Samuel Kaiponen

Deduplikoinnin suorituskyvystä

Tietotekniikan pro gradu -tutkielma

19. kesäkuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Samuel Kaiponen

Yhteystiedot: samuel.m.a.kaiponen@student.jyu.fi

Ohjaaja: Antti Valmari

Työn nimi: Deduplikoinnin suorituskyvystä

Title in English: On the Performance of Deduplication

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 75+0

Tiivistelmä: Deduplikointi säästää tallennustilaa. Siinä etsitään datasta identtisiä alueita, joista yksi säilytetään ja loput korvataan viitteellä tähän säilytettävään alueeseen. Tässä tutkielmassa käsiteltiin kirjallisuuteen perustuen deduplikoinnin eri osa-alueita. Erityistä huomiota kiinnitettiin deduplikoinnin suorituskykyyn ja sen parantamiseen. Katsauksessa selvisi, että deduplikoinnin moninaisiin sovelluskohteisiin tarvitaan hyvin erilaisia deduplikointijärjestelmiä. Niissä tasapainoillaan suorituskyvyn eri alueiden välillä: yhden alueen parantaminen heikentää usein toista. Työssä toteutettiin myös tietokoneohjelma, joka deduplikoi tiedostoja. Sen suoritusaikoja mitattiin kahden muuttujan eri arvoilla. Mittauksissa löydettiin muuttujille arvot, joilla suoritusaika oli yleisesti pienin.

Avainsanat: deduplikointi, suorituskyky, tallennustila, tiiviste

Abstract: Deduplication saves storage space. In deduplication, data is searched for identical sections. One of these sections is stored and the rest are replaced with a reference pointing to the stored section. In this study, various aspects of deduplication were examined based on the literature. Special attention was given to the performance of deduplication and its improvement. In the review it was found that the diverse applications of deduplication require very different deduplication systems. The systems have to balance between the many aspects of performance: improving one aspect often weakens another. A computer program that deduplicates files was also implemented in this work. Its execution times were measured with

different values of two variables. Values were found with which the program's execution times were generally the lowest.

Keywords: deduplication, performance, storage, hash

Termiluettelo

Ensisijainen tallennustila	Tallennustila, joka on aktiivisessa käytössä. Esimerkiksi käyttäjän kotihakemiston sisältävä levy.
Toissijainen tallennustila	Tallennustila, joka ei ole aktiivisessa käytössä. Esimerkiksi levy, joka sisältää vain varmuuskopioita.
Välitön deduplikointi	Deduplikointi, joka suoritetaan kirjoitusoperaation yhteydessä ennen kuin data kirjoitetaan levylle.
Jälkikäteinen deduplikointi	Deduplikointi, joka suoritetaan datalle, joka on jo kirjoitettu levylle.
Ajallinen paikallisuus	Viitattuun osoitteeseen viitataan usein pian uudestaan.
Avaruudellinen paikallisuus	Lähekkäisiin osoitteisiin viitataan usein peräkkäin.
Hajautusfunktion törmäys	Kaksi erilaista hajautusfunktion syötettä kuvautuu samaksi tiivisteeksi.
i-solmu	Joissakin tiedostojärjestelmissä käytetty tietorakenne, joka sisältää tiedoston metatiedot ja osoittimet sen sisältöön.
Symbolinen linkki	Tiedostotyyppi, jonka i-solmu sisältää linkitetyn tiedoston polkunimen.
Kova linkki	Hakemistomerkintä, joka yhdistää tiedoston nimen ja i-solmun. Yhteen i-solmuun voidaan yhdistää useampi nimi.
Reflink	Kun tiedostoon luodaan reflink, luodaan uusi i-solmu, joka osoittaa alkuperäisen tiedoston datalohkoihin. Jos jommankumman tiedoston datalohkoja muokataan, muuttuneille datalohkoille varataan uutta tilaa.

Kuviot

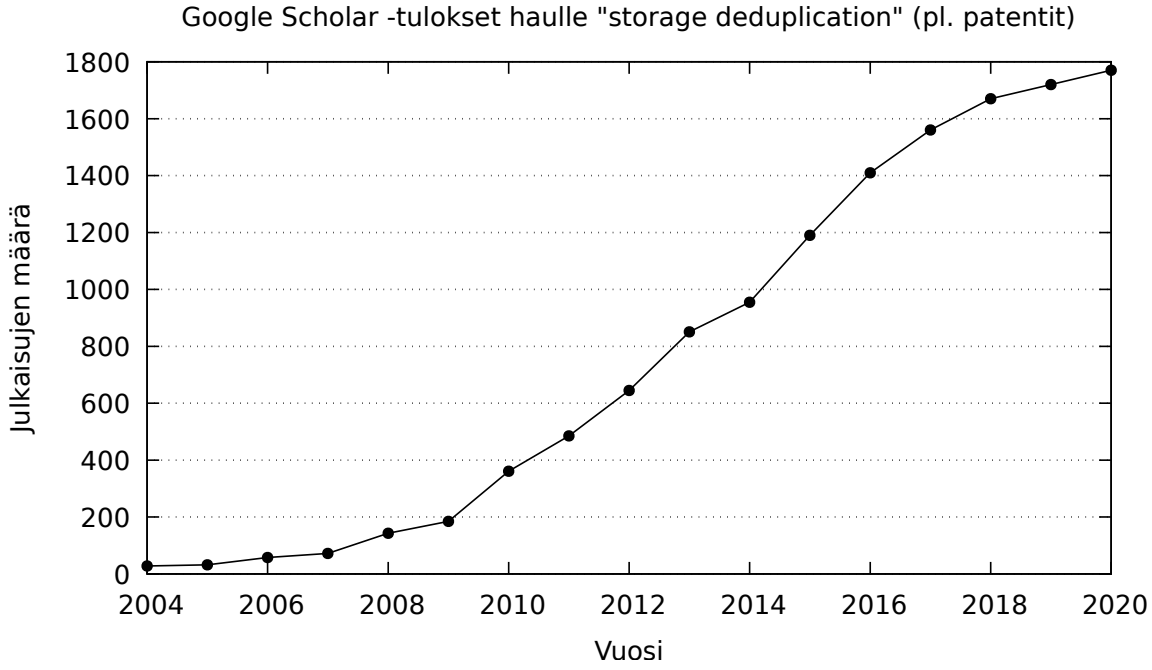
Taulukot

Sisältö

1 Johdanto

Deduplikoinnissa on kyse tilan säästämisestä. Siinä etsitään toistuvia rakenteita, joista säilytetään yksi ja korvataan loput jonkinlaisella merkillä. Jos merkki vie vähemmän tilaa kuin itse rakenne, tilaa säästyy. Toistuvien rakenteiden etsiminen vaatii laskentaa. Deduplikoinnissa siis uhrataan laskentaresursseja tallennustilan säästämiseksi. Deduplikoinnilla voidaan vähentää myös esimerkiksi tarvittavan verkkoliikenteen tai levyoperaatioiden määrää.

Deduplikoinnin laajempi tutkimus on alkanut vuosituhannen vaihteessa (Xia ym. 2016). Deduplikoinnin suosio tutkimuskohteena on kasvanut koko ajan, kuten kuvio 1 ilmaisee.



Kuvio 1. Deduplikoinnin suosio tutkimuskohteena

Deduplikointi on ollut jo kauan laajassa käytössä varmuuskopioiden kanssa. Kehityksen myötä sitä on alettu käyttää myös ensisijaisessa tallennustilassa. Deduplikoinnin suosiota edistää myös flash-muistin ja virtuaaliympäristöjen lisääntyvä käyttö. Flash-levyt ovat kalliimpia ja etenkin satunnaislukunopeudeltaan nopeampia kuin perinteiset kiintolevyt. Korkeampi hinta motivoi deduplikoimaan, ja nopeus auttaa siinä. Virtuaaliympäristöissä taas on usein paljon toistuvia rakenteita, joten ne ovat hyvä kohde deduplikoinnille. (Harnik, Khaitzin ja Sotnikov

2016).

Tässä tutkielmassa luodaan katsaus deduplikointiin. Deduplikoinnin eri osa-alueet käsitellään kirjallisuuteen perustuen. Erityisesti tarkastellaan deduplikoinnin suorituskykyä ja sen parantamista. Lisäksi tutustutaan yhteen deduplikoinnin sovelluskohteeseen ja toteutetaan siinä käytettävä deduplikointiohjelma. Lopuksi ohjelman suoritusnopeutta mitataan kahden muuttujan eri arvoilla ja etsitään parhaan tuloksen tuottavat arvot.

Luvussa 2 käsitellään deduplikoinnin ymmärtämisessä hyödyllisiä esitietoja. Luvussa 3 käydään läpi deduplikointiprosessin vaiheita ja vaihtoehtoja sen ajoitukselle ja paikalle. Luvussa 4 käsitellään deduplikoinnin suorituskykyä ja pyrkimyksiä sen parantamiseksi. Luvussa 5 syvennytään yhteen deduplikoinnin sovelluskohteeseen ja siitä tehtyyn tutkimukseen. Luvussa 6 esitellään työssä toteutetun ohjelman toiminta ja kehitysprosessi. Luvuissa 7 ja 8 kuvataan ohjelmasta tehtyjä mittauksia ja niiden tuloksia.

2 Esitiedot

2.1 Hajautustaulu

Deduplikoinnissa tarvitaan tietorakennetta, joka mahdollistaa avain-arvoparien lisäämisen, hakemisen ja poiston. Haun tehokkuus on erityisen tärkeää. Parien järjestyksellä ei ole väliä. Näihin tarpeisiin vastaa parhaiten tietorakenne nimeltään *hajautustaulu*.

Hajautustaulu on tietyllä tapaa tavallisen taulukon yleistys. Taulukossa päästään vakioajassa käsiksi tietyssä indeksissä olevaan alkioon, koska sen muistipaikka löytyy indeksin perusteella (Cormen ym. 2009, luvun 11 johdanto). Usein alkion indeksiä ei kuitenkaan tiedetä, jolloin alkion haun aikavaativuus on järjestetyllä taulukolla kertaluokkaa $\mathcal{O}(\log n)$ ja järjestämättömällä taulukolla kertaluokkaa $\mathcal{O}(n)$. Haku saadaan vakioaikaiseksi siten, että alkioiden indeksinä toimivat niiden avaimet. Tällöin puhutaan *suorasaantitaulusta*. Suorasaantitaulussa täytyy varata paikka jokaiselle mahdolliselle avaimelle, vaikka vain pieni osa niistä todella esiintyisi taulussa (Cormen ym. 2009, luku 11.2). Tämä rajoittaa sen käyttöä merkittävästi. Ongelma ratkeaa käyttämällä hajautustaulua, jossa tilankäyttö on suhteessa tallennettujen alkioiden määrään.

Hajautustaulussa alkiot tallennetaan taulukkoon, jonka pituus on m. Alkion paikka taulukossa lasketaan sen avaimen perusteella käyttäen *hajautusfunktiota*. Hajautusfunktio siis kuvaa avaimen luvuksi välillä $[0, m-1]$. Koska mahdollisia avainten arvoja on enemmän kuin m kappaletta, täytyy joidenkin eri avainten kuvautua samaksi luvuksi. (Weiss 2014, luku 5.1.) Tätä kutsutaan törmäykseksi, ja se täytyy ottaa huomioon hajautustaulun toteutuksessa. Yksinkertaisin tapa on ketjutus, jossa taulukossa on linkitettyjä listoja. Samaan paikkaan kuvautuneet alkiot laitetaan samaan linkitettyyn listaan. (Cormen ym. 2009, luku 11.2.) Ketjutusta käyttävässä hajautustaulussa alkioiden lisäys-, haku- ja poisto-operaatiot saadaan kaikki tehtyä keskimäärin vakioajassa (Weiss 2014, luku 5.7).

2.2 Muistihierarkia

Ohjelmoijan kannalta olisi kätevää, jos käytettävissä olisi rajaton määrä nopeaa muistia. Siihen ei aivan pystytä, mutta järjestämällä muisti sopivasti päästään lähelle. *Muistihierarkia*, joka hyödyntää muistin tyypillisiä käyttötapoja ja muistityyppien erilaisia ominaisuuksia, auttaa saamaan muistista kaiken irti.

Ohjelmat käyttävät yleensä tietyllä aikavälillä vain pientä osaa niille varatusta muistista. Tätä kutsutaan *paikallisuuden periaatteeksi*. (Jacob, Ng ja Wang 2008, luku Ov.1.1.) Paikallisuutta on kahdenlaista (Patterson ja Hennessy 2013, luku 5.1):

- Ajallinen paikallisuus: viitattuun osoitteeseen viitataan usein pian uudestaan. Näin tapahtuu esimerkiksi silloin, kun ohjelmassa on silmukka.
- Avaruudellinen paikallisuus: lähekkäisiin osoitteisiin viitataan usein peräkkäin, kuten esimerkiksi taulukoita käsiteltäessä.

Kaikki ohjelmat eivät suinkaan käytä muistia paikallisuuden periaatteen mukaisesti, mutta ilmiö on sen verran yleinen, että se kannattaa valjastaa hyötykäyttöön.

Tietokoneen suoritin tarvitsee nopeaa hajasaantimuistia toimiakseen järkevästi. Useimmiten halutaan myös paljon tallennustilaa, jonka sisältö säilyy ilman sähkövirtaa. Näiden vaatimusten täyttäminen yhdellä muistityypillä olisi erittäin kallista. Se ei onneksi ole tarpeen, sillä muisti voidaan toteuttaa hierarkiana. Muistihierarkiassa on monta tasoa muistia, jotka on valmistettu erilaisista komponenteista. Tasot ovat sitä suurempia ja hitaampia, mitä kauempana ne ovat suorittimesta. Paikallisuuden periaatteen ansiosta muistihierarkian suorituskyky voi olla lähellä nopeinta komponenttia, hinta bittiä kohden lähellä halvinta komponenttia, ja virrankulutus datan saantia kohden lähellä vähävirtaisinta komponenttia. (Jacob, Ng ja Wang 2008, luku Ov.1.1; Patterson ja Hennessy 2013, luku 5.1.)

Muistihierarkiassa on yleensä neljä päätasoa (Hennessy ja Patterson 2012, luku 2.1):

- Rekisterit, jotka sijaitsevat suorittimen välittömässä läheisyydessä. Nopein, pienin ja kallein taso.
- Välimuisti, joka sijaitsee myös lähellä suoritinta. Toteutetaan yleensä staattisella RAM-muistilla. Usein jaettu useampaan tasoon.

- Keskusmuisti, joka toteutetaan yleensä dynaamisella RAM-muistilla. Suuri ja halpa verrattuna välimuistiin, ja nopea verrattuna levyyn. Sopii työmuistiksi.
- Massamuisti, joka toteutetaan yleensä SSD- tai kiintolevyllä. Hitain, suurin ja halvin taso. Säilyttää sisältönsä myös ilman sähkövirtaa.

Ajallista paikallisuutta hyödynnetään pitämällä data sitä lähempänä suoritinta, mitä vähemmän aikaa sen viime käytöstä on. Kun ohjelman pyynnöstä noudetaan dataa, siitä tallennetaan kopio välimuistiin. Jos samaa dataa tarvitaan uudestaan, sitä ei tarvitse noutaa kaukaisemmilta tasoilta. Avaruudellista paikallisuutta hyödynnetään noutamalla pyydetyn datan lisäksi samalla myös sen viereistä dataa. (Jacob, Ng ja Wang 2008, luku 1.2.) Välimuistin toiminta on automaattista, eikä ohjelmoijan tarvitse välttämättä ajatella sitä. Asian huomioiminen ohjelmia kirjoittaessa voi kuitenkin tehostaa niiden toimintaa.

Muistihierarkian suorittimesta kaukaisimmat tasot mahdollistavat virtuaalimuistin, jossa prosessit saavat käyttöönsä virtuaalisen muistiavaruuden, joka on suurempi kuin keskusmuisti. Prosessien muisti on jaettu kiinteän mittaisiin osiin, joita kutsutaan sivuiksi. Osa sivuista sijaitsee keskusmuistissa, osa massamuistissa. Kun suoritin viittaa tiettyyn sivuun, käyttöjärjestelmä hakee sen tarvittaessa massamuistista keskusmuistiin. Harvemmin käytetyt sivut päätyvät massamuistiin. Tätä sivujen siirtelyä massamuistin ja keskusmuistin välillä kutsutaan *heittovaihdoksi*. (Arpaci-Dusseau ja Arpaci-Dusseau 2018, osa 2.) Keskusmuisti toimii siis välimuistin tavoin, pitäen paikallisuuden periaatteen mukaisesti sopivan osan muistiavaruudesta suoritinta lähemmällä tasolla muistihierarkiaa.

2.3 Tiedostojärjestelmä

Tiedostojärjestelmä tarjoaa abstraktiokerroksen, joka helpottaa massamuistin käyttöä. Tiedostojärjestelmä jakaa massamuistilla olevan bittien paljouden tiedostoiksi, hakemistoiksi ja erilaisiksi metatiedoiksi. Eräät tiedostojärjestelmien ominaisuudet vaikuttavat deduplikointiin.

Tiedostojärjestelmä käsittelee massamuistin dataa lohkoina. Kaikki luku- ja kirjoitusoperaatiot tehdään vähintään lohkon kokoisille datapätkille. Vaikka prosessi pyytäisi vain yhden tavun tiedostosta, tiedostojärjestelmä lukee kokonaisen lohkon muistissa olevaan puskuriin. (Giampaolo 1999, luku 2.2; Kerola 2019.) Tämän takia tässä työssä toteutetussa ohjelmassa

luetaan oletuksena kustakin tiedostosta yksi lohko.

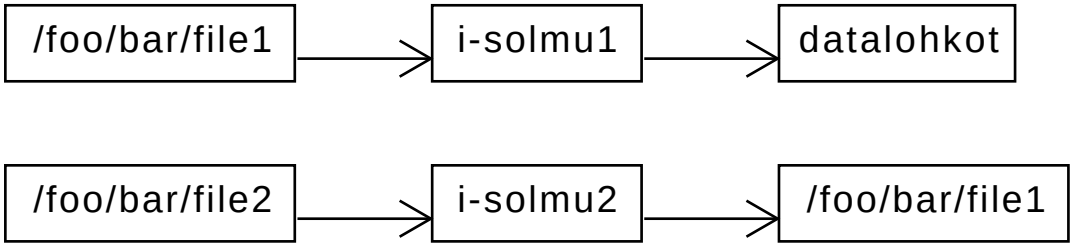
Kuten tietotekniikassa on yleistä, myös massamuistissa olevan datan käsittely on toteutettu kerroksista koostuvana pinona (ks. kuva 2), jossa tiedostojärjestelmä on yksi kerros. Näin yhtenäistetään erilaiset tiedostojärjestelmät ja massamuistilaitteet samanlaisen rajapinnan taakse. Kun prosessi haluaa käsitellä massamuistissa olevaa tiedostoa, pyyntö saapuu ensin virtuaaliselle tiedostojärjestelmälle. Virtuaalinen tiedostojärjestelmä mahdollistaa erilaisten tiedostojärjestelmien käytön samassa käyttöjärjestelmässä määrittämällä rajapinnan, jonka tiedostojärjestelmän pitää toteuttaa. Virtuaalinen tiedostojärjestelmä välittää pyynnön sille tiedostojärjestelmälle, jossa tiedosto sijaitsee. Tiedostojärjestelmä (esimerkiksi Ext4 tai NTFS) hallinnoi tiedostojen ja hakemistojen sisältöä ja metadataa. Se pyytää allaan olevaa lohkokerrosta lukemaan tai kirjoittamaan lohkoja. Lohkokerros vuorontaa saamansa pyynnöt ja välittää ne oikealle laiteajurille. Lisäksi se hallinnoi levyvälimuistia. Laiteajuri keskustelee suoraan tietyn massamuistilaitteen kanssa toteuttaakseen ylempää tulleen pyynnön. (Silberschatz, Galvin ja Gagne 2018, luvut 14.1 ja 15.5; Arpaci-Dusseau ja Arpaci-Dusseau 2018, luku 36.7.)

Kuvio 2. Kerrokset tiedostojärjestelmän ympärillä.

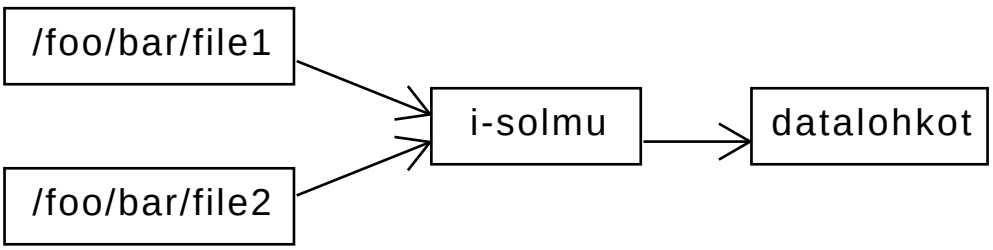
Tiedostojärjestelmissä erotellaan yleensä tiedoston sisältö sen metatiedoista, joihin kuuluvat esimerkiksi tiedoston koko ja käyttöoikeudet (Giampaolo 1999, luku 2). Tietorakennetta,

johon metatiedot tallennetaan, kutsutaan usein *i-solmuksi* (engl. *i-node*). I-solmu sisältää myös osoittimet tiedoston sisältöön. (Arpaci-Dusseau ja Arpaci-Dusseau 2018, luku 40.3.) Tiedoston nimi pidetään erillään sen sisällöstä ja metatiedoista. Kun johonkin hakemistoon luodaan tiedosto, hakemistoon lisätään hakemistomerkintä (engl. *directory entry*), joka linkittää tiedostonimen ja tiedostolle luodun i-solmun (Arpaci-Dusseau ja Arpaci-Dusseau 2018, luku 39.14).

Tiedostoihin voidaan luoda erilaisia linkkejä, joita voidaan hyödyntää deduplikoinnissa. Näitä linkkejä ovat symbolinen linkki, kova linkki ja reflink (Chernov ym. 2018). Symbolinen linkki on erityinen tiedostotyyppi, jonka i-solmu sisältää linkitetyn tiedoston polkunimen. Kova linkki tarkoittaa äsken mainittua hakemistomerkintää. Samalle tiedostolle voidaan luoda useampi kova linkki eli antaa sille useampi nimi. Ei voida sanoa, mikä nimistä on ”alkuperäinen”, vaan ne ovat tasavertaisia viittauksia samaan tiedostoon. (Arpaci-Dusseau ja Arpaci-Dusseau 2018, luku 39.)



