TAMPEREEN TEKNILLINEN YLIOPISTO

Henrik Heino

Muokattava ja deduplikoiva tiedostojen paketointi- ja pakkausjärjestelmä

Diplomityö

Tarkastaja: Antti Valmari
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekunnan tiedekuntaneuvoston
kokouksessa 6.3.2013

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Henrik Heino: Muokattava ja deduplikoiva tiedostojen paketointi- ja pakkausjärjestelmä
Diplomityö, 59 sivua, 0 liitesivua
Marraskuu 2013
Pääaine: Ohjelmistotiede
Tarkastaja: Professori Antti Valmari
Avainsanat: deduplikointi, paketointi, pakkaus, varmuuskopio

Työn tavoitteena oli kehittää pakkaustyökalu, eli ohjelma ja sen käyttämä tiedostoformaatti. Näiden erityisvaatimuksena oli deduplikointi sekä pakettitiedoston muokattavuus. Deduplikoinnilla tarkoitetaan tässä identtisten tiedostojen ja hakemistorakenteiden tallentamista vain kerran, vaikka ne esiintyisivätkin aineistossa jopa miljoonia kertoja. Työssä kehitetty pakkaustyökalu deduplikoi koko paketoitavan aineiston laajuudelta.

Lähtökohtana työlle oli oma varmuuskopiointitapaus, jossa tärkeät tiedostot ja hakemistot tallennetaan joka yö tiettyyn pakettitiedostoon. Muokattavuutta tarvittiin, jotta pakettitiedostoon voidaan lisätä tiedostoja ja hakemistoja.

Ohjelma toteutettiin C++-ohjelmointikielellä. Ohjelman algoritmit keskittyvät lähinnä siihen, miten tiedostot ja hakemistot paketoidaan deduplikoinnin saavuttamiseksi. Varsinaisen tiedon pakkaamisen, eli tiivistämisen tekee zlib-ohjelmistokomponentti. Pakkaustyökalu tukee myös yksinkertaista AES256-pohjaista salausta.

Toteutettu työkalu pakkaa lähtökohtana olleen varmuuskopiointitapauksen erittäin tehokkaasti. Deduplikointi toimii tiedostojen lisäksi kokonaisille hakemistorakenteille ja säästää sen vuoksi paljon tilaa. Työkalua voi käyttää myös tavallisissa pakkaustapauksissa, kuten esimerkiksi ohjelmiston paketoimisessa, mutta näissä työkalun pakkaustehokkuus on korkeintaan keskitasoa.

Ohjelmasta löytyi myös virheitä. Vaikka työkalu onkin käyttökelpoinen päivittäiseen varmuuskopiointiin, tulee pakettitiedostoihin aina silloin tällöin virheitä. Näihin ongelmiin puututtiin toki jo työn aikana, mutta niitä jäi ratkottavaksi myös tulevaisuuteen. Mitkään virheet eivät kuitenkaan olleet sellaisia, että koko paketin sisältö menetettäisiin.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Henrik Heino : A Modifiable and Deduplicating File Archiving System
Master of Science Thesis, 59 pages, 0 Appendix pages
November 2013
Major: Computer Science
Examiner: Professor Antti Valmari
Keywords: deduplication, compression, archiving, backup

The aim of this work was to develop an archiving system. This means a computer program and a file format. The special requirements are deduplication and ability to modify the archive file. Here deduplication means storing identical files and directory structures only once, even if they exist multiple times in the material. The deduplication algorithm uses the entire material to find identical data.

The starting point of this work was a backup case, where important files and directories are stored every night to a certain archive file. Ability to modify the archive file is needed, so that more files and directories can be added.

The program has been written in C++ programming language. Algorithms in the program mainly focus on achieving deduplication. Actual data compression is done using zlib software library. The archiving system also has a simple AES256-based encryption.

The archiving system performs very well on the backup case. Deduplication handles entire directory structures, so the archiving system succeeded in storing lots of identical data into a small space The archiving system can also be used with traditional archiving cases, for example to pack some software into a single file. However in these cases, the compression efficiency is only mediocre at best.

Also some errors were found in the program. The system is usable for daily backupping, although occasionally the archive files contain errors. Most of these errors were fixed, but some still remain to be fixed in the future. However, none of these errors causes a total loss of the archive file.

ALKUSANAT

Tämä työ on jatkoa omalle harrasteprojektilleni. Haluan osoittaa kiitokseni koko Tampereen teknillisen yliopiston kannustavalle ja joustavalle ilmapiirille sekä ihmisille, jotka mahdollistivat oman harrasteprojektini kehittymisen diplomityöksi.

Erityiskiitoksen saa professori Antti Valmari työn rennosta mutta silti erittäin tarkasta opastamisesta sekä työn virheisiin ja algoritmeihin perehtymisestä. Lisäksi haluan kiittää Eero Kokkoa, joka on työn aikana tarjonnut osan laitteistostaan ja verkkoyhteydestään käyttööni työn testaamiseksi käytännössä.

Porissa 16.7.2013

Henrik Heino

SISÄLLYS

TERMIT JA NIIDEN MÄÄRITELMÄT

Data-alue Yksi tässä työssä esitetyn tiedostoformaatin alueista, joka sisältää peräkkäin datayksiköitä. Datayksiköt ovat data-alueella tiiviisti, eli yhden jälkeen alkaa heti toinen.

Datavektori Tietyllä arvoalueella sijaitsevia lukuja sisältävä rajallisen mittainen jono, esimerkiksi tiedoston sisältöä kuvaava tavujen jono. Tässä työssä datavektorit koostuvat lähes aina joko tavuista tai biteistä.

Datayksikkö Data-alueella sijaitseva vaihtelevan mittainen yksikkö. Sisältää käyttämättömiä tavuja tai tietoja rakennepuun yhdestä solmusta.

Hakemistometadata Hakemistossa olevien tiedostojen, alihakemistojen jne. metatietoa, esimerkiksi muokkausaika, omistussuhteet ja käyttöoikeudet.

Hakemistorakenne Puumainen rakenne, joka alkaa yhdestä hakemistosta ja voi sisältää useita alihakemistoja, tiedostoja sisältöineen jne. sekä näiden tietoja, kuten nimiä, luomispäivämääriä yms.

Hakupuu Koko metadata-alueen laajuinen binäärihakupuu, jolla rakennepuun solmuja voi löytää logaritmisessa ajassa niiden tiivisteen perusteella.

Kirjanpito (eng. journaling) Levylle tai tiedostoon kirjoitettava aputietue, jolla varaudutaan varsinaisen kirjoitusoperaation yllättävään keskeytymiseen. Se sisältää tiedot varsinaisista kirjoituksista. Jos varsinainen kirjoitus keskeytyy, voidaan kirjanpidolla viimeistellä se. Tässä työssä kirjanpito on myös viimeinen alue esitetyssä tiedostoformaatissa.

Metadata-alue Yksi tässä työssä esitetyn tiedostoformaatin alueista, joka sisältää peräkkäin vakiomittaisia metadatoja. Vaikka metadata-alue on vektorimaisessa muodossa, voidaan sitä käsitellä myös binäärihakupuun lailla metadatoissa olevien tietojen avulla.

Metadata Tässä työssä metadatalla tarkoitetaan metadata-alueella sijaitsevaa yksikköä, joka sisältää lähes kaikki vakiomittaiset tiedot, mitä rakennepuun yhdellä solmulla on. Näitä ovat solmun tiiviste, viittaus datayksikköön sekä joitain deduplikointiin ja hakupuuhun liittyviä tietoja. Metadata on myös synonyymi hakupuun solmulle.

Otsake (tiedoston) Ensimmäinen alue työssä esitetyssä tiedostoformaatissa. Sisältää erilaisia vakiomittaisia tietueita kuten esimerkiksi onko tiedosto salattu, paljonko solmuja on, mihin data-alue loppuu, löytyykö tiedostosta kirjanpitoa ja onko tiedostossa mahdollisesti virheitä.

Paketointi — Hakemistorakenteen kokoaminen yksittäiseksi tietoyksiköksi, esimerkiksi datavektoriksi, siten, että alkuperäinen hakemistorakenne on mahdollista palauttaa siitä.

Paketti — Tiedosto (tai ryhmä tiedostoja), joka pitää sisällään paketoidun ja mahdollisesti pakatun hakemistorakenteen.

Pakkaus — Tiedon esitystavan muuttaminen siten, että se vie vähemmän tilaa, mutta on kuitenkin mahdollista palauttaa takaisin alkuperäiseen tilaansa. Yleensä tätä kutsutaan *häviöttömäksi* pakkaukseksi.

Rakennepuu — Puumainen tietorakenne, johon on tallennettu paketin sisältämä hakemistorakenne.

Salauslohko — Tavuvektori, joka kirjoitetaan tiettyyn kohtaan tiedostoa. Kaikki tiedoston kirjoitukset tehdään salauslohkoina. Tiedoston mahdollinen salaus tehdään salaamalla kokonaisia salauslohkoja.

Serialisoitu data — Solmun tyyppi, viittaukset lapsiin sekä sen oheisdata muunnettuna tavuvektoriksi tiedostoon kirjoittamista ja tiivisteen laskemista varten.

Symbolinen linkki — Unix-järjestelmien tukema hakemiston jäsen, joka ei ole alihakemisto eikä tiedosto, vaan viittaus johonkin sijaintiin hakemistorakenteessa.

Tavuvektori — Datavektori, jonka yksiköt ovat tavuja eli kokonaislukuja välillä 0–255.

Täytedatayksikkö — Nimitys datayksikölle, joka sisältää käyttämättömiä tavuja.

1. JOHDANTO

Pakkausohjelmat ovat kätevä tapa hallita ja siirtää useita tiedostoja. Hieman harvemmin käytettyjä ominaisuuksia tässä toiminnassa ovat *deduplikointi* ja olemassa olevan paketin muokkaaminen. Deduplikoinnilla tarkoitetaan tässä toistuvan datan laajamittaista havaitsemista, esimerkiksi kaikkien tiedostojen tasolla, ja tallentamista vain kerran. Kaikissa pakkausalgoritmeissa on jonkinlaista toistuvan datan havaitsemista, mutta usein tällainen koskee vain muutamia tavuja ja ulottuu rajalliselle alueelle. Laajamittaisella havaitsemisella tarkoitetaan sitä, että toistuvaa dataa etsitään koko aiemmin havaitun datan laajuudelta.

Näillä ominaisuuksilla saadaan tehostettua esimerkiksi varmuuskopiointia. Tällöin yksi paketti voisi helposti sisältää jopa tuhansia versioita useilta eri ajanhetkiltä samasta varmuuskopioitavasta kohteesta, mutta ei kuitenkaan veisi paljoa lisää tilaa deduplikoinnin ansiosta. Varmuuskopioissa useat eri versiot ovat tarpeellisia esimerkiksi sellaisten vikojen vuoksi, joita ei havaita heti. Kun vika on havaittu, voi kätevästi etsiä viimeisimmän toimineen version ja käyttää sitä tilanteen palauttamiseksi.

Tämä työ lähti tarpeesta saada omat tiedostot päivittäin talteen yhteen helposti hallittavaan tiedostoon joka ei kuitenkaan kasvaisi liian suureksi. Valmiita ratkaisuja löytyy, mutta omat tarpeet rajaavat niistä suuren osan pois. Ratkaisun tulisi tukea deduplikointia vähintään tiedostotasolla, mutta olisi hyvä jos se kykenisi deduplikoimaan myös kokonaisia hakemistohierarkioita. Ratkaisun tulisi olla myös muokattava, tarvittaessa salasanalla suojattava, toimia Linux-järjestelmässä, kyetä toimimaan yhdessä tiedostossa ja sisältää vähintään kohtuullisen virheenkorjauksen. Lisäksi olisi hyvä jos järjestelmä olisi avointa lähdekoodia, ilmainen ja vapaa patenteista.

Mikään olemassa oleva työkalu ei lopulta tuntunut tarpeeseen sopivalta. Tämän vuoksi työssä toteuttiin vaatimusten mukainen työkalu, eli oma pakkausohjelma ja -formaatti. Työssä käytettyjen algoritmien, tietorakenteiden ja muiden ohjelmistotieteellisten seikkojen lisäksi työ on kiinnostava myös yleisluontoisena tietotekniikan projektina. Työn tekemällä sai kohtuullisen kuvan kuinka helppoa tämänkokoisen projektin tekeminen yksin on.

Työn taustoissa käydään aluksi läpi muutamia aihepiiriin liittyviä teorioita sekä aiemmin toteutettuja ratkaisuja. Tiedon organisoinnissa esitellään oma toteutus

ensin tietorakenteen ja sitten tiedostoformaatin osalta. Ohjelma esittelee ohjelman rakennetta, toimintaa ja käyttöä. Mittausten kuvauksessa esitellään mittaustavat, jolla omaa valmista toteutusta verrataan muutamaan olemassa olevaan pakkausohjelmaan ajankäytön ja pakkaustehokkuuden kannalta sekä valmistellaan lukijaa ohjelman reilu puoli vuotta kestäneen käyttökokemuksen kuvaamiseen. Tulokset ja niiden tarkastelu vastaa edellisen luvun asettamiin kysymyksiin. Päätelmistä löytyvät työn keskeisimmät tulokset tiivistettynä.

2. TYÖN TAUSTAT

Tässä luvussa esitellään teoriaa ja historiaa joka on olennaista työn ymmärtämiseksi. Aluksi käydään läpi olennaiset osat tärkeimmistä teorioista ja tekniikoista. Tämän jälkeen esitellään aiempia tutkimuksia. Lopuksi esitellään jo olemassa olevia ratkaisuja. Esitellyt aiheet on valittu pääosin niiden aiheiden joukosta, jotka liittyvät tähän työhön, mutta aiempien ratkaisujen tapauksissa on otettu mukaan joitain aiheita myös niiden yleisyyden vuoksi.

2.1 Teoria

Seuraavaksi esitellään erilaisia teorioita ja tekniikoita, jotka ovat tarpeellisia tämän työn ymmärtämiseksi. Niistä esitellään vain ne osat, jotka ovat olennaisia tämän työn kannalta. Esimerkiksi monia tekniikoita voi käyttää useisiin eri kohteisiin tietotekniikassa, mutta ne eivät ole tämän työn kannalta olennaisia.

2.1.1 Kryptografiset tiivisteet ja tiivistealgoritmit

Kryptografinen tiiviste, jatkossa vain tiiviste, on jostain datavektorista luotu toinen datavektori. Sen luomiseen käytetään tiivistealgoritmia.

Tiivistealgoritmi on funktio, joka ottaa syötteenään mielivaltaisen pitkän datavektorin ja antaa ulostulona vakiomittaisen tiivisteen. Samasta datavektorista seuraa aina sama tiiviste.

Koska syötteet ovat mielivaltaisen pituisia ja tiivisteet vakiomittaisia, on mahdollisia tiivisteitä vähemmän kuin mahdollisia syötteitä. Kyyhkyslakkaperiaatteen mukaan voidaan päätellä, että kahdesta eri syötteestä voi seurata sama tiiviste. Näin ollen tiivisteestä ei voi päätellä mistä datavektorista se on luotu, eli tiivistefunktio on yksisuuntainen. Saman tiivisteen saamista kahdesta eri syötteestä kutsutaan *törmäykseksi*.

Hyvä tiivistealgoritmi on sellainen, jossa törmäyksiä tulee mahdollisimman vähän, eli tiivisteet ovat jakaantuneet mahdollisimman tasaisesti [4, s. 262]. Törmäysten keskimääräiseen todennäköisyyteen on kuitenkin vaikea vaikuttaa. Sen vuoksi tiivistealgoritmeissa pyritään siihen, että syötteessä olevat säännönmukaisuudet eivät kasvattaisi törmäysten mahdollisuutta. Pienikin ero kahden datavektorin välillä, esimerkiksi pisteen puuttuminen lauseen lopusta, johtaa käytännössä aina täysin erilaisiin tiivisteisiin.

Eräs käyttökohde tiivisteelle on mielivaltaisen pituisten datavektorien yksilöinti. Tässä tapauksessa pitkienkin datavektorien käsittely on nopeaa, koska sen sijaan että käsiteltäisiin pitkiä datavektoreita, voidaan käsitellä niiden tiivisteitä esimerkiksi selvittämään onko samanlainen datavektori esiintynyt jo aikaisemmin dataa käsiteltäessä.

Tiivistealgoritmit lukevat syötettä usein pienissä datavektorin paloissa eli *lohkoissa*. Luettuaan yhden lohkon, ne käyttävät lohkon sisältämää informaatiota muokatakseen *sisäistä tilaansa*. Sisäinen tila on datavektori ja se kuvaa algoritmin tilaa tiivisteen laskemisen aikana. Lopulta sisäisestä tilasta muodostetaan lopullinen ulostulo eli tiiviste.

Tiivistealgoritmeja voi vertailla niiden käyttämien datavektorien pituuksien mukaan. Vertailtavia datavektoreita ovat ulostulo, sisäinen tila sekä luettu lohko. Näiden pituudet esitetään bitteinä.

2.1.2 Tiedostojärjestelmä

Ohjelmoijan näkökulmasta kiintolevyn voi ajatella sisältävän vain yksinkertaisen ja todella pitkän datavektorin. Tällaisen yksinkertaisen datavektorin käyttäminen tietovarastona on kuitenkin hankalaa. Tiedostojärjestelmän tarkoituksena on helpottaa tiedon organisoimista kiintolevylle. Se esittää tiedon puumaisena tiedosto- ja hakemistorakenteena ja mahdollistaa sen lukemisen, läpikäymisen ja muokkaamisen.

Kiintolevyllä sijaitseva datavektori on rajallinen ja sen käyttö on hidasta. Tiedon lukeminen, uuden tiedon kirjoittaminen ja vanhan tiedon siirtäminen eteen- tai taaksepäin ovat kaikki hitaita operaatioita. Nämä seikat asettavat rajoituksia tiedostojärjestelmän toteuttamiselle.

Kiintolevyn rajoitusten vuoksi puumainen rakenne on usein toteutettu jonkinlaisella linkityksellä. Jokin ennalta määritelty kohta kiintolevyllä sisältää linkin puun juurisolmuun, josta on sitten viittaukset seuraaviin solmuihin ja niin edelleen.

On tavallista, että tiedostojärjestelmän yksiköt eli tiedostot ja hakemistot on tallennettu kahdessa osassa: ensin osa tiedosta vakiomittaisina yhdessä säiliössä ja loput vaihtelevan mittaisina toisessa säiliössä [9, s. 196]. Kutsukaamme näitä solmujen *otsakkeeksi* ja *dataksi*. Tämän hyöty on siinä, että vakiomittaisia otsakkeita sisältävä säiliö voidaan toteuttaa taulukkona, jolloin sen jäseniin voi viitata nopeasti järjestysnumeron perusteella. Otsakkeista löytyvät ne tiedot, jotka ovat kaikille tiedostoille ja hakemistoille yhteisiä, kuten esimerkiksi viittaus datan sijaintiin. Data sisältää vaihtelevan mittaiset tiedot, esimerkiksi tiedoston sisällön jonka pituus voi olla mitä tahansa nollasta ylöspäin. Tiedostojärjestelmä käyttää kiintolevyä usein kolmessa alueessa, joista edellä mainitut otsakkeet ovat keskellä ja datat lopussa. Alussa sijaitsee tällöin erilaista hallinnollista tietoa, esimerkiksi linkki juurihakemiston solmun otsakkeeseen.

Aluetta, joka sisältää datoja, kutsutaan *data-alueeksi*. Tämän alueen toteuttaminen olisi hyvin yksinkertaista, mikäli ei tarvitsisi varautua tiedostojen ja hakemistojen poistamiseen. Tällöin uudet datat voisi vain kirjoittaa yhtenäisinä edellisten perään.

Poistoihin on kuitenkin varauduttava. Siirrot levyllä ovat hitaita, joten datojen siirtelyä ei voi käyttää suuria määriä. Muutenhan olisi mahdollista pitää datat yhtenäisinä ja poiston tapahtuessa siirtää kaikkia sitä seuraavia datoja poistetun datan pituuden verran, jolloin ei jäisi tyhjää rakoa levyn datavektoriin.

Eräs tapa on jakaa data-alue vakiomittaisiin lohkoihin ja käsitellä sitä kuin taulukkoa. Lohkot ovat joko tyhjiä tai jonkin datan käytössä. Jokin tiedosto voisi koonsa puolesta tarvita esimerkiksi kymmenen lohkoa. Kun tiedosto poistetaan, merkitään nämä kymmenen lohkoa tyhjiksi jonka jälkeen ne voidaan käyttää uudelleen.

