksi ja täyttöaste jää pieneksi.

Mainittujen hyppytekniikoiden asymptoottinen tehokkuus on O(n) johtuen siitä, että hyppyjä jatketaan vapaan paikan löytymiseen saakka tai siihen asti, että koko taulukko on käyty läpi (Yusuf et al., 2021b).

### 4.1.3 Käkihajautus

*Käkihajautus* (Cuckoo hashing) perustuu kahden hajautustaulun käyttöön. Siinä molemmilla tauluilla on käytössä oma hajautusfunktionsa. Lisätessä alkiota varatulle paikalle, uusi alkio vie vanhan paikan ja vanha alkio siirretään seuraavaan paikkaan toisessa taulussa, jossa jälleen tuleva alkio saa paikan ja vanha alkio siirtyy taas toiseen tauluun. Alkioita siis vaihdellaan kahden taulun välillä. Tätä jatketaan, kunnes kaikilla alkioilla on paikka tai prosessi toistaa itseään ja saavutetaan iteraatioiden raja-arvo, jolloin suoritetaan uudelleenhajautus ja haetaan paikka uudestaan. (Pagh & Rodler, 2004)




***Kuva 3.*** *Käkihajautuksen paikan vaihto.*

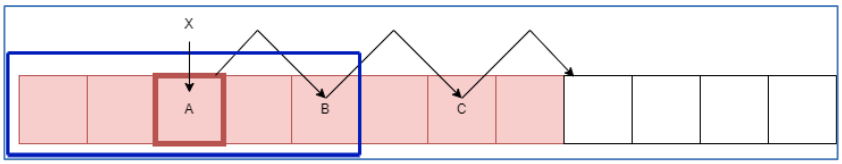
Kuvassa 3 on havainnollistettu käkihajautuksen lisäysmekanismia, jossa uusi alkio syrjäyttää vanhan. Ensin alkio A syrjäyttää alkion B, joka syrjäyttää alkion C. C vie alkion D paikan ja D löytää itselleen vapaan paikan. Käkihajautuksessa alkioiden lisääminen on huonoimmassa tapauksessa lineaarista O(n), mutta hajautuksen toimintaperiaatteen ansiosta avain löytyy vakioajassa O(1) (Yusuf et al., 2021b).

### 4.1.4 Ruutuhyppelyhajautus

*Ruutuhyppelyhajautuksessa* (Hopscotch hashing) hajautustaulu ajatellaan kokoelmana naapurustoja, jotka koostuvat yksittäisestä paikasta ja muista sen vieressä etäisyydellä N-1 olevista paikoista. Jokaisella paikalla on siis oma naapurustonsa, jonka koon taulun määrittäjä voi päättää. (Herlihy et al. 2008)

Kun alkiota yritetään sijoittaa tauluun alkuperäisen paikan ollessa varattuna, etsitään vapaata paikkaa naapuruston sisältä. Jos paikkaa ei löydy naapurustosta, etsitään taulusta seuraava vapaa paikka. Vapaasta paikasta haetaan etäisyydellä N-1 alkuperäisen indeksin ja vapaan paikan välillä olevaa alkiota, joka voidaan siirtää siihen.

Kuvassa 4 on havainnollistettu tilanne, jossa naapurusto on rajattu sinisellä, kun N on 3. Tarkoituksena on siirtää varatun paikan alkio naapuruston sisällä olevalle paikalle, jotta se löytyisi lähes yhtä nopeasti kuin jos se olisi alkuperäisellä paikallaan. Kuvan esimerkissä alkio C vaihdetaan vapaalle paikalle, jolloin vapaa paikka siirtyy C:n paikalle.



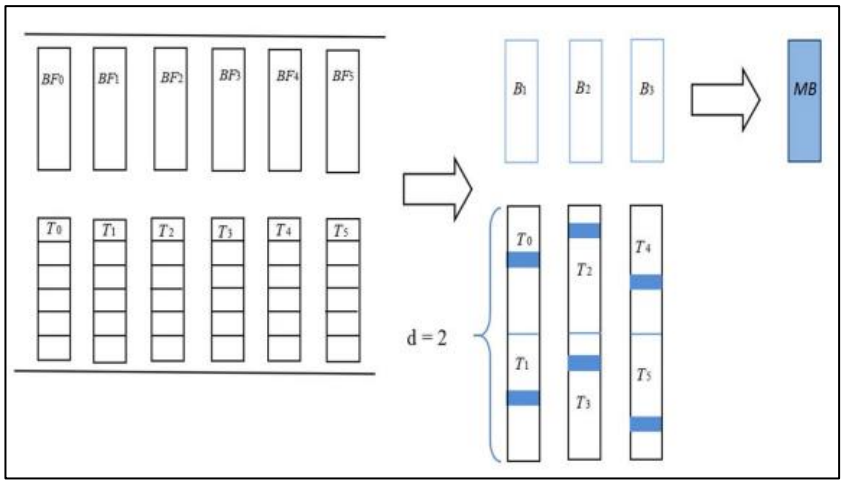

***Kuva 4.*** *Tyhjän paikan siirto lasketun paikan naapurustoon ja edelleen lasketulle paikalle. Kirjaimet kuvastavat eri alkioita ja sininen laatikko naapurustoa.*

Siirron seurauksena vapaa paikka siirtyy lähemmäs alkuperäistä indeksiä. (Herlihy et al. 2008) Näin vapaa paikka siirtyy vähitellen kohti naapurustoa, kunnes vaihtelemalla alkioiden paikkoja saadaan vapaa paikka naapuruston sisään ja siitä edelleen lasketulle paikalle.

### 4.1.5 OE-hajautus

OE-hajautus eli pariton-parillinen hajautus (Odd-Even hash algorithm) on käkihajautuksen pohjalta kehitetty algoritmi. OE-hajautus perustuu hajautustaulun ja sitä avustavan rakenteen käyttöön. Hajautustaulu koostuu parillisesta määrästä *t* saman kokoisia alitauluja, joista viimeinen taulu on linkitetty lista. Alitauluista kaksi peräkkäistä yhdistetään yhdeksi alitauluksi, jolloin lopulta alitauluja on *t*/2 verran. Jokaisessa alitaulussa on *k* määrä paikkoja. (Zhu et al. 2022)