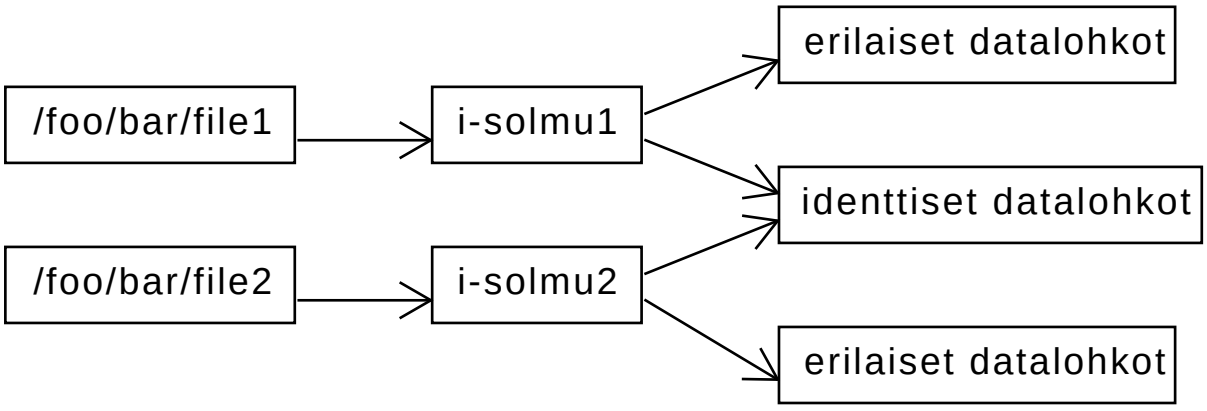
Kuvio 3. Symbolinen linkki.



Kuvio 4. Kova linkki.

Kovien ja symbolisten linkkien käytössä on muutama merkittävä eroavaisuus. Kovaa linkkiä ei voida käyttää, jos halutaan viitata hakemistoon tai toisella laitteella olevaan tiedostoon. Symbolinen linkki taas voi viitata mihin tahansa tiedostoon tai kansioon. Jos symbolisen linkin kohde poistetaan tai siirretään, linkki jää osoittamaan olemattomaan kohteeseen. Jos jokin tiedostoon osoittavista kovista linkeistä (eli tiedostonimistä) poistetaan, voi tiedostoa silti käyttää muiden nimien kautta. Vasta viimeisenkin kovan linkin poistaminen poistaa tiedoston. Tiedoston siirtäminen (saman laitteen sisällä) ei riko kovaa linkkiä, koska tiedoston i-solmu pysyy samana. (Arpaci-Dusseau ja Arpaci-Dusseau 2018, luku 39.)

Kun tiedostoa muokataan, muutokset luonnollisesti näkyvät kaikkien siihen osoittavien kovien tai symbolisten linkkien kautta. Edellä mainittu kolmas linkkityyppi, reflink, toimii eri tavalla. Kun luodaan reflink, sille luodaan uusi i-solmu, joka osoittaa alkuperäisen tiedoston datalohkoihin. Kun tiedostoa muokataan reflinkin kautta, muuttuneille datalohkoille varataan uutta tilaa. (Chernov ym. 2018.) Tiedostojen identtiset datalohkot siis jaetaan ja erilaiset pidetään erillään. Tämä mahdollistaa kopiointia vastaavan käytön, joka kuitenkin säästää tilaa, kun kopioihin ei ole tehty muutoksia. Periaate tunnetaan nimellä *copy-on-write*, ja sitä käytetään myös muun muassa käyttöjärjestelmän muistinhallinnassa (Chernov ym. 2018). Reflink toimii vain joissakin copy-on-writeen perustuvissa tiedostojärjestelmissä, joihin kuuluvat esimerkiksi XFS, Btrfs ja APFS (Aleksandersen 2020).



Kuvio 5. Reflink.

3 Deduplikointi

3.1 Mitä deduplikointi on?

Datan määrä kasvaa räjähdysmäisesti. Reinsel, Gantz ja Rydning (2018) arvioivat, että vuonna 2018 maailmassa oli 33 tsettatavua (tsettatavu = 10^{21} tavua) dataa, ja ennustavat määrän kasvavan 175 tsettatavuun vuodeksi 2025. Informaatioentropia ei kuitenkaan kasva samassa suhteessa (Xia ym. 2016), eli data on osin redundanttia. Datassa siis esiintyy usein samanlaisia osia, joiden koko voi vaihdella muutamasta tavusta suuriin tiedostoihin. Toistoa syntyy esimerkiksi peräkkäisissä varmuuskopioissa tai useille käyttäjille lähetetyissä sähköposteissa. Poistamalla tällaista toisteisuutta voidaan säästää tallennustilaa. Datan toisteisuuden poistamista kutsutaan datan tiivistämiseksi (engl. *data reduction*).

Datan tiivistämiseen käytetään pääasiassa pakkaamista ja deduplikointia. Pakkaaminen toimii pienemmässä mittakaavassa kuin deduplikointi. Yleensä pakataan yksittäisiä tiedostoja tai pieniä tiedostojoukkoja, kun taas deduplikointia voidaan käyttää jopa petatavujen kokoisille datamäärille (Guo ja Efstathopoulos 2011) sen paremman skaalautuvuuden ansiosta. Deduplikointi on yleensä myös käyttäjälle läpinäkyvää, toisin kuin pakkaaminen. Deduplikoituja tiedostoja voi käyttää normaalisti, mutta pakattu tiedosto täytyy ensin purkaa.

Deduplikointi on sekä akateemisen tutkimuksen kohde että useiden tallennusjärjestelmien tarjoama palvelu. Esimerkiksi Dropbox ja Google Drive deduplikoivat dataa (Wang ja Chen 2016), samoin tiedostojärjestelmät ZFS (Kay ja Swearingen 2014) ja Btrfs (“Deduplication” 2021). Deduplikoinnissa jaetaan käsiteltävä data palasiin, joita yleensä kutsutaan lohkoiksi (engl. *chunk*) (Meyer ja Bolosky 2012). Kukin identtinen datalohko tallennetaan vain kerran, ja duplikaatit korvataan viitteellä tähän yhteen tallennettuun lohkoon. Datalohkojen vertailua nopeutetaan laskemalla niistä tiiviste, joka yksilöi lohkon lähes uniikisti.

Deduplikoituun dataan voidaan ajatella olevan kaksi näkökulmaa: looginen näkökulma, jossa esiintyy duplikaatteja, ja laitteelle tallennettu fyysinen näkökulma, josta duplikaatit on poistettu (Paulo ja Pereira 2014). Näiden välissä toimii deduplikointijärjestelmä. Kun dataa kirjoitetaan, deduplikointijärjestelmä poistaa duplikaatit, ja kun dataa luetaan, se kokoaa alkuperäisen

datan. Deduplikointi on häviötöntä (paitsi jos luotetaan siihen, ettei tapahdu törmäyksiä: ks. alaluku 3.3.2), sillä data voidaan palauttaa alkuperäiseen muotoonsa täydellisesti.

Deduplikoinnille on monia sovelluskohteita. Duplikaatteja voi poistaa esimerkiksi aktiivisessa käytössä olevasta tallennustilasta, varmuuskopioista, virtuaalikoneista ja tietokannoista. Deduplikointi voi myös vähentää verkon yli siirrettävän datan määrää. Erilaisiin sovelluskohteisiin tarvitaan erilaisia toteutuksia deduplikoinnista. Esimerkiksi aktiivisessa käytössä olevaa tallennustilaa luetaan paljon useammin kuin varmuuskopioita, mikä pitää ottaa huomioon deduplikointijärjestelmän toteutuksessa.

Sovelluskohde siis vaikuttaa deduplikointijärjestelmän ominaisuuksiin. Usein näiden ominaisuuksien välillä joudutaan käymään vaihtokauppaa: yhden ominaisuuden parantaminen heikentää toista. Esimerkiksi suurempien tilansäästöjen saavuttaminen vaatii yleensä enemmän laskentaa. Toivotut ominaisuudet on siis laitettava kussakin järjestelmässä tärkeysjärjestykseen.

Deduplikointiprosessissa on yleensä ainakin kolme vaihetta: lohkominen, tiivisteiden laskenta sekä tiivisteiden indeksointi. Lisäksi täytyy hallinnoida tiedostoihin ja lohkoihin liittyvää metadataa. Näitä vaiheita käydään läpi alaluvuissa 3.2 ja 3.3. Deduplikointi voidaan suorittaa kirjoitusoperaation yhteydessä tai vasta jälkikäteen, mitä selvitetään alaluvussa 3.4. Erilaisia deduplikoinnin paikkoja käsitellään alaluvussa 3.5.

3.2 Esimerkki deduplikoinnin kulusta

Seuraavassa kuvataan tyypillinen deduplikointitapaus, jossa varmuuskopion viemää tilaa vähennetään tallentamalla vain ne datan osat, joita ei ole aiemmin varmuuskopioitu. Deduplikoinnin yksityiskohdat on valittu tavoitellen mahdollisimman suurta tilansäästöä nopeuden kustannuksella.

Datasta tehdään varmuuskopio verkkolevylle.

1. Verkkolevylle virtaava data jaetaan erimittaisiin lohkoihin sen sisällön perusteella (alaluku 3.3.1).
2. Kullekin lohkolle lasketaan tiiviste, joka on lähes uniikki (alaluku 3.3.2).

3. Hajautustauluun lisätään avain-arvo -pareja, joissa avaimena on tiiviste ja arvona lohkon osoite levyllä. Kun lohkon tiiviste on saatu laskettua, tarkistetaan löytyykö sitä hajautustaulusta (alaluku 3.3.3). Hajautustaulu sisältää myös verkkolevyllä jo aiemmin olleen datan tiedot.

   (a) Jos tiiviste löytyy, lohko korvataan (mahdollisen törmäyksen varalta tarkistamisen jälkeen) viitteellä alkuperäiseen lohkoon, jonka osoite löytyy hajautustaulusta.
   (b) Jos tiivistettä ei löydy, tiiviste-osoite -pari lisätään hajautustauluun.
4. Lisäksi tallennetaan datan kullekin tiedostolle resepti eli lista niiden lohkojen tiedoista, joista tiedosto koostuu. Kun deduplikoituun tiedostoon halutaan päästä käsiksi, siihen kuuluvat lohkot löytyvät reseptin avulla (alaluku 3.3.4).
5. Jos deduplikoitu tiedosto halutaan poistaa, täytyy ottaa selvää, ovatko sen lohkot vielä muussa käytössä. Tässä auttaa viitteidenhallinta (alaluku 3.3.5).

3.3 Deduplikoinnin vaiheet

Yleensä deduplikoinnilla tarkoitetaan tiettyä prosessia, joka voidaan jakaa melko selkeisiin vaiheisiin. Prosessissa jaetaan ensin annettu datajoukko lohkoihin, joiden duplikaatteja etsitään. Lohkoista lasketaan tiiviste, jotta niiden vertailu olisi tehokkaampaa. Tiivisteet tallennetaan indeksiin, josta kukin lohko voidaan löytää tiivisteensä perusteella. Saman tiivisteen tuottaneista lohkoista säilytetään vain yksi, ja loput voidaan korvata pelkällä viitteellä tähän yhteen lohkoon. Lisäksi täytyy tallentaa tieto siitä, mistä lohkoista kukin tiedosto koostuu. Nämä reseptit koostuvat tiivisteistä, joiden perusteella lohkot voidaan noutaa tiivisteindeksistä.

3.3.1 Lohkominen

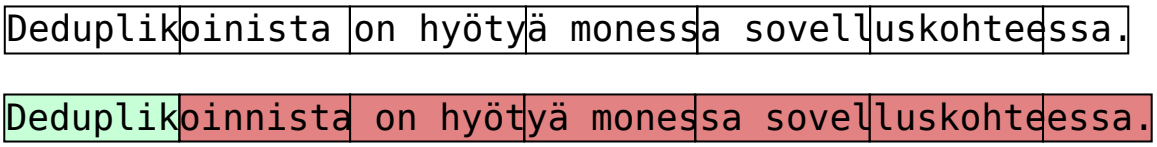
Deduplikointijärjestelmä jakaa käsittelemänsä datan lohkoihin. Lohkoa voidaan pitää deduplikoinnin perusyksikkönä, sillä datasta etsitään juuri identtisiä lohkoja. Lohkojen vertailua nopeutetaan laskemalla hajautusfunktion avulla tiiviste kullekin lohkolle. Tiiviste yksilöi lohkon uniikisti (lukuun ottamatta törmäyksiä, ks. alaluku 3.3.2), joten vertailu voidaan suorittaa tiivisteiden välillä.

Datan jakamiseen lohkoihin on erilaisia menetelmiä (Meister ja Brinkmann 2009):

- tiedosto lohkona, jossa kukin tiedosto on yksi lohko (engl. *whole file hashing/chunking*),
- kiinteä lohkokoko, jossa jokainen lohko on samankokoinen, esimerkiksi 4 Kit, (engl. *fixed block hashing*, *fixed-size chunking* tai *static chunking*),
- vaihteleva lohkokoko, jossa lohkojen rajat määritellään datan sisällön perusteella (engl. *variable block hashing* tai *content-defined chunking*) ja
- tiedostomuodosta riippuva lohkominen, jossa tiedosto jaetaan lohkoihin hyödyntäen tietoa kyseisen tiedostomuodon rakenteesta (engl. *application-aware chunking*).

Usein on helpointa hyödyntää tiedostojärjestelmän valmista jakoa tiedostoihin ja käsitellä kutakin tiedostoa yhtenä lohkona. Tällöin lohkomiseen ei kulu laskentaresursseja, ja indeksi vie vähän tilaa. Toisaalta saavutettu tilansäästö on pienempi kuin muita lohkomistapoja käytettäessä, sillä esimerkiksi kaksi gigatavun kokoista tiedostoa eivät ole duplikaatteja, jos niissä on yhdenkin tavun ero.

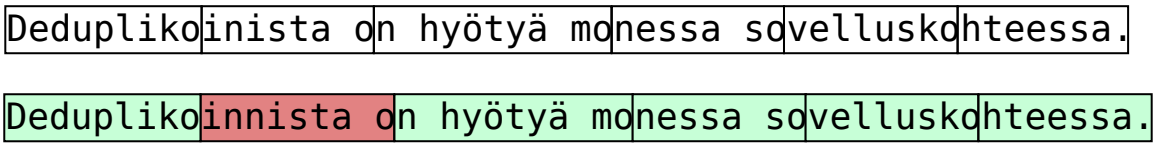
Toinen helppo menetelmä on jakaa data samankokoisiin lohkoihin. Tiedostojärjestelmissä data on yleensä jaettu lohkoihin, ja tätä jakoa voidaan käyttää deduplikoinnissa. Tässä menetelmässä ongelmana on, että jos tiedoston keskeltä poistetaan tai keskelle lisätään jotakin, muuttuvat myös tiedoston kaikki muutoskohtaa seuraavat lohkot. Ongelmaa kutsutaan rajansiirtymisongelmaksi (engl. *boundary shift problem*) (Xia ym. 2016). Kuviossa 6 esitetään tilanne, jossa teksti on jaettu kahdeksan merkin lohkoihin. Kun tekstissä oleva kirjoitusvirhe korjataan lisäämällä yksi kirjain, kaikki lohkot korjauskohdasta alkaen muuttuvat, jolloin ne eivät deduplikoidu.



Kuvio 6. Rajansiirtymisongelma kiinteää lohkokokoa käytettäessä.

Rajansiirtymisongelma ratkeaa käyttämällä vaihtelevaa lohkokokoa, mitä kutsutaan myös sisältöön perustuvaksi lohkomiseksi. Toisin kuin kiinteää lohkokokoa käytettäessä, lohkorajat määritellään perustuen datan sisältöön, ei sen paikkaan. Näin lohkoista tulee erikokoisia, mutta muutos keskelle tiedostoa ei välttämättä muuta sitä seuraavia lohkoja. Alkuperäinen

(Muthitacharoen, Chen ja Mazières 2001) ja nykyäänkin yleinen tapa määrittää lohkorajat sisällön perusteella on käyttää *liukuvaa ikkunaa*. Liukuvan ikkunan tekniikassa tarkastellaan tietynkokoista data-aluetta eli ikkunaa, joka liikkuu tavu kerrallaan dataa pitkin. Jokaisesta osittain päällekkäisestä ikkunasta lasketaan tiiviste. Jos tiivisteen vähiten merkitsevät bitit ovat sama kuin tietty ennalta määrätty arvo, merkitään kyseiseen kohtaan lohkon raja. Tiivisteenä käytetään usein Rabinin sormenjälkeä (Rabin 1981), koska sen laskeminen tietylle ikkunalle vaatii vain pieniä muutoksia sitä edeltävän ikkunan sormenjälkeen. Kuviossa 7 esitetään sama teksti kuin kuviossa 6, mutta se on jaettu vaihtelevan kokoisiin lohkoihin. Lohkoraja määritellään o-kirjaimen kohdalle. Kun kirjoitusvirhe korjataan, vain sen kohdalla oleva lohko muuttuu. Loput pysyvät samoina ja deduplikoituvat.



Kuvio 7. Vaihteleva lohkokoko.

Neljännessä tavassa jakaa data lohkoihin hyödynnetään tietoa tiedostomuodoista. Usein tiedostomuoto määrää pitkälti tiedoston rakenteen. Esimerkiksi esitysgrafiikkatiedosto voitaisiin luontevasti lohkoa joka dian kohdalta. Näin lohkoista saadaan loogisia kokonaisuuksia. Tällainen deduplikointi voi säästää yhtä paljon tilaa kuin edellisessä kappaleessa esiteltyä vaihtelevaa lohkokokoa käyttävä, mutta lohkoja tarvitaan vähemmän, mikä säästää resursseja (Kim, Song ja Choi 2013). Tiedostomuodon perusteella voidaan myös valita, käytetäänkö samankokoisia vai vaihtelevan kokoisia lohkoja, kuten Min, Yoon ja Won (2011) tekevät. He hyödyntävät sitä tosiasiaa, että tiettyjen tiedostojen pienikin muokkaaminen muuttaa niiden koko sisältöä levyllä. Tällaisia tiedostoja ovat pakattua formaattia käyttävät mediatiedostot sekä muut pakatut tai salatut tiedostot. Näiden kohdalla laskennallisesti vaativamman vaihtelevan lohkokoon käyttäminen ei yleensä maksa vaivaa, joten he käyttävät samankokoisia lohkoja. He eivät ollenkaan käsittele kokonaisten tiedostojen deduplikointia, vaikka se sopisi tähän tilanteeseen vielä paremmin. Muille tiedostoille he käyttävät vaihtelevaa lohkokokoa. Tiedostojen rakenteen huomioivia deduplikointimenetelmiä ovat esitelleet myös C. Liu ym. (2008) ja Yan ja Tan (2011).

3.3.2 Tiivisteiden laskenta

Datalohkoja verrataan keskenään duplikaattien löytämiseksi. Lohkoja voitaisiin käydä läpi tavu tavulta, mutta vertailua kannattaa tehostaa käyttämällä hajautusfunktiota.

Hajautusfunktio kuvaa mielivaltaisen pitkän syötteen vakiopituiseksi tiivisteeksi. Jos kaksi tiivistettä ovat erilaisia, ovat syötteetkin varmasti erilaisia. Tiivisteiden identtisyys ei kuitenkaan takaa syötteiden identtisyyttä. Tämä johtuu siitä, että jotta hajautusfunktiosta olisi hyötyä, tiiviste on syötteen maksimipituutta lyhyempi. Tällöin kyyhkyslakkaperiaatteesta seuraa, että osa syötteistä kuvautuu samaksi tiivisteeksi. Tätä tapahtumaa kutsutaan törmäykseksi. Hyvä hajautusfunktio tuottaa törmäyksiä mahdollisimman harvoin.

Jos törmäyksiä ei olisi, tiiviste olisi täysin uniikki tunniste syötteelle. Usein törmäyksen todennäköisyys on kuitenkin niin pieni, että tiivistettä voidaan käyttää kuin se olisi uniikki. Kun lasketaan b bitin pituiset tiivisteet q:lle erilaiselle lohkolle, törmäyksen todennäköisyys p voidaan laskea seuraavalla kaavalla (Quinlan ja Dorward 2002):

$$p \leq \frac{q(q-1)}{2^{b+1}}. \tag{3.1}$$

Quinlan ja Dorward (2002) johtavat kaavan 3.1 vain pintapuolisesti. Seuraavaksi kaava johdetaan mukaillen Boldyrevaa (2005). Olkoon $P(N,q)$ todennäköisyys ainakin yhdelle törmäykselle, kun lasketaan q tiivistettä ja mahdollisia tiivisteitä on N kappaletta. Kuvatkoon E_i sitä tapahtumaa, että i:s tiiviste törmää. Sen todennäköisyys $P(E_i) \leq (i-1)/N$, sillä kun i:s tiiviste lasketaan, on uniikkeja tiivisteitä jo laskettu enintään $i-1$ kappaletta ja i:s tiiviste törmää yhtä suurella todennäköisyydellä minkä tahansa niistä kanssa, jos oletetaan tiivistefunktion jakavan tiivisteet tasaisesti. Tällöin

$$\begin{aligned}
P(N,q) &= P(E_1 \cup E_2 \cup \cdots \cup E_q) \\
&\leq P(E_1) + P(E_2) + \cdots + P(E_q) && \text{(Boolen epäyhtälö)} \\
&\leq \frac{0}{N} + \frac{1}{N} + \cdots + \frac{q-1}{N} \\
&= \frac{q(q-1)}{2N}. && \text{(Aritmeettisen sarjan summa)}
\end{aligned} \tag{3.2}$$

Kun tiivisteet ovat b bittiä pitkiä, on mahdollisia tiivisteitä 2^b kappaletta ja epäyhtälöt 3.1 ja 3.2 vastaavat toisiaan. Yleensä törmäyksen todennäköisyyttä laskettaessa ei haittaa käyttää

ylälikiarvoa, koska on parempi varautua liian suureen todennäköisyyteen kuin liian pieneen. Törmäyksen todennäköisyyden tarkka arvo on

$$p = 1 - \frac{N!}{N^q (N-q)!}.$$

Tämän laskeminen ei kertomien takia ole järkevää niin suurilla luvuilla kuin deduplikoinnissa käytetään. Kertoma voidaan kuitenkin korvata gammafunktiolla, joka voidaan mielivaltaisen tarkkoja lukuja käyttämällä laskea halutulla tarkkuudella. Kaava 3.1 on kuitenkin erittäin tarkka, kun törmäyksen todennäköisyys ei ole suuri.