Tämän tavan ongelma on, että datat *pirstaloituvat* lopulta ympäri levyä [9, s. 189]. Jos äskeisen esimerkin kymmeneen vapautuneeseen lohkoon kirjoittaisi kahdeksan lohkoa vaativan tiedoston, jäisi kaksi lohkoa tyhjiksi. Ajan kuluessa koko levyn vapaa tila on pirstaloitunut useiksi pieniksi tyhjien lohkojen jonoiksi. Tällöin suuret tiedostot eivät enää mahdu näihin jonoihin kokonaisina ja ne täytyy sijoitella osina ympäri levyä. Tällainen hidastaa datan lukemista ja kirjoittamista, sillä levyn fyysistä lukupäätä pitää jatkuvasti siirrellä edestakaisin.

Lohkojen vakiomittaisuus aiheuttaa myös ongelman. Jos tiedosto vaatii esimerkiksi kymmenen lohkoa, on hyvin todennäköistä ettei viimeinen lohko täyty kokonaan. Pahimmassa tapauksessa sinne tulee tallennetuksi vain yksi tavu lopun jäädessä tarpeettomaksi tilaksi. Tämä ongelma ei ole kovin vakava, mutta sitäkin voi olla joskus tarpeen optimoida. Eräs tapa on kerätä useita vajaita lohkoja yhteen yksittäiseen lohkoon.

Myös tyhjän lohkon löytäminen voi olla haasteellista. Niiden etsiminen lohko kerrallaan on liian hidasta. Ratkaisuna on esimerkiksi ketjuttaa tyhjät lohkot tai pitää niistä kirjaa jossain keskitetyssä paikassa levyllä [9, s. 188].

Jotta tiedostojärjestelmän käyttö olisi mahdollisimman sujuvaa, on käyttöjärjestelmään usein myös toteutettu *levyvälimuisti* ja tiedostojärjestelmään *kirjanpito*.

Levyvälimuistin ideana on tallentaa kaikki luku- ja kirjoitusoperaatioissa ilmi tulleet levyn sisällöt vapaana olevaan keskusmuistiin [9, s. 191]. Keskusmuisti on huomattavasti kiintolevyä nopeampi ja sitä on usein vapaana suuriakin määriä. Tämän vuoksi toistuvat lukuoperaatiot ovat usein hyvin nopeita ensimmäisen luvun jälkeen.

Kirjanpito (eng. journal) tarkoittaa suoritettavien muokkausten tallentamista ensin johonkin muualle levylle. Tämän avulla on mahdollista suorittaa muokkaukset loppuun, mikäli ne keskeytyivät esimerkiksi sähkökatkon seurauksena. Tämä takaa tiedostojärjestelmän eheyden.

Myös monimutkaisempia lisäominaisuuksia on. Näistä käydään läpi *deduplikointi*

omassa kappaleessaan.

On olennaista huomata, että kiintolevyllä sijaitsevalla tiedostojärjestelmällä on vakiomittainen tila käytettävissä. Sen sijaan kun kyseessä on muokattava paketointiformaatti, niin on aina tärkeää pyrkiä siirtämään olennaiset tiedot kohti tiedoston alkua. Tällöin muokkauksista syntyneet mutta tarpeettomiksi käyneet tilanvaraukset päätyvät tiedoston loppuun ja tiedostoa voidaan lyhentää. Näin saadaan vapautettua levytilaa muuhun käyttöön.

2.1.3 Tiedon häviötön pakkaaminen

Tiedon häviöttömän pakkaamisen tarkoituksena on säästää tilaa muuttamalla tieto sellaiseen esitysmuotoon, että se vie vähemmän tilaa. Häviöttömyydellä tarkoitetaan sitä, että yhtään informaatiota ei häviä pakkaamisen seurauksena. Pakattu muoto on siis mahdollista muuntaa takaisin alkuperäiseen muotoonsa.

Pakatessa tietoa toistuvat ja yleiset osat, eli *kuviot*, pyritään esittämään mahdollisimman vähäisellä määrällä informaatiota. Esimerkiksi kuvasta voidaan sanoa, että seuraavaksi on sata pikseliä punaista väriä, sen sijaan että toistettaisiin sata kertaa tieto yhdestä punaisesta pikselistä. Voidaan myös merkitä, että seuraavaksi tulee samanlainen kuvio joka esiintyi jo aiemmin. Toisaalta nämä lisäävät tiedon esittämisen monimutkaisuutta ja siten kasvattavat harvinaisten osien pituutta. Niiden tapauksessa kun ei voi viitata mihinkään toistuvuuteen.

Jotta oikeanlainen esitysmuoto tiedolle löytyisi, täytyy tietää mitkä kuviot syötteessä ovat yleisiä ja mitkä harvinaisia. Tämän vuoksi syötettä täytyy analysoida. Analyysin voi tehdä joko koko syötteelle kerralla tai sitten päivittää analyysia samalla kun syötettä käydään läpi. Koko syötteen analysointi on usein raskasta eikä se välttämättä selviä tilanteista, joissa syötteen tilastollinen luonne muuttuu edetessä syötteessä.

Alkuperäisessä muodossaan tiedon yksiköt, esimerkiksi datavektorin tavut, ovat usein yhtä pitkiä, mutta pakatussa muodossaan ne voivat olla eri pituisia juuri sen mukaan, ovatko ne yleisiä vai harvinaisia syötteessä. Näin on esimerkiksi Huffmanin koodauksessa, jossa yleiset tavut vievät vähemmän bittejä [10].

Pienikin osanen syötteessä voi siis vaikuttaa koko sitä seuraavaan ulostuloon. Tämä johtuu ensinnäkin siitä, että jokainen tiedon osanen vaikuttaa analyysiin jolla päätetään esitysmuodosta. Toiseksi tämä johtuu siitä, että mikäli osanen on harvinainen, niin se esitetään pitkällä tavalla. Tämä pidentää ulostuloa. Jos osanen taas on yleinen, aiheuttaa se lyhyemmän ulostulon. Täten se siis vaikuttaa koko sitä seuraavan ulostulon pituuteen.

Tietoa ei voi pakata loputtomasti. Mitä enemmän tietoa pakkaa, sitä vähemmän siinä on toistoa jota voisi hyödyntää. Jotkut tiedot ovat myös luonteeltaan sellaisia, että niissä on toistoa äärimmäisen vähän. Tiedon pakkaaminen ei siis aina ole

hyödyksi. Myöskään kovin pientä tietoa ei kannata pakata, sillä tällöin tiedon esitystavan kuvaaminen ja pakattu tieto yhdessä saattavat viedä enemmän tilaa kuin alkuperäinen pakkaamaton tieto.

2.1.4 Deduplikointi

Deduplikoinnin tarkoituksena on havaita toistuva tieto ja estää sen kirjoittaminen useaan kertaan. Tällä tavalla tilaa saadaan säästettyä muihin tarkoituksiin. Jos esimerkiksi tiedosto sisältää saman datavektorin useaan kertaan, voidaan se pakata pienempään tilaan viittaamalla datavektorin myöhemmissä esiintymissä ensimmäiseen esiintymään.

Deduplikointia esiintyy eritasoisena. Sitä voidaan tehdä joko virtaavaan dataan tai kokonaisuuksiin, esimerkiksi tiedostoihin.

Pakkauksen näkökulmasta tehokas tapa on etsiä datavirrasta mielivaltaisen kokoisia toistuvuuksia mielivaltaisista kohdista. Tällöin datavirtaa käydään läpi ja esiintynyttä dataa kirjataan ylös tietorakenteeseen, josta voi nopeasti tarkastaa uuden datan kohdalla, onko vastaavaa jo esiintynyt aiemmin. Käytäessä datavirtaa läpi, kasvaa tietorakenteen koko nopeasti ja sitä mukaa myös ajan ja muistin käyttö. Sen vuoksi tietorakenne on usein rajallinen: se tyhjennetään joko kokonaan tai osittain sen kasvaessa riittävän suureksi. Tätä periaatetta kutsutaan *liukuvaksi ikkunaksi*. Deduplikointi tässä muodossa on siis luonteeltaan paikallista.

Valitsemalla sopivia kokonaisuuksia, voidaan deduplikointia harjoittaa myös laajemmassa mittakaavassa. Esimerkiksi paljon käyttäjiä sisältävässä järjestelmässä saattaa useilla käyttäjillä olla täysin samansisältöinen tiedosto. Jos kyetään havaitsemaan tiedostojen sisältö samaksi, voidaan toistuvuudet eliminoida viittaamalla jokaisesta tiedostosta samaan sisältöön. Useiden tiedostojen läpikäyminen tavu kerrallaan on kuitenkin äärimmäisen hidasta. Ratkaisuna on käyttää aiemmin esiteltyjä tiivisteitä. Tiedostojen sisällöistä luodaan tiiviste, joka tallennetaan johonkin hakutietorakenteeseen. Mukaan tallennetaan tietysti myös viittaus itse tiedoston sisältöön. Kun uutta tiedostoa ollaan tallentamassa, lasketaan sen tiiviste ja katsotaan tietorakenteesta löytyykö tiedoston sisältö jo jostain muualta. Tällä tavalla suurienkin tiedostojen sisältöjä on mahdollista vertailla nopeasti.

Jotkut tapaukset ovat kuitenkin ongelmallisia tälle järjestelylle. Eräs esimerkki on tilanne, jossa käyttäjä haluaa päivittäin tehdä varmuuskopion tiedostoistaan kopioimalla ne kyseisen päivämäärän hakemistoon. Mikäli hänellä on jokin hyvin suuri tiedosto, johon hän tekee päivittäin pieniä muutoksia, tulee tiedostolle joka päivä eri sisältö. Tällaisia tiedostoja ovat esimerkiksi virtuaalikoneiden levykuvat. Ongelman voi korjata siten, että tiedostokohtaisen deduplikoinnin sijaan jaetaankin tiedostot lohkoihin ja toteutetaan deduplikointi niiden kesken. Tämäkään ratkaisu ei ole täydellinen, mutta se on yksinkertainen toteuttaa ja tarjoaa silti parhaimmillaan suuren

avun. Jos lohkoilla on esimerkiksi vakiokokona yksi megatavu ja käyttäjä muokkaa gigatavun kokoisesta tiedostosta kolmea tavua, täytyy uutta tietoa tallentaa korkeintaan kolmen megatavun verran sen sijaan, että tiedostokohtaisessa deduplikoinnissa olisi pitänyt tehdä kokonaan uusi gigatavun kokoinen tiedosto.

Tässä työssä deduplikoinnilla tarkoitetaan nimenomaan laajassa, koko aineiston mittakaavassa tapahtuvaa deduplikointia, jota harjoitetaan tiedostojen lohkojen tasolla.

2.1.5 Differentiaalinen ja inkrementaalinen varmuuskopiointi

Differentiaalinen ja inkrementaalinen varmuuskopiointi ovat tapoja, joilla on mahdollista pitää useita eri versioita varmuuskopioitavasta kohteesta ilman, että jokainen versio veisi yhtä paljon tilaa kuin täysi kopio kohteesta. Molemmissa tavoissa aloitetaan tekemällä ensin *täyskopio* varmuuskopioitavasta kohteesta, eli varmuuskopioidaan se kokonaisuudessaan. Tämän jälkeen toimitaan eri tavoin:

Differentiaalisessa tavassa myöhemmät varmuuskopiot sisältävät vain muutokset, jotka ovat tapahtuneet alkuperäisen täyskopion ja varmuuskopioitavan systeemin senhetkisen tilan välillä. Varmuuskopion palauttaminen vaatii alkuperäisen täyskopion sekä halutun version. Differentiaalista varmuuskopiointia käyttävä ohjelma on esimerkiksi myöhemmin esiteltävä *eXdupe*. Differentiaalinen tapa käy ajan myötä tilaa vieväksi, sillä jokaisen version täytyy sisältää samat muutokset jotka on jo aiemminkin tallennettu.

Inkrementaalisessa tavassa jokainen myöhempi varmuuskopio sisältää vain muutokset, jotka ovat tapahtuneet edellisen version jälkeen. Inkrementaalinen varmuuskopiointi ei tämän vuoksi vie yhtä paljon tilaa kuin differentiaalinen. Varmuuskopion palauttaminen on kuitenkin hitaampaa, sillä se vaatii jokaisen version läpikäymistä alun täyskopiosta haluttuun versioon asti.

2.1.6 Snapshot

Snapshot on täydellinen kopio koko tiedostojärjestelmästä tai sen osasta. Se kuvaa yleensä mennyttä ajanhetkeä ja järjestelmän tilaa tuolloin.

Koska todellinen kopio veisi suuren määrän tilaa, käytetään snapshoteissa yleensä deduplikointia. Tämän vuoksi snapshot ei aluksi vie käytännössä yhtään tilaa, sillä aitojen kopioiden sijaan se käyttää viittauksia varsinaisen tiedostojärjestelmän tietoihin. Vasta kun tiedostoja aletaan poistaa tai muokata, niin snapshotin voi sanoa vievän tilaa. Tämä johtuu siitä, että vaikka käyttäjä poistaa tiedostoja tai kirjoittaa vanhan sisällön päälle uutta, niin tiedostojen vanhat versiot säilyvät edelleen snapshotin sisällä.

Joissain tiedostojärjestelmissä snapshottien muokkaaminen on estetty. Tällöin ne

kuvastavat paremmin mennyttä ajanhetkeä, johon ei enää pääse vaikuttamaan.

2.1.7 Symmetrinen salaus

Symmetrisellä salauksella tarkoitetaan yhden avaimen salausta. Siinä tieto salataan ja avataan samalla avaimella [24].

Se soveltuu yksinään huonosti esimerkiksi verkkoliikenteeseen, sillä siinä tietoa käyttävät eri henkilöt, ja näiden välille täytyy keksiä jokin turvallinen kanava avaimen vaihtoa varten. Tällainen on esimerkiksi *julkisen avaimen menetelmä*, mutta sitä ei käydä läpi tässä työssä.

Sen sijaan yhden käyttäjän tarpeisiin symmetrinen salaus on käyttökelpoinen. Käyttäjä voi esimerkiksi salata tiedostoja, jotta arkaluontoiset tiedot eivät joudu vääriin käsiin mikäli tallennusväline varastetaan. Avaimen päätymistä vihamieliselle taholle ei tarvitse pelätä, sillä se on tallennettu vain käyttäjän muistiin.

Todellisuudessa käyttäjä harvoin hallitsee itse salausavainta, sillä salausavaimet ovat yleensä vakiopituisia datavektoreita. Tämän vuoksi lopulliset salausavaimet luodaan ottamalla käyttäjän syöttämästä salasanasta tiiviste.

Symmetriset salaimet voidaan jakaa *jono-* ja *lohkosalaimiin*.

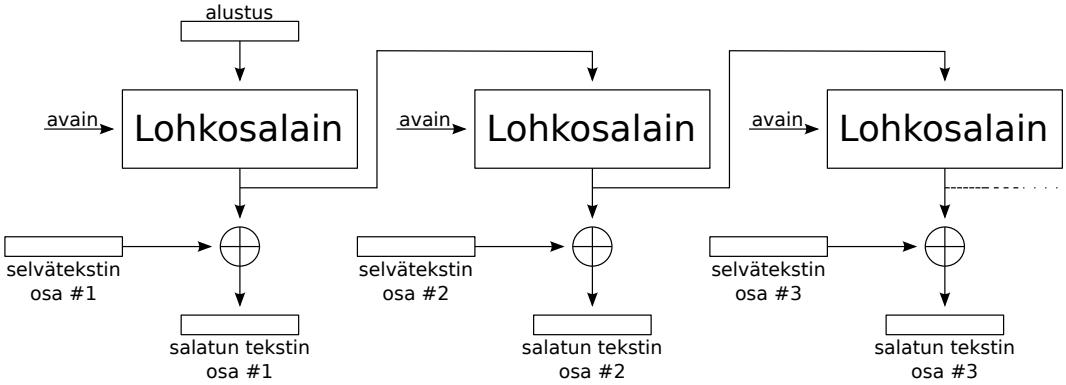
Jonosalaimet lukevat selvätekstistä datavirtaa ja antavat jatkuvaa salattua virtaa. Hyvä jonosalain on sellainen, että syötteessä esiintyvä toisto ei esiinny toistona ulostulossa. Mikäli syötteen toisto aiheuttaisi toistoa ulostulossa, olisi esimerkiksi kuvien salaaminen hyödytöntä, koska samat värialueet muuttuisivat aina samalla tavalla ja kuvista voisi edelleen päätellä minkä muotoinen asia niissä on kuvattuna.

Jonosalain usein alustetaan jollain tiedolla. Tällä vältetään se, että kaksi samansisältöistä viestiä koodautuisi samoiksi. Tiedostoista voidaan käyttää esimerkiksi niiden luontipäivämääriä ja niistä saatu alustustieto voidaan liittää sellaisenaan salatun tiedon mukaan. Sen päätymisestä vääriin käsiin ei ole merkitystä, sillä se on olemassa vain jotta vihamielinen taho ei pystyisi havaitsemaan toistuvuutta.

Lohkosalaimet ovat perusperiaatteeltaan yksinkertaisempia: ne ottavat syötteenään vakiomittaisen avaimen ja vakiomittaisen syötteen, ja antavat ulostulona syötteen pituisen salatun tiedon. Syötteen ja ulostulon pituutta kutsutaan lohkopituudeksi. Samalla avaimella ja samalla syötteellä saadaan aina sama ulostulo.

Suurin osa symmetrisistä salaimista on lohkosalaimia. Ne olisivat yksinään melko hyödyttömiä, mutta ne on mahdollista muuntaa jonosalaimiksi käyttäen jotain *toimintamoodia*. Toimintamoodilla tarkoitetaan sitä, että lohkosalaimen alustustieto, salausavain, syöte ja ulostulo ketjutetaan siten, että lopputulos on jonosalaimen kaltainen. Esimerkiksi OFB-moodi (Output feedback) toimii seuraavasti:

1. Ensimmäiselle lohkolle annetaan syötteenä alustustieto ja avaimena salausavain.



Kuva 2.1: Esimerkki OFB-toimintamoodin salaimesta

2. Ulostulosta ja osasta selvätekstistä syötettä luodaan XOR-operaatiolla osa salatusta ulostulosta.

3. Seuraavalle lohkolle annetaan syötteenä edellisen lohkon ulostulo ja avaimena salausavain.

4. Ulostulon kanssa toimitaan samoin kuin aiemmin ja näitä kahta viimeistä vaihetta jatketaan niin kauan kuin syötettä on tarjolla.

OFB-toimintamoodin salaus on esitetty myös kuvassa 2.1. OFB-toimintamoodi varmistaa sen, että datavektori, jonka kanssa syötteelle ajetaan XOR-operaatio, muuttuu jatkuvasti erilaiseksi. Tämän vuoksi syöte voi sisältää paljonkin toistoa, mutta se ei ilmene ulostulosta. Mikäli syöte ei satu olemaan lohkopituuden monikerta, joudutaan sitä täydentämään esimerkiksi nollilla, jotta loputkin ulostulosta saataisiin. Tällöin pitää kuitenkin purkaessa tietää mikä alkuperäisen syötteen kokonaispituus oli. Muitakin täydennystapoja on, jolloin alkuperäistä kokonaispituutta ei tarvitse tietää, mutta ne eivät ole tämän työn kannalta olennaisia.

2.2 Aikaisempia tutkimuksia

Deduplikoinnin tehoa käytännössä ovat tutkineet esimerkiksi Dutch T. Meyer ja William J. Bolosky [14]. Heidän tutkimuksessaan tutkittiin deduplikointia työpöytäkäytössä olevilla tietokoneilla. Vertailun kohteena oli erityisesti tiedostojen osien ja pelkästään tiedostojen tasolla tapahtuva deduplikointi. Tutkimuksen tuloksena selvisi, että deduplikointi pelkästään tiedostojen tasolla onnistuu säästämään tilaa vain 75 % siitä määrästä, minkä tiedostojen osien tasolla tapahtuva deduplikointi.

Erilaisten pakkausalgoritmien ja -formaattien pakkaustehoa ovat tutkineet Mladen Konecki, Robert Kudelić ja Alen Lovrenčić [11]. Heidän tutkimuksessaan esiteltiin ensin tunnettuja pakkausalgoritmeja ja sen jälkeen vertailtiin tunnettuja sekä vähemmän tunnettuja mutta tehokkaita pakkausformaatteja. Vertailussa käytettiin

yhtä 48 MB kokoista aineistoa, joka vastaa sisällöltään tyypillisen työpöytäkäyttäjän henkilökohtaisia tiedostoja sillä erotuksella, että tekstitiedostoja oli poikkeuksellisen paljon.

Myös Tiina Vekkilä vertailee pakkausformaattien pakkaustehoja [23]. Hänen tutkimuksessaan aineistoja on muutamia ja niiden sisällöt edustavat eri tiedostotyyppejä.

2.3 Aiempia ratkaisuja

Tässä osassa esitellään aihepiiriin kuuluvan tekniikan nykyistä tilaa. Aluksi listataan paketointi- ja pakkausformaatteja. Tämän jälkeen perehdytään tiedostojärjestelmiin, jotka sisältävät työn kannalta mielenkiintoisia elementtejä. Lopuksi on lyhyt esittely tiivistealgoritmeista.