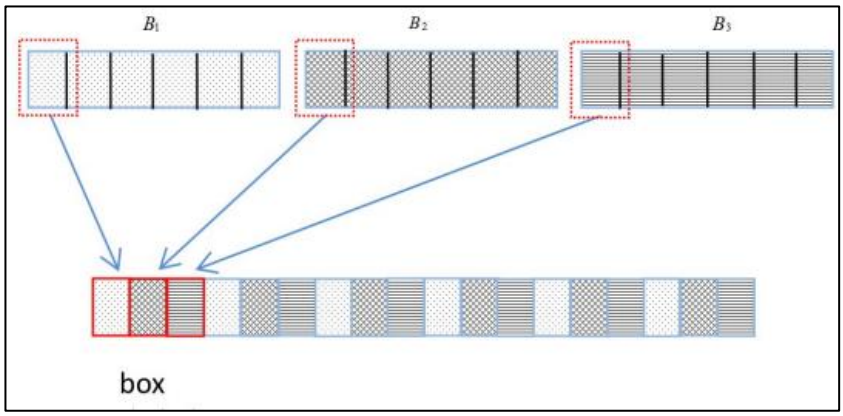
Hajautustaulun lisäksi käytössä on sama *t* määrä *Bloom-suodattimia* (Bloom filter), jotka jokainen vastaavat yhdelle alitaululle. Näille Bloom-suodattimille tehdään sama yhdistäminen eli kaksi peräkkäistä yhdistetään, jolloin niitäkin on *t*/2 määrä. Seuraavaksi suodattimet asetetaan päällekkäin, jotta saadaan yhtenäinen monibittinen Bloom-suodatin. (Zhu et al. 2022) Näin ollen kuvan 6 mukaisesti jokainen monibittisen Bloom-suodattimen paikka sisältää jokaisesta Bloom-suodatinjoukosta suodattimen. Sekä hajautusalitaulujen että Bloom-suodattimien yhdistyminen on havainnollistettu kuvassa 5.




***Kuva 5.*** *Alitaulujen $T_n$ ja Bloom-suodattimien $BF_n$ yhdistäminen monibittiseksi Bloom-suodattimeksi (MB). (Zhu et al. 2022)*

Todellisuudessa Bloom-suodattimien yhdistäminen tapahtuu laitetasolla, mutta alihajautustaulujen yhdistäminen on esitetty vain asian havainnollistamiseksi.

Jokaisella Bloom-suodatinalitaululla on oma bittikarttansa, jossa jokainen bitti vastaa yhdestä paikasta bittikarttaa vastaavasta hajautusalitaulusta. Kun bitti on 1, taulun paikka on varattu ja vastaavasti bitin ollessa 0, paikka on tyhjä.




***Kuva 6.*** *Bloom-suodattimien yhdistäminen. Jokainen monibittisen Bloom-suodattimen laatikko sisältää suodattimen jokaisesta ali-Bloom-suodattimesta. (Zhu et al. 2022)*

OE-hajautuksen nimi tulee siitä, että algoritmi päättää hajautusfunktion laskeman tiivistearvon parillisuuden perusteella, laitetaanko avain-arvo-pari parilliseen vai parittomaan alitauluun. Tämän jälkeen voidaan käyttää Bloom-suodattimen bittikarttoja ja selvittää, onko alitauluissa paikkoja vapaana. Jos vapaita paikkoja löytyy, laitetaan alkio siihen alitauluun, jossa on vapaa paikka ja matalin täyttöaste, jotta alitaulujen täyttöasteet saadaan pidettyä samansuuruisina. Jos paikkoja ei löydy, käytetään siirto-ominaisuutta, jossa otetaan talteen siirrettävä avain-arvo-pari ja laitetaan tilalle lisättävä pari. Nyt siirrettävälle alkiolle yritetään etsiä vapaata paikkaa samalla menetelmällä

bittikartan avulla. Jos tämä ei onnistu, suoritetaan siirto sokkona ja syrjäytettävä alkio siirretään viimeiseen alitauluun, joka on linkitetty lista. (Zhu et al. 2022)

Hakuoperaatiossa selvitetään ensin monibitti-Bloom-suodattimesta, minkä Bloom-suodattimen alaisuudessa arvo on. Sitten lasketaan hajautusfunktion avulla avaimen parillisuus ja selvitetään bittikartasta oikea sijainti. Poisto-operaatio hyödyntää samaa hakua, mutta lopussa arvo poistetaan ja merkintä korjataan bittikartassa takaisin tyhjäksi eli arvoon 0. (Zhu et al. 2022)

OE-hajautuksen tehokkuus perustuu Bloom-suodattimien käyttöön, koska tiedon hakeminen bittikartasta on todella nopeaa. OE-hajautuksessa viimeisen linkitetyn listan ansiosta päästään eroon hajautustauluille ominaisesta ongelmasta, jossa liian monen törmäyksen sattuessa täytyy koko taulu uudelleenhajauttaa.

## 4.2 Suljetun hajautuksen tekniikat

Suljetusta hajautuksesta ei löydy yhtä montaa variaatiota kuin avoimesta. Suljetussa hajautuksessa alkiot talletetaan indeksin osoittamaan kohtaan, vaikka siinä olisikin jo alkio. Tämä onnistuu käyttämällä sopivaa säiliöratkaisua, kuten ketjutuksessa linkitettyä listaa tai merkkijonojen tapauksessa käytössä voi olla taulukko-tietorakenne.

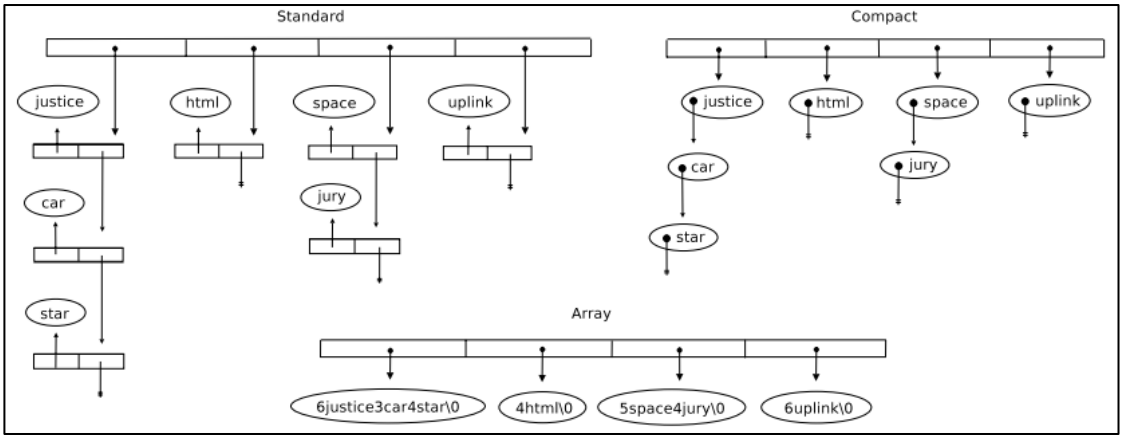
### 4.2.1 Erillisketjutus

*Erillisketjutuksessa* (Separate chaining) luodaan jokaiselle taulun paikalle linkitetty lista tietorakenteeksi. Kun törmäys tapahtuu, lisätään avain-arvo-pari listan jatkoksi. Tämä lähestymistapa on tehokas, kunhan vain listat eivät kasva liian pitkiksi. Ratkaisu liian pitkiin listoihin on taulun koon kasvattaminen ja avainten uudelleenhajautus. (Mailund, 2019, luku 3)

Linkitetyn listan voi toteuttaa tavallisessa muodossa eli siten, että jokainen solmu listassa sisältää osoittimen arvoon ja seuraavaan solmuun. Listan voi myös toteuttaa kompaktisti eli siten, että jokainen arvo on tallennettu suoraan solmuun. Tämä on muistinkäytön kannalta tehokkaampi ratkaisu (Askitis & Zobel, 2005). Asymptoottinen tehokkuus lisättäessä ja haettaessa alkioita linkitetystä listasta on O(n), koska huonolla hajautuksella alkiot päätyvät samaan listaan kaikki ja ne joudutaan käymään lävitse. Toisaalta listan käydessä liian pitkäksi, sen uudelleenhajautus nopeuttaa alkioiden hakemisen lähemmäs vakioaikaisuutta.