Törmäyksen todennäköisyyttä verrataan usein tallennusvälineen korjaamattomissa olevien bittivirheiden suhdelukuun (engl. *uncorrectable bit error rate, UBER*). Se on sekä kiinto- että SSD-levyillä noin 10^{-15} (Gray ja Van Ingen 2005; Tai ym. 2019). Suhdeluku 10^{-15} tarkoittaa, että kun luetaan 10^{15} bittiä eli noin 114 tebitavua, vastaan tulee yksi virhe, joka ei ole korjattavissa. Kaavan 3.1 mukaan sama todennäköisyys törmäykselle saavutetaan 128-bittisellä hajautusfunktiolla, kun deduplikoidaan $8{,}25 \times 10^{11}$ lohkoa. Jos lohkot ovat kooltaan 4 kibitavua, vastaisi määrä 3 pebitavua dataa. Useimmat deduplikointijärjestelmät perustuvat sille oletukselle, että törmäyksen todennäköisyys on häviävän pieni ja tiiviste vastaa uniikkia tunnistetta (Xia ym. 2016, luku II-B).

Mitä enemmän lohkoja deduplikointijärjestelmä käsittelee, sitä pidempi sen käyttämän tiivisteen pitäisi olla. Jos halutaan olla täysin varma, ettei datan menettäminen törmäyksen takia ole mahdollista, voidaan saman tiivisteen tuottaneet lohkot vertailla vielä tavu tavulta. Tällöin tiivisteen ei tarvitse olla kovin pitkä. Tähän ratkaisuun päädyin itse ohjelmassani. Sitä käyttää myös NetApp (“What is Data Deduplication?” 2019). Tavu tavulta vertailu on vaihtoehtona myös Foundationissa (Rhea, Cox ja Pesterev 2008) sekä DEDIS:issä (Paulo ja Pereira 2016).

Tiivisteitä käytetään siis nopeuttamaan deduplikointia. On huomattavasti nopeampaa vertailla lyhyitä tiivisteitä (yleensä enintään 32 tavua), jotka yleensä ovat vieläpä keskusmuistissa, kuin massamuistissa olevia suurempia lohkoja (yleensä joitakin kilotavuja). Tiivisteiden laskenta kuitenkin vaatii suoritinresursseja.

3.3.3 Tiivisteiden indeksointi

Lohkoista lasketut tiivisteet tallennetaan indeksiin. Indeksi eli deduplikointitaulu on hajautustaulu (ks. alaluku 2.1), jossa avaimena on tiiviste ja arvona viittaus lohkon sijaintiin. Viittaus voi olla esimerkiksi polkunimi tai tunnistenumero varmuuskopiojonoon. Jos lohkolle laskettu tiiviste löytyy jo indeksistä, on kyseinen lohko duplikaatti ja sen voi korvata viitteellä aiemmin tallennettuun duplikaattilohkoon. Kuten luvussa 3.3.2 mainittiin, lohkojen identtisyys saatetaan haluta varmistaa vertaamalla niitä tavu tavulta. Jos lohkon tiivistettä ei löydy indeksistä, tiiviste lisätään indeksiin ja lohko tallennetaan.

3.3.4 Tiedostoreseptit

Deduplikointi on yleensä käyttäjälle läpinäkyvää, joten tiedostojen käytön tulisi tapahtua samalla käyttöliittymällä kuin ilman deduplikointia. Tämän takia deduplikointijärjestelmissä on tietorakenne, jonka avulla alkuperäinen data saadaan koottua vastaamaan tavallisen tiedostorajapinnan pyyntöihin eli rekonstruoitua. (Mandal 2015.) Tiedostoresepti on tällainen tietorakenne.

Tiedostoresepti kertoo, mistä lohkoista tiedosto koostuu. Kun tiedostoa halutaan lukea tietystä kohtaa, tarvittavat lohkot selvitetään reseptin avulla. Resepti voi sisältää lohkojen tiivisteet, jolloin lohkojen osoitteet noudetaan deduplikointitaulusta (ks. esim. J. Liu ym. 2014). Toinen vaihtoehto on tallettaa lohkojen osoitteet reseptiin, jotta tiedostoa lukiessa ei tarvitse käyttää deduplikointitaulua (ks. esim. Yin ym. 2021).

3.3.5 Viitteidenhallinta ja roskienkeruu

Deduplikoidussa järjestelmässä samaan datalohkoon voidaan viitata useasta paikasta. Tämä monimutkaistaa tiedostojen poistoa: tiedoston datalohkoja ei voida poistaa ennen kuin tiedetään, oliko kyseessä viimeinen viittaus datalohkoon. Siksi viittauksista täytyy pitää kirjaa.

Viitteidenhallinta mahdollistaa roskienkeruun eli edellä mainitun orpojen (joihin ei enää viitata) datalohkojen etsimisen ja poiston. Orpojen datalohkojen etsimismenetelmät voidaan yleensä jakaa kahteen kategoriaan: viitteiden laskenta sekä merkitse ja lakaise (Xia ym. 2016).

Viitteiden laskenta tarkoittaa yksinkertaisesti sitä, että lohkojen viittausmäärää seurataan. Kun viittausmäärä on nolla, lohko poistetaan. Merkitse ja lakaise -menetelmässä käydään kaikki tiedostot läpi ja merkitään niiden viittaamat lohkot. Lohkot, joita ei merkitty, ovat orpoja ja ne poistetaan.

Viitteiden laskenta tapahtuu välittömästi tiedostoja luodessa, muokatessa ja poistaessa. Merkitse ja lakaise -prosessi taas ajetaan erikseen halutuin väliajoin. Välittömyys lisää viivettä I/O-operaatioihin mutta vapauttaa tallennustilan mahdollisimman nopeasti. Viivästetty roskienkeruu taas ei vaikuta viiveeseen, mutta tallennustilan vapautumista joutuu odottamaan. (Xia ym. 2016.) Molemmissa menetelmissä on ongelmansa, kuten viitteiden laskennassa virheidensietokyky ja merkitse ja lakaise -menetelmässä skaalautuvuus (Guo ja Efstathopoulos 2011). Xia ym. (2016) esittävät koosteen erilaisista roskienkeruun parannusehdotuksista.

3.4 Deduplikoinnin ajoitus

Deduplikoinnin hyödyt realisoituvat nopeimmin, jos se suoritetaan heti kun data ollaan siirtämässä massamuistiin. Deduplikointi on kuitenkin laskennallisesti raskasta ja jos se tehdään kirjoitusoperaation yhteydessä, operaation viive kasvaa. Joskus viiveen on tärkeää olla mahdollisimman pieni, jolloin deduplikointi täytyy siirtää myöhemmäksi.

Välitön (tai synkroninen, engl. *inline/in-band/synchronous*) deduplikointi tarkoittaa, että jokaisen kirjoitusoperaation yhteydessä yritetään ensin deduplikointia, ennen kuin data kirjoitetaan (Mandagere ym. 2008). Silloin duplikaattilohkot eivät koskaan saavuta massamuistia, mikä vähentää kirjoitusoperaatioiden määrää eikä vaadi väliaikaisesti tallennustilaa duplikaateille. I/O-operaatioiden välttäminen on erityisen hyödyllistä hitaita kiintolevyjä käytettäessä. Kirjoitusoperaatioon syntyy kuitenkin viivettä, kun joudutaan laskemaan lohkojen rajoja ja tiivisteitä, tutkimaan tiivisteindeksiä sekä tekemään viitteidenhallintaa. (Fingler, Ra ja Panta 2019.) Duplikaattien kirjoittaminen voi siis olla nopeampaa kuin järjestelmässä jossa ei käytetä deduplikointia, mutta uniikkien lohkojen kirjoittaminen on varmasti hitaampaa. Lisäksi tallennusjärjestelmän täytyy mahdollistaa kirjoitusoperaation sieppaaminen, jotta välitön deduplikointi olisi ylipäätään mahdollista (Paulo ja Pereira 2014).

Jälkikäteisessä (tai asynkronisessa, engl. *offline/out-of-band/asynchronous/post-process*) de-

duplikoinnissa datalohkot tallennetaan aina ensin massamuistille, vaikka ne olisivat duplikaatteja (Paulo ja Pereira 2014). Deduplikointi suoritetaan jälkeenpäin tietyin väliajoin, usein silloin, kun järjestelmä on käyttämättömänä. Kirjoitusoperaatioihin ei siis synny viivettä, mutta tallennustilaa tarvitaan väliaikaisesti enemmän. Lisäksi I/O-operaatioiden kokonaismäärä kasvaa huomattavasti, koska data täytyy lukea uudelleen kun se deduplikoidaan. Joskus järjestelmällä ei ole selvää hetkeä, jolloin se olisi vähäisellä käytöllä, mikä voi aiheuttaa hankaluuksia jälkikäteisen deduplikoinnin ajankohdan päättämisessä. (Paulo ja Pereira 2014; Fingler, Ra ja Panta 2019.) Koska deduplikointi tapahtuu asynkronisesti I/O-operaatioiden kanssa, tarvitaan erillinen mekanismi (kuten copy-on-write, ks. alaluku 2.3) huolehtimaan siitä, ettei jaetun datan muokkaaminen aiheuta datan korruptiota (Paulo ja Pereira 2016).

3.5 Deduplikoinnin paikka

Deduplikointia käytetään eniten toissijaisessa tallennustilassa (Xia ym. 2016). Siinä on yleensä eniten duplikaatteja. Suuria hyötyjä voidaan kuitenkin saavuttaa myös muissa käyttökohteissa.

3.5.1 Ensisijainen ja toissijainen tallennustila

Ensisijaisessa tallennustilassa oleva data on aktiivisessa käytössä. Sitä luodaan, muokataan, luetaan ja poistetaan. Esimerkiksi sähköpostipalvelimet ja käyttäjien kotihakemistot sijaitsevat ensisijaisessa tallennustilassa. Koska ensisijainen tallennustila on ihmisten jatkuvassa suorassa käytössä, sen pitää toimia nopeasti ja tasaisesti. Toimintojen viiveen pitää olla pieni eikä suorituskyky saa heitellä liikaa. (Yin ym. 2018.) Deduplikointikaan ei siis saisi juurikaan hidastaa ensisijaisen tallennustilan toimintaa.

Välittömän deduplikoinnin aiheuttama viive on joskus mahdotonta saada tarpeeksi pieneksi, jotta sitä voitaisiin käyttää ensisijaisessa tallennustilassa. Silloin on turvauduttava jälkikäteiseen deduplikointiin, jossa on kuitenkin omat ongelmansa. Kummankin menetelmän käytöstä ensisijaisessa tallennustilassa on runsaasti esimerkkejä (Paulo ja Pereira 2014).

Toissijainen tallennustila tarkoittaa tallennustilaa, joka ei ole aktiivisessa käytössä, kuten varmuuskopioita tai arkistoja. Toissijaisessa tallennustilassa pidettävää dataa muutetaan harvoin, eikä joistakin arkistoista poisteta koskaan mitään (Quinlan ja Dorward 2002). Suurin

kiinnostus suorituskyvyn suhteen kohdistuu datan siirtonopeuteen tallennustilaan ja sieltä ulos. Toissijaisessa tallennustilassa käytetään useammin välitöntä kuin jälkikäteistä deduplikointia (Srinivasan ym. 2012).

Usein esimerkiksi tietty hakemisto varmuuskopioidaan määrätyin väliajoin. Tällöin varmuuskopioiden välillä on paljon samoja datalohkoja, ja deduplikointi säästää erittäin paljon tilaa. Datassa tietyllä ajanhetkellä ei ole yhtä paljon toisteisuutta.

Ensisijaisessa tallennustilassa käsitellään usein pieniä tiedostojen osia, toisin kuin toissijaisessa tallennustilassa, johon usein syötetään kerralla suuria määriä tiedostoja (Lu ym. 2012). Ensisijaisen tallennustilan I/O-operaatiot ovat siis useammin satunnaisia, toissijaisen tallennustilan taas peräkkäisiä. Ensisijaisen tallennustilan deduplikoinnissa kannattaa siksi optimoida viive, toissijaisen tallennustilan deduplikoinnissa taas tiedonsiirtonopeus (Paulo ja Pereira 2014; Srinivasan ym. 2012).

Ensisijaisessa tallennustilassa ei yleensä ole yhtä paljon duplikaatteja kuin toissijaisessa. Esimerkkejä redundantin datan määristä ovat 42–68 prosenttia ensisijaisessa tallennustilassa ja 69–97 prosenttia toissijaisessa (Xia ym. 2016; Meyer ja Bolosky 2012). Ensisijainen tallennustila on yleensä kalliimpaa kuin toissijainen, mikä motivoi deduplikoimaan sitä huolimatta sen vähäisemmästä duplikaattien määrästä (Yin ym. 2018).

3.5.2 Verkot ja palvelimet

Kun siirrytään yhden datajoukon deduplikoinnista deduplikoimaan dataa, joka tulee useasta erillisestä lähteestä, täytyy järjestelmän suunnittelussa huomioida uusia seikkoja. Duplikaatteja voidaan etsiä kustakin lähteestä erikseen tai kaikista yhdessä, ja prosessoinnin paikalle on vaihtoehtoja.

Hajautetuissa järjestelmissä, verkoissa ja klustereissa on useita tietokoneita eli solmuja, jotka on yhdistetty toisiinsa verkkoyhteydellä. Tässä yhteydessä käsitellään niitä samankaltaisina järjestelminä ja kutsutaan niitä nimellä verkko.

Verkoissa voidaan deduplikoida kahdella tapaa. Jos halutaan, että duplikaatti löytyy, vaikka sen esiintymät olisivat eri solmuissa, täytyy deduplikoinnin olla globaalia. Siinä kunkin solmun

dataa verrataan kaikkien muiden solmujen dataan. Toinen vaihtoehto on etsiä duplikaatteja kustakin solmusta erikseen, jolloin deduplikointi on paikallista. (Kaur, Chana ja Bhattacharya 2018.)

Globaali deduplikointi löytää luonnollisesti enemmän duplikaatteja kuin paikallinen. Se vaatii kuitenkin globaalin indeksin, jonka käyttö lisää viivettä I/O-operaatioihin (Bansal ja Sharma 2021). Paikallinen deduplikointi skaalautuukin hieman paremmin kuin globaali.

Verkoista erillisenä arkkitehtuurina käsitellään asiakas-palvelin-arkkitehtuuria. Tässä yhteydessä sillä tarkoitetaan datan yhdensuuntaista kulkua asiakkaalta palvelimelle, kuten esimerkiksi varmuuskopiointia. Deduplikoinnin voi suorittaa asiakas, palvelin tai niiden välissä oleva erillinen laite (Mandagere ym. 2008).

Asiakkaan suorittama deduplikointi (engl. *client/source deduplication*) tarkoittaa duplikaattien poistamista ennen kuin data lähetetään palvelimelle. Se vähentää huomattavasti tiedonsiirron määrää. (Kaur, Chana ja Bhattacharya 2018.) Jos asiakas deduplikoi täysin itsenäisesti, ei asiakkaiden välisiä duplikaatteja löydetä. Vaihtoehtoisesti asiakas voi lähettää palvelimelle datalohkojen tiivisteitä, joiden avulla palvelin tunnistaa datalohkot, joita sillä ei vielä ole ja pyytää asiakasta lähettämään ne. Tällöin asiakkaiden väliset duplikaatit löydetään. (Kim, Song ja Choi 2017, luku 2.4.2.)

Palvelimen suorittama deduplikointi (engl. *server/target/destination deduplication*) poistaa kaikkien asiakkaiden väliset duplikaatit. Palvelimella on usein enemmän resursseja käytettävissä deduplikointiin kuin asiakkailla. Tiedonsiirtoa tarvitaan enemmän kuin asiakkaan suorittamassa deduplikoinnissa, koska kaikki duplikaatitkin siirretään palvelimelle. (Kim, Song ja Choi 2017, luku 2.4.1.)

Asiakkaiden ja palvelimen välissä voi olla erillinen laite, joka hoitaa deduplikoinnin (engl. *deduplication appliance*). Jos laite käyttää välitöntä deduplikointia, se saattaa muodostua tiedonsiirron pullonkaulaksi. (Mandagere ym. 2008.) Erillinen laite vie deduplikoinnin taakan asiakkailta ja palvelimelta.

3.5.3 SSD

SSD-levyn deduplikoinnista on monenlaista hyötyä. Säästynyttä tilaa voivat käyttäjän lisäksi hyödyntää SSD:n sisäiset kulutuksen tasaus- ja roskienkeruumekanismit. SSD:n käyttämä flash-muisti kestää vain rajatun määrän kirjoittamista, mutta duplikaattien kirjoittaminen voidaan estää välittömällä deduplikoinnilla. (Paulo ja Pereira 2014; Chen, Luo ja Zhang 2011.)

SSD voi hoitaa deduplikoinnin itsenäisesti. SSD:n sisäinen flash-muunnostaso kuvaa loogiset osoitteet fyysisiksi, mikä sopii hyvin myös deduplikoinnissa käytettyihin viittauksiin. Myös SSD:n roskienkeruu voidaan valjastaa deduplikoinnin käyttöön. Koska SSD:n satunnaislukunopeus on hyvä, deduplikoinnin aiheuttama pirstoutuminen ei haittaa merkittävästi. SSD:n prosessointikyky on kuitenkin vähäinen, mikä asettaa haasteita tiivisteiden laskemiseen. Lisäksi SSD:n välimuistina käytettävä DRAM-muisti on yleensä liian pieni täydelle tiivisteindeksille. (Paulo ja Pereira 2014; Kim ym. 2012.)

4 Deduplikoinnin suorituskyvyn parantaminen

Deduplikoinnin tutkimuksen keskiössä on sen suorituskyvyn parantaminen. Alaluvussa 4.1 käydään läpi deduplikoinnin suorituskyvyn mittarit. Lopuissa alaluvuissa esitellään erilaisia tapoja deduplikoinnin suorituskyvyn parantamiseen.

4.1 Deduplikoinnin suorituskyky

Deduplikoinnin suorituskykyä voidaan mitata monella mittarilla. Usein eniten huomiota kiinnitetään tilansäästöä kuvaavaan deduplikointisuhteeseen (engl. *deduplication ratio*), joka kertoo, kuinka paljon vähemmän tilaa deduplikoitu data vie suhteessa alkuperäiseen. Deduplikointisuhde on alkuperäisen datan koko jaettuna datan koolla deduplikoinnin jälkeen (Fu ym. 2015).

Muita suorituskyvyn mittareita ovat deduplikoinnin nopeus, rekonstruointinopeus, resurssien kulutus sekä skaalautuvuus (Mandagere ym. 2008; Guo ja Efstathopoulos 2011). Deduplikoinnin nopeudella tarkoitetaan sitä, kuinka nopeasti data kulkee deduplikointijärjestelmän läpi. (Guo ja Efstathopoulos 2011). Rekonstruointinopeus taas tarkoittaa deduplikoidun datan lukunopeutta (Mandagere ym. 2008). Niitä mitataan siis tavuina sekunnissa. Deduplikoinnin kuluttamia resursseja ovat suoritin, keskusmuisti, tallennustilan kaista ja joskus verkon kaista (El-Shimi ym. 2012; Mandagere ym. 2008). Deduplikointijärjestelmän skaalautuvuus tarkoittaa sen kykyä selviytyä suuristakin määristä dataa ilman huomattavaa suorituskyvyn laskua (Guo ja Efstathopoulos 2011).

Kun optimoidaan deduplikointijärjestelmää keskittyen tiettyyn suorituskyvyn osa-alueeseen, jokin toinen osa-alue usein heikkenee. Esimerkiksi indeksin säilyttäminen keskusmuistissa massamuistin sijaan parantaa deduplikoinnin nopeutta, mutta keskusmuistin määrä asettaa rajan skaalautuvuudelle. (Guo ja Efstathopoulos 2011.) Toisena esimerkkinä vaihtelevan lohkokoon käyttäminen parantaa deduplikointisuhdetta verrattuna kokonaisten tiedostojen käyttämiseen lohkoina, mutta heikentää nopeutta, rekonstruointinopeutta, resurssien kulutusta ja skaalautuvuutta.

4.2 Lohkominen

Lohkomistapa vaikuttaa kaikkiin deduplikoinnin suorituskyvyn mittareihin. Siihen onkin kiinnitetty tutkimuksissa paljon huomiota.

Yleisesti ottaen voidaan sanoa, että deduplikointisuhde paranee siirryttäessä kokonaisten tiedostojen deduplikoinnista kiinteään lohkokokoon ja kiinteästä lohkokoosta vaihtelevaan lohkokokoon. Muut suorituskyvyn mittarit noudattavat päinvastaista järjestystä. Samankaltainen vaikutus on lohkokoolla: suurempi lohkokoko huonontaa deduplikointisuhdetta, mutta parantaa muita suorituskyvyn mittareita. (Mandagere ym. 2008; Paulo ja Pereira 2014; El-Shimi ym. 2012.)

Kokonaisten tiedostojen käyttäminen lohkoina on yksinkertaisin ratkaisu. Deduplikointi on silloin nopeinta, koska aikaa ei kulu lohkorajojen määrittämiseen ja käsiteltäviä lohkoja on vähemmän. Rekonstruointinopeus laskee vähiten, koska tiedostojen osat eivät päädy eri paikkoihin. Koska laskentaa tarvitaan vähiten, suoritinresursseja kuluu vähiten. Lohkojen määrä on pienin, joten metadataa (johon kuuluvat tiivisteindeksi ja tiedostoreseptit) ja siten keskusmuistia tarvitaan vähiten. Huono puoli kokonaisten tiedostojen käyttämisessä lohkoina on deduplikointisuhde, joka on huonoin. Huonoimman duplikaattien tunnistamismahdollisuuden takia myös I/O-operaatioiden ja verkon kaistan säästäminen on epätodennäköisintä.