2.3.1 Paketointi- ja pakkausformaatteja

Seuraavaksi esitellään tavallisia ja varmuuskopiointiin tarkoitettuja paketointi- ja pakkausformaatteja. Mukaan on pyritty valitsemaan sellaisia formaatteja, jotka kaikista parhaiten täyttäisivät työn vaatimukset, mutta joukossa on myös joitain yleisiä formaatteja.

Formaattien esittelyssä keskitytään työn kannalta olennaisiin asioihin. Näitä ovat esimerkiksi onko formaatti muokattavissa, avointa lähdekoodia, ilmainen, vapaa patenteista, deduplikoiva ja/tai usealla alustalla toimiva.

Apple Time Machine

Apple Time Machine on varmuuskopiointiohjelma, joka tukee vain Macintosh-tietokoneita [1]. Nimensä mukaisesti se tarjoaa mahdollisuuden tarkastella myös aikaisempia versioita siihen tallennetun järjestelmän historiasta.

Apple Time Machinea varten valitaan jokin ulkoinen tallennusmedia kohdelaitteeksi. Kohdelaite voi olla esimerkiksi ulkoinen kiintolevy.

Aluksi ohjelma varmuuskopioi koko järjestelmän tiedostot laitteeseen. Tämän jälkeen ohjelma varmuuskopioi tunnin välein ne osat järjestelmän tiedostoista, jotka ovat muuttuneet.

Näin säästetään tilaa, mutta käyttäjällä on silti mahdollisuus tarkastella järjestelmää kokonaisuutena. Ohjelma osaa esittää varmuuskopiointihetkien tilat yhtenäisinä kokonaisuuksina yhdistelemällä tietoa alun täyskopiosta ja sen jälkeen tallennetuista muutostiedoista.

Vuorokauden jälkeen säilytetään vain päivittäisiä versioita ja kuukauden jälkeen vain viikottaisia versioita. Viikottaisia versioita säilytetään niin kauan, kunnes kohdelaitteesta loppuu tila.

eXdupe

eXdupe on useilla eri käyttöjärjestelmillä toimiva pakkaus- ja paketointiohjelma, joka tukee myös differentiaalista varmuuskopiointia [5]. Se käyttää kahta tiedostoformaattia. `.full`-päätteinen vastaa tyypillistä pakattua paketointiformaattia. Tätä käytetään tehtäessä varmuuskopioinnin ensimmäinen, täydellinen versio. Toinen tiedostoformaatti on `.diffN`-päätteiset tiedostot, joihin tallennetaan alkuperäisen `.full`-päätteisen paketin ja varmuuskopioitavan järjestelmän nykyhetken väliset erot. eXdupe on tällä hetkellä ilmainen ja avointa lähdekoodia, mutta tulee muuttumaan tulevaisuudessa maksulliseksi.

eXdupe keskittyy vahvasti deduplikointiin usean ytimen järjestelmillä. Se käyttää liukuvan ikkunan periaatetta. Näiden avulla se pääsee parempiin suoritusaikoihin ja pakkaustehoihin kuin useat muut tunnetut pakkausohjelmistot.

Reasonable Archiver

Reasonable Archiver on deduplikointia usealla tasolla tukeva paketointi- ja pakkausohjelma Windowsille [19]. Käyttäjä voi valita joko nopean moodin tai parhaan moodin. Ensimmäinen toteuttaa deduplikointia vain tiedostojen tasolla ja jälkimmäinen myös datan tasolla. Ohjelma on suljettua lähdekoodia ja maksullinen. Sen kehitys on lopetettu.

Flyback

Flyback on ilmainen, avoimen lähdekoodin varmuuskopiointiohjelma Linuxille [6]. Se pyrkii toteuttamaan Apple Time Machinen kaltaisen käyttökokemuksen.

Tiedostojen ja hakemistojen aikaisempien versioiden hallinnan taustalla on Git-versionhallintaohjelmisto. Koska Git on toteutettu pääasiassa ohjelmointia varten, on siinä joitain rajoituksia jotka vaikuttavat myös Flybackiin. Git ei esimerkiksi kykene käsittelemään tiedostoja, jotka ovat liian suuria mahtuakseen keskusmuistiin.

Zip

Zip on yksi yleisimpiä pakkaus- ja paketointiformaatteja [17]. Se on avointa lähdekoodia ja toimii lukuisilla eri käyttöjärjestelmillä.

Zip-formaatti ei ole suunniteltu varmuuskopiointia varten, mutta sitä voisi mahdollisesti käyttää siihen, sillä zip-paketeissa on otettu muokattavuus huomioon. Paketin sisältöä voi muokata, uutta tietoa lisätä ja vanhaa poistaa.

Zip-paketti koostuu kahdesta osasta: *keskushakemistosta* (eng. central directory) ja kokoelmasta *tiedostoalkioita* (eng. file entry). Keskushakemisto sijaitsee zip-tiedoston lopussa.

Keskushakemistosta zip-tiedoston sisältö on nopea listata. Se sisältää tiedostojen nimet, niiden hakemistometadatat jne. Tiedostojen sisällöt eivät sijaitse keskushakemistossa vaan tiedostoalkioissa, joihin keskushakemistosta viitataan.

Tiedostoalkiot sisältävät tiedostojen sisällön lisäksi useita samoja tietoja kuin vastaava alkio keskushakemistossa. Tähän on syynä tiedon varmennus. Tiedostoalkiot voivat sijaita eri järjestyksessä kuin vastaavat alkiot keskushakemistossa ja niiden välissä voi olla käyttämätöntä dataa. Tämän vuoksi tiedostoalkioita ei koskaan tule selata suoraan, vaan aina keskushakemiston kautta. Tiedostoalkiot sisältävät myös vapaamuotoisia lisätietoja. Ohjelmat voivat tallentaa näihin lisäominaisuuksiaan, jotka eivät kuulu formaattiin vakiona. Esimerkiksi WinZip-ohjelma toteuttaa AES-salauksen näin.

Keskushakemiston alkiot eivät sisällä mitään tiivistetietoa tiedostojen sisällöistä, joten nopea tiedostokohtainen deduplikointi ei ole ainakaan oletuksena tuettu.

Rar

Rar on yleinen pakkaus- ja paketointiohjelma ja -formaatti [18]. Sen uusin versio on suljettua lähdekoodia, mutta tukee silti useita käyttöjärjestelmiä. Sen käyttämät algoritmit ovat patentoituja.

7z

7z on yleistyvä ja tehokas pakkaus- ja paketointiformaatti [16]. Se on osa 7-Zip-ohjelmaa, jolla 7z-tiedostoja voidaan käsitellä. 7-Zip on avointa lähdekoodia ja sille löytyy versiot useille eri käyttöjärjestelmille. Myös formaatin arkkitehtuuri on avoin, eli se sallii uusien pakkaus- ja salausmenetelmien lisäämisen tulevaisuudessa. 7z sisältää useita nykyaikaisia ominaisuuksia, esimerkiksi salauksen, tuen suurille tiedostoille ja unicode-tiedostonimet.

7-Zip osaa esikäsitellä pakattavaa dataa siten, että useat samankaltaiset tiedostot voidaan pakata kerralla ja näin hyötyä niiden samankaltaisuudesta. 7z mahdollistaa useiden eri pakkausmenetelmien käytön, mikä tekee siitä tehokkaan formaatin. Myös tiedostojen muokkaaminen 7z-paketin sisältä on mahdollista.

7z ei tallenna tiedostojen ja hakemistojen omistajuuksia tai käyttöoikeuksia, joten se ei ainakaan sovellu täydelliseen varmuuskopiointiin. 7-Zip -ohjelmaa voi kuitenkin hieman soveltaen käyttää siten, että se vertaa yhden 7z-paketin ja halutun hakemistorakenteen tilaa ja luo näistä differentiaalista varmuuskopiointia useina tiedostoina [3].

ZPAQ

ZPAQ on erittäin edistynyt pakkaus- ja paketointiformaatti [13]. Se on avointa lähdekoodia ja ilmainen. Sille löytyy ohjelmia useille eri käyttöjärjestelmille. ZPAQ tukee esimerkiksi deduplikointia, kirjanpitoa sekä inkrementaalista varmuuskopiointia. ZPAQ:ia kehitetään aktiivisesti.

ZPAQ:n käyttämät algoritmit ovat erittäin tehokkaita pakkausteholtaan, kuten selviää Koneckin ja muiden tutkimuksista [11]. Toisaalta samaiset tutkimukset myös osoittavat, että algoritmit ovat hyvin hitaita.

tar.gz ja tar.bz2

tar.gz ja tar.bz2 ovat Unix-maailmassa yleisiä tapoja paketoida ja pakata hakemistorakenteita. Kyseessä on yhdistelmä kahdesta eri formaatista. Pääte tar viittaa *tape archive* -formaattiin, joka ainoastaan paketoi hakemistorakenteen yhdeksi pitkäksi tiedostoksi eli datavektoriksi. Tämän jälkeen datavektoriksi muunnettu tieto annetaan gzip- tai bzip2-ohjelmalle, joka pakkaa sen pienempään tilaan [7][20].

tar.gz- ja tar.bz2-paketit ovat hyviä esimerkkejä formaatista, jonka sisällön muokkaaminen on käytännössä mahdotonta. Kuten kohdassa 2.1.3 mainittiin, pienikin muutos syötteessä vaikuttaa koko sitä seuraavaan ulostuloon. Vaikka vain yhtäkin bittiä yhdessä tiedostossa tar-paketin sisällä muokattaisiin, saattaisi sen aiheuttama muutos vaikuttaa siihen, miten datavektori pakataan siitä eteenpäin lopullisessa tar.gz- tai tar.bz2-paketissa. Vaikutukset olisivat vielä suurempia jos esimerkiksi tar-paketin keskelle lisättäisiin jokin tiedosto.

2.3.2 Tiedostojärjestelmiä

Alla on lueteltu kolme edistynyttä tiedostojärjestelmää: ZFS, BTRFS ja SDFS. Niissä on useita työn aihepiiriin liittyviä ominaisuuksia.

ZFS

ZFS on Sun Microsystemsin kehittämä, alunperin Solarikselle tarkoitettu hyvin edistynyt tiedostojärjestelmä [15]. Se tukee esimerkiksi usean eri tason deduplikointia [2], salausta, pakkausta, snapshotteja ja tiedon automaattista eheyden tarkastamista sekä korjaamista.

B-tree Filesystem (BTRFS)

BTRFS on Linuxille kehitetty vastine ZFS:lle. Tämän tiedostojärjestelmän kehitys aloitettiin sen vuoksi, että ZFS:ää ei voitu lisenssiongelmien vuoksi suoraan liittää

Linuxin ytimeen. Projekti on edelleen kehityksen alla, mutta monet ZFS:n tärkeistä ominaisuuksista ovat suunnitteilla myös BTRFS:ään [12].

SDFS

SDFS on verkossa toimiva deduplikoiva tiedostojärjestelmä [21]. Se on avointa lähdekoodia ja toteutettu Javalla, joten se toimii useissa käyttöjärjestelmissä. Toisin kuin useita muita tiedostojärjestelmiä, SDFS:ää ei ajeta käyttöjärjestelmän ytimen tasolla, vaan sovellustasolla. Sen kehityksessä on huomioitu erityisesti virtuaalikoneiden tarpeet, eli suuret levytiedostot jotka saattavat sisältää paljon identtisiä osia ja ovat jatkuvasti muutoksen kohteina. Koska tiedostojärjestelmä toimii verkossa, on sen deduplikointi globaalia.

Tiedostot on jaettu pieniin osiin ja niistä pidetään kirjaa tiivisteiden avulla. Tiivisteiden kesken voi tapahtua törmäyksiä, jotka rikkovat järjestelmän eheyden. Näiden todennäköisyys oletetaan kuitenkin mitättömäksi.

Kun asiakas lähettää tiedoston osaa, se laskee siitä ensin tiivisteen. Tällä tavoin voidaan deduplikoinnin lisäksi myös välttyä turhilta tiedonsiirroilta mikäli tiedoston osa sijaitsee jo valmiiksi palvelimella.

Palvelimella tietojen tallennus on jaettu tiiviste- ja datavarastoksi. Tiivistevarastosta on mahdollista hakea nopeasti tiedostojen osia niiden tiivisteiden perusteella. Kun haluttu osa on löytynyt, haetaan sen sisältö datavarastosta.

2.3.3 Tiivistealgoritmeja

Tiivistealgoritmeja on kehitelty useita useilla eri bittimäärillä. On yleistä, että yksittäinen tiivistealgoritmi on ajan kanssa laajentunut kokonaiseksi *tiivistealgoritmiperheeksi*, joka käsittää useita eri versioita samasta tiivistealgoritmista. Eri versiota on tehty tietoturvan lisäämiseksi ja niissä on usein enemmän bittejä kuin edeltäjissään. Myös toimintaperiaate on saattanut muuttua.

Yleisiä tiivistealgoritmiperheitä ovat esimerkiksi *MD*, *SHA* ja *RIPEMD*. Kaikille nykyisille MD- ja SHA-perheen versioille on löytynyt törmäyksiä, mutta RIPEMD-128/256 ja RIPEMD-320 ovat toistaiseksi välttyneet tältä.

Yleisimmät tiivistealgoritmiperheet on esitelty taulukossa 2.1.

Taulukko 2.1: Yleisimmät tiivistealgoritmiperheet. Yksiköt ovat bittejä.

Algoritmi	Ulostulo	Sisäinen tila	Lohkon koko
MD2	128	384	128
MD4	128	128	512
MD5	128	128	512
RIPEMD	128	128	512
RIPEMD-128/256	128/256	128/256	512
RIPEMD-160	160	160	512
RIPEMD-320	320	320	512
SHA-0	160	160	512
SHA-1	160	160	512
SHA-256/224	256/224	256	512
SHA-512/384	512/384	512	1024

3. TIEDON ORGANISOINTI

Tässä luvussa kuvataan miten tieto on järjestelmässä organisoitu. Aluksi perehdytään itse tietorakenteeseen algoritmien kannalta. Tämän jälkeen selitetään millaisia vaatimuksia tämä asettaa tiedostoformaatille.

3.1 Tietorakenne

Koko tietorakenteen ydin on yksi hakemistorakenne, joka on toteutettu puuna. Tätä puuta kutsutaan *rakennepuuksi*. Seuraavaksi esitellään rakennepuun solmujen ominaisuudet ja sen jälkeen puussa käytetty deduplikointi.

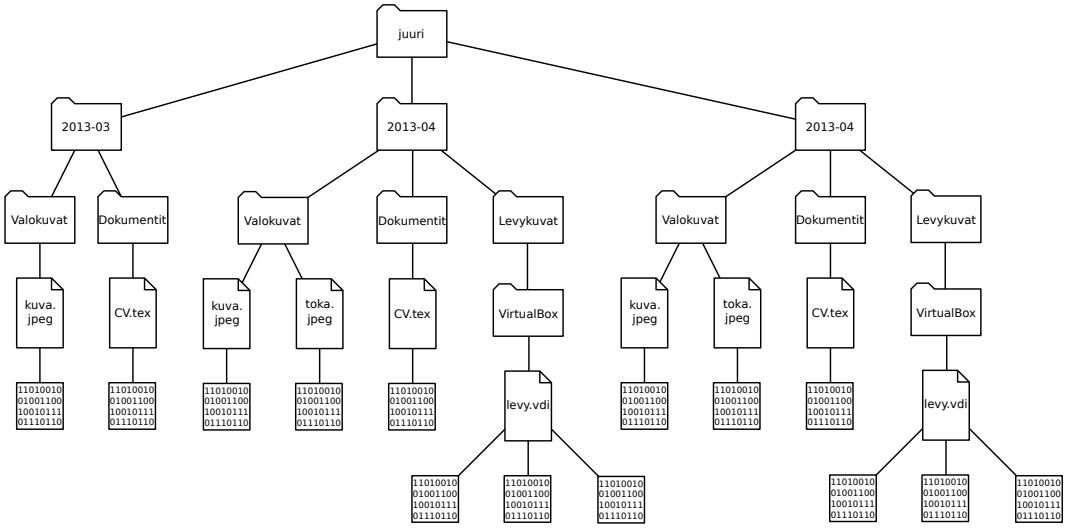
3.1.1 Solmujen ominaisuudet

Yhdellä rakennepuun solmulla on tyyppi, viittaukset mielivaltaiseen määrään lapsisolmuja sekä mahdollinen oheisdata. Solmuilla ei ole viittauksia niiden vanhempiin. Viittaukset lapsisolmuihin ovat aina jossain tietyssä järjestyksessä. Kaikki hakemistorakenteen osat ja tiedot on tallennettu joko rakennepuun solmuina tai solmujen oheisdatana.

Jokainen rakennepuun solmuista on jotain seuraavista tyypeistä: *hakemisto*, *tiedosto*, *datalohko* tai *symbolinen linkki*. Solmun tyyppi määrää millaista oheisdataa ja millaisia lapsia sillä voi olla. Tiivistelmä niiden tiedoista on taulukossa 3.1. Eri solmutyypit esitellään seuraavaksi tarkemmin.

Taulukko 3.1: Solmujen tyypit sekä lapsisolmujen ja oheisdatan käyttötapa tyypissä.

Tyyppi	Lapsisolmut	Oheisdatan käyttö
Hakemisto	Hakemiston sisältämät alihakemistot, tiedostot ja symboliset linkit	Lapsien nimet tässä hakemistossa sekä niiden hakemistometadatat
Tiedosto	Datalohkot jotka kuvaavat tiedoston sisältöä	Ei oheisdataa
Datalohko	Ei lapsia	Yksi yhtenäinen osa jonkin tiedoston datasisällöstä
Symbolinen linkki	Ei lapsia	Polku symbolisen linkin kohteeseen



Kuva 3.1: Esimerkki rakennepuusta. Puussa on havaittavissa kolmen tyyppisiä solmuja: hakemisto-, tiedosto- ja datalohkosolmuja.

Hakemisto on solmutyypeistä monimutkaisin. Sillä voi olla lapsisolmuina toisia hakemistoja, tiedostoja sekä symbolisia linkkejä. Hakemistosolmun oheisdataa käytetään tallentamaan sen lapsien tiedostonimet kyseisessä hakemistossa. Oheisdata voi sisältää myös lapsien hakemistometadatat eli muokkausajat, omistussuhteet, käyttöoikeudet jne.

Tiedostosolmujen lapsina voi olla vain datalohkoja. Tiedoston sisältö koostuu datalohkoista, joten niiden järjestys määrää mikä lapsista sisältää tiedoston ensimmäiset tavut, mikä niitä seuraavat jne. Oheisdataa tiedostolla ei ole.

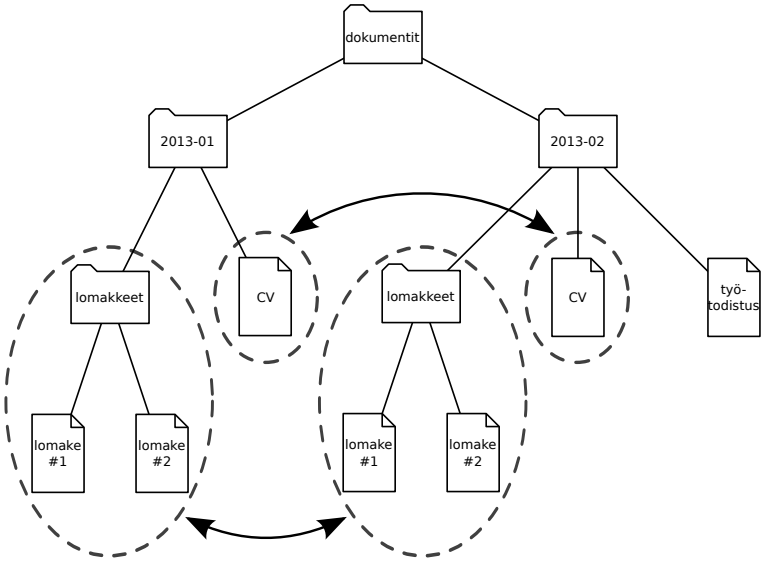
Datalohkosolmuilla ei ole lapsia. Niillä on vain oheisdata joka sisältää osan jostain tiedostosta eli tavuvektorin. Tavuvektorin pituus voi olla mitä tahansa, mutta tässä työssä se on aina neljä megatavua, lukuunottamatta tiedoston viimeistä datalohkoa, joka saattaa olla lyhyempi.

Symbolinen linkki -solmuilla ei myöskään ole lapsia. Niiden oheisdata koostuu polusta, johon symbolinen linkki osoittaa.

Kuvassa 3.1 on esitetty esimerkki rakennepuusta, johon on tallennettu dokumentteja kolmelta eri ajanhetkeltä.