### 4.2.2 Taulukkorakenne

Erityisesti tekstisyötettä hajauttaessa Askitis ja Zobel (2005) esittävät linkitetyn listan sijaan käytettäväksi *taulukkorakennetta* (array). Kun taulukossa tapahtuu törmäys, uusi alkio lisätään samalle paikalle edellisten perään osaksi samaa merkkijonoa (Askitis & Zobel, 2005). Huono puoli taulukkoratkaisulla on se, että uusia alkioita lisättäessä taulukon koko täytyy määritellä aina uudelleen. Kuitenkin taulukkorakenne on tehokkuudeltaan linkitetyn listan kanssa yhtäläinen, mutta se kuluttaa vähemmän muistia.




***Kuva 7.*** *Tavallinen ketjutusratkaisu, kompaktiketjutus ja taulukkorakenne. (Askitis & Zobel, 2005)*

# 5. SUORITUSKYVYN ARVIOINTI

Uusia menetelmiä esittelevillä lähdeteksteillä on tapana tehdä aiempien menetelmien kanssa suorituskykyvertailuja, joissa vertaillaan esimerkiksi välimuistihutien (cache miss) tai tapahtuvien törmäysten määrää. Vertailussa käytettyjä mittareita ovat myös muistinkulutus ja operaatioiden suoritusnopeus.

Tarkkoja arvoja eri kokeiden välillä ei kannata vertailla, sillä koetilanteessa käytetty laitteisto vaikuttaa tulokseen. Kokeissa tärkeää on havaita jokaisen menetelmän tehokkuus suhteessa muihin menetelmiin ja vertailla sitä muiden vastaavien kokeiden kesken.

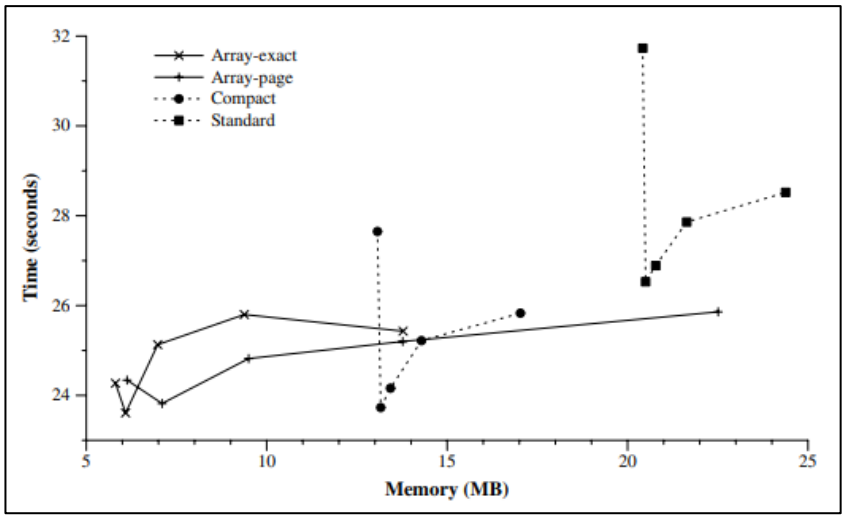
## 5.1 Merkkijonotaulut

Askitis ja Zobel (2005) tekivät kokeita merkkijonoja sisältäville datasarjoille, joita olivat sarja sanoja, url-osoitteita ja uniikkeja sanoja. Heidän tavoitteena oli selvittää välimuistin kannalta tehokkain algoritmi törmäyksien ratkaisemiseksi. Työssään he tutkivat pelkästään suljetun hajautuksen tekniikoita, sillä taulun täyttöasteen lähentyessä yhtä, avoin hajautus ei ole enää riittävän tehokas vaihtoehto.

Työssä vertailtiin tavallista ketjutusta, kompaktia ketjutusta sekä rinnakkaista taulukko-rakennetta (kuva 7). Rinnakkaisesta taulukkoratkaisusta käytettiin kahta vaihtoehtoa: täsmällistä (exact fit), jossa jokaista paikkaa laajennetaan vain tarvittava määrä sekä sivullistamista (paging), jossa paikkoja laajennetaan 64 bitin kertoimilla.

Kuvasta 8 voidaan nähdä nopeuden ja muistinkulutuksen suhde eri tekniikoilla sekä paikkojen määrillä luodessa hajautustaulua. Kuvassa jokaisella tekniikalla yksi piste vastaa yhtä paikkamäärää. Arvoja paikkojen määrille ovat: 10 000, 31 622, 100 000, 316 228 ja 1 000 000, ja ne ovat jakautuneet kuvaajassa vasemmalta oikealle. (Askitis & Zobel, 2005)

Paikkojen määrän ollessa liian pieni tai liian suuri, aikaa kuluu paljon, kuten kuva 8 havainnollistaa. Toisaalta, jos paikkoja on paljon, kuluu myös muistia runsaasti. Syy sille miksi pienellä määrällä paikkoja aikaa kuluu paljon on nähtävissä taulukossa 1, joka ilmaisee välimuistihutien määrää suhteessa käytettyjen paikkojen määrään. Siitä voidaan huomata, että kompaktilla ja tavallisella ketjutuksella tapahtuu eniten välimuistihuteja pienimmällä paikkamäärällä. Rinnakkaisella taulukolla niitä tapahtuu harvemmin.




***Kuva 8.*** *Algoritmien muistinkäyttö tavallisella sanalistalla. (Askitis & Zobel, 2005)*

Taulukosta 1 nähdään välimuistihutien määrä eri kokoisilla tauluilla taulujen luonnin ja hakujen aikana. Pienemmillä tauluilla taulukkoratkaisu on selkeästi parempi kuin linkitetyt listat, mutta suuremmilla kompakti linkitetty lista tuottaa vähemmän huteja. Toisaalta ero on pieni ja kuvan 8 mukaisesti taulukko pärjää kuitenkin paremmin yleisellä tasolla.

Taulukko 1. *Välimuistihutien määrä tavallisella sanalistalla (Askitis & Zobel, 2005)*

| Paikkojen määrä | Taulukkorakenne | Kompakti ketjutus | Tavallinen ketjutus |
|---|---|---|---|
| 10 000 | 68 506 659 | 146 375 946 | 205 795 945 |
| 100 000 | 89 568 430 | 80 612 383 | 105 583 144 |
| 1 000 000 | 102 474 094 | 94 079 042 | 114 275 513 |

Lopputuloksena Askitis ja Zobel kokosivat taulukkoon 2 hajautustaulun muistinkäytön, joka koostuu merkkijonoille, osoittimille, paikoille ja yleisrasitteelle varatusta tilasta. Taulukossa on mitattuna muistinkäyttö (MB) jokaiselle datasarjalle jokaisella tekniikalla ja kahdella eri paikkamäärällä. Tuloksista voidaan havaita, että taulukkorakenne on muistinkäytöltään tehokkain ja erityisesti täsmällinen taulukko kuluttaa vähiten tilaa. Toisaalta kuvasta 8 voidaan havaita, että sivullistettu taulukko on kuitenkin nopeampi, vaikka se viekin tilaa.