Tiedostoa pienemmät lohkot lisäävät lohkojen kokonaismäärää, mikä kasvattaa metadatan kokoa ja siten keskusmuistin tarvetta. Lisäksi laskentaa tarvitaan enemmän, mikä hidastaa deduplikointia ja lisää suorittimen käyttöä. Rekonstruointinopeus laskee enemmän, koska tiedostojen osat päätyvät eri paikkoihin. Syy tiedostoa pienempien lohkojen käyttöön on usein huomattavasti parempi deduplikointisuhde. Duplikaattien paremman tunnistamisen myötä myös I/O-operaatioita ja verkon kaistaa säästyy enemmän.

Vaihtelevaa lohkokokoa käytetään, kun halutaan maksimoida deduplikointisuhde. Lohkorajojen määrittäminen vaatii silloin muihin tapoihin verrattuna ylimääräistä laskentaa. Se mahdollistaa kuitenkin enemmän joustavuutta: esimerkiksi Bobbarjung, Jagannathan ja Dubnicki (2006) esittelevät algoritmin, joka luo pienempiä lohkoja muokatuille data-alueille ja suurempia lohkoja muokkaamattomille. Näin metadatan koko pienenee, mutta deduplikointisuhde ei huonone.

Lohkorajojen ja tiivisteiden laskenta saattaa muodostua deduplikointijärjestelmän pullonkaulaksi, jos keskusmuistia on runsaasti ja tallennustilana käytetään nopeita SSD-levyjä (Kim, Park ja Park 2012). Laskentaa voidaan nopeuttaa rinnakkaistamalla se usealle ytimelle tai säikeelle tai käyttämällä siihen yleiskäyttöisiä grafiikkaprosessoreita (engl. *GPGPU* eli *general-purpose computing on graphics processing units*) (Xia ym. 2016).

Metadatan koko on lähes aina pieni verrattuna deduplikoitavaan datajoukkoon. Se voi kuitenkin vaikuttaa suuresti deduplikoinnin nopeuteen, jos indeksi ei mahdu keskusmuistiin, kuten alaluvussa 4.3 kerrotaan (Mandagere ym. 2008). Kokonaisten tiedostojen deduplikoinnin vaatiman metadatan koko vaihtelee tiedostojen määrän funktiona, kun taas tiedostoa pienempiä lohkoja käytettäessä datan määrän funktiona. Se vaikuttaa järjestelmän skaalautuvuuteen.

4.3 Indeksointi

Välittömän deduplikoinnin nopeuden kannalta on tärkeää, että tiivisteindeksi on tallennettu mahdollisimman nopeaan muistiin. Tiivisteindeksiä luetaan välittömän deduplikoinnin aikana satunnaislukuna. Keskusmuistin lukuviive on tyypillisesti 2–3 kertaluokkaa pienempi kuin SSD-levyn, ja SSD-levyn taas 2 kertaluokkaa pienempi kuin kiintolevyn (Zhang ja Swanson 2015; Zambelli ym. 2017). Usein deduplikoitavia lohkoja on kuitenkin niin paljon, ettei indeksi mahdu keskusmuistiin. Jos datajoukossa esimerkiksi on 10 teratavua uniikkia dataa ja se deduplikoidaan 4 kilotavun lohkokoolla, 128-bittiset tiivisteet vievät 40 gigatavua tilaa. Lisäksi indeksissä vievät tilaa viittaukset lohkojen sijaintiin. Jos keskusmuisti ei riitä, ainakin osa indeksistä pidetään massamuistissa. Tällaista tilannetta, jossa joudutaan deduplikoidessa käyttämään hidasta massamuistia, kutsutaan *levypullonkaulaksi*. Levypullonkaula hidastaa deduplikointia ja huonontaa sen skaalautuvuutta.

Monet deduplikointimenetelmät uhraavat osan tilansäästöstä estääkseen levypullonkaulan syntymisen. Tällaista deduplikointia, jossa kaikkia duplikaatteja ei löydetä, kutsutaan likimääräiseksi. Sen vastinpari on eksakti deduplikointi, jossa kaikki duplikaatit löydetään. (Xia ym. 2016.)

Levypullonkaulan välttämiseen on kehitetty useita menetelmiä. Ne perustuvat esimerkiksi datan paikallisuuden tai samankaltaisuuden taikka Bloom-suodattimien tai flash-muistin hyö-

dyntämiseen (Xia ym. 2016; Lu, Nam ja Du 2012). Kolme ensin mainittua menetelmäluokkaa pyrkii vähentämään tarvetta massamuistissa sijaitsevan indeksin lukemiseen, viimeksi mainittu taas käyttää nopeampaa massamuistia indeksille. Näitä menetelmiä käsitellään tämän luvun alaluvuissa.

4.3.1 Paikallisuus

Alaluvussa 2.2 käsitelty paikallisuuden periaate pätee myös deduplikoinnissa. Ajallinen paikallisuus tarkoittaa tässä yhteydessä sitä, että jos tietty lohko kirjoitetaan kerran, se todennäköisesti kirjoitetaan pian uudestaan. Sitä voidaan hyödyntää toteuttamalla tiivisteindeksiin välimuisti. Keskusmuistissa sijaitseva välimuisti vähentää tarvetta massamuistin lukemiseen. Korvausalgoritmina käytetään LRU:ta (engl. *least recently used*), eli välimuistin täyttyessä siitä poistetaan se alkio, jonka käyttämisestä on kulunut eniten aikaa. (Paulo ja Pereira 2014; Quinlan ja Dorward 2002.) Avaruudellinen paikallisuus tarkoittaa deduplikoinnin yhteydessä sitä, että lohkot esiintyvät todennäköisesti eri datavirroissa samassa järjestyksessä. Kun kirjoitetaan lohkot A, B ja C, voidaan siis arvata, että lohkoa A seuraavat vastaisuudessakin lohkot B ja C. Tätä voidaan hyödyntää tallentamalla lohkot ja niiden tiivisteet siten, että niiden datavirrassa vallinnut järjestys säilyy. Kun lohko A kirjoitetaan toisen kerran, lohkojen B ja C tiivisteet voidaan noutaa välimuistiin. (Paulo ja Pereira 2014; Zhu, Li ja Patterson 2008.)

Guo ja Efstathopoulos (2011) hyödyntävät avaruudellista paikallisuutta järjestelmässään, joka deduplikoi likimääräisesti. Siinä keskusmuistin indeksi sisältää vain osan tiivisteistä, mikä ehkäisee levypullonkaulan syntymistä. Massamuistissa säilytetään kaikki datalohkot ja tiivisteet, ja ne on ryhmitelty säiliöihin (engl. *container*), joissa on vierekkäisiä lohkoja. Kun tiiviste löytyy indeksistä, sen sisältävä säiliö noudetaan välimuistiin. Tämä vähentää osittaisen indeksoinnin aiheuttamaa deduplikointisuhteen heikentymistä.

4.3.2 Samankaltaisuus

Tavallisessa varmuuskopioinnissa esiintyy yleensä paljon paikallisuutta, kun samoja hakemistoja varmuuskopioidaan useaan kertaan. Jos taas varmuuskopioidaan useasta lähteestä satunnaisessa järjestyksessä saapuvia tiedostoja, ei paikallisuutta ole. (Bhagwat ym. 2009.)

Silloin on keksittävä jokin muu keino levypullonkaulan välttämiseksi. Eräs vaihtoehto on etsiä datasta samankaltaisia alueita.

Samankaltaisuuteen perustuvat indeksit koostuvat ainakin kahdesta kerroksesta. Ensimmäisessä kerroksessa pyritään löytämään deduplikoitavalle lohkolle samankaltaisten lohkojen ryhmä, jonka jäseniin sitä myöhemmissä kerroksissa verrataan tarkemmin. Ryhmiä on vähemmän kuin niiden jäseniä, joten aiemmat kerrokset vievät vähemmän tilaa kuin myöhemmät. Aiemmat kerrokset voidaankin pitää keskusmuistissa ja myöhemmät massamuistissa. (Tolič ja Brodnik 2015.)

Extreme Binning (Bhagwat ym. 2009) on likimääräinen deduplikointijärjestelmä, joka hyödyntää samankaltaisuutta. Siinä kutakin tiedostoa edustamaan valitaan yksi sen lohkoista. Valintaan käytetään Broderin 1997 teoreemaa. Sen perusteella kahdella tiedostolla on todennäköisesti paljon samoja lohkoja, jos niiden pienin lohkon tiiviste on sama. Kunkin tiedoston edustava lohko on siis se, jonka tiiviste on pienin. Vain edustavat lohkot pidetään keskusmuistissa. Niiden perusteella etsitään samankaltaiset tiedostot, joiden loput lohkot haetaan massamuistista ja deduplikoidaan.

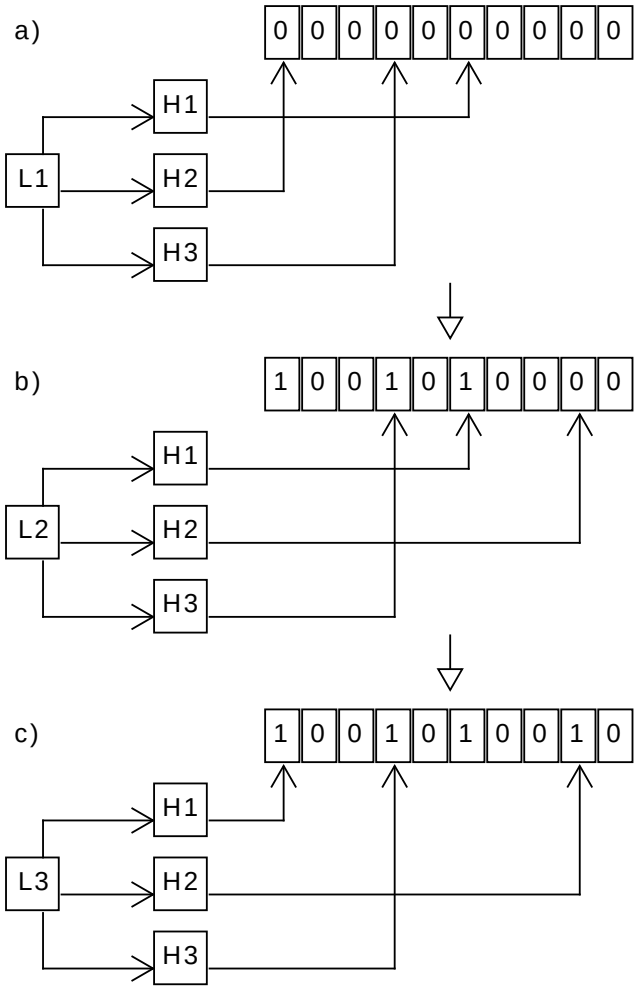
4.3.3 Bloom-suodatin

Bloom-suodatin (Bloom 1970) on tietorakenne, jota käytetään testaamaan, kuuluuko alkio tiettyyn joukkoon. Bloom-suodatin kertoo joko että alkio ei varmasti kuulu joukkoon tai että se mahdollisesti kuuluu joukkoon. Väärät positiiviset tulokset ovat siis mahdollisia, mutta väärät negatiiviset eivät. Bloom-suodattimen etuna verrattuna muihin vastaaviin tietorakenteisiin on, että sen aika- ja tilavaativuus eivät riipu alkioiden määrästä (Phyu ja Sinha 2021). Tämän hintana on väärien positiivisten tulosten mahdollisuus.

Tyhjä Bloom-suodatin on m:n bitin bittitaulukko, jonka kaikki bitit ovat nollia. Kun suodattimeen lisätään alkio, sille lasketaan k tiivistettä k:lla erilaisella hajautusfunktiolla, jotka palauttavat arvoja väliltä $[0, m-1]$. Tiivisteitä käytetään indeksinä bittitaulukkoon, johon asetetaan kunkin tiivisteen kohdalle arvoksi 1. Kun halutaan selvittää, onko alkio jo aiemmin lisätty suodattimeen, lasketaan sillekin tiivisteet. Jos bittitaulukossa on yhdenkään tiivisteen kohdalla 0, alkiota ei ole aiemmin lisätty suodattimeen. Jos kaikkien tiivisteiden kohdalla on

1, alkio on saatettu lisätä jo aiemmin. On kuitenkin mahdollista, että eri alkiot ovat asettaneet bitit ykkösiksi. (Bloom 1970.)

Kun Bloom-suodatinta käytetään deduplikoinnissa, alkion roolissa on datalohkon tiiviste ja joukkona tiivisteindeksi. Jos tiiviste on uusi, ei tiivisteindeksiä tarvitse välttämättä käydä läpi. Tämä nopeuttaa uusien lohkojen kirjoittamista. Kuviossa 8 esitetään Bloom-suodattimen toiminta. Kohdassa a) lisätään datalohko tyhjään suodattimeen. Kohdassa b) lisätään toinen datalohko ja huomataan että se on uusi. Tiivisteindeksiä ei siis tarvitse käydä läpi. Kohdassa c) lisätään kolmas datalohko ja huomataan että se ei välttämättä ole uusi. Tiivisteindeksistä selviää, että kyseessä on väärä positiivinen tulos ja kolmaskin lohko on uusi.



Kuvio 8. Bloom-suodatin.

4.3.4 Flash-muisti

Vaikka aiemmin esitellyillä menetelmillä voidaan huomattavasti vähentää massamuistissa olevan indeksin lukemista, sitä kuitenkin välillä luetaan. Indeksin lukuoperaatio kestää keskusmuistista noin 1 mikrosekunnin, flash-muistista noin 100 mikrosekuntia ja kiintolevyltä noin 10 millisekuntia. Lukuoperaatio voidaan saada osumaan keskusmuistiin lähes 99 prosentissa tapauksista. Kiintolevy on niin hidas, että vaikka vain yksi sadasta lukuoperaatiosta menisi keskusmuistin sijaan kiintolevylle, on lukuoperaation keskimäärin viemä aika satakertainen verrattuna pelkästään keskusmuistista luettuun indeksiin. Jos kiintolevyn sijasta käytetään flash-muistia, voi lukuoperaation keskimäärin viemä aika olla jopa 50 kertaa pienempi. (Debnath, Sengupta ja Li 2010.)

Flash-muisti tarjoaa indeksin säilyttämiseen enemmän tilaa kuin keskusmuisti ja suuremman nopeuden kuin kiintolevy. Indeksin vaatimat pienten datamäärien kirjoitusoperaatiot eivät kuitenkaan sovi kovin hyvin flash-muistille (Paulo ja Pereira 2014). Flash-muistissa data säilötään lohkoina, jotka koostuvat sivuista. Sivun koko on neljän kilotavun luokkaa, ja lohkossa on noin 128 sivua. Dataa luetaan ja kirjoitetaan sivu kerrallaan, mutta poistetaan lohko kerrallaan. Lisäksi sivut täytyy kirjoittaa lohkon sisällä peräkkäin, ja jo kirjoitetun sivun päivittäminen vaatii, että lohko on tyhjä. (Chen, Luo ja Zhang 2011.) Siksi pienetkin kirjoitusoperaatiot vaativat usein kokonaisten lohkojen sisällön siirtelyä. Ongelma voidaan ratkaista käyttämällä lokimuotoista indeksiä, johon kirjoitetaan kerralla monta sivua. (Debnath, Sengupta ja Li 2010.)

4.4 Muokkaustiheyden huomioiminen

Jotkin järjestelmät (esim. (El-Shimi ym. 2012)) pyrkivät deduplikoimaan vain tiedostoja, jotka eivät ole aktiivisen muokkauksen kohteena. Esimerkiksi vain tiettyä rajaa vanhemmat tiedostot deduplikoidaan. Näin vähennetään työtä, joka säästäisi tilaa todennäköisesti vain hetkellisesti.

Jos tiedosto on jatkuvan muokkauksen kohteena, sen deduplikoinnista ei ole juuri hyötyä. Joka kerta kun tiedostoa muokataan, joudutaan deduplikoimaan uudestaan, eivätkä tulokset ole päteviä kuin seuraavaan muutokseen asti. Laskentaresursseja kuluu tällöin suhteellisen

turhaan. Deduplikointi kannattaisi tehdä vasta, kun tiedostoa ei aktiivisesti muokata.

4.5 Pirstoutumisen välttäminen

Tiedostoille pyritään yleensä löytämään peräkkäisiä lohkoja tallennustilasta, jotta niiden lukeminen olisi mahdollisimman nopeaa (Park ja Park 2016). Deduplikointi kuitenkin kuvaa duplikaattilohkot paikkoihin, joissa niiden alkuperäiset kappaleet ovat. Tällöin tiedoston lohkot voivat sijaita aivan eri paikoissa massamuistia, eli ne ovat pirstoutuneet. Tämä hidastaa niiden lukua (eli rekonstruointinopeutta) etenkin perinteiseltä kiintolevyltä.

Pirstoutumista vältetään deduplikoinnissa muutamilla eri tavoilla. Deduplikointi voidaan rajata vain riittävän pitkiin peräkkäisiin lohkoryhmiin (ks. esim. Srinivasan ym. 2012; Mao ym. 2014). Pirstoutuneita lohkoja voidaan myös kopioida parempiin paikkoihin deduplikoinnin jälkeen (ks. esim. Kaczmarczyk ym. 2012; Lillibridge, Eshghi ja Bhagwat 2013; Fu ym. 2014). Pirstoutumisen määrä riippuu myös lohkokoosta: mitä pienempää lohkokokoa käytetään, sitä enemmän tiedostot voivat pirstoutua. Kokonaisten tiedostojen deduplikointi ei aiheuta pirstoutumista.

Esimerkkinä tilanteesta, jossa syntyy pirstoutumista, voidaan käyttää varmuuskopioiden välitöntä deduplikointia. Ensimmäinen varmuuskopio tallennetaan levylle yhtenäisenä jaksona. Toisen varmuuskopion uniikit lohkot tallennetaan ensimmäisen varmuuskopion perään, ja duplikaattilohkot viittaavat ensimmäiseen varmuuskopioon. Kun varmuuskopioita tulee lisää, ajan kanssa viimeisin varmuuskopio on fyysisesti erittäin hajallaan pitkin levyä. (Kaczmarczyk ym. 2012.) Koska uusin varmuuskopio on yleensä todennäköisin palautettava, olisi parempi, että se on levyllä yhtenäisenä. Se onnistuu edellä mainitulla tavalla: kopioimalla lohkoja siten, että pirstoutuminen siirtyy uusista varmuuskopioista vanhoihin.

5 Ensisijaisen tallennustilan jälkikäteinen deduplikointi

Tässä luvussa käsitellään kirjallisuudessa esitettyjä ratkaisuja ensisijaisen tallennustilan jälkikäteiseen deduplikointiin. Työssä toteutettu ohjelma, jota käsitellään luvussa 6, kuuluu tähän kategoriaan. Luku on jaettu alalukuihin, joissa kussakin käsitellään aihetta, joka nousi esiin useassa kirjallisuuslähteessä.

Ensisijaisessa tallennustilassa oleva data on erilaista kuin muualla. Varmuuskopioinnin voidaan odottaa tuottavan paljon duplikaatteja, kun samoja datajoukkoja varmistetaan useaan kertaan. Ensisijaisessa tallennustilassa tällaista yksinkertaista tilaisuutta merkittävään tilansäästöön ei yleensä ole.

Ensisijaiseenkin tallennustilaan syntyy ajan kanssa toisteisuutta, kun käyttäjät ja sovellukset käsittelevät tiedostoja. Toisteisuutta synnyttävät esimerkiksi (Tarasov ym. 2012):

- identtiset ladatut tiedostot,
- käyttäjien käsin tekemät kopiot,
- versiohallintajärjestelmien tekemät kopiot sekä
- sovellusten luomat standardit ylä- ja alatunnisteet sekä mallipohjat.

Ensisijaisen tallennustilan deduplikointi on hyvin usein jälkikäteistä (Paulo ja Pereira 2014). Jälkikäteinen deduplikointi ei huononna kirjoitusoperaatioiden viivettä, mikä on eduksi aktiivisessa käytössä olevassa tallennustilassa.

5.1 Kokonaisten tiedostojen deduplikointi

Deduplikoimalla kokonaisia tiedostoja säästetään ensisijaisessa tallennustilassa usein lähes yhtä paljon tilaa kuin deduplikoimalla pienempiä lohkoja. Constantinescun, Gliderin ja Chamblissin (2011) tutkimuksessa kokonaisten tiedostojen deduplikointi pärjäsi hyvin etenkin verkkosivudatan ja versiohallinnan tietovarastojen kohdalla. Se toimii parhaiten mediatiedostoille ja huonoiten osista koostuville tiedostoille, kuten pakatuille tiedostoille ja virtuaalikoneiden levykuville. Lohkomistapojen erot pienenevät entisestään, jos deduplikointiin yhdistetään pakkaaminen. Samankaltaisia tuloksia saivat myös Meyer ja Bolosky (2012),

jotka totesivat kokonaisten tiedostojen deduplikoinnin saavuttavan noin kolme neljäsosaa aggressiivisimman lohkoperusteisen deduplikoinnin tilansäästöstä. Heidän mittauksissaan kokonaisten tiedostojen duplikaatteja oli eniten ohjelmistojen binääritiedostoissa. Myös Lu ym. (2012) huomasivat kokonaisten tiedostojen deduplikoinnin toimivan hyvin kotihakemistoissa sekä ohjelmistoprojekteissa ja huonosti virtuaalikoneiden levykuvissa. Tässä työssä kehitetyssä ohjelmassa (ks. luku 6) käytetään kokonaisten tiedostojen deduplikointia.