3.1.2 Deduplikointi

Johdannossa esitettiin vaatimus, että paketin on kyettävä säästämään tilaa kun siihen tallennetaan identtisiä tiedostoja tai jopa kokonaisia identtisiä hakemistorakenteita. Kohdassa 3.1.1 mainittiin, että solmut eivät tiedä mitään vanhemmistaan eivätkä täten myöskään tiedä polkua jossa ne sijaitsevat. Tästä voidaan päätellä, että samanlaisia tiedostoja tai hakemistorakenteita kuvaavat solmut ovat myös ra-



Kuva 3.2: Esimerkki identtisistä tiedostoista ja hakemistorakenteista samassa isossa hakemistorakenteessa. Identtiset osat on ympyröity katkoviivalla ja yhdistetty nuolella. Jotta tiedostot ja hakemistot olisivat identtisiä, tulee niiden sisältöjen, oikeuksien, päivämäärien jne. olla täysin samoja. Tässä kuvassa tiedostojen datalohkot eivät ole näkyvissä.

kennepuussa täysin samanlaisia. Tätä on havainnollistettu kuvassa 3.2.

Tätä päätelmää käytetään hyväksi deduplikoinnin toteutuksessa. Rakennepuun solmut on yksilöity SHA512-tiivisteiden avulla. Tarkoituksena on, että identtisille solmuille tulee aina sama tiiviste. Identtisillä solmuilla tarkoitetaan sellaisia solmuja, jotka ovat täysin samanlaisia ja joilla on täysin samanlaiset lapset, mutta joiden vanhemmat eli sijainti rakennepuussa saa vaihdella. Solmuihin viitataan käyttämällä niiden tiivisteitä.

Solmun tiivisteen laskemiseen käytetään kaikkia tietorakenteen kannalta olennaisia solmun tietoja. Tämä tarkoittaa solmun tyyppiä, solmun oheisdataa sekä solmun lapsisolmujen tiivisteitä, joilla viittaaminen lapsisolmuun tapahtuu.

Koska solmun tiivisteen laskemiseen käytetään myös sen lapsien tiivisteitä, voi tästä päätellä, että pienikin muutos mihin tahansa solmuun muuttaa myös kaikkien sen vanhempien tiivisteitä rakennepuun juureen asti. Tämä tekee mahdolliseksi yksilöidä yhdellä tiivisteellä yksittäisen solmun lisäksi kokonaisia solmuhierarkioita. Myös koko rakennepuu on mahdollista yksilöidä tällä tavoin.

Tiivisteiden avulla on mahdollista tarkastaa onko jokin solmu tai solmuhierarkia jo tallennettu rakennepuuhun. Jos esimerkiksi tietokoneelta ollaan kopioimassa jotain valtavaa hakemistorakennetta, joka jo löytyy järjestelmästä, ei rakennepuuhun tarvitse kirjoittaa ainoatakaan uutta solmua, joka perustuisi hakemistorakenteen sisältöön.

Ainoat muutokset rakennepuuhun ovat tällaisessa tapauksessa peräisin muuttu-

neista viittauksista. Ensin täytyy tehdä uusi viittaus ja nimi siihen hakemistosolmuun, johon hakemistorakenne kopioidaan. Tämä muutos hakemistosolmuun muuttaa sen tiivistettä, jonka vuoksi myös sen vanhemman täytyy muuttua, kuten myös sen vanhemman jne. aina juureen asti.

Tiivisteiden törmäyksien vuoksi on teoriassa mahdollista, että kaksi erilaista solmua saisivat saman tiivisteen. Tällainen tapahtuma tekisi paketista todennäköisesti käyttökelvottoman. SHA512-tiiviste oletetaan tässä työssä kuitenkin riittävän luotettavaksi, eikä tiivisteiden törmäyksiä oleteta realistiseksi riskiksi.

3.2 Tiedostoformaatti

3.2.1 Yleisrakenne

Tiedostoformaatti on jaettu neljään osaan. Osat ovat tiedostossa seuraavassa järjestyksessä: Ensimmäisenä on *otsake*, joka sisältää monipuolista tietoa paketista itsestään. Seuraavana ovat *metadata-* ja *data-alue*, joihin paketin sisältämän rakennepuun solmut on tallennettu. Viimeisenä on alue kirjanpitoa varten. Alueet käydään läpi yksityiskohtaisesti myöhemmin tässä luvussa.

Otsakkeen koko ei muutu paketin luomisen jälkeen. Muiden alueiden koot sen sijaan muuttuvat pakettia muokatessa. Alueiden koon muuttuminen on olennaista, sillä paketin käyttämä tiedosto pyritään jatkuvasti pitämään mahdollisimman lyhyenä. Tämän vuoksi myös tiedostojen osien täytyy olla aivan peräkkäin toisissaan kiinni.

Ellei toisin mainita, on luvut ja merkkijonot muunnettu tiedostoon kirjoitettaessa tavuvektoreiksi seuraavasti:

1. Luvut ovat ei-negatiivisia kokonaislukuja ja muutos tavuvektoriksi tehdään bigendian-muodossa.

2. Merkkijonot muunnetaan käyttäen UTF8-koodausta.

3.2.2 Salaus

Paketissa on mahdollista käyttää salausta. Salauksen esittelyn yksinkertaistamiseksi tiedoston osat esitetään siten kuin salaus olisi aina päällä. Mitään ei kuitenkaan salata, mikäli paketin salaus on kytketty pois päältä.

Salaus tehdään *salauslohkoina*, joista jokainen on oma salattava yksikkönsä. Tämä tarkoittaa sitä, että erillisten salauslohkojen sisältö ei vaikuta toisiinsa kun taas salauslohkon sisällä yhdenkin bitin muuttaminen vaatii koko salauslohkon kirjoittamista tiedostoon uudelleen. Tämän vuoksi salauslohko koostuu yhdestä tai useammasta tietueesta, jotka on loogista kirjoittaa tiedostoon kerralla. Tällaisia ovat esimerkiksi:

1. Jokin itsenäisesti muuttuva luku.
2. Ryhmä lukuja, joiden toiminta liittyy selkeästi toisiinsa ja jotka on tallennettu tiedostoon peräkkäin.
3. Yhtenäinen osa ison tiedoston sisältöä.

Tiedoston osien esittelyssä on mainittu miten tietueet muodostavat salauslohkoja.

Lähes koko tiedosto on salattu. Ainoastaan tiedoston otsakkeessa on muutamia tietueita, joita ei salata. Niitä tarvitaan salauksesta ilmoittamiseen ja mahdollisen salauksen alustamiseen, joten niiden salaaminen ei olisi mielekästä. Otsakkeen esittelyssä on erikseen mainittu mitkä tietueet ovat salattuja ja mitkä eivät.

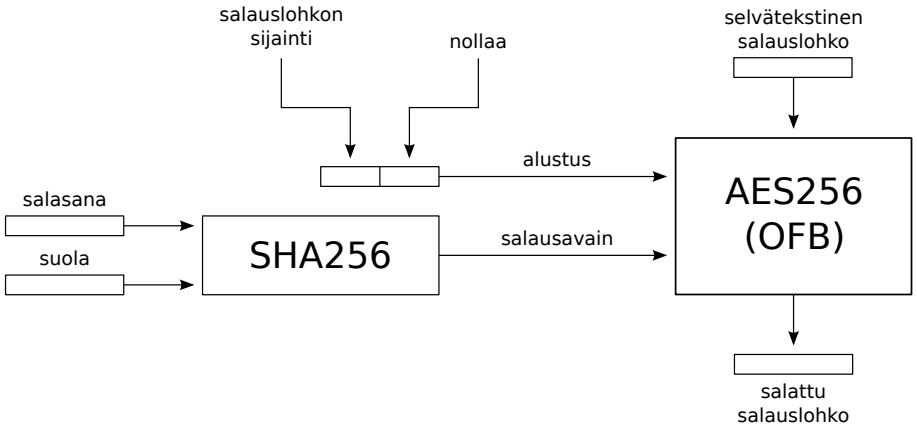
Koska salaamattomia tietueita on vähän, tehdään kaikki kirjoitukset tiedostoon salauslohkoina. Tämä koskee myös salaamaattomia tietueita. Niiden tapauksessa pidetään muistissa tietoa, että salausta ei tehdä vaikka se olisi muuten kytketty päälle.

Salauksessa käytetään *suolaa*. Suola on tässä tapauksessa 64 tavua pitkä satunnainen tavuvektori. Se luodaan paketin luomisen yhteydessä ja se pysyy sen jälkeen muuttumattomana. Sen tarkoituksena on yksilöidä jokaisen paketin salaus. Esimerkiksi kaksi täysin samansisältöistä ja samalla salasanalla suojattua pakettia näyttävät salattuina erilaisilta. Suolan luomiseen käytetään käyttöjärjestelmän tarjoamaa turvallista satunnaislukulähdettä.

Itse salaus toimii seuraavasti: Aluksi luodaan suola. Käyttäjän antamasta UTF8-koodatusta salasanasta ja suolasta muodostetaan SHA256-tiiviste. Tämä tiiviste on 256 bittiä pitkä datavektori, ja sopii siten suoraan salausavaimeksi AES256-symmetriselle salaimelle. AES256-salainta käytetään kohdassa 2.1.7 esitellyssä OFB-toimintamoodissa ja sen alustusvektorina käytetään 16 tavua pitkää datavektoria, josta ensimmäinen puolikas on täynnä nollaa. Toinen puolikas sisältää 64-bittisen positiivisen kokonaisluvun joka on muunnettu tavuvektoriksi. Kokonaisluvun arvoksi asetetaan salauslohkon sijanti tavuina mitattuna tiedoston alusta. Koko salausprosessi on esitetty kuvassa 3.3.

Salauksen vahvistamiseksi jotkin lipputietueet tallennetaan käyttäen satunnaislukuja seuraavasti: Jos lippu on pois päältä, tavun arvoksi tulee jokin luku väliltä 0–127. Jos lippu on päällä, tavun arvo on välillä 128–255.

Salaus tässä työssä, kuten muutkaan tietoturvallisuuteen liittyvät ratkaisut, eivät ole minkään salausasiantuntijan tarkistamia. Tarkoituksena on ollut lähinnä kevyen turvallisuuden saaminen ja näistä ratkaisuista löytyisi varmasti paljonkin paranneltavaa.



Kuva 3.3: Salauslohkon salaaminen käyttäen suolaa, käyttäjän antamaa salasanaa ja salauslohkon sijaintia. Huomaa, että tavuvektorit on ilmaistu siten, että niiden ensimmäinen tavu on oikealla.

3.2.3 Otsake

Otsakkeen tarkoituksena on toimia yleisenä varastona kaikille paketin tilaa kuvaaville tiedoille joita on yksi kappale ja jotka ovat vakiokokoisia. Tällä tavalla otsakkeen koko saadaan pidettyä myös vakiona, eikä otsaketta seuraavien osien tarvitse huolehtia otsakkeen koon muutoksista.

Otsakkeen sisältämät tietueet ovat monipuolisia ja ne on lueteltu alla samassa järjestyksessä kuin ne tiedostossa sijaitsevat. Ne löytyvät myös taulukosta 3.2.

1. Tiedoston tunniste. Tunniste sisältää neljä ASCII-koodattua merkkiä jotka muodostavat sanan `ROSA`. Tunniste on neljä tavua pitkä eikä sitä ole salattu.

2. Versionumero yhdellä tavulla ilmaistuna ja salaamattomana. Tämän työn kirjoittamisen aikana ainoa hyväksytty arvo on nolla. Tulevaisuudessa muut arvot ovat mahdollisia. Tiedoston rakenne on tällöin erilainen.

3. Lippu, joka kertoo onko paketti salattu vai ei. Se vie tilaa yhden tavun ja on salaamaton. Nolla tarkoittaa, että salausta ei käytetä ja muut arvot että käytetään.

4. Suola salaamattomana. Sen pituus on 64 tavua. Mikäli salaus ei ole käytössä, ei paketti sisällä suolaa.

5. Salasanan varmistin. Tämä tietue on olemassa vain mikäli salaus on käytössä. Se sisältää saman satunnaisen 32 tavua pitkän tavuvektorin kaksi kertaa peräkkäin. Tämä on ensimmäinen tietue joka salataan. Lisäksi se on samalla kokonainen salauslohko, eli tämä tietue salataan yksinään ilman muita tietueita. Kaikki tietueet tästä eteenpäin on salattu myös. Nimensä mukaisesti tätä

Taulukko 3.2: Otsakkeen sisältö. Sarake *Pakollinen* kertoo onko kyseinen tietue mukana salaamattomassa paketissa.

Tietue	Tavuja	Pakollinen	Salaus
Tunniste	4	Kyllä	Ei
Versionumero	1	Kyllä	Ei
Salauslippu	1	Kyllä	Ei
Suola	64	Ei	Ei
Salasanan varmistin	64	Ei	Salauslohkona
Kirjanpidon lippu	1	Kyllä	Salauslohkona
Kirjanpidon sijainti	8	Kyllä	Salauslohkona
Orpojen solmujen lippu	1	Kyllä	Salauslohkona
Viittaus rakennepuun juureen	64	Kyllä	Yhteisenä salaus-lohkona
Solmujen määrä	8	Kyllä	
Hakupuun alku	8	Kyllä	
Data-alueen loppu	8	Kyllä	

käytetään varmistamaan, että käyttäjä kirjoitti salasanan oikein. Salasanan huomataan menneen väärin, mikäli siitä ja suolasta luodulla salausavaimella ei pystytä purkamaan salasanan varmistinta sellaiseksi, että se sisältäisi kaksi identtistä tavuvektoria.

6. Lippu kirjanpidolle. Tämä lippu kertoo, sisältääkö tiedosto kirjanpidon jostain kesken jääneestä kirjoitusoperaatiosta. Sen pituus on yksi tavu ja se on salattu yhtenä salauslohkona. Mikäli kirjanpitoa ei löydy, tämän tavun arvo on välillä 0–127 ja mikäli löytyy, se on välillä 128–255.

7. Mahdollisen kirjanpidon sijainti tavuina tiedoston alusta mitattuna ja 64-bittisellä luvulla ilmaistuna. Mikäli kirjanpitoa ei ole, tämä sisältää määrittelemätöntä dataa. Tämä tietue on salattu yhtenä salauslohkona.

8. Orpojen solmujen lippu. Tämä lippu kertoo, löytyykö paketista mahdollisesti sellaisia rakennepuun solmuja, jotka eivät ole minkään muun solmun lapsia. Tätä käytetään selviämään virhetilanteista, kun pakettia muokanneen ohjelman suoritus on keskeytynyt. Sen pituus on yksi tavu ja sitä luetaan kuin kirjanpidon lippua. Se kirjoitetaan tiedostoon omana salauslohkonaan.

9. Viittaus rakennepuun juurisolmuun. Tämä tietue sisältää sen solmun tiivisteen, joka toimii rakennepuun juurena. Tiiviste on tavuvektori ja sen pituus on 64 tavua. Se on salattu yhteisenä salauslohkona kolmen seuraavan tietueen kesken.

10. Rakennepuun solmujen määrä ilmaistuna 64-bittisellä luvulla.

11. Metadata-alueella sijaitsevan *hakupuun* aloitussolmun järjestysluku ilmaistuna 64-bittisellä luvulla. Metadata-alueen hakupuu esitellään myöhemmin.

12. Data-alueen loppu laskettuna tavuina tiedoston alusta. Se on ilmaistu 64-bittisellä luvulla.

3.2.4 Metadata-alue

Solmujen vaihtelevien ominaisuuksien vuoksi niiden vaatima tilantarve voi olla mitä tahansa muutamista tavuista miljooniin tavuihin. Tämä asettaa samanlaisia haasteita mitä tiedostojärjestelmissä on tiedostojen, hakemistojen ja muiden vaihtelevan mittaisten tietorakenteiden vuoksi. Myös ratkaisu on samankaltainen: vakiokokoiset tiedot solmuista on koottu eri alueelle kuin vaihtelevan mittaiset tiedot.

Tässä ratkaisussa vakiomittaisten tietojen aluetta kutsutaan metadata-alueeksi. Se on jaettu yksiköihin, joita kutsutaan *metadatoiksi*. Jokainen metadata on saman pituinen ja sisältää solmujen vakiomittaisten tietojen lisäksi tietoja, joita käytetään deduplikoinnissa, hakupuun muodostamisessa ja data-alueelle viittaamisessa.

Solmun tyyppi tekee tässä poikkeuksen. Vakiomittaisuudestaan huolimatta se on tallennettu vaihtelevan mittaisten tietojen kanssa data-alueella oleviin *datayksiköihin*, joita on yksi jokaista metadataa kohden. Datayksiköt sekä syy solmun tyypin poikkeukseen esitellään yhdessä data-alueen kanssa myöhemmin.

Rakennepuun lisäksi paketti sisältää myös hakupuun, joka sijaitsee metadata-alueella. Metadatat ovat hakupuun solmuja. Seuraavaksi käydään läpi hakupuun rakenne ja tämän jälkeen solmun metadata-tietue.

Hakupuu

Järjestelmän sulava käyttö vaatii, että solmuja on nopea hakea tiedostosta niiden tiivisteiden perusteella. Tästä on hyötyä varsinkin deduplikoinnissa. Myös normaali rakennepuun käyttö ja läpikäyminen vaatii tätä, sillä solmuihin viitataan aina niiden tiivisteiden perusteella.

Koska tiedosto on yksi suuri tavuvektori, ovat myös metadatat tallennettuina peräkkäin kuten vektorissa. Tätä vektoria kuitenkin käytetään binäärihakupuuna, joka on järjestetty solmujen tiivisteiden perusteella. Tiivisteet ovat vakiomittaisia tavuvektoreita eli binäärilukuja, ja siten vertailukelpoisia. Tällä tavoin on mahdollista löytää solmuja logaritmisessa suoritusajassa.

Tiedoston otsakkeessa on järjestysluku, joka kertoo monesko metadata on hakupuun juuri. Jokaisesta metadatasta on viittaus kahteen muuhun metadataan niiden järjestyslukujen perusteella. Näitä kutsutaan *pienemmäksi* ja *suuremmaksi lapseksi*. Ensimmäinen näistä viittaa johonkin metadataan, jonka solmun tiiviste on pienempi kuin viittaavan solmun tiiviste. Toinen viittaa vastaavasti sellaiseen metadataan,

Taulukko 3.3: Metadatan sisältö

Tietue	Tavuja
Tiiviste	64
Viittausten määrä tähän solmuun rakennepuussa	3
Vanhemman järjestysluku hakupuussa	8
Pienemmän lapsen järjestysluku hakupuussa	8
Suuremman lapsen järjestysluku hakupuussa	8
Datayksikön sijainti data-alueella	8

jonka solmun tiiviste on suurempi kuin viittaavan solmun tiiviste. Perinteisen binäärihakupuun tapaan hakupuu ei voi muodostaa silmukoita viittauksillaan ja jokainen hakupuun solmu eli metadata on löydettävissä juuresta lähtien. Jotta hakupuu ei jatkuisi loputtomiin, pitää olla myös mahdollista viitata tyhjään. Järjestysluvut esitetään 64-bittisinä kokonaislukuina, joten tyhjään on loogista viitata heksaluvulla `ffff ffff ffff ffff`. Se on mainitun arvoalueen suurin mahdollinen arvo.

Metadata-tietue

Metadata sisältää lopulta hyvin vähän tietoa, joka liittyisi suoraan varsinaiseen tietorakenteeseen eli rakennepuuhun. Suurin osa sen tiedoista liittyy deduplikointiin, hakupuuhun ja data-alueeseen, kuten selviää taulukosta 3.3. Metadatan tietueet on esitelty myös yksityiskohtaisesti alla.

Metadata on 99 tavua pitkä ja se salataan yhtenä salauslohkona. Siinä on seuraavat tietueet:

1. Tiiviste, joka on ainoa suoraan rakennepuuhun liittyvä tietue. SHA512-tiivisteenä se on 64 tavua pitkä.

2. Viittausten määrä tähän solmuun rakennepuussa. Deduplikoinnin vuoksi yhteen solmuun voi viitata mielivaltainen määrä muita solmuja. Tämän määrän selvittäminen vaatisi lähes koko tiedoston läpikäymistä, joten sen vuoksi metadatassa pidetään kirjaa tästä luvusta. Kun tämä menee nollaksi, voi solmun poistaa. Otsakkeessa sijaitseva viittaus juurisolmuun tulkitaan yhdeksi viittaukseksi. Ilman tätä tulkintaa koko rakennepuu tuhoutuisi, sillä silloin juurisolmu pitäisi poistaa, kuten myös sen lapset ja niin edelleen. Tämä luku on poikkeuksellisesti ilmaistu 24 bitillä, sillä mistään solmusta ei uskota tulevan yli 16 000 000 duplikaattia.

3. Vanhemman järjestysluku hakupuussa 64-bittisellä luvulla ilmaistuna. Tätä vaaditaan, jotta poistot hakupuusta ovat mahdollisia. Tähän aiheeseen palataan myöhemmin, kun tutustutaan ohjelman toimintaan.