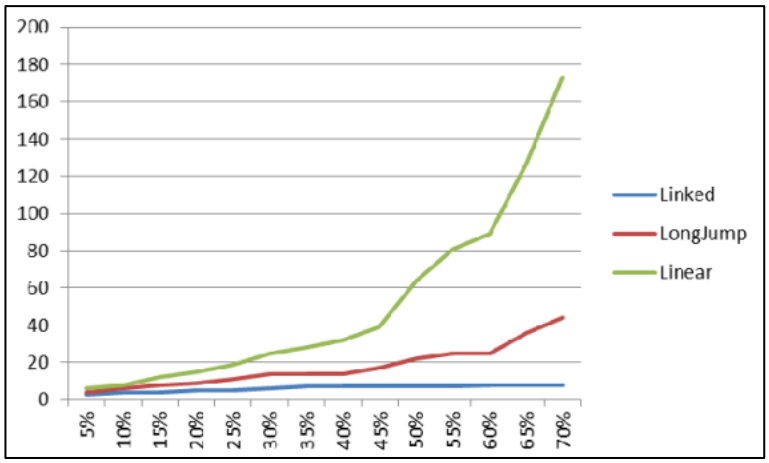
Taulukko 2. *Muistinkäyttö (MB) (Askitis & Zobel, 2005)*

| Käytetyt sanalistat | Sanat | | URL-osoitteet | | Uniikit arvot | | |
|---|---|---|---|---|---|---|---|
| Sanamäärät | 10 000 | 1 000 000 | 10 000 | 1 000 000 | 10 000 | 1 000 000 | 10 000 000 |
| Täsmällinen taulukko | 5.81 | 13.78 | 36.77 | 57.16 | 205.52 | 218.39 | 322.64 |
| Sivullistettu taulukko | 6.13 | 22.51 | 47.04 | 72.94 | 205.83 | 249.66 | 463.29 |
| kompakti ketjutus | 13.07 | 17.03 | 62.16 | 66.12 | 445.43 | 449.39 | 485.39 |
| tavallinen ketjutus | 20.42 | 24.38 | 77.64 | 81.60 | 685.43 | 689.39 | 725.39 |

## 5.2 Suuret datamäärät

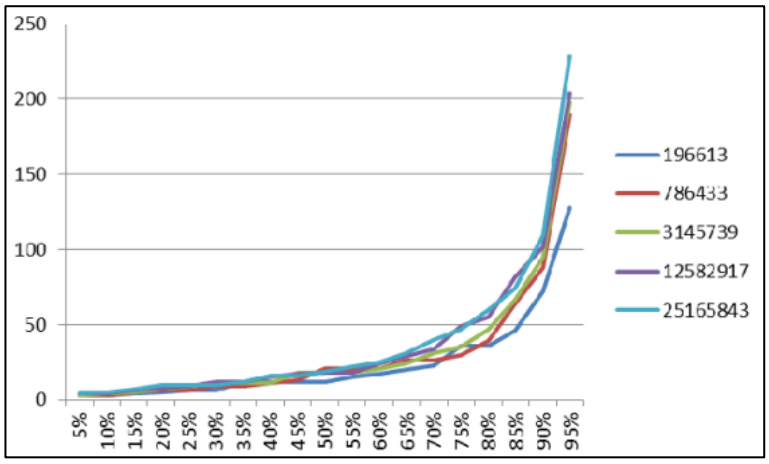
Liu ja Xu (2015) tutkivat empiirisesti törmäysten ratkaisutekniikoita suurten datamäärien näkökulmasta. Ratkaisutekniikoiden vertailussa olivat mukana suljetusta hajautuksesta linkitetty lista ja avoimesta lineaarinen kokeilu sekä hyppytekniikka, jossa hypyn pituus riippuu taulun koosta.

Kuvasta 9 voidaan havaita, että samalla kun täyttöaste kasvaa, myös avoimen hajautuksen tekniikoiden törmäyksien määrä kasvaa. Linkitetyllä listalla ei ole samanlaista ongelmaa.

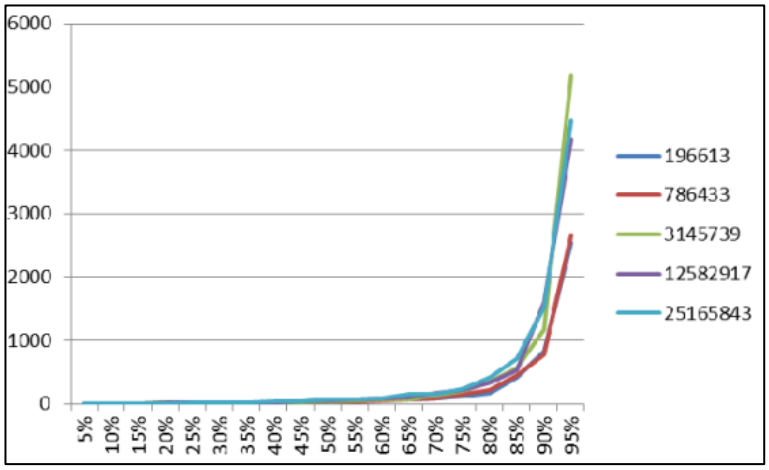



***Kuva 9.*** *Eri algoritmien törmäyksien määrä (y-akseli) täyttöasteen (x-akseli) suhteen taulun koolla 100 663 319. (Liu & Xu, 2015)*

Kuvista 10 ja 11 voidaan havaita, että kaikilla taulukon kooilla noin 70 % täyttöasteen kohdalla avoimen hajautuksen tekniikoilla tapahtuu jo niin paljon törmäyksiä, ettei niitä voida pitää käyttökelpoisina suuremmilla täyttöasteen arvoilla. Toisaalta pienemmillä täyttöasteen arvoilla lineaarisen kokeilun ja hyppytekniikkan törmäyksien määrä on melko pieni.