Joskus kokonaisten tiedostojen deduplikointi on kuitenkin tilansäästön kannalta selvästi huonompi vaihtoehto kuin pienempien lohkojen deduplikointi. El-Shimi ym. (2012) tutkivat palvelimia, joissa säilytetään käyttäjien itse tekemiä tiedostoja. Niissä säästettiin $2,3-15,8$ kertaa enemmän tilaa vaihtelevan lohkokoon deduplikoinnilla verrattuna kokonaisten tiedostojen deduplikointiin. Eron Meyerin ja Boloskyn (2012) tutkimukseen arveltiin johtuvan siitä, että käyttäjien itse tekemissä tiedostoissa on todennäköisemmin hienojakoisempaa toisteisuutta kuin ohjelmistojen binääritiedostoissa.

5.2 Tiedostojärjestelmät

Jotkin tiedostojärjestelmät tukevat jälkikäteistä deduplikointia. Näihin kuuluvat ainakin Btrfs (“Deduplication” 2021) ja XFS (“XFS” 2022). Deduplikointi käyttää Linux-ytimen virtuaalisen tiedostojärjestelmän iotctl_fideduperange-operaatiota (Chernov ym. 2018; “ioctl_fideduperange” 2021). Tälle operaatiolle annetaan argumentteina deduplikoitavien lohkojen tiedot, jotka on kerätty erillisellä työkalulla, kuten duperemove-ohjelmalla (“Deduplication” 2021). Operaatio deduplikoi lohkot käyttämällä reflinkejä (ks. alaluku 2.3). Tässä työssä toteutettua ohjelmaa olisi mahdollista jatkokehittää tähän sopivaksi työkaluksi.

5.3 Tallennusverkot

Hong ym. (2004) esittelevät menetelmän jälkikäteiseen deduplikointiin SAN (Storage Area Network) -tallennusverkossa. Se on myös tutkielmaan löydetyistä lähteistä ensimmäinen, joka käyttää jälkikäteistä deduplikointia ensisijaiseen tallennustilaan. Heidän menetelmässään tarvitaan keskitetty metadatapalvelin, jolle verkon asiakkaat lähettävät datalohkojensa tiivisteitä. Palvelin hoitaa deduplikoinnin.

Keskitetyn palvelimen huono puoli on, että sen toimintahäiriö pysäyttää koko järjestelmän toiminnan. Se saattaa myös ruuhkautua tai aiheuttaa rinnakkaisuusongelmia. Clementsin ym. (2009) ratkaisussa deduplikointitaulu on jaetulla levyllä. Verkon asiakkaat keräävät lokia omista kirjoitusoperaatioistaan ja tietyin väliajoin päivittävät deduplikointitaulua, jolloin ne löytävät duplikaatit. Asiakkaiden määrän kasvaessa taulua jaetaan osiin, ja sitä suojataan konflikteilta tiedostolukoilla. Tämä ratkaisu ei kärsi edellä mainituista ongelmista.

Edelliset menetelmät toimivat kumpikin vain yhden tietyn tiedostojärjestelmän kanssa. Paulo ja Pereira (2016) kehittivät yleiskäyttöisemmän järjestelmän, joka vaatii toimiakseen vain rajapinnan jaetuille lohkoille. Se on myös optimoitu tarpeeksi hyvin, jotta deduplikointi voi tapahtua samaan aikaan normaalin käytön kanssa. Aiemmissa menetelmissä tämän esti etenkin copy-on-write -operaatioiden aiheuttama yleisrasite. Järjestelmässä ei myöskään ole mitään keskitettyä osaa.

5.4 Virtuaalikoneet

Suuri osa ensisijaisen tallennustilan deduplikointiin liittyvästä tutkimuksesta käsittelee tallennusverkkoja ja pilvitallennuspalveluita. Näissä järjestelmissä ajetaan usein hyvin monia virtuaalikoneita, jotka käyttävät samoja käyttöjärjestelmiä ja ohjelmia (Clements ym. 2009). Ne ovat hyvä kohde deduplikoinnille.

Usein halutaan ajaa useaa samanlaista tai lähes samanlaista virtuaalikonetta. Silloin kannattaa tallettaa copy-on-write-levykuva, jonka pohjalta virtuaalikoneet luodaan. Tällaisen staattisen toisteisuuden lisäksi voidaan deduplikoida myös virtuaalikoneissa käytettyjen ohjelmien luomaa dynaamista dataa. Duplikaatteja voidaan etsiä myös useamman virtuaalikoneen välisesti. (Paulo ja Pereira 2016.)

5.5 Luokittelu

Tiedostojen ominaisuudet ja käyttötavat kannattaa ottaa huomioon deduplikoinnissa. Jakamalla tiedostot luokkiin, jotka määräävät niille käytettävän deduplikointitavan ja -järjestyksen, voidaan parantaa tilansäästöä ja vähentää pirstoutumisen haittavaikutuksia. Pirstoutumisen

haittavaikutusten välttäminen on erityisen tärkeää ensisijaisessa tallennustilassa.

Tässä käsitellyissä tutkimuksissa luokittelua käytettiin seuraavissa järjestelmissä: SAUD (Tang, Yin ja Lo 2015), Mudder (Tang, Yin ja Wu 2016), D3 (Yin ym. 2018), SLADE (Wu ym. 2018) ja Parkin ja Parkin (2016) nimetön järjestelmä, josta käytetään tässä nimeä Selective. Käytettyjä luokittelutapoja olivat:

- tiedoston koko (SAUD, Mudder, D3),
- tiedostomuoto (SAUD, D3),
- tiedoston viimeisimmän käytön ajankohta (SAUD, Mudder, D3, SLADE, Selective),
- tiedoston käytön peräkkäisyys (Selective) ja
- tiedoston sisältämän pisimmän redundantin datajakson pituus (D3).

Suurien tiedostojen deduplikointi on intuitiivisestikin järkevämpää kuin pienten. Siten säästetään enemmän tilaa vähemmillä laskentaresursseilla. Jos pienien tiedostojen deduplikointia ei vain lykätä vaan ne jätetään kokonaan huomiotta, voidaan vähentää pirstoutumista.

Tiedostomuodosta voidaan varsin hyvin päätellä, miten hyvin tiedosto deduplikoituu milläkin lohkomistavalla. Esimerkiksi virtuaalikoneiden levykuvat sisältävät usein paljon identtisiä lohkoja. Mediatiedostot taas eivät juuri deduplikoidu muutoin kuin kokonaisten tiedostojen tasolla. Tiedostomuodolla voi olla jonkin verran yhteyttä myös tiedoston kokoon ja käyttömäärään. (Tang, Yin ja Lo 2015; Yin ym. 2018.) Tiedostomuoto on siis tehokas peruste valittaessa tiedostojen deduplikointitapaa ja -järjestystä.

Ensisijaisen tallennustilan käytöstä tehdyissä tutkimuksissa on huomattu, että yleensä vain pientä osaa tiedostoista käytetään säännöllisesti. Suurin osa tiedostoista on levyllä käyttämättömänä. Lisäksi tiedostoja muokataan tai poistetaan eniten samana päivänä kuin ne on luotu. (Gibson ja Miller 1998; Leung ym. 2008.) Näitä havaintoja hyödynnetään deduplikoimalla ensin tiedostoja, joiden käytöstä on jo aikaa tai jopa estämällä vähän aikaa sitten käytettyjen tiedostojen deduplikointi. Siten vältetään mahdollisten muokkausten turhaksi tekemää deduplikointia, joka myös hidastaisi tiedoston lukua.

Tiedostojen käyttötapoja tarkkailemalla voidaan valita myös niille sopiva lohkomistapa. Jos tiedostoa luetaan pääasiassa peräkkäisistä kohdista, sitä voi olla järkevää deduplikoida suurella

lohkokoolla, jolloin pirstoutuminen vähenee. Jos taas tiedostoa luetaan satunnaisista kohdista, sille sopii paremmin pieni lohkokoko. Näin maksimoidaan tilansäästö, eikä pirstoutuminen juuri haittaa, koska tiedostoa luetaan muutoinkin satunnaisista kohdista. Selective (Park ja Park 2016) jakaa tiedostot neljään luokkaan:

1. alle kaksi päivää sitten luodut tai muokatut tiedostot (ei deduplikoida),
2. alle kaksi viikkoa sitten luetut tiedostot, joiden luku on ollut pääasiassa peräkkäistä (deduplikoidaan suurella lohkokoolla),
3. alle kaksi viikkoa sitten luetut tiedostot, joiden luku on ollut pääasiassa satunnaista (deduplikoidaan pienellä lohkokoolla) ja
4. yli kaksi viikkoa sitten viimeksi luetut tiedostot (deduplikoidaan pienellä lohkokoolla).

Luokka 1 estää turhan deduplikoinnin niiltä tiedostoilta, joita muokataan paljon. Luokka 2 mahdollistaa nopean peräkkäisluvun niille tiedostoille, joilla sitä tarvitaan. Luokka 3 maksimoi tilansäästön tiedostoille, joiden lukunopeuteen pirstoutuminen vaikuttaa tavallista vähemmän. Luokka 4 maksimoi tilansäästön tiedostoille, joiden lukunopeudella on tavallista vähemmän merkitystä.

D3:lla (Yin ym. 2018) on tutkituista järjestelmistä monimutkaisin tapa määrittää tiedostojen deduplikointijärjestys. Siihen vaikuttavat tiedoston koon, muodon ja viimeisimmän käyttöajan lisäksi sen pisimmän redundantin datajakson pituus sekä järjestelmän viimeaikainen luku- ja kirjoitusoperaatioiden suhde. Kahden viimeksi mainitun huomioimisella pyritään entisestään vähentämään luku- ja kirjoitusoperaatioiden viivettä.

Tässä työssä toteutettua ohjelmaa olisi mahdollista jatkokehittää siten, että käyttäjä voisi määritellä suodattimia deduplikointiin. Niiden perusteella voitaisiin hakemistoja läpikäytäessä jättää deduplikoinnista pois tiedostoja, jotka ovat esimerkiksi liian pieniä tai väärää tiedostomuotoa.

6 Ohjelma

Työssä toteutettiin ohjelma, joka etsii samansisältöisiä tiedostoja. Ohjelmalle annetaan komentoriviargumentteina tiedosto- tai hakemistopolkuja, joista löydetyt duplikaattitiedostot voidaan poistaa tai korvata linkeillä. Ohjelman toteutuskieli on C++. Ohjelman deduplikointi on jälkikäteistä ja eksaktia kokonaisten tiedostojen deduplikointia. Ohjelmaa voi käyttää sekä ensisijaisen että toissijaisen tallennustilan deduplikointiin.

Ohjelmani eroaa tutkimuskirjallisuudessa käsitellyistä ohjelmista siten, että se toimii sovellustasolla eikä tiedostojärjestelmätasolla tai sen alapuolisella lohkotasolla. Tästä johtuen ohjelmani toiminnassa on niihin verrattuna suuria eroja. Sekä välittömästi että jälkikäteen deduplikoivat ohjelmat vahtivat tiedostojärjestelmässä tapahtuvia muutoksia, ja niillä on koko ajan ajantasainen tieto tiedostojärjestelmän tilasta. Ne ovat jatkuvasti suorituksessa. Ohjelmani ei pysty sieppaamaan tiedostojärjestelmän operaatioita, joten se ei voi luottaa tiedostoista keräämiensä tietojen ajantasaisuuteen. Se ei siksi säilytä näitä tietoja kuin suorituksensa ajan, koska ne olisivat ensi kerralla vanhentuneita. Ohjelmani ei ole jatkuvasti suorituksessa, vaan se ajetaan haluttaessa eräänlaisena ”siivousoperaationa”.

Ohjelma toteutettiin sovellustasolla, koska se oli yksinkertaisinta ja mahdollisti ohjelman yleiskäyttöisyyden. Deduplikoivan tiedostojärjestelmän luominen olisi ollut liian vaativaa, varsinkaan sellaisen, jota kukaan haluaisi siirtyä käyttämään. Lohkotason järjestelmä olisi varsin yleiskäyttöinen, mutta sekin olisi vaativa toteuttaa.

Ohjelma deduplikoi kokonaisia tiedostoja, mikä on merkittävästi yksinkertaisempaa toteuttaa kuin tiedostojen osien deduplikointi. Suurin ero on se, että tiedoston osien deduplikointi vaatisi ohjelman, joka on jatkuvassa suorituksessa vahtimassa tiedosto-operaatioita, toisin kuin kokonaisten tiedostojen deduplikointi. Tämä johtuu siitä, että tiedostojärjestelmät tarjoavat yleensä linkityksen vain kokonaisille tiedostoille. Kun duplikaattitiedosto korvataan linkillä, tiedostojärjestelmä hoitaa linkin hallinnan. Jos taas linkitettäisiin redundantteja tiedostojen osia, pitäisi linkin hallinta toteuttaa itse. Esimerkiksi lukiessa tiedostoa, jossa on redundantteja osia, ohjelman täytyisi seurata linkkejä.

6.1 Ohjelman toiminta

Ohjelman suorituksen vaiheet ovat pääpiirteissään seuraavat:

1. Komentorivillä annetaan argumentit, joista ainoa pakollinen on tiedosto- tai hakemistopolku tai -polut.
2. Annetut polut skannataan eli niissä olevien tiedostojen polut kerätään muistiin sekä lasketaan tiedostojen yhteismäärä ja -koko.
3. Kerätyistä tiedostoista etsitään duplikaatit.
4. Löydetyt duplikaatit käsitellään argumenttina annetulla tavalla.

Vaiheet käydään seuraavaksi läpi tarkemmin.

6.1.1 Vaihe 1: Komentoriviargumentit

Ohjelman käyttöliittymänä toimii komentorivi. Ohjelmalle voi antaa seuraavanlaisia argumentteja:

- Tiedosto- tai hakemistopolku tai -polut, joiden sisältämät tiedostot halutaan deduplikoida.
- Käydäänkö polut läpi rekursiivisesti vai ei.
- Tiiviste lasketaan tiedoston alussa olevasta osasta. Tämän osan pituus voidaan antaa argumenttina. Pituus voi olla mikä tahansa tavumäärä. Jos pituudeksi annetaan 0, tiiviste lasketaan koko tiedostosta.
- Tiedostoista laskettavan tiivisteen pituus. Vaihtoehdot ovat 1, 2, 4 ja 8 tavua.
- Duplikaateille tehtävä toimenpide. Toimenpiteen voi valita seuraavista vaihtoehdoista: pelkkä yhteenveto duplikaateista, duplikaattien listaus, poistaminen tai linkittäminen.
- Duplikaattiehdokkaiden säilytystapa eli indeksin tyyppi. Vaihtoehtoja ovat kahden tiivisteen hajautustaulu, tiivistevektori sekä tiivisteitä käyttämätön vektori.

6.1.2 Vaihe 2: Tiedostojen skannaus

Toisessa vaiheessa ohjelma skannaa annetut polut. Polut käydään läpi rekursiivisesti, jos ohjelmalle on annettu kyseinen argumentti. Kustakin tiedostosta tallennetaan sen polkunimi,

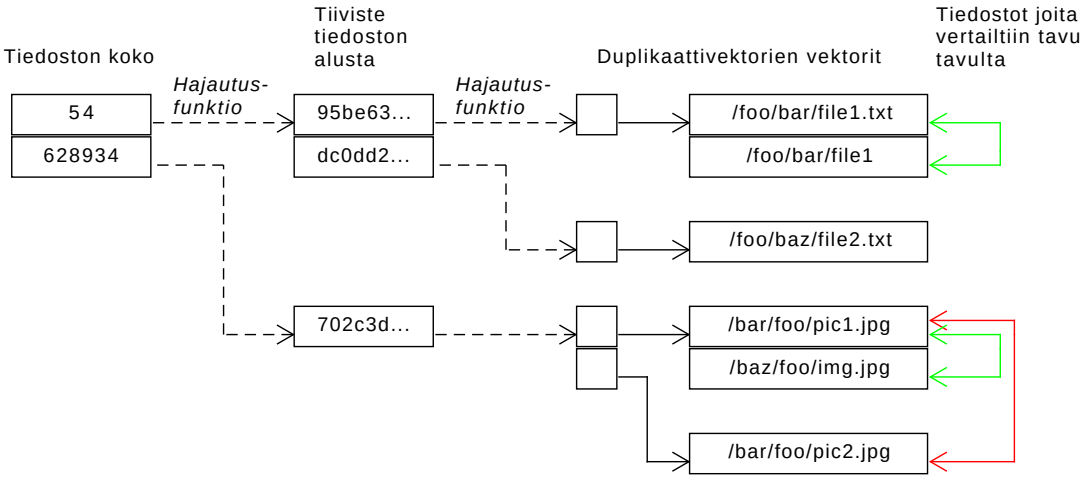
koko ja viimeisen muutoksen ajankohta. Tyhjistä tiedostoista ei kuitenkaan tallenneta mitään. Myöskään muista kuin ”tavallisista tiedostoista”, joihin eivät kuulu esimerkiksi symboliset linkit, nimetyt putket tai laitetiedostot, ei tallenneta mitään. Jos löydetään kaksi tiedostoa, jotka ovat samalla laitteella ja joiden tiedoston tunnistenumero on sama, ne ovat kovia linkkejä jotka osoittavat samaan tiedostoon. Näistä vain ensin löydetyn tiedot tallennetaan. Lisäksi lasketaan niiden tiedostojen, joiden tietoja tallennettiin, kokonaismäärä ja -koko. Tiedostojen kokonaismäärän avulla voitiin toteuttaa deduplikointivaiheeseen edistymispalkki.

6.1.3 Vaihe 3: Duplikaattien etsiminen

Ennen duplikaattien etsimistä hyödynnetään sitä tosiseikkaa, etteivät tiedostot voi olla sisällöltään identtiset, jos ne ovat eri kokoiset. Edellisessä vaiheessa kerätyistä tiedostoista poistetaan ne tiedostot, joiden kokoisia on vain yksi kappale. Sitten aletaan lisäämään tiedostoja indeksiin, jonka avulla duplikaatit löydetään.

Oletuksena indeksinä (ks. kuvio 9) käytetään kaksikerroksista hajautustaulua. Ulomman taulun avaimena on tiedoston pituus ja arvona sisempi taulu. Sisemmän taulun avaimena on tiedoston alusta laskettu tiiviste ja arvona vektori, joka sisältää kaikki tiedostot, jotka tuottivat kyseisen tiivisteen. Vektorin alkioina on duplikaattien vektoreita, jotka sisältävät täysin identtisiä tiedostoja (jotka on vertailtu tavu tavulta). Vektori on C++:n säiliöluokka, joka säilyttää alkionsa taulukossa ja muuttaa kokoaan tarpeen mukaan automaattisesti (“std::vector” 2021).

Kun tiedosto (siis siitä skannausvaiheessa kerätyt tiedot) lisätään deduplikointitauluun, sen alusta lasketaan ensin tiiviste. Tiivisteessä huomioonotettavan osan pituus ja itse tiivisteen pituus annetaan ohjelmalle argumenttina. Tiedosto lisätään sitten kokonsa ja tiivisteensä mukaiseen kohtaan deduplikointitaulua. Jos sillä kohdalla sijaitseva vektori ei ole tyhjä, verrataan lisättävää tiedostoa tavu tavulta kunkin duplikaattien vektorin ensimmäiseen tiedostoon. Kunkin tiedoston kohdalta vertailu voidaan luonnollisesti keskeyttää heti jos löydetään eroavaisuus, eli tiedostoja ei tarvitse välttämättä käydä kokonaan läpi. Jos identtinen tiedosto löytyy, tiedosto laitetaan kyseiseen duplikaattien vektoriin. Jos tiedosto on uniikki, se laitetaan uuteen duplikaattien vektoriin. Identtisillä tiedostoilla on sama sisältö, mutta tässä ohjelmas-



Kuvio 9. Ohjelman oletusindeksi.

sa niiden muita ominaisuuksia, kuten nimeä ja käyttöoikeuksia ei huomioida duplikaattien tunnistamisessa. Kun kaikki tiedostot on lisätty deduplikointitauluun, duplikaattitiedostot löytyvät duplikaattien vektoreista, joissa on enemmän kuin yksi tiedosto.

Ohjelman vaihtoehtoisia indeksityyppejä ovat kahden tiivisteen hajautustaulu, tiivistevektori ja sisältövektori. Kahden tiivisteen hajautustaulussa käytetään tiedostojen vertailuun ensin tiedoston alkuosasta laskettua tiivistettä, kuten oletusindeksissäkin. Jos se tiiviste on sama, lasketaan tiiviste koko tiedostosta ja käytetään sitä, ja jos sekin on sama, verrataan vielä tiedostoja tavu tavulta. Tiivistevektoria käytettäessä kerätään vektoriin pareja, jotka muodostuvat tiedoston alkuosasta lasketusta tiivisteestä ja tiedoston muista tiedoista. Sitten vektori järjestetään tiivisteen perusteella, jolloin saman tiivisteen tuottaneet tiedostot ovat vierekkäin. Lopuksi kunkin tiivisteryhmän tiedostot vertaillaan keskenään, jolloin duplikaatit löytyvät. Sisältövektori toimii muutoin samalla tavalla kuin tiivistevektori, mutta tiivisteiden sijaan vektoriin kerätään tiedostojen alut sellaisinaan.

Oli indeksin tyyppi mikä hyvänsä, kerätään löydetyt duplikaatit lopuksi erilliseen vektorien vektoriin. Tämä kokoelma välitetään sitten seuraavalle vaiheelle.

6.1.4 Vaihe 4: Duplikaattien käsittely

Tässä vaiheessa duplikaateille tehdään argumenttina annettu toimenpide. Toimenpide on yksi seuraavista:

- Tulostetaan pelkkä yhteenveto löydettyjen duplikaattien määrästä ja koosta. Yhteenveto tulostetaan aina, vaikka oltaisiin valittu jokin muu toimenpide.
- Tulostetaan lista kaikkien löydettyjen duplikaattien poluista. Samansisältöisten duplikaattien ryhmät erotellaan toisistaan tyhjällä rivillä.
- Interaktiivinen toiminto, jossa kustakin duplikaattien ryhmästä valitaan, mikä tai mitkä tiedostot halutaan säilyttää.
- Ei-interaktiivinen toiminto, jossa kustakin duplikaattien ryhmästä säilytetään yksi tiedosto. Loput joko poistetaan tai linkitetään alkuperäiseen kovalla linkillä tai symbolisella linkillä. Säilytettävä tiedosto valitaan seuraavalla periaatteella: jos argumenttina on annettu useampi polku, aiemmin annetuista poluista löytyneet tiedostot ovat etusijalla. Jos useampi tiedosto on löytynyt samasta annetusta polusta, etusijalla on tiedosto, jonka viimeisimmän muutoksen ajankohta on aikaisempi. Jos molemmat tekijät ovat samat useammalla tiedostolla, ei ole määritelty, mikä niistä valitaan säilytettäväksi.