4. Pienemmän lapsen järjestysluku hakupuussa 64-bittisellä luvulla ilmaistuna.
5. Suuremman lapsen järjestysluku hakupuussa 64-bittisellä luvulla ilmaistuna.
6. Solmun datayksikön sijainti data-alueella. Tämä on laskettu tavuina tiedoston alusta. Luku on 64-bittinen.

3.2.5 Data-alue

Data-alue koostuu vaihtelevan mittaisista tietueista, joita kutsutaan datayksiköiksi. Nämä sisältävät solmujen ne tiedot, joita ei voitu metadata-alueelle kirjoittaa. Data-alueen tiedot sisältävät myös kaiken sen, mitä tarvitaan solmun tiivisteen laskemiseksi.

Metadata-alueelle kirjoitettujen tietojen jälkeen solmusta on vielä kirjoittamatta sen tyyppi, mahdollinen oheisdata sekä mahdolliset viittaukset lapsiin. Solmun tyyppi on ainoa tietue, joka löytyy jokaisesta solmusta. Muut tiedot ovat vaihtelevan mittaisia, vaihtelevan tyyppisiä eikä niitä ole kaikissa solmutyypeissä. Tämä vaikeuttaa datayksikön luomista näistä tiedoista.

Ratkaisu on muuntaa mahdollinen oheisdata ja mahdolliset lapsiviittaukset tavuvektoriksi. Jokaisella solmutyypillä on oma tapansa muunnoksen toteuttamiseksi. Muunnoksesta seuraavaa tavuvektoria kutsutaan solmun *serialisoiduksi dataksi*.

Solmun serialisoidun datan muodostaminen

Alla on lueteltu miten eri solmutyypeille tehdään muunnos serialisoiduksi dataksi. Monista tiedoista puuttuu pituus, jota tarvitaan kun serialisoitua dataa muunnetaan olioksi ohjelman käyttöön. Se on kuitenkin pääteltävissä koko serialisoidun datan pituudesta, joka saadaan luettaessa datayksikkö levyltä.

1. Hakemiston tapauksessa sen lapsiviittausten tiedot eli viittaus ja siihen liittyvä hakemistometadata, muunnetaan ensin tavuvektoreiksi siten, että jokaisesta lapsiviittauksesta tulee oma tavuvektorinsa. Tämän jälkeen ne asetetaan peräkkäin, jolloin ne muodostavat koko hakemiston serialisoidun datan. Yhden lapsiviittauksen serialisointi on esitetty taulukossa 3.4.

   Lapsiviittausten serialisoinnissa lapsen tyyppi koodataan seuraavasti: Nolla tarkoittaa, että lapsen tyyppi on hakemisto, yksi tarkoittaa tiedostoa ja kaksi symbolista linkkiä.

   Lapsen hakemistometadata kuvataan listalla avainarvopareja. Avaimina toimivat merkkijonot. Myös lukuarvot esitetään poikkeuksellisesti merkkijonoina. Tämän työn kirjoittamisen aikana mahdollisia avaimia ovat `user` ja `group`, jotka kuvaavat lapsen omistajan käyttäjätunnusta ja ryhmää sekä `crtime` ja

`mtime`, jotka kuvaavat lapsen luontiaikaa ja muokkausaikaa. Nämä ajat on ilmaistu kuluneina sekunteina vuoden 1970 alusta ja lisätarkkuutta voi antaa nanosekunteina avaimilla `crtime_nsec` ja `mtime_nsec`. Uusia avaimia on mahdollista keksiä tiedostoformaatin kehittyessä. Mitkään avainarvoparit eivät kuitenkaan ole pakollisia, eli ne voi tarvittaessa jättää pois.

Sekä avainarvoparit että lapsiviittaukset on hyvä järjestää lopulliseksi tavuvektoriksi niiden avainten ja tiedostonimien mukaan. Pakollista se ei ole, mutta siitä on hyötyä deduplikoinnin kannalta. Järjestämisen avulla hakemistot, jotka sisältävät samat lapset samoilla hakemistometadatoilla, muodostavat aina saman serialisoidun datan ja siten myös saman tiivisteen.

2. Tiedosto muunnetaan tavuvektoriksi listaamalla peräkkäin kaikista sen datalohkolapsista sekä datalohkon pituus että sen tiiviste eli viittaus siihen. Datalohkon pituus muunnetaan tavuvektoriksi 32-bittisenä lukuna ja tiivisteet ovat jo valmiiksi tavuvektoreita, joiden pituus on 64 tavua. Tiedoston serialisoitu data sisältää siis ensimmäisenä ensimmäisen datalohkon pituuden, sitten viittauksen ensimmäiseen datalohkoon, sitten toisen datalohkon pituuden jne. Muunnoksen koko on yhteensä $(4 + 64) \times N$ tavua, missä N on tiedoston käyttämien datalohkojen määrä. Datalohkon pituutta käytetään, jotta tiedostoa voisi lukea myös keskeltä ilman että sen datalohkojen pituuksia tarvitsisi selvittää datalohkoista itsestään.

3. Datalohkot ovat jo valmiiksi tavuvektorimuodossa, sillä ne sisältävät vain yhtenäisen osan jonkin tiedoston sisällöstä. Täten muunnosta ei tehdä, vaan serialisoitu data saadaan suoraan datalohkon sisällöstä.

4. Symbolisten linkkien serialisoitu data saadaan muuntamalla niiden sisältämä merkkijono tavuvektoriksi.

Tiivisteen laskeminen

Edellä kuvattu serialisoitu data on myös olennainen tiivisteen laskemiseksi. Solmun tiiviste lasketaan antamalla SHA512-algoritmille ensin solmun tyyppi yhtenä tavuna ja sen jälkeen solmun serialisoitu data.

Tyyppiä kuvaavan tavun arvo määräytyy seuraavasti: nolla tarkoittaa hakemistoa, yksi tiedostoa, kaksi datalohkoa ja kolme symbolista linkkiä.

Solmun tyyppi on mukana tiivisteen laskemisessa jotta voitaisiin varmistaa, että eri tyyppiset solmut saavat erilaiset tiivisteet. Syy tähän on lähinnä historiallinen. Eräässä vaiheessa ohjelman kehitystä tietynlainen hakemistorakenne muodosti ongelman, kun sekä tyhjät hakemistot että tyhjät tiedostot saivat saman tiivisteen.

Taulukko 3.4: Hakemistosolmun lapsiviittauksen serialisointi

Tietue		Tavuja
Nimen pituus		2
Nimi		1–65535
Lapsen tyyppi		1
Lapsen tiiviste		64
Avainarvoparien määrä		2
Avainarvo-pari	Avaimen pituus	1
	Avain	0-255
	Arvon pituus	2
	Arvo	0-65535

Ongelma korjattiin tällöin varmistamalla eri solmutyyppien eri tiivisteet. Vaikka ongelmaa ei välttämättä nykyisin enää olisikaan, on varmistus jätetty mahdollisten samankaltaisten ongelmien varalta.

Tiivisteen laskemisen kannalta on olennaista, että kaikki siihen vaaditut tiedot löytyvät datayksiköstä. Datayksiköstä ei ole viittausta sen metadataan, joten tiivisteen laskeminen on ainoa tapa löytää metadata jonka kanssa datayksikkö muodostaa solmun. Tämä on myös syy siihen, miksi solmun tyyppi on tallennettu datayksikköön eikä metadataan.

Viittauksia datayksiköstä metadataan päin tarvitaan silloin, kun datayksiköitä siirrellään data-alueella ja metadataan pitää päivittää datayksikön uusi sijainti.

Datayksiköiden rakenne ja sijainti data-alueella

Koska datayksiköt ovat vaihtelevan mittaisia, ei data-aluetta voi mieltää vektoriksi joka koostuu vakiokokoisista korkeamman tason yksiköistä, kuten metadata-alue. Sen sijaan data-alueen voi mieltää pitkäksi tavuvektoriksi, joka on eri mittaisten datayksiköiden täyttämä.

Datayksiköt ovat tässä tavuvektorissa tiiviisti peräkkäin, eli yhden datayksikön perässä on heti seuraava. Datayksiköihin on myös tallennettu tieto niiden pituudesta, joten niitä on mahdollista käydä läpi siirtyen eteenpäin datayksikkö kerrallaan. Läpikäynti täytyy aloittaa aina jostain tunnetusta datayksiköstä, esimerkiksi alusta.

Samaan tapaan kuin tiedostojärjestelmässä, solmuja syntyy ja poistuu järjestelmää muokattaessa. Kuten tiedostojärjestelmässä, ei tässäkään ole triviaalia keinoa täyttää syntyneitä rakoja. Jotta vaatimus datayksiköiden peräkkäisyydestä kuitenkin toteutuisi, on datayksiköt mahdollista merkitä käyttämättömäksi tilaksi silloin, kun sen omistanut solmu on poistettu.

Kuten kohdassa 2.1.2 todettiin, joissain tiedostojärjestelmissä vaihtelevan mit-

taiset yksikötkin on jaettu joihinkin vakiokokoisiin lohkoihin tehostamaan tyhjän tilan käsittelyä. Tässä työssä datayksiköitä ei kuitenkaan yksinkertaistamisen vuoksi ole jaettu tällä tavalla, vaan datayksiköt sijaitsevat koko pituudeltaan yhtenäisinä, joskin tiedostojen sisältö on jaettu solmujen tasolla erillisiksi datalohkoiksi.

Datayksikkö koostuu neljä tavua pitkästä otsakkeesta sekä joko solmun serialisoidusta datasta tai käyttämättömäksi merkityistä tavuista. Solmun serialisoitu data on pakatussa muodossa. Otsake on 32-bittinen kokonaisluku, joka on muunnettu samaan tapaan tavuvektoriksi kuin muutkin luvut. Tämä luku sisältää kuitenkin useita tietoja, jotka ovat:

1. Yhdellä bitillä ilmaistu tieto siitä, sisältääkö datayksikkö vain käyttämätöntä tilaa.

2. Mahdollisen solmun tyyppi kahdella bitillä ilmaistuna. Tyyppiä kuvaavan luvun arvo valitaan samalla tavalla kuin tiivistettä laskettaessa.

3. Datayksikön pituus tavuissa mitattuna ja 29 bitillä ilmaistuna. Pituus ei sisällä otsakkeen pituutta, eli se kertoo joko käyttämättömien tavujen määrän tai solmun serialisoidun datan pakatun pituuden.

Otsakkeen 32-bittinen kokonaisluku kootaan pienemmistä luvuista siten, että yllä olevan listan ensimmäinen jäsen käyttää eniten merkitsevää bittiä, toinen jäsen toiseksi ja kolmanneksi merkitsevimpiä bittejä ja datayksikön pituus loppuja bittejä. Tämä kokoaminen tehdään siis ennen luvun muuntamista 32-bittisestä luvusta neljäksi tavuksi bigendian-muodossa.

Otsakkeen jälkeen seuraa joko pakattu serialisoitu data tai käyttämättömän tilan tapauksessa tavuja, jotka ovat arvoiltaan määrittelemättömiä. Serialisoidun datan pakkaamiseen käytetään patenteista vapaata zlib-kirjastoa [8].

Vaikka otsake ei sisällä serialisoidun datan pakkaamatonta pituutta, on se mahdollista selvittää purkamalla pakattu serialisoitu data. Pakatun serialisoidun datan pituus löytyy datayksikön otsakkeesta.

Datayksikkö kirjoitetaan tiedostoon salauslohkoina seuraavasti: Otsake kirjoitetaan aina omana salauslohkonaan, oli kyseessä sitten serialisoitua dataa tai käyttämättömiä tavuja sisältävä datayksikkö. Serialisoitu data kirjoitetaan yhtenä salauslohkona. Käyttämättömien tavujen tapauksessa ei tiedostoon kirjoiteta otsakkeen lisäksi muuta, eli käyttämättömät tavut sisältävät saman mitä tiedostossa oli aiemminkin samassa kohdassa.

3.2.6 Kirjanpito

Kun paketin tietoja kirjoitetaan tietokoneen muistista tiedostoon, on vaarana että paketin eheys rikkoutuu ohjelman yllättävän keskeytyksen vuoksi kun osa olennaisis-

ta tiedoista jää kirjoittamatta. Tämän ongelman estämiseksi tiedoston loppuun täytyy ennen varsinaista tiedostoon kirjoittamista luoda kirjanpito. Kirjanpito sisältää kokoelman salauslohkoja, jotka on tarkoitus kirjoittaa tiedostoon. Lisäksi kirjanpito sisältää tiedot niiden sijainneista sekä pitääkö ne salata vai ei.

Toisin kuin muut tiedoston osat, kirjanpito on vain väliaikainen osa. Sen vuoksi sen on luontevaa sijaita tiedoston lopussa, sillä tiedoston loppuun on nopea kirjoittaa väliaikaista tietoa. Myös väliaikaisten tietojen poisto tiedoston lopusta on nopeaa ja lyhentää tiedostoa välittömästi. Kirjoitettavista salauslohkoista sekä otsakkeesta löytyvästä tiedoston data-alueen lopusta on myös helppo päätellä mihin muut tiedoston osat loppuvat.

Tiedostoon kirjoitus ja kirjanpidon luominen tapahtuvat seuraavassa järjestyksessä:

1. Luodaan kirjanpito kirjoitettavista salauslohkoista tiedoston loppuun ja tallennetaan kirjanpidon sijainti otsakkeeseen.
2. Kytketään otsakkeesta kirjanpidon lippu päälle.
3. Kirjoitetaan salauslohkot tiedostoon.
4. Kytketään otsakkeesta kirjanpidon lippu pois päältä.

Jokaisen vaiheen jälkeen kutsutaan tiedostojärjestelmän *flush*-operaatiota, joka varmistaa että tiedot todella tulevat kirjoitetuksi levylle, eivätkä esimerkiksi jää johonkin puskureihin, jotka voisivat sähkökatkoksen seurauksena kadota. Kirjanpitoa ei ole pakko poistaa heti, sillä tiedoston lopussa saa olla ylimääräistä dataa. Sen lippu on kuitenkin pakko poistaa käytöstä heti kun varsinaiset salauslohkot on saatu kirjoitettua.

Kirjoittamisen vaiheista voi päätellä, että vaikka ohjelman suoritus keskeytyisi, niin tiedosto jää kuitenkin johonkin seuraavista eheistä tiloista:

1. Salauslohkoja ja kirjanpitoa ei kirjoitettu ollenkaan tiedostoon.
2. Kirjanpito kirjoitettiin joko osittain tai kokonaan, mutta kirjanpidon lippua ei ehditty kytkeä päälle, joten kirjanpito jätetään huomiotta.
3. Salauslohkojen kirjoittaminen keskeytyi, mutta kirjanpidon avulla ne voidaan kuitenkin kirjoittaa kun tiedostoa seuraavan kerran käytetään.
4. Salauslohkot saatiin kirjoitettua täydellisesti.

Taulukko 3.5: Salauslohko osana kirjanpitoa

Tietue	Tavuja
Kirjoitettavan salauslohkon sijainti	8
Salataanko salauslohko	1
Salauslohkon pituus	4
Salauslohkon sisältö	1–$(2^{32}-1)$

Tiedoston eheän tilan ylläpito vaatii tietysti, että tiedostojärjestelmän flush-operaatio toimii ja että tiedostoon kirjoitetaan kerralla vain loogisia salauslohkoja. Jos ollaan esimerkiksi päivittämässä rakennepuun juurisolmun tietuetta tiedoston otsakkeesta, täytyy samalla myös päivittää uuden ja vanhan juurisolmun viittauslaskureita.

Mikäli tiedostoa avattaessa huomataan, että se sisältää kirjanpidon, täytyy se kirjoittaa uudestaan. Tämä kuvataan myöhemmin ohjelman toimintaa esiteltäessä.

Kirjanpidon sisältämät salauslohkot on tallennettu kirjanpidon näkökulmasta selvätekstisenä. Ne salataan vasta kirjoitettaessa lopulliseen sijaintiinsa. Sen sijaan kirjanpito itse on kirjoitettu ja salattu kahtena salauslohkona, joten tiedot eivät pääse vuotamaan ulkopuolisille.

Ensimmäinen salauslohko on neljä tavua pitkä ja sisältää kirjanpidon pituuden. Heti tämän jälkeen seuraa toinen salauslohko, joka sisältää kirjanpitoon tallennetut salauslohkot peräkkäin. Yhden tällaisen salauslohkon tavumuunnos on esitetty taulukossa 3.5. Lippua luetaan siten, että tavun arvot 0–127 tarkoittavat lipun olevan epätosi ja 128–255 tosi.

Kirjanpito on sikäli joustava, että se ei määrittele kuinka paljon tiedostoon ollaan kirjoittamassa. Se ei myöskään määrittele mitä tyyppiä kirjoitettavat yksiköt ovat.

Kahdessa tapauksessa halutut tiedot kirjoitetaan tiedostoon ilman kirjanpitoa. Ensimmäinen näistä on orpojen solmujen lipun kirjoitus. Toinen tapaus on tyhjän paketin luominen. Molemmille tapauksille on yhteistä se, että vaikka kirjoitusoperaatio epäonnistuisi, ei siitä seuraa kovin suuria menetyksiä. Lisäksi ensimmäinen tapaus on ilman apukeinojakin riittävän atominen.

4. OHJELMA

Ohjelma on toteutettu C++:lla. Kieleksi valittiin C++, sillä vaikka se tarjoaa useita edistyneitä ominaisuuksia, ovat sillä toteutetut ohjelmat kuitenkin hyvin nopeita. C++:n edistyneet ominaisuudet ovat helpottaneet monien algoritmien ohjelmointia tässä työssä.

Ohjelmakoodia kertyi yhteensä 4 853 riviä. Tästä kommentteja on 691 ja varsinaista koodia 4 162 riviä.

Ohjelmaan perehdytään ensin sen rakenteen, sitten sen toiminnan ja lopuksi sen käytön kautta. Myös rakennetta esiteltäessä käydään läpi toimintaa, mutta sen pääpaino on kuitenkin esitellä olennaisia osia ohjelmasta.

4.1 Ohjelman rakenne

Tässä osassa esitellään kolme luokkaa, jotka voidaan nähdä omiksi erillisiksi osikseen. Toki ohjelmassa on myös joitain muita luokkia, mutta nämä ovat oleellisimmat:

1. FileIO, jonka kautta kaikki lukeminen, kirjoittaminen sekä salauksen käsittely tehdään.

2. Node-luokka ja siitä periytetyt muut luokat, joiden avulla rakennepuun käsittely saadaan luontevaksi. Lisäksi tässä nähdään olio-ohjelmoinnin hyödyt, kun on toteutettavana yksi kantatyyppi eli solmu, jolla on useita alityyppejä eli hakemisto-, tiedosto-, datalohko- ja symbolisen linkin solmut.

3. Archive-luokka, joka kuvastaa pakettia.

4.1.1 FileIO-luokka

FileIO-luokka tarjoaa edistyneen kerroksen tiedoston lukemiseen ja kirjoittamiseen. Se huolehtii esimerkiksi luku- ja kirjoitusoperaatioiden välimuistista ja salaamisesta sekä kirjanpidosta ja tiedoston lyhentämisestä tarvittaessa.

Kaikki FileIO-luokalle annetut kirjoitukset ja sillä tehdyt lukemiset ovat aina kokonaisia, yksittäisiä salauslohkoja. Vaikka luokka ei itse tätä vaadikaan, on tämä pakollista salauksen toimimisen vuoksi.

Kuten tiedoston kirjanpidosta jo todettiin, jotkut toisiinsa liittyvät salauslohkot on kirjoitettava tiedostoon kerralla, jotta paketin eheys säilyisi. FileIO-luokka tarjoaa tähän tarkoitukseen kirjoitusoperaatioiden välimuistin. Sen avulla on mahdollista

tehdä useita kirjoitusoperaatioita keskusmuistiin odottamaan tiedostoon kirjoitusta. Vasta varsinaisen pyynnön jälkeen ne kirjoitetaan tiedostoon.

Kirjoitusoperaatioiden välimuisti muodostaa kuitenkin ongelman. Muistiin voi olla merkitty, että jokin tiedoston kohta tulee sisältämään päivitettyä tietoa. Tiedostossa on kuitenkin edelleen päivittämätön tieto. Tämän vuoksi onkin luontevaa, että myös lukuoperaatiot tapahtuvat FileIO-luokan kautta. Tällöin luokalla on mahdollisuus tarkastaa löytyykö jostain tiedostosta luettavasta osasta uudempi versio muistista.

FileIO-luokka sisältää myös lukuoperaatioiden välimuistin. Tämä tarkoittaa, että kaikki tiedostosta tehtävät luvut ja sinne tehtävät kirjoitukset päivittävät myös erillistä puskuria keskusmuistissa. Tästä puskurista on erittäin nopea tarkastaa, mikä tiedoston sisältö tietyssä kohtaa on, mikäli kyseinen salauslohko on tallennettu puskuriin.