***Kuva 10.*** *Hyppytekniikan törmäyksien määrä (y-akseli) täyttöasteen (x-akseli) suhteen eri taulun kooilla. (Liu & Xu, 2015)*

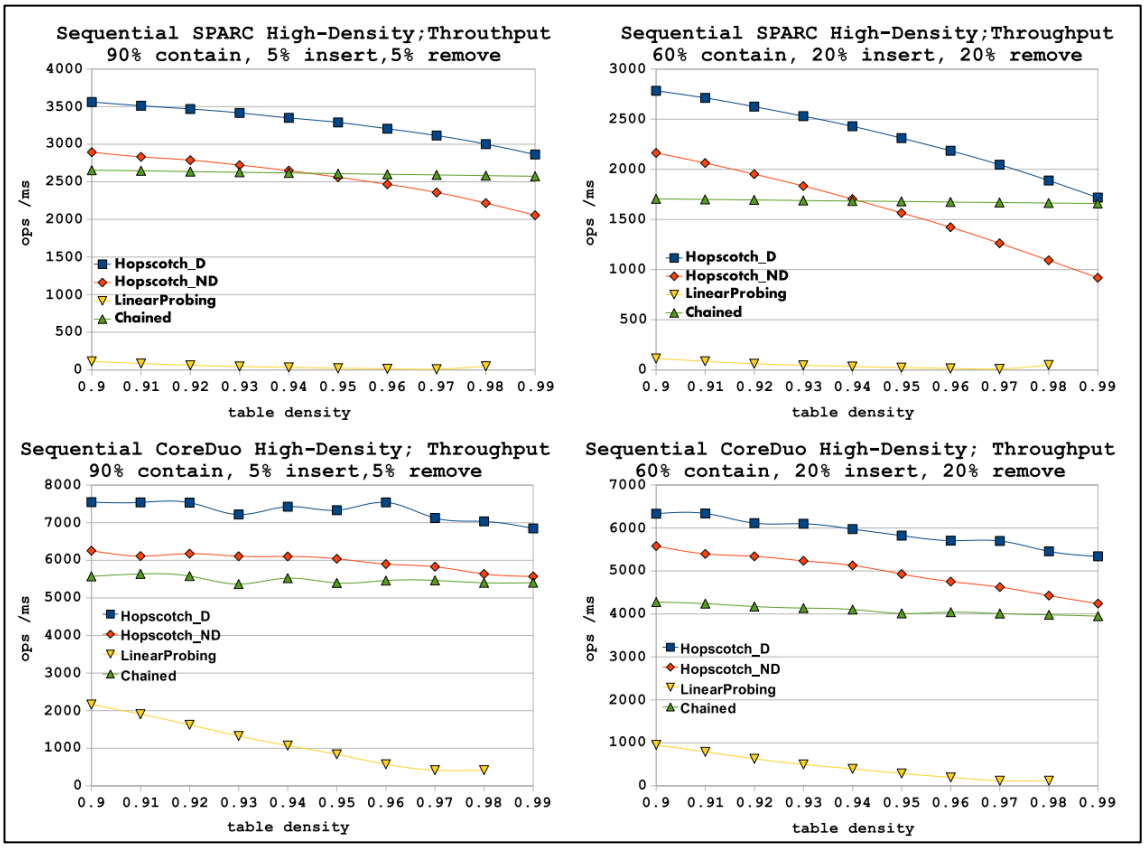



***Kuva 11.*** *Lineaarisen kokeilun törmäyksien määrä (y-akseli) täyttöasteen (x-akseli) suhteen eri taulun kooilla. (Liu & Xu, 2015)*

## 5.3 Ruutuhyppelyhajautuksen tulokset

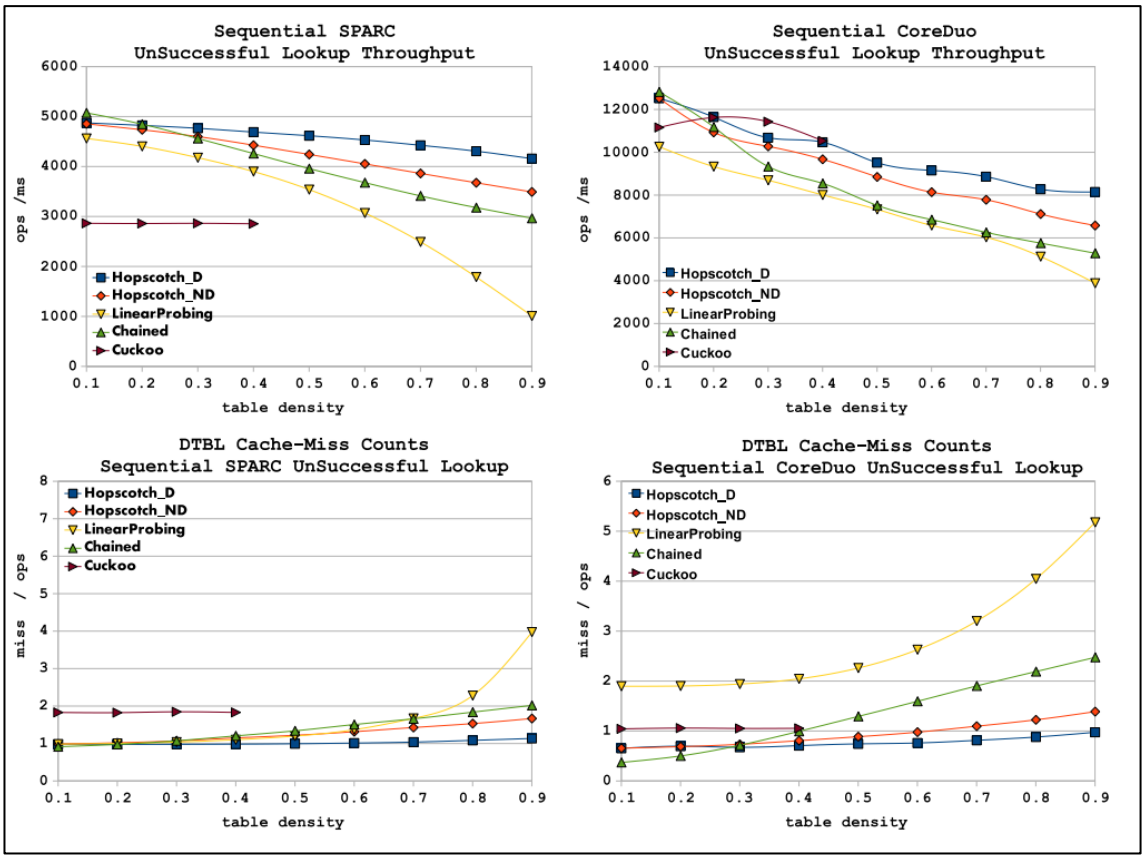
Herlihy et al. (2008) suorittivat työssään tehokkuustestejä hajautustauluille ottaen huomioon myös rinnakkaisuuden. Tässä työssä ei keskitytä rinnakkaisuutta hyödyntävien taulujen tuloksiin. Ruutuhyppelyhajautusta verrattiin käkihajautukseen, ketjutukseen ja lineaariseen kokeiluun. Kokeissa käytetty data koostuu $2^{23}$ kokoisesta alkiojoukosta, joille tehtiin 10 toistoa. Kokeet suoritettiin kahdella eri prosessorilla: Sparc

ja CoreDuo. Ruutuhyppelyhajautuksesta kokeiltiin kahta versiota: Hopscotch_D ja Hopscotch_ND, joista D versio siirtää alkioita niin, että ne ovat linjassa välimuistilohkon kanssa ja ND ei.