6.2 Ohjelman kehitysprosessi

Ohjelma kirjoitettiin C++-kielellä. Käytettyjä ei-standardeja kirjastoja olivat

- cxxopts komentoriviargumenttien jäsentämiseen,
- xxHash tiivisteiden laskemiseen tiedostojen sisällöstä sekä
- Catch2 yksikkötestejä varten.

Tärkeä osa ohjelmaa on myös C++17:ssä standardikirjastoon lisätty Filesystem-kirjasto, joka helpottaa tiedostojärjestelmän käsittelyä.

Hajautustauluun täytyy kunkin tiivisteen kohdalle tallentaa keinot, joilla tiivisteen tuottaneet tiedostot löydetään. Päädyin käyttämään tiedoston polkunimeä. Vaihtoehtona olisi ollut tiedoston ja laitteen tunnistenumeron (jotka Unix-järjestelmissä löytyvät i-solmusta) yhdistelmä. Polkunimi toimii suoraan Filesystem-kirjaston kanssa ja on alustariippumaton. Tunniste-

numerot taas ovat erilaisia eri käyttöjärjestelmissä, eikä Filesystem-kirjasto tunnista niitä. Tunnistenumeron avulla ei myöskään saa suoraan selville tiedoston polkunimeä, vaan se on etsittävä käymällä läpi hakemistorakennetta, kunnes kyseistä tunnistenumeroa vastaava tiedosto löytyy. Tunnistenumeroiden etuna olisi ollut pienempi muistinkäyttö.

Kukin polkunimi tallennetaan kokonaisena. Muistinkäyttöä oltaisiin voitu pienentää käyttämällä esimerkiksi pakattua trie-rakennetta, jossa solmuja olisivat voineet olla polun osat eli hakemistot ja tiedostonimet. En kuitenkaan käyttänyt sellaista, koska kokonaisten polkunimien tallentaminen ei normaaleilla tiedostojen ja keskusmuistin määrillä nouse ongelmaksi. Kun ajoin ohjelman tietokoneeni juurihakemistoon, jossa oli 3 850 170 tiedostoa, ohjelma käytti enimmillään 1,35 gigatavua muistia (käyttöjoukon koko, ks. alaluku 8.5).

Kiinnitin ohjelmassa erityistä huomiota siihen, ettei dataa voi menettää vahingossa. Kun tiedostoja skannataan (ks. alaluku 6.1.2), niiden viimeisimmän muokkauksen aikaleima tallennetaan. Jos duplikaatteja poistettaessa tai linkitettäessä tiedoston senhetkinen aikaleima on uudempi kuin tallennettu aikaleima, tiedostoa on saatettu muokata erilaiseksi. Toimenpiteen jatkaminen johtaisi silloin datan menetykseen, joten se perutaan. Kyseessä on time-of-check to time-of-use -tyyppinen ongelma (Bishop ja Dilger 1996).

Dataa voidaan menettää myös, jos sama tiedosto lisätään deduplikoitavaksi kahteen kertaan. Näin voisi käydä, jos argumenttina annetaan kaksi samaan paikkaan johtavaa polkunimeä tai jos skannauksen aikana linkit johtavat samaan paikkaan. Tiedostoa luultaisiin tällöin itsensä duplikaatiksi, ja duplikaattien poistaminen johtaisi tiedoston datan menetykseen. Ohjelmassa kuitenkin tarkistetaan, etteivät argumenttina annetut polut johda samaan paikkaan. Skannausvaiheessa löydetyt symboliset ja kovat linkit jätetään huomiotta. Olisi myös mahdollista tarkistaa, ettei linkkien takia lisätä samoja tiedostoja, mutta oletin, että linkit on yleensä tehty tarkoituksella eikä niitä haluta deduplikoida. Niiden deduplikoinnilla ei myöskään säästettäisi juurikaan tilaa.

Kolmas riski datan menetykseen johtuu tiivisteiden törmäyksestä (ks. alaluku 3.3.2). Ohjelmassani tätä riskiä ei ole, sillä pelkkiin tiivisteisiin ei luoteta. Tiedostoja, joilla on sama tiiviste, vertaillaan tavu tavulta, kunnes niiden väliltä löytyy ero tai tiedostot on luettu kokonaan. Koska tiivisteen ei tässä menetelmässä tarvitse olla kuin alustava vihje tiedostojen

mahdollisesta samanlaisuudesta, voidaan sen vahvuutta heikentää ohjelman nopeuttamiseksi. Tiiviste lasketaan vain tiedoston alkuosasta, mikä tarkoittaa, ettei tiedostoja välttämättä lueta kuin tämän alkuosan verran. Tiedostoa joudutaan lukemaan pidemmälle vain jos päädytään tavu tavulta vertailuun. Itse tiivisteen ei myöskään tarvitse olla kovin pitkä.

7 Mittaukset

7.1 Suoritusajan mittaaminen

Kun ajetaan samaa tietokoneohjelmaa samoilla syötteillä useaan kertaan, on suoritusajoissa lähes aina eroa. Syitä tähän ovat muun muassa (Touati, Worms ja Briais 2010):

- Taustalla ajettavat muut ohjelmat
- Keskeytykset
- Prosessien vuoronnus
- Välimuistit
- I/O-operaatiot
- Suorittimen hidastuminen liian suuren lämpötilan takia
- Mittauksen epätarkkuus

Jos halutaan mitata ohjelman suoritusaikaa luotettavasti, täytyy suorituskertoja siis olla paljon. Usein mittauksen tuloksia halutaan kuvata yhdellä luvulla. Vaihtoehtoja luvuksi ovat ainakin suoritusaikojen minimi, keskiarvo ja mediaani.

Suoritusaikojen minimi ei yleensä ole hyvä mittari. Mikään ei takaa sitä, että pienempi minimi johtuisi ohjelman optimoinnista tai että se olisi ideaali suoritus, jossa tietokoneen toiminnasta johtuva kohina on vähäisintä. Mikä tärkeintä, minimi on yleensä tapauksena harvinainen eikä siten kuvasta tyypillistä suoritusaikaa. (Touati, Worms ja Briais 2010)

Mediaani on keskiarvoa parempi suure kuvaamaan suoritusaikojen jakauman sijaintia. Mediaani ei ole yhtä herkkä poikkeaville havainnoille. Suoritusaikojen jakaumat ovat usein myös vinoja, jolloin mediaani on lähempänä tyypillisintä suoritusaikaa. (Touati, Worms ja Briais 2013). Esimerkiksi jakauman {600, 605, 580, 1500, 595} keskiarvo on 776 ja mediaani 600.

Suurin ero suoritusajassa ohjelmani eri suorituskertojen välillä johtuu tiedostojärjestelmän välimuistista. Jos ohjelmani tarvitsemat tiedostot ovat jo välimuistissa, suoritusaika lyhenee merkittävästi. Poistan tämän muuttujan tyhjentämällä välimuistin ennen jokaista suorituskertaa. Tällöin suorituskerrat ovat myös toisistaan riippumattomia.

Ohjelmani toimii deterministisesti, jos syötteenä annettujen polkujen sisältö ei muutu eikä niiden käsittelyssä ilmene ongelmia esimerkiksi käyttöoikeuksien kanssa. Tehdyissä mittauksissa nämä oletukset pätevät. Tällöin suoritusaikojen erot eivät johdu ohjelman toiminnan logiikasta.

7.2 Mittausten kuvaus

Mittausten tavoitteena on selvittää, miten tiivisteen laskemisessa huomioonotettavan tiedoston osan pituus sekä indeksin tyyppi vaikuttavat ohjelman suoritusnopeuteen.

Hypoteesina on, että sopiva mitta tiedoston osalle on sama kuin levyn lohkokoko. Tämä johtuu siitä, että levyltä noudetaan joka tapauksessa vähintään lohkon koon verran dataa. On järkevää hyödyntää se kokonaan, sillä tiivisteen laskeminen on hyvin paljon nopeampaa kuin datan noutaminen levyltä. Jos mitta on lyhyt, useampi tiedosto päätyy samaan listaan. Levyn lohkokoko ei kuitenkaan olisi paras mitta, jos käsiteltävillä tiedostoilla olisi yhteinen, tunnettu rakenne, jonka mukaan niillä olisi identtinen, lohkoa pidempi alkuosa. Tällöin kannattaisi käsitellä pidempää alkuosaa. On myös mahdollista, että niin suuresta osasta käsiteltäviä tiedostoja löytyy eroja niin läheltä niiden alkua, että on nopeinta huomioida levyn lohkokokoa lyhyempi osa tiedostoista.

Indeksin tyypin vaikutusta on vaikeampi ennustaa. Jonkinlaisia arvioita voidaan kuitenkin esittää. Kahden tiivisteen hajautustaulu on yhden tiivisteen hajautustaulua nopeampi silloin, kun koko tiedoston tiivisteen käytöstä on hyötyä. Se, milloin näin on, riippuu varsin monesta asiasta. Jos tiedostojen aluissa on eroja, molemmat indeksit toimivat samalla tavalla. Jos tiedostojen alut ovat samanlaiset, molemmissa indekseissä on ensin laskettu ja verrattu niiden alun tiivisteitä ja todettu ne samoiksi. Sen jälkeen tiedoston sisällölle on kaksi vaihtoehtoa:

1. Tiedostot ovat identtiset. Yhden tiivisteen hajautustaululla luetaan tiedostot kokonaan läpi. Kahden tiivisteen hajautustaululla lasketaan ensin koko tiedostoista tiivisteet, todetaan ne samoiksi ja luetaan tiedostot vielä uudelleen läpi. Tässä tilanteessa yhden tiivisteen hajautustaulu on siis nopeampi.
2. Tiedostot eroavat loppuosaltaan. Yhden tiivisteen hajautustaululla luetaan tiedostoja sen verran, että ero löytyy. Kahden tiivisteen hajautustaululla taas lasketaan koko tiedostois-

ta tiivisteet ja vertaillaan niitä. Tässä tilanteessa nopeamman indeksin määräytyminen on monimutkaisempaa. Jos samalla tavalla alkaneita tiedostoja on enemmän kuin kaksi, kahden tiivisteen hajautustaululla voidaan selvitä vähemmällä tiedostojen lukemisella kuin yhden tiivisteen hajautustaululla. Tämä riippuu kuitenkin siitä, missä kohtaa tiedostoja ero löytyy.

Mittauksissa oli alun perin tarkoitus tutkia myös tiivisteen pituuden vaikutusta ohjelman suoritusnopeuteen. Vaikutus osoittautui kuitenkin hyvin pieneksi. Tulosten esityksen yksinkertaistamiseksi tämä muuttuja jätettiin pois. Kaikissa esitetyissä mittauksissa on käytetty 8 tavun pituista tiivistettä.

Mittauksissa ohjelma ajetaan erilaisilla aineistoilla. Aineistoja ovat:

- Linux-ytimen version 5.9 lähdekoodi,
- Linux-ytimen versioiden 5.0, 5.1, 5.2, . . . , 5.9 (10 versiota) lähdekoodi,
- kirjoittajan henkilökohtainen kuvakokoelma sekä
- FUSE-korpuksesta (Barik ym. 2015) poimitut 3000 laskentataulukkotiedostoa.

Taulukossa 1 esitetään tietoja aineistoista.

	Linux	Linuxit	Kuvat	Laskenta-taulukot
Tiedostojen määrä	69 972	663 382	16 416	3 000
Koko (GiB)	0,91	8,19	119	0,32
Duplikaattitiedostoja (%)	0,43	71,3	4,56	67,7
Duplikaattitiedostojen osuus tiedostojen yhteiskoosta (%)	0,13	58,3	9,37	74,4
Tiedostoja, joilla uniikki koko (%)	18,7	2,34	84,5	4,20
Tiedostojen mediaanikoko (KiB)	3,76	3,80	3770	18,6
Tallennusväline	SSD	SSD	HDD	HDD
Ajokertojen määrä per konfiguraatio	30	15	15	30

Taulukko 1. Aineistot

Ohjelma ajettiin kullakin aineistolla kaikilla tutkittavien muuttujien yhdistelmillä. Tiivisteen

laskemisessa huomioonotettavan tiedoston osan pituuden arvoja olivat 128, 512, 1024, 2048, 4096 ja 8192 tavua. Indeksityyppejä olivat yhden tiivisteen hajautustaulu, kahden tiivisteen hajautustaulu, tiivistevektori ja sisältövektori. Muuttujien yhdistelmiä eli konfiguraatioita oli siis $6 \times 4 = 24$ kappaletta. Kutakin konfiguraatiota ajettiin 15 tai 30 kertaa aineistosta riippuen.

Ohjelma on käännetty GCC 8.4.0 -kääntäjällä käyttäen optiota -O3. Aineistoista kaksi sijaitsee sisäisellä SSD-levyllä, jonka tiedostojärjestelmä on Ext4 ja joka on yhdistetty tietokoneen SATA III -liitäntään. Kaksi muuta aineistoa sijaitsevat ulkoisella kiintolevyllä, jonka tiedostojärjestelmä on NTFS ja joka on yhdistetty tietokoneen USB 3 -liitäntään. Kummankin levyn kohdalla liitännät tukevat suurempaa nopeutta kuin mihin levyt yltävät. Ext4- ja NTFS-tiedostojärjestelmien oletuslohkokoko on 4096 tavua ("Filesystems in the Linux Kernel" 2022; "Default cluster size for NTFS, FAT, and exFAT" 2022).

7.3 Tilastollinen analyysi

Mittauksissa tutkittiin kahden riippumattoman muuttujan (tiedoston alusta huomioitavien tavujen määrä ja indeksin tyyppi) vaikutusta yhteen riippuvaan muuttujaan (suoritusaika) erilaisilla aineistoilla. Kukin riippumattomien muuttujien yhdistelmä muodosti oman ryhmänsä. Tähän käyttöön sopiva tilastollinen testi olisi ollut yksisuuntainen varianssianalyysi (eli ANOVA). Se on parametrinen testi, eli se olettaa tutkittavan perusjoukon noudattavan parametrista jakaumaa, joka on tässä tapauksessa normaalijakauma (Montgomery ja Runger 2003, luku 15.1.). Riippuvan muuttujan arvojen pitäisi siis olla likimain normaalijakautuneita kussakin ryhmässä. Lisäksi ryhmien varianssien täytyisi olla likimain samansuuruisia. (Dinno 2015.) Normaaliutta testattiin Shapiro-Wilkin testillä ja varianssien samansuuruisuutta Levenen testillä. Ryhmien varianssit olivat erisuuruisia jokaisella aineistolla, eivätkä arvot olleet osalla ryhmistä normaalisti jakautuneita ($p < 0,05$). Tämän takia tilastolliseen analyysiin käytettiin yksisuuntaisen varianssianalyysin sijaan sitä vastaavaa parametritonta testiä: Kruskal-Wallisin testiä.

Kruskal-Wallisin testi on parametriton, eli se ei oleta tutkittavan perusjoukon jakaumasta mitään. Testissä asetetaan kaikki havainnot järjestykseen ja verrataan kunkin ryhmän jär-

jestyslukujen keskiarvoja. (Montgomery ja Runger 2003, luku 15.) Toisin kuin ANOVA, Kruskal-Wallisin testi ei siis etsi eroja ryhmien keskiarvojen välillä, vaan niiden stokastisen dominanssin välillä. Ryhmä on stokastisesti dominantti, jos siitä satunnaisesti poimittu havainto on todennäköisesti suurempi kuin muista ryhmistä satunnaisesti poimittu havainto. Jos riippumattoman muuttujan jakauma on jatkuva ja ryhmien jakaumat ovat samanmuotoiset, Kruskal-Wallisin testin voidaan lisäksi sanoa etsivän eroa ryhmien mediaanien välillä. (Dinno 2015.)

Kruskal-Wallisin testin nollahypoteesi on, että ryhmien järjestyslukujen keskiarvot ovat yhtä suuria (McDonald 2014). Jos testistä saadaan merkitsevä tulos, ainakin yhden ryhmän järjestyslukujen keskiarvo poikkeaa merkittävästi muista. Kruskal-Wallisin testi ei kuitenkaan kerro, minkä ryhmien välillä on eroa. (Corder ja Foreman 2014, luku 6.2.) Sitä varten täytyy tehdä parittaisia vertailuja erillisellä testillä. Testiksi valittiin Dunnin testi, koska sitä pidetään suosituimpana testinä tähän tarkoitukseen (Mangiafico 2015). Kun parittaisia testejä tehdään monta, on suurempi todennäköisyys hylätä ainakin yksi nollahypoteesi virheellisesti. Siksi merkitsevyystasoa (jota pienemmillä p-arvoilla nollahypoteesi hylätään) täytyy pienentää (McDonald 2014.) Siihen käytettiin Holmin korjausta, joka on varsin yleisesti käytetty eikä yhtä konservatiivinen kuin yleisimmin käytetty Bonferronin korjaus (Aickin ja Gensler 1996).

Suoritusaikojen eroja tutkittiin erikseen kullakin aineistolla ja riippumattomien muuttujien yhdistelmällä. Vertailu aloitettiin Kruskal-Wallisin testillä. Jos siitä saatiin merkitsevä tulos, ryhmien väliset erot selvitettiin Dunnin testillä. Tilastollinen analyysi suoritettiin R-ohjelmointiympäristössä.

8 Tulokset

Mittausten tuloksia kuvataan laatikko-janakuvioilla. Tällaisessa kuviossa on laatikko, jonka poikki kulkeva viiva on havaintojen mediaanin kohdalla. Laatikon ala- ja yläreuna kuvaavat arvojen ala- ja yläkvartiilia, eli laatikko sisältää puolet havainnoista. Laatikon ylä- ja alareunoista alkavat janat, joiden pituus vaihtelee laatikko-janakuvion eri varianteissa. Tässä tutkielmassa käytäntönä on, että ylempi jana ulottuu suurimpaan arvoon, joka on enintään puolentoista kvartiilivälin päässä yläkvartiilista. Vastaavasti alempi jana ulottuu pienimpään arvoon, joka on enintään puolentoista kvartiilivälin päässä alakvartiilista. Näitä poikkeavammat havainnot esitetään erillisillä ympyröillä.

Indeksin tyypeistä käytetään seuraavia lyhenteitä: oletuksena käytetty, yhden tiivisteen hajautustaulu on 1TH, kahden tiivisteen hajautustaulu on 2TH, tiivistevektori on TV ja sisältövektori SV.

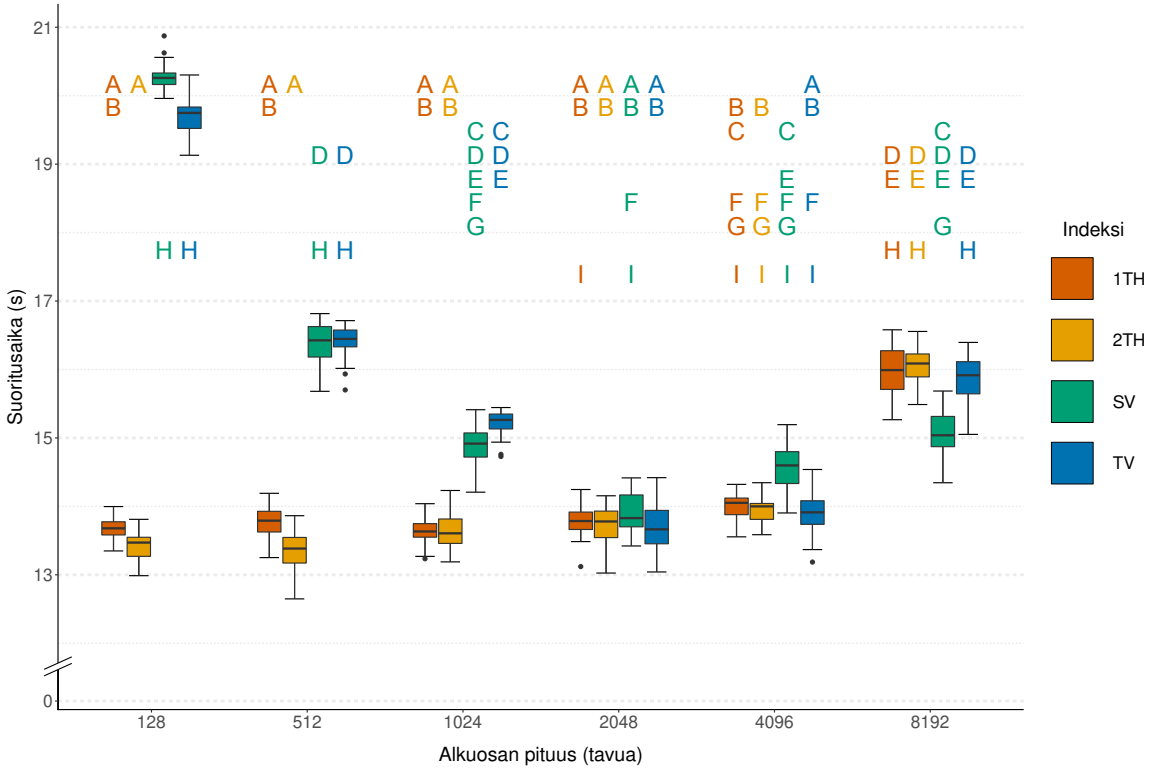
Jokaisessa vertailussa saatiin Kruskal-Wallisin testistä merkitsevä tulos. Kaikki p-arvot olivat pienempiä kuin $2{,}2 \times 10^{-16}$. Täten jokaisessa vertailussa suoritettiin myös Dunnin testi.