Lukuvälimuistilla on myös jokin raja, mitä se ei saa ylittää. Tämä raja ei ole aivan ehdoton, mutta ohjelma pyrkii pitämään lukuvälimuistiin tallennettujen salauslohkojen yhteiskoon sitä pienempänä. Mikäli raja ylitetään, aletaan lukuvälimuistiin tallennettuja salauslohkoja poistaa. Poistot kohdistuvat niihin salauslohkoihin, jotka ovat pisimpään olleet käyttämättöminä.

Kuten kohdassa 2.1.2 mainittiin, lukuvälimuisti löytyy usein myös käyttöjärjestelmästä, mutta joissain tapauksissa voi olla hyvä, että ohjelma toteuttaa sen itse. Tällöin ohjelma voi pitää esimerkiksi suurempaa välimuistia kuin mitä järjestelmän keskusmuisti on. Välimuisti menee tällöin hitaaseen swap-muistiin, mutta se voi silti olla pienempi paha kun tiedostoa käytetään esimerkiksi verkon yli. Lisäksi keskusmuisti on prioriteetiltaan levyvälimuistia korkeampaa. Täten minkään muun ohjelman samanaikainen suoritus ei vähennä tämän ohjelman lukuvälimuistiin käytetyn muistin määrää.

Koska FileIO-luokka huolehtii kirjoitusoperaatioiden välimuistista, on luontevaa että se huolehtii myös kirjanpidosta. Lisäksi, koska FileIO-luokka käsittelee varsinaista tiedostoa, on myös luontevaa että se ohjelman sulkeutuessa lyhentää tiedostoa sen verran, että turhaksi käynyt kirjanpito saadaan poistettua.

Toinen kirjoitusoperaatioiden välimuistiin liittyvä ominaisuus on kirjoitusoperaatioiden ketjuttaminen. Varsinkin verkon yli toimittaessa voi olla haitaksi jos pienetkin kirjoitusoperaatiot toteutetaan heti, varsinkin kun niiden jälkeen saattaa tulla uusi kirjoitusoperaatio juuri samaan kohtaan. Tämän vuoksi FileIO-luokalle voi antaa jonkin rajan, kuinka suureksi se kasvattaa kirjoitusvälimuistia, ennen kuin kirjoittaa sen kerralla tiedostoon. Lisäksi ominaisuudessa poistetaan päällekkäiset kirjoitukset. Jos ensin tulee käsky kirjoittaa johonkin kohtaan ja sen jälkeen samaan kohtaan uudelleen, jää ketjuun merkintä vain viimeisimmästä kirjoituksesta.

Näiden ominaisuuksien lisäksi FileIO huolehtii myös salaamisesta. Koska salaa-

minen vie melko paljon tehoa, voidaan se tällä tavoin tehdä vasta juuri ennen tiedostoon kirjoittamista. Tässä vaiheessa ollaan varmistuttu esimerkiksi siitä, että välimuistissa olevat kirjoitukset eivät sisällä päällekkäisyyksiä. Kaikki välimuistissa olevat salauslohkot ovat siis aina salaamattomia.

4.1.2 Node-luokka ja siitä periytetyt luokat

Olio-ohjelmoinnin perintä on luonteva ratkaisu rakennepuun erityyppisten solmujen toteuttamiseen. Sen avulla on mahdollista saada kätevästi esimerkiksi eri solmutyyppien serialisoidun datan muodostus yhden rajapinnan alle.

Node-luokasta on periytetty luokat Folder, File, Datablock ja Symlink, jotka vastaavat suomenkielisiä vastineitaan. Node-luokassa on määritelty kolme jäsenfunktiota, jotka periytettyjen luokkien pitää toteuttaa. Ne ovat:

1. `getType()`, joka palauttaa solmun tiivisteen laskennassa käytetyn tyypin numeroarvon.

2. `serialize()`, joka palauttaa solmun serialisoidun datan yhtenä tavuvektorina.

3. `getNonUniqueChildren()`. Tätä jäsenfunktiota käytetään laskettaessa viittauksien määriä solmuista toisiin solmuihin. Solmu palauttaa viittaukset lapsiinsa, eli näiden tiivisteet. Kuten nimestä voi päätellä, tämän funktion palauttama taulukko tiivisteitä voi sisältää viittauksen samaan lapseen useaan kertaan.

   Laskettaessa solmuihin olevia viittauksia, erottelee kantaluokka tästä duplikaatit. Näin solmuihin ei tule turhaan suuria määriä viittauksia. Suuria viittausmääriä voisi tulla esimerkiksi sellaiseen solmuun, joka kuvastaa nollia täynnä olevaa datalohkoa. Suuri osa nollaa sisältävistä tiedostoista viittaa tähän solmuun nytkin, mutta jokainen niistä vain kerran. Mikäli duplikaattiviittauksia ei poistettaisi, voisi jokainen viitata tähän useita kertoja. Tällöin solmun viittauslaskurin arvoalue ei välttämättä riittäisi.

   Varsinaiset viittaukset, eli esimerkiksi tieto siitä, että tiedostossa on peräkkäin sata samanlaista nolladatalohkoa, löytyvät serialisoidusta datasta.

4.1.3 Archive

Archive kuvastaa yhtä avattua pakettia. Se omistaa yhden FileIO-luokan olion.

Olennaisin osa luokan rajapintaa liittyy paketin sisällön lukemiseen ja muokkaamiseen. Rajapinta tarjoaa esimerkiksi mahdollisuuden käsitellä suuria hakemistorakenteita, joita voi sekä kopioida pakettiin sisälle että sieltä ulos. Rajapinnan kautta

voi myös poistaa sisältöä paketista ja luoda sinne uutta, esimerkiksi tyhjiä hakemistoja. Viittaukset paketin sisälle näissä luku- ja muokkausoperaatioissa tehdään käyttäen tavallisia polkuja.

Lisäksi rajapinnassa on joitain muita jäsenfunktioita, esimerkiksi paketin luomiseksi, avaamiseksi ja sen sisällön eheyden tarkastamiseksi.

Archive-olio voisi sisältää paljonkin valmiiksi luettua tietoa paketista, esimerkiksi suuren osan solmujen metadatoista tai muuta vastaavaa. Nämä tiedot on kuitenkin jo tallennettu tavuvektorimuodossa FileIO-olion lukuvälimuistiin. Sieltä ne on tarvittaessa hyvin nopea hakea ja muuntaa varsinaisiksi ohjelman käsiteltäviksi olioiksi.

Olioon on tallennettu lähinnä tiedoston otsakkeen sisältämiä tietoja, sillä niitä tarvitaan niin useasti, ettei niitä ole luontevaa hakea aina uudelleen lukuvälimuistista tai tiedostosta. Varsinkin pienen lukuvälimuistin tapauksessa tämä on hyväksi, sillä silloin nämä tiedot eivät pyyhkiinny välimuistista muiden luku- tai kirjoitusoperaatioiden vuoksi. Muistissa pidetyt tiedot ovat juurisolmun tiiviste, solmujen määrä, hakupuun juurisolmun järjestysluku, data-alueen loppu sekä tieto siitä onko paketissa mahdollisesti orpoja solmuja.

Lisäksi Archive-olio pitää muistissaan käyttäjän salasanasta ja suolasta muodostettua salausavainta, mikäli salaus on kytketty päälle.

4.2 Ohjelman toiminta

Ohjelman toiminta etenee seuraavasti: Riippuen haluaako käyttäjä tehdä muutoksia, avataan tiedosto vain luettavaksi tai sitten myös kirjoitettavaksi. Näissä tiloissa on muutamia eroavaisuuksia.

Molemmissa tarkastetaan ensin, onko tiedosto jo olemassa. Samalla tarkastetaan onko käyttäjä antanut salasanan oikein, mikäli tiedostossa on salaus kytkettynä päälle. Mikäli tiedostoa ei ole olemassa ja tiedosto on avattu vain luettavaksi, annetaan käyttäjälle virheilmoitus tiedoston puuttumisesta ja lopetetaan suoritus. Mikäli tarkoitus on tehdä myös muokkauksia eikä tiedostoa löydy, luodaan tyhjän paketin sisältävä tiedosto. Tämä sisältää sellaisen rakennepuun, jossa on vain tyhjä hakemistosolmu juurisolmuna.

Kun olemassa oleva tiedosto avataan, pitää siitä ensimmäiseksi tarkastaa sisältääkö se muutamia virheitä, jotka ovat peräisin ohjelman suorituksen yllättävästä keskeytymisestä. Näitä virheitä ovat keskeytyneet kirjoitusoperaatiot sekä keskeytyneet orpojen solmujen poistamiset. Näiden korjaustoimet, eli *keskeytyneen kirjoitusoperaation viimeistely* ja *orpojen solmujen poistaminen*, käydään läpi myöhemmin.

Kun virheet on korjattu, päästään toteuttamaan varsinaisia käyttäjän pyytämiä operaatioita. Rakennepuusta joko haetaan käyttäjän pyytämiä tietoja tai siihen tehdään muokkauksia.

Lopuksi, mikäli käyttäjä muokkasi tiedostoa, suoritetaan sille vielä pientä optimointia eli *tyhjän tilan täyttämistä data-alueella*. Tähän sekä muihin olennaisiin toimintoihin ohjelmassa perehdytään seuraavaksi hieman syvällisemmin.

4.2.1 Keskeytyneen kirjoitusoperaation viimeistely

Keskeytynyt kirjoitusoperaatio havaitaan siitä, että tiedosto sisältää kirjanpidon. Sen viimeistely tapahtuu hieman eri tavoilla riippuen siitä, onko tiedosto avattu vain lukua vai myös kirjoittamista varten. Mikäli ollaan luku- ja kirjoitustilassa, voidaan kirjoitusoperaatio viimeistellä suoraan tiedostoon.

Kun vain lukuoperaatiot ovat sallittuja, kirjoitusoperaation viimeistely tehdään FileIO-luokan lukuvälimuistiin. Tällä tavalla FileIO-luokan käyttäjälle näyttää siltä, kuin kirjoitusoperaatio olisi viimeistelty tiedostoon. Lukuvälimuistiin on tällöin tietysti merkittävä, että tätä nimenomaista puskuroitua tiedoston kohtaa ei saa milloinkaan vapauttaa muistista. Mikäli näin ei tehtäisi, saattaisi korjattu kohta vapautua muistista, ja kun sitä tarvittaisiin uudestaan, luettaisiin sen tila tiedostosta, missä sitä ei välttämättä ole saatu loppuun asti kirjoitetuksi.

4.2.2 Orpojen solmujen poistaminen

Orvot solmut poistetaan vain kun tiedosto on avattu myös kirjoittamista varten. On huomionarvoista, että orpojen solmujen lippu ei kerro mitkä solmut ovat orpoja. Tämän vuoksi ne on etsittävä käymällä kaikki metadatat läpi ja tarkastamalla niiden viittauslaskurit. Niistä metadatoista, joiden viittauslaskurit ovat nollia, lisätään tiiviste *poistettavien solmujen listaan*.

Poistettavien solmujen lista on olemassa vain orpojen solmujen poistamisen ajan. Sitä käydään läpi valitsemalla yksi sen solmuista, joka tullaan poistamaan seuraavaksi. Ensin katsotaan mihin solmuihin poistettava solmu viittaa. Näiden viittauslaskuria vähennetään yhdellä. Mikäli jokin näistä menee nollaksi, myös sen tiiviste listään poistettavien solmujen listaan. Näin on huolehdittu lapsisolmuista ja itse solmu voidaan poistaa. Solmun poisto tapahtuu poistamalla sen metadata ja datayksikkö. Tätä jatketaan kunnes poistettavien solmujen lista on tyhjä.

4.2.3 Muutosten toteuttaminen rakennepuuhun

Rakennepuun solmuja ei koskaan muokata. Tämä johtuu ensinnäkin siitä, että samaan solmuun voidaan deduplikoinnin vuoksi viitata useasta kohtaa rakennepuuta. On epätodennäköistä, että sama muutos haluttaisiin tehdä kaikkiin näihin ilmentymiin. Toinen syy on, että solmun muokkaaminen muuttaisi sen tiivistettä. Täten myös siihen viittaavia solmuja pitäisi muokata, eikä näiden viittaavien solmujen joukkoa tiedetä.

Muokkaamisen sijaan rakennepuusta luodaan kopio, jossa muokattavat solmut ovat erilaisia kuin alkuperäisessä puussa. Solmujen erilaisuus vaikuttaa jälleen kerran uuden rakennepuun juureen asti tiivisteiden muuttumisen myötä. Kun uusi rakennepuu on saatu valmiiksi, korvaa se vanhan puun. Tämä tehdään vaihtamalla paketin juurisolmu uuden rakennepuun juureksi.

Deduplikoinnin ansiosta uusi rakennepuu vie tilaa vain sen verran, kuinka paljon siinä on uusia ja muuttuneita solmuja. Näitä ovat kaikki solmut juuresta muokattavaan solmuun asti sekä mahdolliset uudet solmut, kuten ulkopuolelta pakettiin kopioitu hakemistorakenne. Muut solmut löytyvät jo vanhasta rakennepuusta, joten ne eivät aiheuta tilanlisäystä.

Rakennepuun vaihdoksen myötä vanhan rakennepuun juuren viittauslaskuri menee nollaan, joten se ja mahdolliset muut orvoiksi jääneet solmut on poistettava. Tämä ja muut mainitut asiat käydään yksityiskohtaisemmin läpi seuraavassa esimerkissä. Siinä tiettyyn kohtaan paketin hakemistorakennetta kopioidaan hakemistorakenne paketin ulkopuolelta:

1. Aluksi etsitään paketin sisällä olevan kohdehakemiston solmu kulkemalla siihen juurisolmusta. Kaikki tällä polulla kohdatut solmut tulevat muuttumaan, joten ne luetaan tiedostosta Folder-olioiksi muistiin.

2. Hakemistorakenteen kopioiminen paketin ulkopuolelta tapahtuu luomalla sille oma, aluksi muusta rakennepuusta irtonainen puu. Mikäli tätä hakemistorakennetta ei löydy jo rakennepuun sisältä, tulee sen juureen olemaan nolla viittausta. Tämän vuoksi mahdollisten orpojen solmujen lippu on kytkettävä päälle, jotta mahdollisen keskeytyksen sattuessa tiedetään korjata tilanne. Myös muut osat tässä irtonaisessa rakennepuussa saattavat olla hetkittäin orpoja, sillä sen muodostaminen aloitetaan lehtisolmuista.

3. Muunnetaan pakettiin kopioitava hakemistorakenne omaksi rakennepuukseen käymällä se rekursiivisesti läpi seuraavasti:

   (a) Tiedostot ja symboliset linkit muunnetaan tiedosto-, datalohko- ja symbolinen linkki -solmuiksi kohdatessa.

   (b) Hakemistot muunnetaan hakemistosolmuiksi vasta, kun niiden kaikki lapset on saatu luoduiksi. Vasta tällöin hakemistosolmu tietää kaikkien lastensa tiivisteet.

4. Nyt luodaan kohdehakemistosolmusta kopio, joka on muuten samanlainen kuin alkuperäinen, mutta se sisältää viittauksen äsken luotuun uuteen rakennepuuhun, joka sisältää pakettiin kopioidun hakemistorakenteen. Kopion luomiseen käytetään viimeisintä Folder-oliota, joka luettiin aikaisemmin muistiin.

5. Kohdehakemistosolmun kopion ominaisuudet ovat nyt erilaiset kuin alkuperäisen kohdehakemistosolmun. Sen tiiviste on siis laskettava uudelleen.

6. Tehdään alkuperäisen kohdehakemistosolmun vanhemmasta kopio, jossa alkuperäisen kohdehakemistosolmun viittauksen kohdalla on viittaus tähän uuteen solmuun. Kopioon vaadittava Folder-olio löytyy jo muistista valmiiksi. Tätä toistetaan juureen asti.

7. Nyt paketti sisältää kaksi rakennepuuta. Paketin juurisolmuksi asetetaan uuden puun juurisolmu. Paketin nykyinen tila on nähtävissä kuvassa 4.1.

8. Nyt vanhan rakennepuun juurisolmuun ei ole yhtään viittausta ja on aika poistaa se. Tämä tehdään muuten samalla tavalla kuin orpoja solmuja poistettaessa, paitsi että nyt ei tarvitse selvittää mitkä solmut kuuluvat orpojen solmujen listaan. Sinne kuuluu aluksi vain vanha juurisolmu.

9. Nyt orvoksi käynyt vanhan rakennepuun juuri sekä sen sisältämät muut orvot solmut on poistettu. Orpojen solmujen lippu voidaan asettaa siihen tilaan, missä se oli ennen muokkauksen aloittamista.

Tällä tavalla pakettiin on myös mahdollista luoda atomisuus. Muutokset tapahtuvat vasta sillä hetkellä, kun uuden rakennepuun juuri vaihdetaan. Kirjanpito tekee juuren vaihtamisesta atomisen operaation, joten siten kokonainen suuri muokkaus voi olla atominen operaatio.

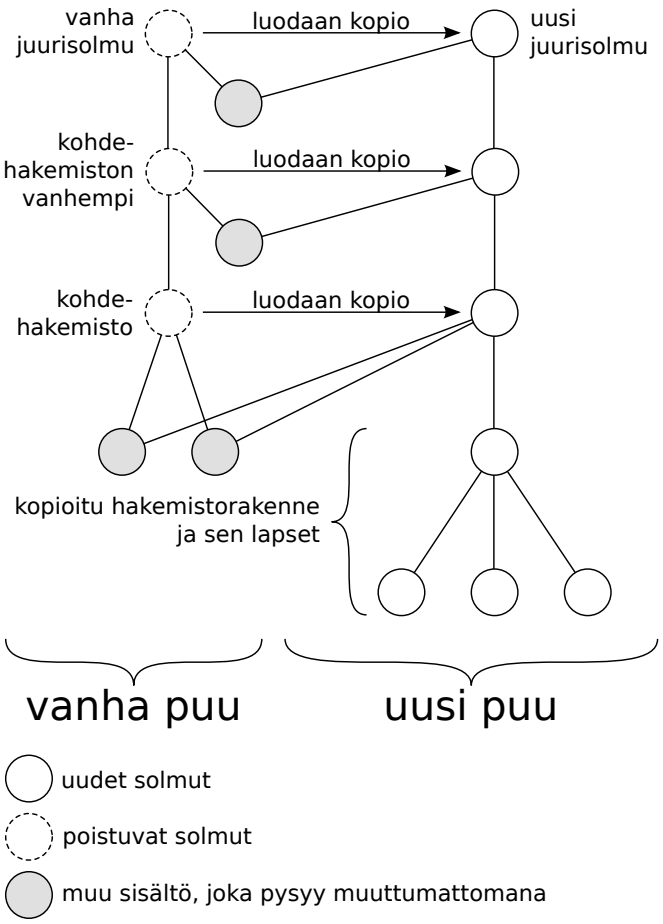
On mahdollista, että ohjelman suoritus katkeaa kun orvoksi jääneitä solmuja ollaan poistamassa. Tämä ei kuitenkaan riko atomisuuden periaatetta. Paketti on edelleen täysin käyttökelpoinen. Orpojen solmujen siivoamista osataan jatkaa seuraavalla käyttökerralla, sillä orpojen solmujen lippu kertoo, että näin pitää tehdä.

4.2.4 Hakupuun käsittely

Metadatan poistaminen hakupuusta

Metadata-tietuetta esiteltäessä mainittiin, että poistamista varten tarvitaan tieto vanhemmasta. Sitä tarvitaan ensinnäkin siihen, että vanhemmalle voidaan kertoa sen lapsen tulleen poistetuksi.

Mikäli poistettavalla metadatalla on hakupuussa lapsia, täytyy niille löytää uusi vanhempi hakupuusta. Tämä on helppoa, mikäli toinen lapsihaara viittaa tyhjään. Silloin voidaan olemassa olevalle lapselle merkitä poistettavan metadatan vanhempi uudeksi vanhemmaksi. Poistetun metadatan lapsen voi siis hakupuun näkökulmasta nähdä korvaavan poistetun metadatan, koska se ottaa sen paikan vanhempansa lapsena. Sen vuoksi sitä kutsutaan korvaajaksi.

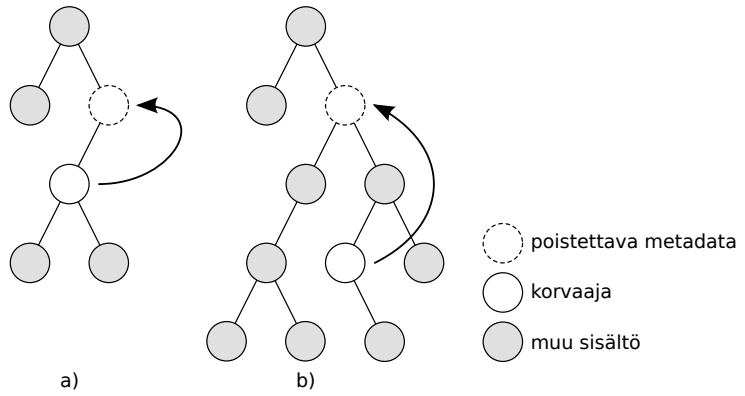


Kuva 4.1: Esimerkki muokkaamisesta. Pakettiin kopioidaan hakemistorakenne paketin ulkopuolelta. Kuvassa näkyvät sekä vanha että valmiiksi saatu uusi rakennepuu.