***Kuva 12.*** *Tuloksia korkeilla täyttöasteilla kahdella eri suorittimella ja hajautustaulun operaatioiden eri suhteilla. (herlihy et al. 2008)*

Kuvasta 12 voidaan nähdä, että lineaarinen kokeilu ei ole hyvä ratkaisu korkealla täyttöasteella. Tämä on linjassa myös aiempien tulosten kanssa. Molemmat versiot ruutuhyppelyalgoritmista ovat parempia kuin ketjutettu hajautus lähes loppuun saakka. Noin 95 % ja siitä korkeammilla täyttöasteilla ketjutus päihittää ND version ruutuhyppelystä. Toisaalta näin korkeita täyttöasteita harvemmin saavutetaan, joten ruutuhyppelyn voidaan todeta suoriutuvan vertailtavista tekniikoista parhaiten. Käkihajautus ei toimi läheskään näin korkeilla täyttöasteilla, joten sitä ei ole mukana näissä vertailuissa.



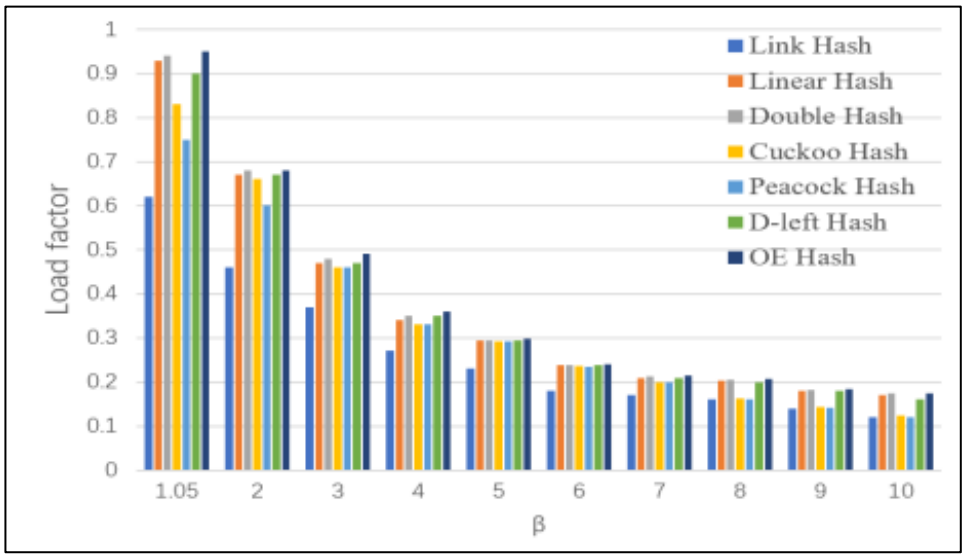

***Kuva 13.*** *Epäonnistuneiden hakujen sekä välimuistihutien määrä kahdella eri suorittimella. (herlihy et al. 2008)*

Eri ratkaisualgoritmien suoritusteho epäonnistuneiden hakujen suhteen on melko samaa tasoa kuin välimuistihutien määrä täyttöasteen kasvaessa. Kuvan 13 käyrissä voi myös havaita korrelaatiota keskenään. Samalla kun ylemmissä kuvaajissa suoritustehoa mittaavat käyrät laskevat, alemmissa kuvaajissa välimuistihutien määrät nousevat. Ruutuhyppelyhajautus säilyttää tehokkuutensa kokonaisuudessaan parhaiten molemmilla variaatioilla ja niillä on myös vähiten välimuistihuteja.

## 5.4 Pariton-parillinen hajautuksen tulokset

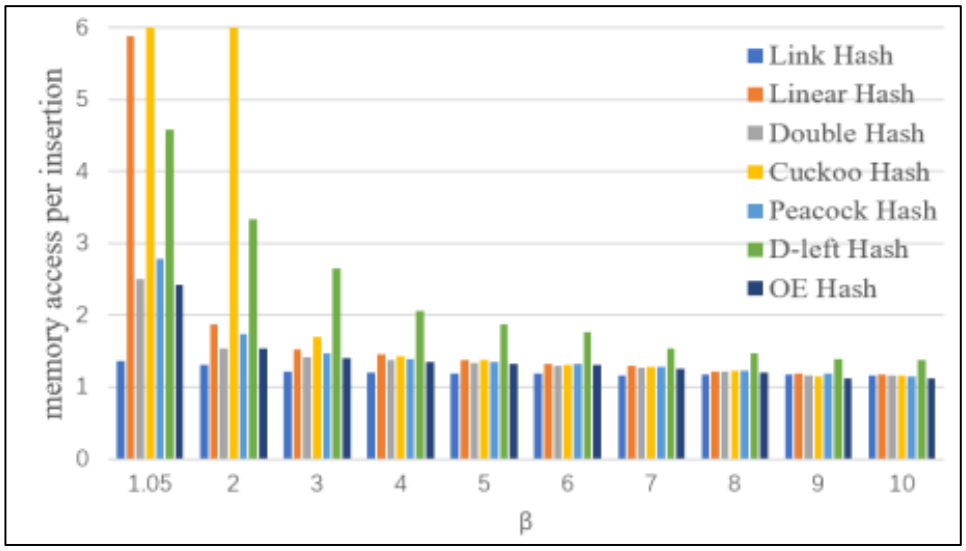
Zhu et al. (2022) esittelivät työssään käkihajautuksesta kehitettyä versiota, jossa hyödynnetään käkihajautuksessa esitettyä siirto-ominaisuutta. Työssä vertailtiin pariton-parillinen hajautuksen eli OE-hajautuksen tehokkuutta muiden yleisten törmäyksien ratkaisualgoritmien kanssa. Kuvaajissa on yleisesti käytössä muuttuja β, joka merkitsee suhdetta kaikissa alitauluissa olevien vapaiden paikkojen määrän ja lisättävien arvojen määrän välillä. Kokeet suoritettiin kahdella eri datasarjalla: 80 000 ja 480 000 alkiolla, mutta koska tulokset olivat datasarjasta riippumatta samanlaisia, esittelen vain toisella datasarjoista saadut tulokset.

Kuvassa 14 nähdään OE-hajautuksen saavuttavan eri β:n arvoilla parhaimman täyttöasteen. Korkeita täyttöasteita saavuttavat myös lineaarinen, tupla sekä D-vasen hajautus. Toisaalta korkea täyttöaste ei ole välttämättä hyvä asia, koska paikan löytäminen tai hakeminen voi kestää pidempään taulun ollessa täysi.