Kuvioissa esitetään myös Dunnin testin tulokset kirjainten avulla. Kahden ryhmän välinen ero ei ole tilastollisesti merkitsevä, jos niiden kohdalle on merkitty sama kirjain. Silloin niiden välisen Dunnin testin p-arvo on suurempi kuin 0,05. Esimerkiksi kuvio 10 kertoo, ettei ohjelman suoritusajassa ole tilastollisesti merkitsevää eroa konfiguraatioiden [128 tavua, 1TH] ja [128 tavua, 2TH] välillä, kun aineistona on Linux-ytimen version 5.9 lähdekoodi. Sen sijaan konfiguraatioiden [128 tavua, 1TH] ja [128 tavua, SV] välillä ohjelman suoritusajassa on tilastollisesti merkitsevä ero.

Kuvioita tarkastellessa on syytä huomata, että y-akseli on katkaistu nollan jälkeen.

8.1 Linux 5.9

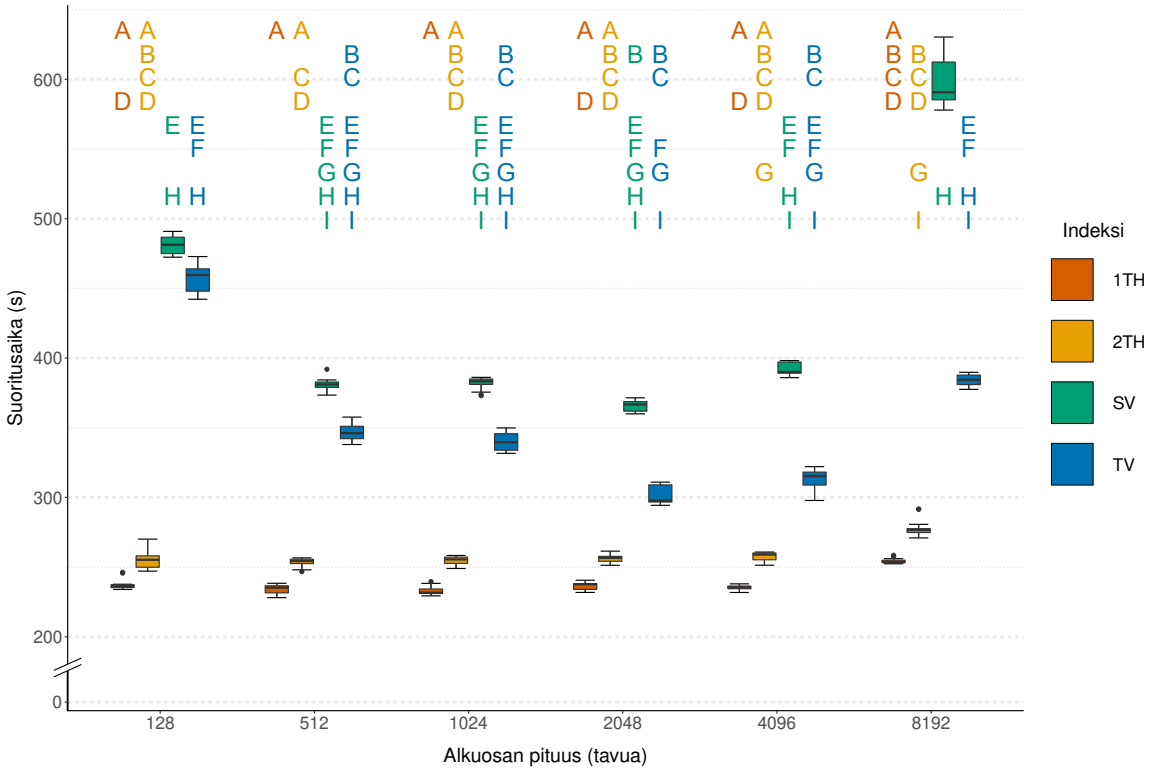
1TH:n ja 2TH:n suoritusajat ovat hyvin lähellä toisiaan, samoin SV:n ja TV:n. Alkuosan pituus ei juurikaan vaikuta 1TH:n ja 2TH:n suoritusaikoihin, paitsi 8192 tavun kohdalla. SV:n ja TV:n suoritusaikoihin se vaikuttaa enemmän.



Kuvio 10. Ohjelman suoritusaika kullakin indeksillä ja tiedoston alusta huomioitavan osan pituudella, aineistona Linux 5.9.

8.2 Linux 5.0–5.9

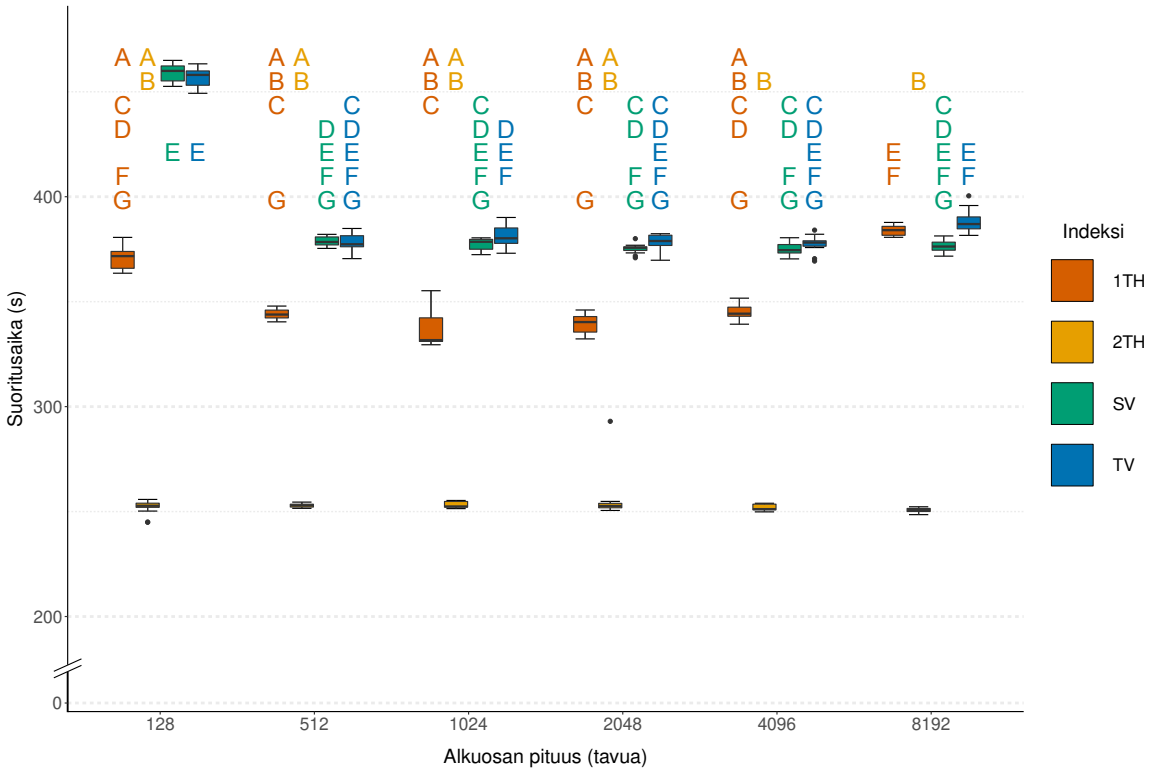
1TH:n ja 2TH:n suoritusajat ovat hyvin lähellä toisiaan, mutta 1TH on hieman nopeampi. Alkuosan pituus ei vaikuta niiden suoritusaikoihin, paitsi hieman 8192 tavun kohdalla. TV on jonkin verran nopeampi kuin SV. Niiden suoritusaikoihin vaikuttavat eniten alkuosan pituuksien ääripäät.



Kuvio 11. Ohjelman suoritusaika kullakin indeksillä ja tiedoston alusta huomioitavan osan pituudella, aineistona Linux 5.0–5.9.

8.3 Kuvakokoelma

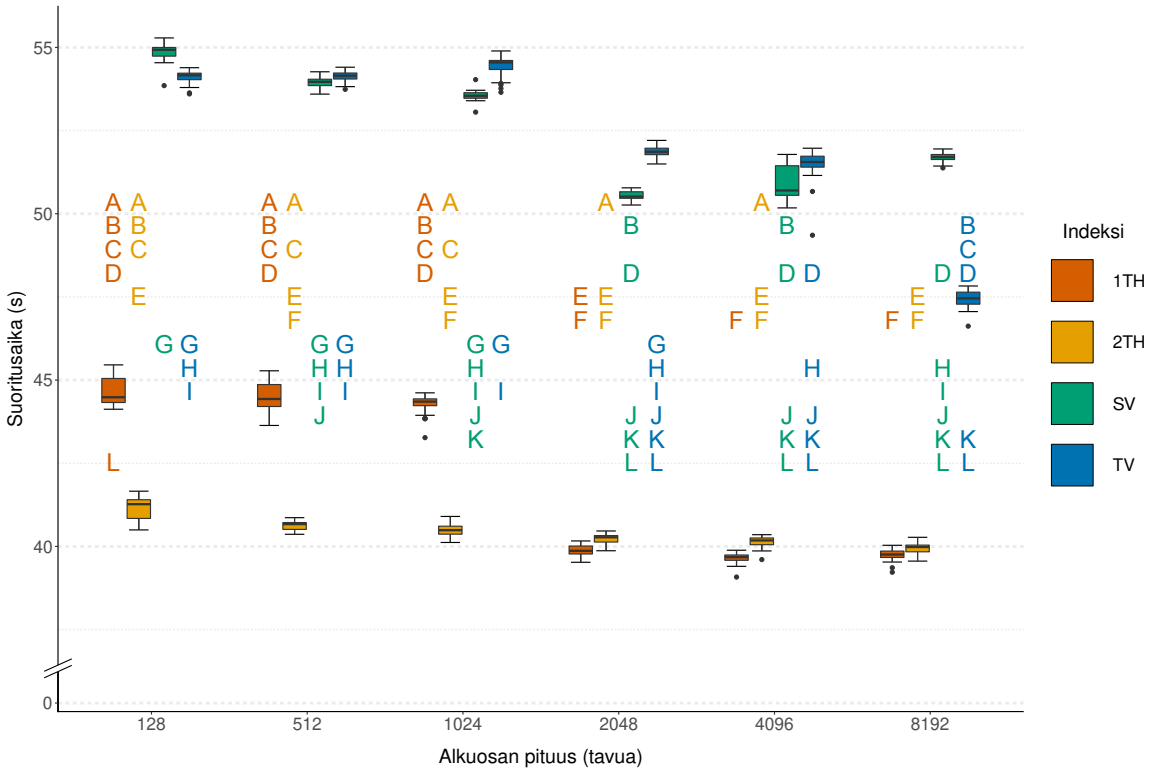
2TH on nopein, eikä sen nopeuteen vaikuta alkuosan pituus. 1TH:n nopeus heikkenee alkuosan pituuden ääripäissä. SV:n ja TV:n suoritusajat ovat hyvin lähellä toisiaan, eikä alkuosan pituus vaikuta niiden nopeuteen kuin 128 tavun kohdalla.



Kuvio 12. Ohjelman suoritusaika kullakin indeksillä ja tiedoston alusta huomioitavan osan pituudella, aineistona kuvakokoelma.

8.4 Laskentataulukkotiedostot

1TH:n ja 2TH:n suoritusajat ovat melko lähellä toisiaan, samoin SV:n ja TV:n. Suuremmat alkuosan pituudet nopeuttavat suoritusaikoja hieman, mikä viittaa siihen, että aineistossa on paljon samalla tavalla alkavia tiedostoja.



Kuvio 13. Ohjelman suoritusaika kullakin indeksillä ja tiedoston alusta huomioitavan osan pituudella, aineistona laskentataulukkotiedostot.

8.5 Mittaustulosten yhteenveto

Kuvioiden 10–13 perusteella nähdään, että ohjelman suoritusajat ovat melko lähellä toisiaan 1TH:lla ja 2TH:lla. 1TH on kuitenkin selvästi hitaampi kuvakokoelman kohdalla. SV:n ja TV:n suoritusajat ovat yleensä lähellä toisiaan, ja ne ovat muita hitaampia. Alkuosan pituus vaikuttaa suoritusaikoihin merkittävästi yleensä vain 128 tai 8192 tavun kohdalla.

Taulukoissa 2 ja 3 esitetään erilaiset yhteenvedot suoritusajan mittaustuloksista. Taulukoissa olevat luvut on saatu seuraavasti:

1. Kunkin aineiston kohdalla etsitään muuttujien yhdistelmä eli konfiguraatio, jolla suoritusaikojen mediaani on pienin.
2. Kyseisen aineiston kaikki suoritusaikojen mediaanit skaalataan jakamalla ne tällä pienimmällä mediaanilla. Pienin mediaani saa siis arvon 1 ja muut sitä suurempia arvoja.
3. Kullekin konfiguraatiolle lasketaan skaalattujen arvojen keskiarvo eli arvojen summa jaetaan neljällä, joka on aineistojen lukumäärä. Nämä keskiarvot esitetään taulukossa 2.
4. Kunkin konfiguraation skaalattujen arvojen maksimi esitetään taulukossa 3.

		Alkuosan pituus (tavua)					
		128	512	1024	2048	4096	8192
Indeksi	1TH	1,1616	1,1343	1,1148	1,1044	1,1100	1,2056
	2TH	1,0391	**1,0328**	1,0366	1,0399	1,0445	1,1008
	SV	1,7024	1,4353	1,4070	1,3468	1,3864	1,6193
	TV	1,6626	1,3980	1,3743	1,2810	1,3020	1,3968

Taulukko 2. Suoritusajan skaalattujen mediaanien keskiarvo kullakin konfiguraatiolla.

		Alkuosan pituus (tavua)					
		128	512	1024	2048	4096	8192
Indeksi	1TH	1,4821	1,3711	1,3225	1,3568	1,3728	1,5314
	2TH	1,1019	**1,0976**	1,1030	1,1076	1,1179	1,2020
	SV	2,0774	1,6447	1,6546	1,5829	1,6835	2,5500
	TV	1,9836	1,5046	1,5160	1,5108	1,5078	1,6591

Taulukko 3. Suoritusajan skaalattujen mediaanien maksimi kullakin konfiguraatiolla.

Taulukosta 2 nähdään, että konfiguraatiolla [512 tavua, 2TH] suoritusajat ovat yleisesti nopeimmat. 2TH pärjää hyvin muillakin alkuosan pituuksilla. 1TH on seuraavaksi nopein, ja SV ja TV ovat hitaimpia. Alkuosan pituuden vaikutus suoritusaikoihin on merkittävä vain ääriarvoissaan.

Taulukko 3 kertoo, miten kukin konfiguraatio pärjää huonoimmassa aineistossaan. Konfiguraatio [512 tavua, 2TH] on nytkin nopein. 2TH on tasaisen varma suoriutuja, eikä sen nopeus

juuri heikkene millään alkuosan pituudella eikä missään aineistossa. 1TH on toiseksi nopein, sitten TV ja lopuksi SV. Alkuosan pituus ei vaikuta hitaimpaan suoritusaikaan juurikaan, paitsi ääriarvoissaan.

Suoritusajan lisäksi mitattiin muistinkulutusta. Indekseillä 1TH, 2TH ja TV muistinkulutus ei riippunut tiedoston alkuosan pituudesta, mutta indeksillä SV riippui. Muistinkulutus selvitettiin käyttämällä GNU Time -ohjelmaa ("time(1) – Linux manual page" 2022). Se raportoi mitattavan ohjelman käyttöjoukon koon (engl. *resident set size*) maksimiarvon. Muistinkulutus on esitetty taulukoissa 4 ja 5. 1TH ja 2TH käyttivät vähiten muistia, TV jonkin verran enemmän ja SV selvästi eniten.

		Testiaineisto			
		Linux	Linuxit	Kuvat	Laskenta-taulukot
Indeksi	1TH	20,48	133,5	7,060	4,916
	2TH	18,98	176,0	7,042	5,016
	TV	24,49	247,1	7,177	5,227

Taulukko 4. Muistinkulutus (mebitavua), viiden mittauksen keskiarvo

		Testiaineisto			
		Linux	Linuxit	Kuvat	Laskenta-taulukot
Alkuosan pituus (tavua)	128	28,16	261,2	7,396	5,262
	512	49,10	493,5	7,423	6,513
	1024	78,34	812,8	7,661	8,020
	2048	134,4	1445	10,14	10,88
	4096	245,5	2710	15,25	16,82
	8192	467,6	5238	25,42	28,80

Taulukko 5. Muistinkulutus (mebitavua), indeksinä sisältövektori, viiden mittauksen keskiarvo

Tulosten perusteella voidaan sanoa, että muuttujien oletusarvoiksi sopivat hyvin 2TH ja 512 tavua. Niillä ohjelma toimi nopeiten eikä ollut missään aineistossa kovin hidas. Myös muistinkulutus oli 2TH:lla lähellä vähäisintä. Alkuosan pituudeksi kävisi myös 1024, 2048 tai 4096 tavua, sillä ne eivät olleet tuloksissa kaukana 512 tavusta.

Hypoteesina oli, että levyn lohkokoko eli tässä tapauksessa 4096 tavua olisi paras mitta alkuosan pituudelle. 4096 tavua oli lähellä parasta, mutta sitä pienemmätkin arvot 512 tavuun asti suoriutuivat suunnilleen yhtä hyvin. 128 tavua kuitenkin jo yleensä hidasti ohjelman toimintaa. Tämä johtuu todennäköisesti siitä, että aineistoissa oli paljon tiedostoja, joiden 128 ensimmäistä tavua olivat samanlaiset. 512 tavua riitti jo erottelemaan enemmän tiedostoja. 8192 tavua taas oli turhan paljon, eikä sen vaatimasta toisen levylohkon noutamisesta ollut hyötyä. Koska 512 tavua oli nopein alkuosan pituus ja 128 tavua hitaimpien joukossa, jatkotutkimuksissa voitaisiin selvittää esimerkiksi 256 tavun nopeutta.

Ennen mittauksia ohjelmassa käytettiin oletuksena yhden tiivisteen hajautustaulua. Se jäi kuitenkin mittauksissa jälkeen kahden tiivisteen hajautustaulusta. Kuten luvussa 7.2 pohdittiin, näiden indeksien välisten nopeuserojen syyt ovat varsin monimutkaisia. Erot riippuvat duplikaattien osuudesta, samalla tavalla alkavien tiedostojen määrästä ja siitä, mistä kohtaa ensimmäiset tiedostojen väliset erot löytyvät. Vektori-indeksit olivat hitaampia kuin hajautustauluindeksit. Indeksityyppien toimintaperiaatteet ovat niin erilaiset, että nopeuserojen syitä on hankala eritellä.

9 Yhteenveto

Datan määrä kasvaa kiihtyvästi, mikä luo painetta ottaa kaikki irti tallennusvälineistä. Siinä auttaa datan deduplikointi. Deduplikointi on ollut käytössä reilut parikymmentä vuotta, minä aikana sen suorituskykyä on pyritty jatkuvasti parantamaan.

Tässä työssä käytiin läpi deduplikointiprosessia ja siinä olevia vaihtoehtoja. Deduplikoinnilla on paljon erilaisia sovelluskohteita, ja prosessi kannattaa räätälöidä kuhunkin kohteeseen sopivaksi. Lisäksi tutustuttiin deduplikoinnin suorituskykyyn ja sen parantamispyrkimyksiin. Deduplikoinnin suorituskyvyllä on useita osa-alueita, jotka ovat usein keskenään ristiriidassa ja joista täytyy siksi valita tärkeimmät tavoiteltavat.

Työssä toteutettiin myös tiedostoja deduplikoiva ohjelma. Ohjelman deduplikointitaululle ja tiivisteen laskennassa huomioonotettavan tiedoston osan pituudelle oli erilaisia vaihtoehtoja, joiden vaikutusta ohjelman suoritusnopeuteen tutkittiin erilaisilla testiaineistoilla. Mittauksissa löydettiin deduplikointitaulu, jolla suoritusnopeus oli selkeästi paras. Tiedoston osan pituudelle löydettiin neljä arvoa, joilla suoritusnopeus oli yleisesti parempi kuin lopuilla kahdella arvolla. Lisäksi löytyi näiden kahden muuttujan arvojen yhdistelmä, joka oli sekä yleisesti nopein että huonoimmassaan aineistossa vähiten hidas. Nämä arvot sopivat siis hyvin ohjelman muuttujien oletusarvoiksi. Ohjelmaa olisi mahdollista jatkokehittää deduplikoimaan tiedostojen osia. Lisäksi voitaisiin lisätä mahdollisuus käsiteltävien tiedostojen suodattamiseen esimerkiksi koon tai tiedostomuodon perusteella.

Lähteet

Aickin, Mikel, ja Helen Gensler. 1996. “Adjusting for multiple testing when reporting research results: the Bonferroni vs Holm methods.” *American journal of public health* 86 (5): 726–728. doi:10.2105/AJPH.86.5.726.

Aleksandersen, Daniel. 2020. “Which file systems support file cloning”. Viitattu 10. toukokuuta 2022. https://www.ctrl.blog/entry/file-cloning.html.

Arpaci-Dusseau, Remzi H., ja Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces.* 1.00. Arpaci-Dusseau Books, elokuu.

Bansal, Sulabh, ja Prakash Chandra Sharma. 2021. “Classification criteria for data deduplication methods”. Teoksessa Thwel ja Sinha 2021, 69–96.

Barik, Titus, Kevin Lubick, Justin Smith, John Slankas ja Emerson R. Murphy-Hill. 2015. “Fuse: A Reproducible, Extendable, Internet-Scale Corpus of Spreadsheets”. Teoksessa *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015,* toimittanut Massimiliano Di Penta, Martin Pinzger ja Romain Robbes, 486–489. IEEE Computer Society. doi:10.1109/MSR.2015.70.

Bhagwat, Deepavali, Kave Eshghi, Darrell D. E. Long ja Mark Lillibridge. 2009. “Extreme Binning: Scalable, parallel deduplication for chunk-based file backup”. Teoksessa *17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2009, September 21-23, 2009, South Kensington Campus, Imperial College, London, UK,* 1–9. IEEE Computer Society. doi:10.1109/MASCOT.2009.5366623.

Bishop, Matt, ja Michael Dilger. 1996. “Checking for Race Conditions in File Accesses”. *Computing Systems* 9 (2): 131–152. https://www.usenix.org/publications/compsystems/1996/spr_bishop.pdf.