Jos poistettavalla metadatalla on molemmat lapset, on tilanne hankalampi. Tällöin toisen lapsen voisi kyllä valita korvaajaksi, mutta toinen lapsi jäisi ilman vanhempaa.

Tämän vuoksi korvaaja etsitään syvemmältä poistettavan metadatan jälkeläisten joukosta. Korvaajaksi valitaan hakupuun suuruusjärjestystä noudattaen poistettavaa metadataa seuraava metadata, eli suuremman lapsen muodostamasta alipuusta järjestysarvoltaan pienin mahdollinen metadata. Tämä voi siis olla myös suurempi lapsi itse, mikäli se on oman alipuunsa järjestysarvoltaan pienin metadata. Tämä algoritmi on kuvattu tarkemmin kirjassa Introduction to Algorithms [4, s. 295–298]. Esimerkit molemmista tilanteista, joissa metadatalla on lapsia, on esitetty kuvassa 4.2.

Näiden lisäksi on myös varmistuttava, ettei ehto metadatavektorin tiiviydestä rikkoudu. Poisto aiheuttaa tyhjän kolon, joten se on tukittava. Tämä tehdään siirtämällä vektorin viimeinen metadata kolon tilalle. Samalla solmujen, ja siten myös metadatojen määrää pienennetään yhdellä.



Kuva 4.2: Esimerkki metadatan poistamisesta hakupuusta. Kohdassa a) metadatalla on vain yksi lapsi. Kohdassa b) metadatalla on molemmat lapset.

Metadatojen väheneminen tarkoittaa myös koko metadata-alueen kutistumista, mikä vaikuttaa heti sen jälkeen alkavaan data-alueesen. Tämä käsitellään myöhemmin tutustuttaessa data-alueen käsittelyn haastaviin tilanteisiin.

Viimeisen metadatan siirtyminen tukittavan kolon kohdalle on toinen tilanne, missä tarvitaan tietoa metadatan vanhemmasta hakupuussa. Näin kyetään löytämään siirrettävän metadatan vanhempi ja merkitsemään sen lapsen muuttunut järjestysluku metadatavektorissa. Toki myös siirretyn metadatan lapsien tiedot omasta vanhemmastaan, eli siirretystä metadatasta, pitää päivittää.

Uuden metadatan lisääminen

Uuden metadatan lisääminen on helpompaa kuin poistaminen. Tällöin solmujen määrää ja siten metadatavektorin pituutta kasvatetaan yhdellä. Tätä ennen on kuitenkin varmistettava, että kasvava metadata-alue ei ylikirjoita mitään oleellista data-alueelta. Data-alueen varautuminen tähän käsitellään myöhemmin.

Kun data-alue on varmistettu, voidaan uusi metadata lisätä metadata-alueen perään. Tämän jälkeen aletaan käymään hakupuuta sen juuresta asti läpi ja etsitään oikea kohta uudelle metadatalle sen tiivisteen mukaan. Kun oikea kohta löytyy, merkitään hakupuu viittaamaan siitä kohdasta uuteen metadataan. Vastaavasti uuden metadatan vanhemmaksi merkitään viittaava metadata hakupuussa. Lisäys on nyt valmis.

Tasapainotus

Hakupuu ei nykyisessä toteutuksessa sisällä minkäänlaista tasapainotusta. Tasapainotukseen olisi voinut käyttää esimerkiksi punamustaa puuta [4, s. 308]. Tasapainotus jätettiin pois pääasiassa sen vuoksi, että työ olisi pysynyt yksinkertaisempana.

Toinen syy miksi tasapainotusta ei tehty, on epävarmuus siitä millaista algoritmia

kannattaisi käyttää. Esimerkiksi punamusta puu sopii jatkuva-aikaiseen tasapainotukseen, mutta tämän työn hakupuuta ei välttämättä tarvitsisi tasapainottaa ikinä.

Syynä on tiivisteiden käyttö hakupuun avaimina. Binäärihakupuiden kanssa toimiessa saattaa vastaan tulla tilanteita, joissa avaimia lisätään aivan arvoalueen ääripäästä. Eräs esimerkki on avaimen suhteen järjestyksessä olevan aineiston lisääminen binäärihakupuuhun. Tällainen aiheuttaa valtavasti lisätasoja, joita hakupuuta selatessa pitää käydä läpi. Tiivisteiden voidaan kuitenkin olettaa olevan tasaisesti jakautuneita. Vaikka juureen tulisikin yksi avain arvoalueen ääripäästä, aiheuttaisi tämä vain yhden lisätason ja seuraava saattaisikin olla jo keskeltä arvoaluetta.

4.2.5 Data-alueen käsittelyn perusoperaatiot

Tässä sekä seuraavassa osassa serialisoitua dataa sisältäviä datayksiköitä kutsutaan yksinkertaistamisen vuoksi pelkiksi datayksiköiksi ja käyttämättömiä tavuja sisältäviä täytedatayksiköiksi.

Koska rakennepuun solmuja ei muokata, ei myöskään niiden käyttämiä datayksiköitä muokata. Datayksiköitä täytyy kuitenkin lisätä ja poistaa. Tähän perehdytään seuraavaksi.

Datayksiköiden poistaminen

Solmun poistuessa sen käyttämän datayksikön hylkääminen on helppoa. Tähän ei tarvita muuta kuin täytedatayksikkö, joka kirjoitetaan poistettavan datayksikön kohdalle. Mikäli poistettava datayksikkö on data-alueen viimeinen tai sen jälkeen ei ole muuta kuin täytedatayksiköitä, jätetään uusi täytedatayksikkö kirjoittamatta, ja sen sijaan siirretään data-alueen loppu siihen kohtaan, mistä poistettava datayksikkö alkoi. Tässä tapauksessa tiedosto tulee siis lyhenemään.

Solmujen jälkeensä jättämää tilaa kutsutaan *tyhjäksi tilaksi*. Se on epätoivottua, sillä se pidentää tiedostoa. Sen poistaminen on kuitenkin haastavaa, kuten tullaan myöhemmin havaitsemaan.

Peräkkäin olevat täytedatayksiköt pyritään aina muuntamaan yhdeksi täytedatayksiköksi, jotta datayksiköiden läpikäyminen ei vaatisi turhia siirtymisiä datayksiköistä seuraaviin. Datayksikköä poistettaessa sitä ennen olevasta mahdollisesta täytedatayksiköstä ei kuitenkaan tiedetä mitään, joten poistossa tapahtuva täytedatayksikköjen yhdistäminen toimii vain kohti tiedoston loppua.

Lisäksi datayksiköiden pituutta kuvaava kokonaisluku on 29-bittisenä suhteellisen lyhyt. Se sallii vain noin puolen gigatavun mittaiset datayksiköt. Myös tämä rajoittaa täytedatayksiköiden yhdistelyä.

Datayksiköiden lisääminen

Uuden datayksikön luominen on haastavampaa. Sen voisi luoda helposti data-alueen perään, mutta parempi olisi jos se ei pidentäisi tiedostoa, vaan korvaisi data-alueella esiintyvää tyhjää tilaa.

Sopivan mittaisen tyhjän tilan löytäminen on vaikeaa. Paras olisi, jos löytyisi täsmälleen saman mittainen pätkä tyhjää tilaa, minkä uusi datayksikkö vaatii. Tällaisia ei kuitenkaan aina ole, ja vaikka olisikin, on niiden etsintä hidasta. Sen vuoksi data-aluetta käydään läpi datayksikkö kerrallaan ja tyydytään ensimmäiseen tyhjään tilaan, johon uusi datayksikkö mahtuu.

Sopivan tilan etsiminen aloitetaan satunnaisesta kohdasta. Satunnaista kohtaa ei voi valita suoraan data-alueelta, sillä data-alueen selaamisen voi aloittaa vain alusta tai jostain olemassa olevasta datayksiköstä. Sen sijaan valitaan satunnaisesti yksi metadatoista ja aloitetaan etsintä sen datayksiköstä.

Kooltaan pienemmät tyhjät tilat eivät tietystikään kelpaa. Myös kooltaan suuremmat tyhjät tilat saattavat olla ongelmallisia seuraavasta syystä: Jos uuden datayksikön jälkeen jää tyhjää tilaa, täytyy se merkitä täytedatayksiköksi. Mikäli tämä tila on kuitenkin pienempi kuin datayksikön otsake eli neljä tavua, ei täytedatayksikkö mahdu siihen. Tästä syystä sopivia tyhjiä tiloja, jotka uusi datayksikkö voisi täyttää, ovat joko täsmälleen sen mittaiset tyhjät tilat tai sellaiset, jotka ovat neljä tai enemmän tavuja sitä pidempiä. Kun tällainen tila on löytynyt, lisätään uusi datayksikkö tilan alkuun ja tarvittaessa heti sen perään täytedatayksikkö, jotta datayksiköiden vaatimus peräkkäisyydestä säilyisi. Täytedatayksikköä ei siis lisätä silloin, mikäli löydetty tyhjä tila sattui olemaan tarkalleen uuden datayksikön pituinen.

Mitä vähemmän data-alueella on tyhjää tilaa, sitä kauemmin uuden tyhjän tilan etsiminen kestää. Sen vuoksi tyhjän tilan etsimiselle on asetettu raja. Mikäli datayksiköitä käydään läpi yli sata kappaletta ilman että yhtään sopivan mittaista tyhjää tilaa tulee vastaan, luodaan uusi datayksikkö data-alueen loppuun pidentäen samalla data-aluetta.

4.2.6 Haastavat tilanteet data-alueen käsittelyssä

Lisäämisen ja poistamisen lisäksi on muutamia muita haastavia tilanteita. Näitä ovat esimerkiksi rakennepuun solmujen määrän muutoksista aiheutuva metadata-alueen kutistuminen ja pidentyminen. Koska data-alue alkaa välittömästi metadata-alueen jälkeen, aiheuttaa tämä data-alueen alkukohdan siirtymistä molempiin suuntiin. Metadatoja lisätään tai poistetaan aina yksi kerrallaan.

Lisäksi solmujen poiston yhteydessä syntyvä tyhjä tila kasvaa ajan kanssa, joten myös sille on tehtävä jotain. Käyttäjän suorittamien operaatioiden jälkeen varataan

aina hieman aikaa tyhjän tilan täyttämiselle. Tämä sekä kaksi edellistä haastavaa tilannetta kuvataan yksityiskohtaisemmin seuraavaksi.

Metadata-alueen kutistuminen

Tämä on yksinkertaisin tapaus. Yksi metadata poistuu, jonka seurauksena metadata-alue kutistuu. Tämä siirtää data-alueen alkua lähemmäs tiedoston alkua. Data-alue vaatii, että siellä pitää alusta asti olla peräkkäin datayksiköitä. Nyt alussa ei kuitenkaan ole datayksikköä vaan siinä aiemmin sijainneen metadatan tavuja. Tämä tilanne on helppo ratkaista luomalla data-alueen uuteen alkuun täytedatayksikkö. Sen pituus on otsake mukaan luettuna sama, mikä yhden metadatan pituus on.

Mikäli data-alueella oli jo valmiiksi tyhjää tilaa alussa, pidennetään nyt luotua täytedatayksikköä kattamaan myös tämä. Pidentämistä ei tehdä mikäli siitä seuraavan täytedatayksikön pituus ylittäisi datayksikön maksimipituuden.

Metadata-alueen pidentyminen

Tämä on vaikeampi tapaus. Jotta metadata-alue voisi kasvaa, pitäisi data-alueen alun siirtyä tarkalleen yhden metadatan pituuden verran kohti tiedoston loppua. Tähän vaaditaan, että data-alueen alussa on täytedatayksikkö, jonka pituus on otsake mukaan luettuna yhden metadatan pituus. Tätä ensimmäistä täytedatayksikköä voi sitten seurata toinen täytedatayksikkö tai varsinainen serialisoitua dataa sisältävä datayksikkö.

Toisin sanoen data-alueen alkuun tulisi saada tyhjää tilaa joko tarkalleen yhden metadatan pituuden verran tai vähintään yhden metadatan sekä datayksikön otsakkeen verran. Niitä alussa olevia datayksiköitä, jotka estävät tämän vaatimuksen, siirretään kauemmaksi alusta joko data-alueen loppuun tai sopiviin tyhjän tilan rakoihin. Siirtelyn aikana tilalle kirjoitetaan täytedatayksiköitä, jotta paketti pysyisi jatkuvasti eheänä. Siirtelyn vaatimien kohdesijaintien etsiminen tapahtuu pääasiassa samalla tavalla kuin uuden datayksikön lisääminen. Ainoa ero on, että sopivaksi sijainniksi ei tietenkään kelpaa sellainen kohta, joka estäisi alkuun vaaditun tyhjän tilan muodostumisen.

Datayksikköä siirrettäessä on päivitettävä myös siihen viittaavaa metadataa. Metadata löydetään laskemalla siirrettävän datayksikön tyypistä ja serialisoidusta datasta tiiviste ja hakemalla metadataa sen perusteella hakupuusta. Kun metadata on löytynyt, se päivitetään viittaamaan uuteen kohtaan data-alueella.

Kun data-alueen alkuun on saatu sopivasti tilaa ja tila on jaettu siten, että ensimmäisenä on metadatan pituinen täytedatayksikkö ja sitä seuraa joko toinen täytedatayksikkö tai datayksikkö, voidaan metadata-aluetta pidentää. Tällöin alun täytedatayksikkö ylikirjoittuu ja sitä seuraavasta datayksiköstä tulee data-alueen en-

simmäinen datayksikkö.

Tyhjän tilan täyttäminen

Tyhjän tilan täyttämisen tarkoituksena on päästä eroon täytedatayksiköistä joita esiintyy data-alueella poistojen seurauksena. Tämä tapahtuu siirtämällä datayksiköitä lopusta alkua kohti. Tällöin ne korvaavat alun täytedatayksiköitä kokonaan tai osittain ja luovat uusia täytedatayksiköitä loppuun. Aina kun data-alue loppuu täytedatayksikköön, voidaan tiedostoa lyhentää täytedatayksikön pituuden verran.

Koska sekä täytedatayksiköt että datayksiköt ovat vaihtelevan mittaisia ja koska toimitaan järjestelmässä, missä datavektorien siirrot ovat hitaita, on tämä ongelma hyvin vaikea ratkaista. Sen vuoksi myöhemmin esitettyä algoritmia ei oleteta täydelliseksi. Sillä pyritään lähinnä tuomaan pientä parannusta paketin koon pienentymiseen kun käyttäjä poistaa sen sisältä tiedostoja.

Mikäli datayksiköt olisi jaettu vakiokokoisiin lohkoihin, saattaisi tyhjän tilan täyttäminen olla helpompaa, mutta kuten kohdan 3.2.5 alakohdassa Datayksiköiden rakenne ja sijainti data-alueella todettiin, tämä on jätetty tiedostoformaatista pois yksinkertaistamisen vuoksi. Kuten algoritmin esittelyssä tullaan huomaamaan, on yksinkertaistamisessa epäonnistuttu sikäli, että yhden asian yksinkertaistaminen on johtanut toisen asian monimutkaistumiseen.

Optimaalisessa tapauksessa data-alueen viimeiset datayksiköt siirrettäisiin ensimmäiseen tyhjään tilaan joka data-alueella on siten, että ne korvaisivat sen kokonaan. Optimaaliseen tai edes osittain optimaaliseen tilanteeseen on hankala päästä seuraavista syistä:

1. Kaikista haastavinta on tyhjän tilan löytäminen. Parasta olisi jos tyhjät tilat saisi täytettyä alusta alkaen järjestyksessä. Ensimmäisen tyhjän tilan, kuten myös sitä seuraavien löytäminen on kuitenkin sikäli vaikeaa, että ei ole mitään takeita kauanko datayksiköitä on vielä selattava ennen kuin tyhjä tila löytyy. Jos esimerkiksi edellinen täyttäminen on tiivistänyt data-alueen alusta suuren osan, tulee tehtyä turhaa läpikäyntiä, joka varsinkin verkkopohjaisessa järjestelmässä saattaa olla erittäin hidasta.

2. Löydetty tyhjä tila saattaa olla sen kokoinen, etteivät lopussa olevat datayksiköt pysty täyttämään sitä kokonaan. Itseasiassa ei ole mitään takeita, että koko tiedostosta löytyy sellaista datayksikköjen joukkoa, joiden yhteispituus täyttäisi tyhjän tilan täydellisesti. Sopivien datayksiköiden löytäminen tyhjän tilan täytteeksi on todennäköisempää, mikäli muistiin ladataan suurempi määrä datayksikköjen tietoja. Tämä tosin pidentää datayksikköjen lukuaikaa ja kasvattaa muistinkäyttöä.

3. Myös täyttämiseen käytettävien datayksiköiden tietojen lukeminen aiheuttaa ongelman, sillä data-alueella ei voi suoraan siirtyä haluamaansa, esimerkiksi viimeiseen datayksikköön.

4. Koska tämä tehdään vasta muiden operaatioiden päätyttyä, ikäänkuin siivouksena, ei tähän voi varata kovin paljon suoritusaikaa.

Seuraavaksi esitellään algoritmi, joka täyttää tyhjää tilaa. Kuten aiemmin mainittiin, täydellistä ratkaisua ei odoteta. Sen sijaan ohjelma pyrkii sopivaan kompromissiin. Algoritmia kuvatessa oletetaan, että kyseessä on tiedosto, joka sisältää valtavan kokoisen rakennepuun. Pienen puun tapauksessa optimoinnit eivät ole tärkeitä.

Koska tämän operaation vaatiman ajan määrää ei tiedetä ja koska tämä on vain muiden operaatioiden jälkeistä vapaaehtoista viimeistelyä, on tähän varattu rajallinen määrä aikaa. Ajan loppuessa tyhjien tilojen täyttäminen keskeytetään. Täyttämiselle annetaan aikaa saman verran kuin kului varsinaisten operaatioiden suorittamiseen tai viisi minuuttia, mikäli varsinaisiin operaatioihin meni kauemmin.

Aluksi ohjelma valitsee satunnaisesti tuhat metadataa ja lukee näiden datayksikköjen alkukohdat ja pituudet muistiin. Näitä kutsutaan *alkudatayksiköiksi* ja niiden avulla on tarkoitus löytää muiden datayksiköiden tiedot eli aloituskohta ja pituus. Alkudatayksiköt järjestetään niiden aloituskohtien mukaan. Tämän jälkeen valitaan kauimpana alusta oleva alkudatayksikkö ja lähdetään selaamaan datayksiköitä kohti loppua. Vastaantulleiden datayksiköiden tiedot kerätään talteen tässä vaiheessa. Kun data-alueen loppu tulee vastaan, valitaan toiseksi kauimmainen alkudatayksikkö ja edetään siitä äsken läpikäydyn joukon alkuun. Tätä jatketaan kunnes 25 000 datayksikön raja saavutetaan tai kunnes kaikki alkudatayksiköt on käyty läpi. Näin on saatu kerättyä datayksiköiden joukko, joiden oletetaan sijaitsevan data-alueen loppupäässä. Datayksiköiden tiedot pidetään muistissa järjestettynä niiden pituuden mukaan.

Seuraavaksi on löydettävä sopivia tyhjiä tiloja, joihin loppupään datayksiköitä voi siirtää. Koska data-alueella saattaa olla pitkiä pätkiä ilman yhtään täytedatayksikköä, ei etsintää kannata tehdä kauaa samasta kohdasta. Sen sijaan haetaan useista eri kohdista, ja yhdestä kohtaa mennään korkeintaan 30 000 datayksikköä eteenpäin.

Koko tyhjien tilojen etsinnän ajan pidetään kirjaa kohdasta, jota ennen tiedetään olevan pelkkää tiivistä dataa eli ei yhtään täytedatayksiköitä. Aluksi tämä kohta on tietysti data-alueen alussa, mutta täyttöä tehdessä tämä kohta lähenee kohti loppua. Tätä kohtaa kutsutaan *tiiviin alueen lopuksi*.

Aloituskohdat haulle valitaan samaan tapaan kuin siirrettäviä datayksiköitä etsittäessä, mutta nyt suositaan datayksiköitä data-alueen alkupuolelta. Jälleen valitaan tuhat metadataa satunnaisesti ja poimitaan niiden datayksiköiden aloituskohdat. Näihin lisätään myös tiiviin alueen loppu ja pudotetaan kaikki sitä aikaisemmat

aloituskohdat pois. Näistä kohdista aletaan käydä datayksiköitä läpi tyhjiä tiloja etsien. Etsintä aloitetaan alkupäästä ja sitä käydään aina seuraavan aloituskohdan alkuun tai kunnes edellä mainittu 30 000 datayksikön raja tulee täyteen.