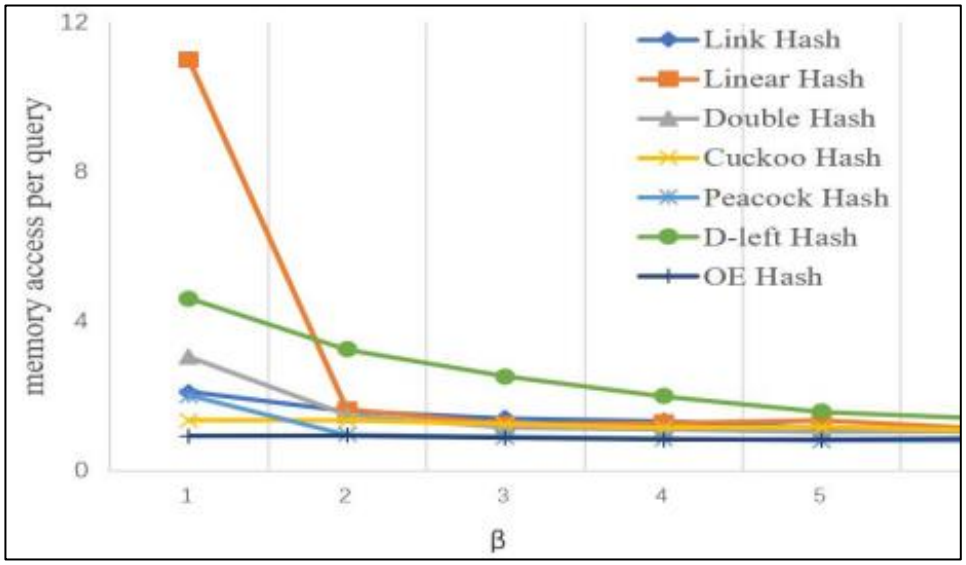

***Kuva 14.*** *Hajautustaulun täyttöaste β:n eri arvoilla. (Zhu et al. 2022)*

Kuvassa 15 esitetään eri algoritmien suoriutuminen alkion lisäämisestä vertaamalla muistinhakujen määrää suhteessa β:n. Kuvasta nähdään, että suurimmat määrät muistihakuja on samoilla algoritmeilla, kuin kuvan 14 korkeimman täyttöasteen omaavilla algoritmeilla. Poikkeuksena tähän on OE-hajautus, jolla on toisiksi pienin määrä muistihakuja heti linkitetyn listan jälkeen. Kuvassa 15 voidaan havaita tulosten tasoittumista β:n kasvaessa, koska täyttöaste pienenee kuvan 14 mukaisesti, jolloin pienempää täyttöastetta suosivat avoimen hajautuksen tekniikat pärjäävät myös.




***Kuva 15.*** *Muistihakujen määrä lisättäessä tauluun β:n eri arvoilla. (Zhu et al. 2022)*

Kuva 16 esittää muistihakujen määrää alkion hakemisen aikana. Siinä selkeimmät erot tapahtuvat, kun vapaiden paikkojen ja alkioiden suhde β on 1. Tällöin OE-hajautus pärjää parhaiten, mutta erot tasoittuvat nopeasti β:n kasvaessa.




***Kuva 16.*** *Muistihakujen määrä etsiessä taulusta β:n eri arvoilla. (Zhu et al. 2022)*

OE-hajautus on tehokkuutensa vuoksi lupaava algoritmi, joka pystyy saavuttamaan suuren täyttöasteen säilyttäen samalla suorituskykynsä.

# 6. TULOSANALYYSI

Tämän luvun tarkoituksena on koota sekä analysoida empiiristen tutkimusten tuloksista sekä kirjallisuuskatsauksesta saatua tietoa. Aluksi käydään läpi eri algoritmien suoriutuminen eri kokeiden välillä, jonka jälkeen on yleistä pohdintaa.

Lineaarinen kokeilu on useassa lähteessä mukana vertailuissa, vaikka sen suoriutuminen on jokaisessa kokeessa ollut huonoin kaikista. Syy tälle mitä luultavimmin on se, että sitä voidaan pitää vertailukohtana eli kuinka algoritmi x suoriutuu suhteessa lineaariseen kokeiluun. Kokeista voidaan havaita lineaarisen kokeilun osalta kirjallisuudessa mainittu noin 70% täyttöasteen raja (herlihy et al., 2008; Liu & Xu, 2015). Siis raja, jonka jälkeen lineaarinen kokeilu muuttuu käyttökelvottomaksi. Sama tulos on myös havaittavissa Mailundin (2019, luku 3) kokeissa.

Hyppytekniikoita ei ole mukana kuin vain Liun ja Xun (2015) kokeessa LongJump nimellä ja siinäkin se pärjäsi vain lineaarista kokeilua paremmin. Voidaan siis päätellä, että hyppytekniikat sinällään eivät ole parhaita ratkaisuja, mutta niiden perusperiaate on kuitenkin käytössä tehokkaammissakin ratkaisuissa.

Käkihajautus on tehokas algoritmi pienillä täyttöasteilla. Herlihy et al. (2008) sekä Zhu et al. (2022) molemmat sisällyttävät käkihajautuksen kokeisiinsa, mutta kummassakaan hajautus ei erottaudu edukseen. Käkihajautuksen muistihakujen määrä verrattuna muihin on korkea, eikä se suoriudu korkeilla täyttöasteilla ollenkaan. Lisäksi käkihajautuksella tapahtuu välimuistihuteja selvästi helpommin kuin muilla hajautuksilla lineaarista hajautusta lukuunottamatta.

Ruutuhyppelyhajautuksesta ei ole muuta tutkimusta tehty kuin itse hajautuksen tekijöiden Herlihy et al. (2008) oma tutkimus. Siinä ruutuhyppelyhajautus pärjää reilusti paremmin kuin muut tutkittavat algoritmit sekä korkeilla että matalilla täyttöasteilla, vaikka ketjutettu hajautus on lähellä ruutuhyppelyn suorituskykyä aivan korkeimmilla 0,95-0,99 samoin kuin matalimmilla 0,1-0,2 täyttöasteilla.

OE-hajautus on niin tuore algoritmi, että siitä ei ole tehty muuta tutkimusta. Silti Zhu et al. (2022) tekemät omat tutkimukset lupaavat hyvää algoritmin tehokkuuden kannalta, vaikka tutkimus ei olekaan vertaisarvioitu. Olennaista OE-hajautuksessa on havaita suuri täyttöaste ja muistihakujen määrän vähyys, mikä puolestaan kasvattaa suorituskykyä. Alkion lisäys pitäisi olla aina mahdollista, minkä ansiosta OE-hajautus ei tarvitse uudelleenhajautusta.

Erillisketjutus on mukana jokaisessa tutkimuksessa vertailukohtana, koska se on yksi harvoista sekä yleisimmin käytetty suljetun hajautuksen tekniikka. Se on myös tehokkuudeltaan kilpailukykyinen. Liu ja Xu (2015) näyttivät kokeissaan, että linkitetyn listan käyttö päihitti selvästi lineaarisen kokeilun sekä hyppytekniikan. Herlihy et al. (2008) saivat saman tuloksen samoin kuin Zhu et al. (2022). Toisaalta erillisketjutus ei suoriudu paremmin kuin ruutuhyppelyhajautus tai OE-hajautus, mutta se pärjää kuitenkin muita algoritmeja paremmin.

Erillisketjutuksen kilpailija taulukkorakenne on Askitiksen ja Zobelin (2005) tutkimusten mukaan tehokkaampi vaihtoehto merkkijonosyötteitä käyttäessä. Aiheesta ei muuta tutkimusta löydy, mutta linkitettyyn listaan verrattaessa taulukkoratkaisu vie vähemmän tilaa sekä vähemmän aikaa. Toisaalta suuremmilla taulun kooilla erot pienenevät.