Bloom, Burton H. 1970. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. *Commun. ACM* 13 (7): 422–426. doi:10.1145/362686.362692.

Bobbarjung, Deepak R., Suresh Jagannathan ja Cezary Dubnicki. 2006. “Improving duplicate elimination in storage systems”. *ACM Transactions on Storage* 2 (4): 424–448. doi:10.1145/1210596.1210599.

Boldyreva, Alexandra. 2005. “The Birthday Problem”. Viitattu 10. toukokuuta 2022. https://www.cc.gatech.edu/~aboldyre/teaching/Fall05cs6260/m-birthday.pdf.

Broder, Andrei Z. 1997. “On the resemblance and containment of documents”. Teoksessa *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings,* toimittanut Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro ja James A. Storer, 21–29. IEEE. doi:10.1109/SEQUEN.1997.666900.

Chen, Feng, Tian Luo ja Xiaodong Zhang. 2011. “CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives”. Teoksessa *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011,* toimittanut Gregory R. Ganger ja John Wilkes, 77–90. USENIX. https://www.usenix.org/conference/fast11/caftl-content-aware-flash-translation-layer-enhancing-lifespan-flash-memory-based.

Chernov, Ilya, Evgeny Ivashko, Alexander Rumiantsev, Vadim Ponomarev ja Anton I. Shabaev. 2018. “Survey on Deduplication Techniques in Flash-Based Storage”. Teoksessa *22nd Conference of Open Innovations Association, FRUCT 2018, Jyväskylä, Finland, May 15-18, 2018,* 25–33. IEEE. doi:10.23919/FRUCT.2018.8468295.

Clements, Austin T., Irfan Ahmad, Murali Vilayannur ja Jinyuan Li. 2009. “Decentralized Deduplication in SAN Cluster File Systems”. Teoksessa *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009,* toimittanut Geoffrey M. Voelker ja Alec Wolman. USENIX Association. https://www.usenix.org/conference/usenix-09/decentralized-deduplication-san-cluster-file-systems.

Constantinescu, Cornel, Joseph S. Glider ja David D. Chambliss. 2011. “Mixing Deduplication and Compression on Active Data Sets”. Teoksessa *2011 Data Compression Conference (DCC 2011), 29-31 March 2011, Snowbird, UT, USA,* toimittanut James A. Storer ja Michael W. Marcellin, 393–402. IEEE Computer Society. doi:10.1109/DCC.2011.46.

Corder, Gregory W., ja Dale I. Foreman. 2014. *Nonparametric Statistics: A Step-by-Step Approach.* 2. painos. John Wiley & Sons, Inc.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest ja Clifford Stein. 2009. *Introduction to Algorithms.* 3. painos. The MIT Press.

Debnath, Biplob K., Sudipta Sengupta ja Jin Li. 2010. "ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory". Teoksessa *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010.* USENIX Association. https://www.usenix.org/conference/usenix-atc-10/chunkstash-speeding-inline-storage-deduplication-using-flash-memory.

"Deduplication". 2021. Viitattu 5. toukokuuta 2022. https://btrfs.wiki.kernel.org/index.php/Deduplication.

"Default cluster size for NTFS, FAT, and exFAT". 2022. Viitattu 23. helmikuuta. https://support.microsoft.com/en-us/topic/default-cluster-size-for-ntfs-fat-and-exfat-9772e6f1-e31a-00d7-e18f-73169155af95.

Dinno, Alexis. 2015. "Nonparametric Pairwise Multiple Comparisons in Independent Groups using Dunn's Test". *The Stata Journal* 15 (1): 292–300. doi:10.1177/1536867X1501500117.

El-Shimi, Ahmed, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li ja Sudipta Sengupta. 2012. "Primary Data Deduplication - Large Scale Study and System Design". Teoksessa *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012,* toimittanut Gernot Heiser ja Wilson C. Hsieh, 285–296. USENIX Association. https://www.usenix.org/conference/atc12/technical-sessions/presentation/el-shimi.

"Filesystems in the Linux Kernel". 2022. Viitattu 24. helmikuuta. https://www.kernel.org/doc/html/latest/filesystems/ext4/overview.html.

Fingler, Henrique, Moo-Ryong Ra ja Rajesh Krishna Panta. 2019. “Scalable, Efficient, and Policy-Aware Deduplication for Primary Distributed Storage Systems”. Teoksessa *31st International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2019, Campo Grande, Brazil, October 15-18, 2019,* 180–187. IEEE. doi:10.1109/SBAC-PAD.2019.00038.

Fu, Min, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang ja Qing Liu. 2014. “Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information”. Teoksessa *2014 USENIX Annual Technical Conference, USENIX ATC ’14, Philadelphia, PA, USA, June 19-20, 2014,* toimittanut Garth Gibson ja Nickolai Zeldovich, 181–192. USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/fu_min.

Fu, Min, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang ja Yujuan Tan. 2015. “Design Tradeoffs for Data Deduplication Performance in Backup Workloads”. Teoksessa *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015,* toimittanut Jiri Schindler ja Erez Zadok, 331–344. USENIX Association. https://www.usenix.org/conference/fast15/technical-sessions/presentation/fu.

Giampaolo, Dominic. 1999. *Practical File System Design with the Be File System.* 1. painos. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 1558604979.

Gibson, Timothy, ja Ethan L. Miller. 1998. “Long-Term File Activity Patterns in a UNIX Workstation Environment”. Teoksessa *Proceedings of the 6th Goddard Conference on Mass Storage Systems and Technologies / 15th IEEE Symposium on Mass Storage Systems,* 355–372. Maaliskuu.

Gray, Jim, ja Catharine Van Ingen. 2005. *Empirical Measurements of Disk Failure Rates and Error Rates.* Tekninen raportti MSR-TR-2005-166. Joulukuu. https://www.microsoft.com/en-us/research/publication/empirical-measurements-of-disk-failure-rates-and-error-rates/.

Guo, Fanglu, ja Petros Efstathopoulos. 2011. “Building a High-performance Deduplication System”. Teoksessa *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011,* toimittanut Jason Nieh ja Carl A. Waldspurger. USENIX Association. https://www.usenix.org/conference/usenixatc11/building-high-performance-deduplication-system.

Harnik, Danny, Ety Khaitzin ja Dmitry Sotnikov. 2016. “Estimating Unseen Deduplication - from Theory to Practice”. Teoksessa *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016,* toimittanut Angela Demke Brown ja Florentina I. Popovici, 277–290. USENIX Association. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harnik.

Hennessy, John L., ja David A. Patterson. 2012. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann. ISBN: 978-0-12-383872-8.

Hong, Bo, Demyn Plantenberg, Darrell D. E. Long ja Miriam Sivan-Zimet. 2004. “Duplicate Data Elimination in a SAN File System”. Teoksessa *21st IEEE Conference on Mass Storage Systems and Technologies / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, Greenbelt, Maryland, USA, April 13-16, 2004,* toimittanut Ben Kobler ja P. C. Hariharan, 301–314. IEEE.

“ioctl_fideduperange”. 2021. Viitattu 5. toukokuuta 2022. https://man7.org/linux/man-pages/man2/ioctl_fideduperange.2.html.

Jacob, Bruce L., Spencer W. Ng ja David T. Wang. 2008. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann. ISBN: 978-0-12-379751-3.

Kaczmarczyk, Michal, Marcin Barczynski, Wojciech Kilian ja Cezary Dubnicki. 2012. “Reducing impact of data fragmentation caused by in-line deduplication”. Teoksessa *The 5th Annual International Systems and Storage Conference, SYSTOR ’12, Haifa, Israel, June 4-6, 2012,* 11. ACM. doi:10.1145/2367589.2367600.

Kaur, Ravneet, Inderveer Chana ja Jhilik Bhattacharya. 2018. “Data deduplication techniques for efficient cloud storage management: a systematic review”. *The Journal of Supercomputing* 74 (5): 2035–2085. doi:10.1007/s11227-017-2210-8.

Kay, Dominic, ja Cindy Swearingen. 2014. “How to Determine Memory Requirements for ZFS Deduplication”. Viitattu 6. toukokuuta 2022. https://www.oracle.com/technical-resources/articles/it-infrastructure/admin-o11-113-size-zfs-dedup.html.

Kerola, Teemu. 2019. “Tiedostojärjestelmät, tiedostot ja massamuisti (Tietokoneen toiminnan jatkokurssi)”. Viitattu 10. toukokuuta 2022. https://tietokoneen-toiminnan-jatkokurssi.mooc.fi/luku-8/2-tiedostot-massamuisti.

Kim, Chulmin, Ki-Woong Park ja Kyu Ho Park. 2012. “GHOST: GPGPU-offloaded high performance storage I/O deduplication for primary storage system”. Teoksessa *Proceedings of the 2012 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2012, New Orleans, LA, USA, February 26, 2012,* toimittanut Minyi Guo ja Zhiyi Huang, 17–26. ACM. doi:10.1145/2141702.2141705.

Kim, Daehee, Sejun Song ja Baek-Young Choi. 2013. “SAFE: Structure-aware file and email deduplication for cloud-based storage systems”. Teoksessa *IEEE 2nd International Conference on Cloud Networking, CloudNet 2013, San Francisco, CA, USA, November 11-13, 2013,* toimittanut Xiaoming Fu, Puneet Sharma, Dijiang Huang ja Deep Medhi, 130–137. IEEE. doi:10.1109/CloudNet.2013.6710567.

———. 2017. *Data Deduplication for Data Optimization for Storage and Network Systems.* Springer. ISBN: 978-3-319-42278-7. doi:10.1007/978-3-319-42280-0.

Kim, Jonghwa, Choonghyun Lee, Sang Yup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won ja Jaehyuk Cha. 2012. “Deduplication in SSDs: Model and quantitative analysis”. Teoksessa *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA,* 1–12. IEEE Computer Society. doi:10.1109/MSST.2012.6232379.

Leung, Andrew W., Shankar Pasupathy, Garth R. Goodson ja Ethan L. Miller. 2008. “Measurement and Analysis of Large-Scale Network File System Workloads”. Teoksessa *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings,* toimittanut Rebecca Isaacs ja Yuanyuan Zhou, 213–226. USENIX Association. https://www.usenix.org/conference/2008-usenix-annual-technical-conference/measurement-and-analysis-large-scale-network-file.

Lillibridge, Mark, Kave Eshghi ja Deepavali Bhagwat. 2013. “Improving restore speed for backup systems that use inline chunk-based deduplication”. Teoksessa *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013,* toimittanut Keith A. Smith ja Yuanyuan Zhou, 183–198. USENIX. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lillibridge.

Liu, Chuanyi, Yingping Lu, Chunhui Shi, Guanlin Lu, David H. C. Du ja Dong-Sheng Wang. 2008. “ADMAD: Application-Driven Metadata Aware De-duplication Archival Storage System”. Teoksessa *2008 Fifth IEEE International Workshop on Storage Network Architecture and Parallel I/Os,* 29–35. doi:10.1109/SNAPI.2008.11.

Liu, Jian, Yunpeng Chai, Xiao Qin ja Yuan Xiao. 2014. “PLC-cache: Endurable SSD cache for deduplication-based primary storage”. Teoksessa *IEEE 30th Symposium on Mass Storage Systems and Technologies, MSST 2014, Santa Clara, CA, USA, June 2-6, 2014,* 1–12. IEEE Computer Society. doi:10.1109/MSST.2014.6855536.

Lu, Guanlin, Youngjin Nam ja David H. C. Du. 2012. “BloomStore: Bloom-Filter based memory-efficient key-value store for indexing of data deduplication on flash”. Teoksessa *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA,* 1–11. IEEE Computer Society. doi:10.1109/MSST.2012.6232390.

Lu, Maohua, David D. Chambliss, Joseph S. Glider ja Cornel Constantinescu. 2012. “Insights for data reduction in primary storage: a practical analysis”. Teoksessa *The 5th Annual International Systems and Storage Conference, SYSTOR ’12, Haifa, Israel, June 4-6, 2012,* 17. ACM. doi:10.1145/2367589.2367606.

Mandagere, NagaPramod, Pin Zhou, Mark A. Smith ja Sandeep Uttamchandani. 2008. “Demystifying data deduplication”. Teoksessa *Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008, Companion Proceedings,* toimittanut Fred Douglis, 12–17. ACM. doi:10.1145/1462735.1462739.

Mandal, Sonam. 2015. *Design and Implementation of an Open-Source Deduplication Platform for Research.* Tekninen raportti FSL-15-03. Stony Brook University.

Mangiafico, Salvatore S. 2015. *An R Companion for the Handbook of Biological Statistics.* 1.32. https://rcompanion.org/rcompanion.

Mao, Bo, Hong Jiang, Suzhen Wu ja Lei Tian. 2014. “POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud”. Teoksessa *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014,* 767–776. IEEE Computer Society. doi:10.1109/IPDPS.2014.84.

McDonald, J.H. 2014. *Handbook of Biological Statistics.* 3. painos. Baltimore, Maryland: Sparky House Publishing. https://www.biostathandbook.com.

Meister, Dirk, ja André Brinkmann. 2009. “Multi-level comparison of data deduplication in a backup scenario”. Teoksessa *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference 2009, Haifa, Israel, May 4-6, 2009,* toimittanut Miriam Allalouf, Michael Factor ja Dror G. Feitelson, 8. ACM International Conference Proceeding Series. ACM. doi:10.1145/1534530.1534541.

Meyer, Dutch T., ja William J. Bolosky. 2012. “A study of practical deduplication”. *ACM Trans. Storage* 7 (4): 14:1–14:20. doi:10.1145/2078861.2078864.

Min, Jaehong, Daeyoung Yoon ja Youjip Won. 2011. “Efficient Deduplication Techniques for Modern Backup Operation”. *IEEE Transactions on Computers* 60, numero 6 (kesäkuu): 824–840. doi:10.1109/TC.2010.263.

Montgomery, Douglas C., ja George C. Runger. 2003. *Applied Statistics and Probability for Engineers.* 3. painos. John Wiley & Sons, Inc.

Muthitacharoen, Athicha, Benjie Chen ja David Mazières. 2001. “A Low-Bandwidth Network File System”. Teoksessa *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001,* toimittanut Keith Marzullo ja Mahadev Satyanarayanan, 174–187. ACM. doi:10.1145/502034.502052.

Park, Sejin, ja Chanik Park. 2016. “Offline Selective Data Deduplication for Primary Storage Systems”. *IEICE TRANSACTIONS on Information and Systems* E99.D (2): 370–382. doi:10.1587/transinf.2015EDP7034.

Patterson, David A., ja John L. Hennessy. 2013. *Computer Organization and Design: The Hardware/Software Interface.* 5. painos. Morgan Kaufmann.

Paulo, João, ja José Pereira. 2014. “A Survey and Classification of Storage Deduplication Systems”. *ACM Computing Surveys* 47 (1): 11:1–11:30. doi:10.1145/2611778.

———. 2016. “Efficient Deduplication in a Distributed Primary Storage Infrastructure”. *ACM Transactions on Storage* 12 (4): 20:1–20:35. doi:10.1145/2876509.

Phyu, Myat Pwint, ja G.R. Sinha. 2021. “Efficient data deduplication scheme for scale-out distributed storage”. Teoksessa Thwel ja Sinha 2021, 153–182.

Quinlan, Sean, ja Sean Dorward. 2002. “Venti: A New Approach to Archival Storage”. Teoksessa *Proceedings of the FAST ’02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA,* toimittanut Darrell D. E. Long, 89–101. USENIX. http://www.usenix.org/publications/library/proceedings/fast02/quinlan.html.

Rabin, Michael O. 1981. *Fingerprinting by random polynomials.* Tekninen raportti TR-15-81. Center for Research in Computing Technology, Harvard University.

Reinsel, David, John Gantz ja John Rydning. 2018. “Data age 2025: the digitization of the world from edge to core”. Viitattu 10. toukokuuta 2022. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

Rhea, Sean C., Russ Cox ja Alex Pesterev. 2008. “Fast, Inexpensive Content-Addressed Storage in Foundation”. Teoksessa *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings,* toimittanut Rebecca Isaacs ja Yuanyuan Zhou, 143–156. USENIX Association. https://www.usenix.org/conference/2008-usenix-annual-technical-conference/fast-inexpensive-content-addressed-storage.

Silberschatz, Abraham, Peter Baer Galvin ja Greg Gagne. 2018. *Operating System Concepts.* 10. painos. Wiley.

Srinivasan, Kiran, Timothy Bisson, Garth R. Goodson ja Kaladhar Voruganti. 2012. “iDedup: Latency-aware, Inline Data Deduplication for Primary Storage”. Teoksessa *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012,* toimittanut William J. Bolosky ja Jason Flinn. USENIX Association. https://www.usenix.org/conference/fast12/idedup-latency-aware-inline-data-deduplication-primary-storage.

“std::vector”. 2021. Viitattu 28. syyskuuta. https://en.cppreference.com/w/cpp/container/vector.

Tai, Amy, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman ja Asaf Cidon. 2019. “Who’s Afraid of Uncorrectable Bit Errors? Online Recovery of Flash Errors with Distributed Redundancy”. Teoksessa *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019,* toimittanut Dahlia Malkhi ja Dan Tsafrir, 977–992. USENIX Association. https://www.usenix.org/conference/atc19/presentation/tai.

Tang, Yan, Jianwei Yin ja Wei Lo. 2015. “SAUD: Semantics-Aware and Utility-Driven Deduplication Framework for Primary Storage”. Teoksessa *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015,* 190–197. IEEE. doi:10.1109/HPCC-CSS-ICESS.2015.226.

Tang, Yan, Jianwei Yin ja Zhaohui Wu. 2016. “Try Managing Your Deduplication Fine-Grained-ly: A Multi-tiered and Dynamic SLA-Driven Deduplication Framework for Primary Storage”. Teoksessa *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016,* 859–862. IEEE Computer Society. doi:10.1109/CLOUD.2016.0123.

Tarasov, Vasily, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning ja Erez Zadok. 2012. “Generating Realistic Datasets for Deduplication Analysis”. Teoksessa *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012,* toimittanut Gernot Heiser ja Wilson C. Hsieh, 261–272. Boston, MA: USENIX Association. https://www.usenix.org/conference/atc12/technical-sessions/presentation/tarasov.

Thwel, Tin Thein, ja G.R. Sinha, toimittaneet. 2021. *Data Deduplication Approaches: Concepts, Strategies, and Challenges.* Academic Press. ISBN: 978-0-12-823395-5. doi:10.1016/C2020-0-00104-0.

“time(1) – Linux manual page”. 2022. Viitattu 25. helmikuuta. https://man7.org/linux/man-pages/man1/time.1.html.

Tolič, Andrej, ja Andrej Brodnik. 2015. “Deduplication in unstructured-data storage systems”. *Elektrotehniški vestnik* 82 (5): 233–242. https://www.dlib.si/?URN=URN:NBN:SI:DOC-Q2XD5BKE.

Touati, Sid, Julien Worms ja Sébastien Briais. 2010. *The Speedup Test.* Tekninen raportti inria-00443839v2. INRIA. https://hal.inria.fr/inria-00443839v2.

———. 2013. “The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation”. *Concurrency and Computation: Practice and Experience* 25 (10): 1410–1426. doi:10.1002/cpe.2939.

Wang, Jianfeng, ja Xiaofeng Chen. 2016. “Efficient and Secure Storage for Outsourced Data: A Survey”. *Data Science and Engineering* 1 (3): 178–188. doi:10.1007/s41019-016-0018-9.

Weiss, Mark Allen. 2014. *Data Structures and Algorithm Analysis in C++*. 4. painos. Addison-Wesley. ISBN: 978-0132847377.

"What is Data Deduplication?" 2019. Viitattu 10. toukokuuta 2022. https://www.netapp.com/us/info/what-is-data-deduplication.aspx.

Wu, Huijun, Chen Wang, Yinjin Fu, Sherif Sakr, Kai Lu ja Liming Zhu. 2018. "A Differentiated Caching Mechanism to Enable Primary Storage Deduplication in Clouds". *IEEE Transactions on Parallel and Distributed Systems* 29 (6): 1202–1216. doi:10.1109/TPDS.2018.2790946.

"XFS". 2022. Viitattu 5. toukokuuta 2022. https://wiki.archlinux.org/title/XFS.

Xia, Wen, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang ja Yukun Zhou. 2016. "A Comprehensive Study of the Past, Present, and Future of Data Deduplication". *Proceedings of the IEEE* 104 (9): 1681–1710. doi:10.1109/JPROC.2016.2571298.

Yan, Fang, ja YuAn Tan. 2011. "A Method of Object-based De-duplication". *Journal of Networks* 6 (12): 1705–1712.

Yin, Jianwei, Yan Tang, Shuiguang Deng, Ying Li ja Albert Y. Zomaya. 2018. "D^3: A Dynamic Dual-Phase Deduplication Framework for Distributed Primary Storage". *IEEE Trans. Computers* 67 (2): 193–207. doi:10.1109/TC.2017.2743199.

Yin, Jianwei, Yan Tang, Shuiguang Deng, Bangpeng Zheng ja Albert Y. Zomaya. 2021. "MUSE: A Multi-Tierd and SLA-Driven Deduplication Framework for Cloud Storage Systems". *IEEE Transactions on Computers* 70 (5): 759–774. doi:10.1109/TC.2020.2996638.

Zambelli, Cristian, Gabriele Navarro, Veronique Sousa, Ioan Lucian Prejbeanu ja Luca Perniola. 2017. "Phase Change and Magnetic Memories for Solid-State Drive Applications". *Proceedings of the IEEE* 105 (9): 1790–1811. doi:10.1109/JPROC.2017.2710217.

Zhang, Yiying, ja Steven Swanson. 2015. "A study of application performance with non-volatile main memory". Teoksessa *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015,* 1–10. IEEE Computer Society. doi:10.1109/MSST.2015.7208275.

Zhu, Benjamin, Kai Li ja R. Hugo Patterson. 2008. “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System”. Teoksessa *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA,* toimittanut Mary Baker ja Erik Riedel, 269–282. USENIX. `https://www.usenix.org/events/fast08/tech/zhu.html`.