Yleinen ongelma eri algoritmeilla on taulujen väliin jäävät tyhjät paikat samoin kuin tarve uudelleenhajautukselle, kun paikkaa ei vain löydy (Yusuf et al. 2021a). Tässä OE-hajautus on vahva vaihtoehto taulun tasaisen täyttymisen ja uudelleenhajautuksen tarpeen puuttumisen ansiosta.

Tuloksista voidaan johtaa se, että avoimen hajautuksen tekniikoista vain Ruutuhyppelyhajautus ja OE-hajautus ovat käyttökelpoisia ja jopa parempia verrattuna linkitetyn listan käyttämiseen. Taulukkorakenne on merkkijonosyötteiden tapauksessa myös kilpailukykyinen vaihtoehto.

Keskenään näitä algoritmeja on vaikea vertailla, koska eri tutkimuksissa on erilaiset tavat mitata tehokkuutta. Tutkimuksissa tehokkuutta mitataan muistihakujen määrällä, muistin käytöllä, törmäyksien määrällä ja operaatioiden määrällä millisekunnissa. Parempia tuloksia saisi, jos käytössä olisi samanlaiset mittaukset tutkimuksesta riippumatta. Tämä herättää kysymyksen siitä, ovatko mittaustavat valittu sen takia, että oma algoritmi ei välttämättä pärjääkään yhtä hyvin toisilla mittareilla.

Törmäysten ratkaisutekniikoista eri ohjelmointikielien hajautustaulutoteutuksissa on käytössä erillisketjutus C++ unordered_map -tietorakenteessa sekä Javan HashMap-tietorakenteessa (cppreference, 2023; javarevisited, 2023). Pythonin dict-tietorakenteessa käytetään avointa hajautusta (python, 2023). Eri ohjelmointikielien välillä ei siis ole selkeästi yhtenäistä linjaa käytettyjen tekniikoiden suhteen.

# 7. YHTEENVETO

Tässä työssä tutkittiin hajautustaulujen törmäyksien ratkaisualgoritmeja ja niiden tehokkuutta. Tavoitteena oli tutkia eri tekniikoita ja selvittää, oliko jokin niistä parempi kuin muut. Tekniikat olivat rajattu koskemaan ainoastaan hajautustauluja ilman rinnakkaisuutta. On kuitenkin syytä huomioida, että yhä useammin ratkaisuja haetaan myös rinnakkaisille toteutuksille. Silti tutkimusta tehdään myös yksittäisen säikeen ratkaisuille.

Vertailu eri algoritmien välillä ei ole helppoa, koska tehokkuus on monen eri osa-alueen vaikutuksen summa ja eri tutkimukset painottivat kokeissaan eri asioita, kuten muistinkulutusta, suoritusnopeutta ja välimuistihutien määrää.

Työn tuloksena saatiin selville, että yhtä ainoaa toimivaa ratkaisua ei löytynyt. Työssä onnistuttiin kuitenkin erottamaan muutamia algoritmeja, jotka toimivat paremmin kuin muut. Näihin lukeutui yleisesti datajoukkoja käsitellessä Ruutuhyppelyhajautus, erillisketjutus ja OE-hajautus. Taulukkorakenne erottautui edukseen tekstisyötteitä käsitellessä.

Seuraavana vaiheena hajautustaulujen törmäysten ratkaisutekniikoiden tutkimuksessa voisi suorittaa valittujen algoritmien kesken tehokkuutta mittaavia kokeita, tavoitteena löytää toimivin ratkaisu. Kannattaa myös huomata, että OE-hajautusta ei ole tuoreena algoritmina vielä vertaisarvioitu, mikä lisää tarvetta jatkotutkimukselle.

Aihealueena hajautusfunktioiden törmäykset ovat paljon tutkittu alue, mutta tutkimusta tehdään jatkuvasti tavoitteena löytää vieläkin tehokkaampia ratkaisuja. Motivaationa tälle on hajautustaulujen laajakäyttöisyys tietotekniikan eri osa-alueilla. Vanhojen löydöksien pohjalta kehitetään vieläkin tehokkaampia ratkaisuja hajautustauluille rinnakkaisuudella tai ilman. Esimerkiksi tässä työssä käsitelty OE-hajautus on kehitetty ja Ruutuhyppelyhajautus on ottanut vaikutteita käkihajautuksen pohjalta.

# LÄHTEET


Askitis, N., & Zobel, J. (2005). Cache-Conscious Collision Resolution in String Hash Tables. Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 3772, 91–102. https://doi.org/10.1007/11575832_11

Cppreference. (2023, viitattu 6.4.2023.). Std::unordered_map. https://en.cppreference.com/w/cpp/container/unordered_map

Herlihy, M., Shavit, N., & Tzafrir, M. (2008). Hopscotch Hashing. Distributed Computing, 5218, 350–364. https://doi.org/10.1007/978-3-540-87779-0_24

Javarevisited. (-, viitattu 6.4.2023.). How does Java HashMap or LinkedHashMap handles collisions?. https://javarevisited.blogspot.com/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html#axzz7y0psn3gm

Liu, D., & Xu, S. (2015). Comparison of hash table performance with open addressing and closed addressing: An empirical study. The International Journal of Networked and Distributed Computing (Online), 3(1), 60–68. https://doi.org/10.2991/ijndc.2015.3.1.7

Mailund, T. (2019). The joys of hashing hash table programming with C (1st ed. 2019.). Apress. https://doi.org/10.1007/978-1-4842-4066-3

OpenDSA. (-, viitattu 1.3.2023.). CS3 Data Structures & Algorithms. https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/index.html

Pagh, R., & Rodler, F. F. (2004). Cuckoo hashing. Journal of Algorithms, 51(2), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

Python. (2012, viitattu 6.4.2023.). CPython mercurial repositories. https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l296

Spraul, V. A. (2015). How software works: the magic behind encryption, CGI, search engines, and other everyday technologies (1st edition). No Starch Press.

Tapia-Fernández, S., García-García, D., & García-Hernandez, P. (2022). Key Concepts, Weakness and Benchmark on Hash Table Data Structures. Algorithms, 15(3), 100–. https://doi.org/10.3390/a15030100

Yusuf, A. D., Abdullahi, S., Boukar, M. M., & Yusuf, S. I. (2021a). Collision Resolution Techniques in Hash Table: A Review. International Journal of Advanced Computer Science and Applications, 12(9), 757–762. https://doi.org/10.14569/IJACSA.2021.0120984

Yusuf, A. D., Abdullahi, S., Boukar, M. M., & Yusuf, S. I. (2021b). Evaluation of Collision Resolution Methods Using Asymptotic Analysis. 2021 16th International Conference on Electronics Computer and Computation (ICECCO), 1–6. https://doi.org/10.1109/ICECCO53203.2021.9663778

Zhu, H., Wan, J., Li, N., Deng, Y., He, G., Guo, J., & Zhang, L. (2022). Odd-Even Hash Algorithm: A Improvement of Cuckoo Hash Algorithm. 2021 Ninth International Conference on Advanced Cloud and Big Data (CBD), 1–6. https://doi.org/10.1109/CBD54617.2021.00010