Tyhjän tilan löytyessä se yritetään täyttää täydellisesti. Aluksi tämä pyritään tekemään niillä loppupään datayksiköillä, joiden tiedot luettiin muistiin. Nämä tiedot on järjestetty datayksikköjen pituuksien mukaan, joten tästä joukosta on helppo kokeilla mahdollisimman monia eri kombinaatioita, jotta täydellinen täyttö löytyisi.

Näitä kokeillaan järjestyksessä suosien pisimpiä siten, että aluksi tilaan yritetään saada mahtumaan pisintä mahdollista sinne mahtuvaa datayksikköä. Tämän jälkeen jäljelle jäävään tilaan kokeillaan pisintä sinne mahtuvaa ja näin jatketaan kunnes tyhjä tila on liian pieni millekään datayksikölle tai kunnes täyttö menee tasan. Jos jäljelle jäänyt tyhjä tila käy liian pieneksi millekään jäljellä olevalle datayksikölle, peruutetaan rekursiossa ja kokeillaan pienempiä datayksiköitä aloittaen rekursion loppupäästä. Kuvassa 4.3 on esimerkki eräästä täytöstä.

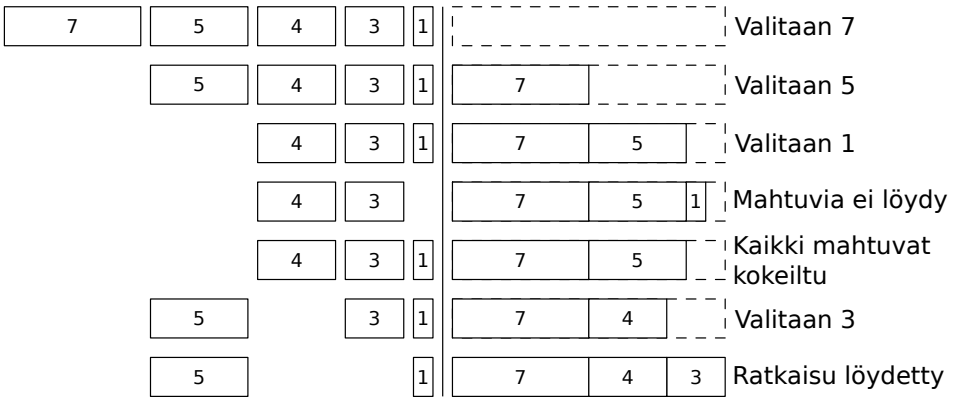
Sopiva kombinaatio tai tieto siitä, ettei sopivaa kombinaatiota ole, löydetään usein alle sadalla yrityksellä. Joskus harvoin tähän kuitenkin kuluu valtavia määriä yrityksiä. Sen vuoksi yrityksille on asetettu rajaksi 250. Datayksiköitä valitessa hylätään sellaiset, jotka aiheuttaisivat siirtoja alusta loppua kohti. Nämä poistetaan muistista lopullisesti, sillä kaikki tämän kierroksen tulevat siirrot tulevat olemaan pidemmällä alkua.

Mikäli sopivaa kombinaatiota ei löydy, joudutaan tyhjä tila täyttämään väkisin. Tällöin tyhjän tilan jälkeinen datayksikkö siirretään siihen kohtaan, mistä tyhjä tila alkaa. Se peittää tyhjän tilan joko kokonaan tai osittain mutta samaan aikaan muodostaa lisää tyhjää tilaa jälkeensä. Näin tyhjä tila siirtyy siirretyn datayksikön pituuden verran kohti loppua. Tätä jatketaan kunnes kohdataan uusi tyhjä tila, joka yhdistyy tähän siirtyvään tyhjään tilaan. Tällöin tyhjän tilan koko on vaihtunut ja datayksiköiden kombinaatiota voidaan taas kokeilla. Tämä väkinäinen toimintatapa on hyvin hidas, tosin sitä käytetään vain kun muutakaan ei ole tehtävissä.

Tätä prosessia jatketaan kunnes aikaraja tulee vastaan tai kunnes tiiviin alueen loppu saavuttaa koko data-alueen lopun. Mikäli kaikki tyhjien tilojen etsimiset ehditään käydä läpi tai mikäli käytettävissä olevat datayksiköt loppuvat kesken, aloitetaan alusta. Aikarajaa ei nollata, kuten ei myöskään tiiviin alueen loppua, mutta kaikki datayksiköiden tiedot ladataan uudestaan. Samaten arvotaan uudet aloituskohdat tyhjän tilan etsimiselle, lukuunottamatta tiiviin alueen loppua, jota ennen olevat etsinnät hylätään.

4.3 Ohjelman käyttö

Ohjelma ei ole graafinen, vaan sitä käytetään komentoriviltä. Tällä hetkellä ohjelmassa on mahdollisuus suorittaa seuraavia perustoimintoja:



Kuva 4.3: Esimerkki täyttämisestä. Vasemmalla ovat käytettävissä olevat datayksiköt ja oikealla täytettävä tila. Jokainen rivi kuvaa yhtä vaihetta ja niiden suoritus aloitetaan ylhäältä. Selkeyden vuoksi oletetaan, että yhden pituiset datayksiköt olisivat mahdollisia. Ensin havaitaan, että täyttö ei onnistu, koska sopivan mittaista datayksikköä ei löydy täyttämään jäljellä olevaa tyhjää tilaa. Sitten peruutetaan rekursiossa, kokeillaan viiden sijaan neljän pituista ja sen avulla ratkaisu lopulta löytyy.

1. Tiedostojen, hakemistojen ja symbolisten linkkien kopioiminen pakettiin.
2. Tiedostojen, hakemistojen ja symbolisten linkkien kopioiminen paketista.
3. Tiedoston, hakemiston tai symbolisen linkin poistaminen paketin sisältä.
4. Hakemiston lapsien listaaminen paketista.
5. Uuden hakemiston luominen pakettiin.

Näiden lisäksi on mahdollisuus käsitellä kohdassa 2.1.6 esiteltyjen snapshottien kaltaisia tietovarastoja. Koska paketti ei ole samaan tapaan jatkuvan muutoksen kohteena kuin tiedostojärjestelmä, ei paketista itsestään ole tarvetta ottaa snapshotteja. Tässä ohjelmassa snapshottien tarkoituksena on tarjota oikopolku tilanteeseen, jossa käyttäjä haluaisi päivittäin kopioida samat tiedostot tai hakemistot paketin sisälle. Snapshotit eivät tässä ohjelmassa ole teknisesti mitenkään erityisiä. Ne ovat tavallisia alihakemistoja paketin juuressa, eli niitä voi käsitellä myös ohjelman perustoiminnoilla. Snapshot-komennoilla snapshotteja voi luoda, poistaa ja niiden sisällön voi kopioida paketista.

Perus- ja snapshot-komentojen lisäksi ohjelmassa on joitain erityiskomentoja. Näitä ovat:

1. Sekalaisen informaation antaminen. Tämä on tarkoitettu ohjelman kehittäjän avuksi. Se listaa useita eri tietoja paketista, esimerkiksi juurisolmun tiivisteen, solmujen määrän, data-alueen pituuden, täytedatayksiköiden yhteispituuden, metadatat, datayksiköt ja koko tiedoston hakemistorakenteen puumaisena esityksenä.

2. Paketin tarkastaminen. Tämän avulla on mahdollista ajaa muutamia testejä, joiden avulla pyritään selvittämään onko paketti edelleen eheä.

3. Paketin korjaaminen. Sama kuin paketin tarkastaminen, mutta tämä pyrkii korjaamaan löytyneitä virheitä.

4. Paketin optimoiminen. Tämä suorittaa tyhjän tilan täyttämisen loppuun asti ilman aikarajoitusta. Tulevaisuudessa tämä voisi myös tasapainottaa hakupuun.

Ohjelmaan ei ole toteutettu ominaisuutta, joka havaitsisi käyttäjän haluavan keskeyttää ohjelman suorituksen. Toisaalta kirjanpidon ja atomisuuden vuoksi on täysin turvallista lopettaa suoritus tarvittaessa väkisin.

5. MITTAUSTEN KUVAUS

Oman ohjelman suorituskykyä verrattiin kolmeen tunnettuun ohjelmaan. Vertailussa mitattiin ajankäyttöä ja pakkaustehokkuutta eri aineistoilla.

Lisäksi ohjelma on ollut päivittäisessä käytössä johdannossa esitetyssä omassa varmuuskopiointitapauksessa. Mittausten yhteydessä kuvataan kokemukset myös tästä.

5.1 Ajankäytön ja pakkaustehokkuuden mittaus

Ajankäytön ja pakkaustehokkuuden mittaukseen valittiin oman ohjelman lisäksi zip, tar.gz ja 7-Zip. Zip ja tar.gz valittiin mukaan siksi, että ne ovat hyvin yleisesti käytössä. 7-Zip valittiin mukaan pääasiassa sen saamien kehujen vuoksi, mutta myös siksi että sekin on melko hyvin tunnettu.

7-Zipissä olisi hieman soveltaen ollut mahdollista käyttää differentiaalisen varmuuskopioinnin toimintoa mikä olisi helpottanut muutamien aineistojen pakkaamista. Tämä olisi kuitenkin luonut useita paketittitiedostoja mitä nimenomaan pyrittiin välttämään johdannossa esitetyssä omassa varmuuskopiointitapauksessa. Tämän sekä sen käytön hankaluuden vuoksi differentiaalista tapaa ei käytetty.

Aineistojen tavukokoja ilmoitettaessa käytetään *kibi-*, *mebi-* ja *gibitavuja*. Näiden yksiköiden kerrannaisyksiköt ovat binäärisiä, eli 1024-kantaisia tavanomaisten 1000-kantaisten sijaan. Lyhenteet näille yksiköille ovat KiB, MiB ja GiB.

5.1.1 Aineisto

Aineistoja oli viisi erilaista. Aineistoiksi on valittu tyypillisiä käyttötapauksia, pakkausalgoritmeille vaikeita tapauksia sekä johdannossa esitetty varmuuskopiointitapaus. Aineistojen numeeriset tiedot on listattu taulukossa 5.1 ja aineistot kuvataan sanallisesti seuraavaksi.

1. Valokuvia JPEG-formaatissa. Jonkin tapahtuman valokuvien kokoaminen yhdeksi paketiksi on tyypillinen käyttötapaus. Kuvat ovat usein valmiiksi pakattuja, joten pakkausalgoritmien on hankala saada niitä pienempään tilaan.

2. Linux-ytimen erään version lähdekoodi. Tämä on melko tyypillinen avoimen

Taulukko 5.1: Aineistojen numeeriset tiedot.

	Valokuvat	Lähdekoodi	Video	Lähdekoodin eri versiota	Varmuuskopiot
Tiedostoja	169	42424	1	164583	940506
Uniikkeja tiedostoja	169	42093	1	71233	46283
Duplikaatteja tiedostoja (%)	0	0,78	0	57	95
Koko (GiB)	0,17	0,55	0,69	2,2	360
Tiedostojen keskimääräinen koko (KiB)	1090	13,6	720000	13,7	404

lähdekoodin levityspaketti. Toisaalta multimedian, eli grafiikan, videon ja äänen määrä on tässä ohjelmassa mitätön. Tiedostot ovat pääasiassa tekstitiedostoja, joten pakkausalgoritmien tulisi selvitä niistä helposti.

3. Yksi videotiedosto. Videotiedostojen pakkaaminen on epätavallista, sillä ne ovat jo valmiiksi pakatussa muodossa. Tämä aineisto on mukana lähinnä sen vuoksi, että se on pakkausalgoritmeille hankala.

4. Linux-ytimen lähdekoodin eri versioita. Tämä aineisto on muuten samankaltainen kuin edellä esitelty yhden version lähdekoodi, mutta duplikaattien tiedostojen määrä on suurempi. Tämän tulisi helpottaa deduplikoivaa ohjelmaa.

5. Varmuuskopioita tärkeistä tiedostoista 12 eri ajanhetkeltä. Tämä aineisto on peräisin johdannossa esitetystä omasta varmuuskopiointitapauksesta. Sen tiedostot ovat pääasiassa valokuvia, asiakirjoja, ohjelmistoprojekteja, 3d-mallinnuksia ja pelipalvelinten tiedostoja. Vaikka varmuuskopiot ovat eri ajanhetkiltä, on valtaosa tiedostoista täysin samoja. Esimerkiksi valokuvia ei tallentamisen jälkeen ole muokattu ikinä, joten kaikista varmuuskopioista löytyvät lähes samat valokuvat. Ohjelmistoprojektit lisäävät duplikaattien tiedostojen määrää entisestään, sillä ne ovat samalta tekijältä ja siten sisältävät paljon samoja tiedostoja, jotka on kopioitu ohjelmistoprojektista toiseen. Deduplikointia tukevan ohjelman pitäisi kyetä pakkaamaan tämä aineisto erittäin pieneen tilaan.

5.1.2 Mittausten suoritus

Mittauksia suoritettiin yksi aineisto kerrallaan. Tällä pyrittiin siihen, että sama aineisto pysyisi käyttöjärjestelmän levyvälimuistissa niin hyvin kuin vain on vapaan

muistin puolesta mahdollista. Tämä mahdollistaa mittausten suorittamisen nopeammin.

Aineisto käytiin läpi kolmena kierroksena ja jokaisen kierroksen aikana jokainen neljästä ohjelmasta pakkasi aineiston vuorotellen. Jotta aineisto olisi löytynyt levyvälimuistista myös ensimmäisen kierroksen ensimmäisen ohjelmiston tapauksessa, ajettiin sitä ennen ohjelma, joka luki koko aineiston kerran läpi. Jokaisella ajokierroksella mitattiin ohjelman käyttämä aika sekä ensimmäisellä kierroksella myös aineiston koko pakatussa muodossa.

Mittauksia tekevässä tietokoneessa oli 7,8 gibitavua keskusmuistia, josta vähintään puolet oli suurimman osan ajasta vapaana levyvälimuistia varten. Täten kaikki aineistot viimeistä lukuunottamatta mahtuivat helposti levyvälimuistiin.

Tietokoneessa oli kaksiytiminen prosessori ja sen käyttöjärjestelmänä oli Linux-pohjainen Ubuntu 12.04. Käytetyt ohjelmat olivat tässä työssä toteutettua ohjelmaa lukuunottamatta samoja, jotka tulevat käyttöjärjestelmän mukana. Koska eri asetusten kokeileminen olisi pidentänyt mittaustuloksia valtavasti, ajettiin ohjelmia perusasetuksilla, ja mikäli mahdollista, ilman ohjelman ulostuloa terminaaliin.

Tässä työssä toteutetun ohjelman perusasetukset tarkoittavat sitä, että ohjelmassa käytettyä zlib-kirjastoa ajetaan kirjaston vakiopakkaustehokkuudella, ohjelman sisäisen lukuvälimuistin koko on 64 mebitavua ja kirjoitusvälimuistin 320 kibitavua.

Aineistot sijaitsivat eri kovalevyllä kuin itse käyttöjärjestelmä. Kaikki paitsi viimeinen aineisto onnistuttiin mittaamaan yhden yön aikana. Yöllä tietokoneen muu käyttö oli hyvin vähäistä, joten mittaus sai toimia häiriöttä. Viimeisen aineiston läpikäynnissä kesti kuitenkin useita vuorokausia ja tällöin käytettiin sekä kovalevyä että järjestelmää satunnaisesti. Tämän ei kuitenkaan oleteta vaikuttavan mittaustuloksiin merkittävästi, sillä satunnainen käyttö ei ollut kovin kuormittavaa.

5.2 Ohjelman päivittäinen käyttö

Ohjelmaa on käytetty säännöllisesti öisin varmuuskopioimaan tärkeitä tiedostoja kahdelta tietokoneelta johdannossa esitetyn oman tarpeen mukaisesti. Molemmissa tapauksissa käyttö on tapahtunut SSHFS-verkkotiedostojärjestelmää apuna käyttäen [22]. Tämä tiedostojärjestelmä käyttää SFTP-protokollaa. Molemmissa tapauksissa on myös ilmennyt satunnaisia verkkoyhteyden katkeamisia. Käyttökokemukset ovat reilun puolen vuoden ajalta.

Ensimmäinen tietokone sijaitsi ensin Tampereella ja myöhemmin Porissa. Siitä on tehty varmuuskopioimista sekä Tampereelle että Saksaan. Yksi päivittäinen varmuuskopiointikerta käsitti Tampereelle vajaat 30 gibitavua ja Saksaan noin seitsemän gibitavua, tosin deduplikoinnin vuoksi vain pieni murto-osa tästä siirtyi verkon ylitse. Tampereelle varmuuskopioiminen käytti salausta.

Toinen tietokone sijaitsi Saksassa. Siitä on tehty varmuuskopioita ensin Tampe-

reelle ja myöhemmin Poriin. Päivittäinen varmuuskopiointi tältä koneelta vei noin kaksi gibitavua. Luonnollisesti myös tässä tapauksessa deduplikointi vähensi verkkoliikenteen minimaaliseksi.

Molemmat koneista huolehtivat omasta varmuuskopioinnistaan joka päivä. Aamuyön tunteina ne toimivat seuraavasti:

1. Ensin liitettiin etäkoneen hakemisto SSHFS-verkkotiedostojärjestelmää käyttäen. Tällä tavoin näytti siltä, kuin etäkoneella oleva pakettitiedosto olisi sijainnut varmuuskopioivassa tietokoneessa itsessään.

2. Seuraavaksi paketin hakemistorakenteen juureen tehtiin sen hetkisen päivämäärän mukaan nimetty hakemisto ja sinne kopioitiin kaikki tärkeät tiedot.

3. Lopuksi SSHFS-verkkotiedostojärjestelmän liitos purettiin.

On olennaista, että ohjelmaa ajettiin nimenomaan sillä tietokoneella, missä varmuuskopioitavat tiedostot sijaitsivat. Tämä johtuu siitä, että tehdessään deduplikointia ohjelma lukee varmuuskopioitavat tiedostot täydellisesti kun taas paketissa jo olevista tiedostoista tarvitsee lukea vain muutamia tiivisteitä.

Näin ollen raskaat lukuoperaatiot kohdistuvat paikalliseen levyyn ja kevyet lukuoperaatiot verkkolevyyn. Mikäli ohjelmaa olisi ajettu sillä tietokoneella missä paketti sijaitsee, olisivat raskaat lukuoperaatiot keskittyneet verkkolevyyn ja kevyet lukuoperaatiot paikalliseen levyyn.

6. TULOKSET JA NIIDEN TARKASTELU

Seuraavaksi luetellaan ajankäytön ja pakkaustehokkuuden mittaustulokset sekä selvitetään niiden syitä. Luvun lopussa myös kuvataan kokemukset ohjelman päivittäisestä käytöstä.

6.1 Ajankäyttö

Ajankäytön mittaustulokset on listattu taulukossa 6.1. Zip ja tar.gz olivat nopeuksiltaan melko identtisiä ja lähes kaikissa tilanteissa nopeimpia. Vastaavasti 7-Zip ja oma ohjelma olivat hitaampia, todennäköisesti monimutkaisuutensa vuoksi.

Näistä kahdesta oma ohjelma oli muulloin 7-Zippiä nopeampi paitsi Linux-ytimen lähdekoodien tapauksessa. Syy tähän on todennäköisesti siinä, että nämä aineistot sisälsivät suuret määrät pieniä tiedostoja, jolloin oma ohjelma joutui tekemään valtavat määrät hakupuuoperaatioita tarkistaessaan löytyvätkö tiedostot jo paketista.

Varmuuskopioaineisto noudattaa melko samaa kaavaa muiden ohjelmien tapauksessa, mutta oma ohjelma selviytyi tästä tilanteesta erittäin hyvin. Huolimatta sen monimutkaisista toimintatavoista, onnistui se olemaan nopein. Tämä tapaus osoittaa deduplikoinnin hyödyt. Siinä missä deduplikoimaton ohjelma joutuu lukemaan tiedoston, pakkaamaan sen ja kirjoittamaan pakettiin, voi deduplikoiva ohjelma tehdä pelkän tiedoston lukemisen mikäli käsiteltävä tiedosto on jo aiemmin lisätty pakettiin.

6.2 Pakkaustehokkuus

Pakkaustehokkuuden mittaustulokset on listattu taulukossa 6.2.

Vaikeissa tapauksissa, eli valokuvissa ja videossa, saatiin identtisiä tuloksia. Näissä ohjelmat eivät saaneet pakattua materiaalia käytännössä ollenkaan.

Linux-ytimen lähdekoodissa 7-Zip suoriutui hyvin muiden ohjelmien jäädessä hieman jälkeen siitä. Oma ohjelma oli tässä huonoin, johtuen todennäköisesti suuresta määrästä pieniä tiedostoja. Tällöin pakettitiedostoon tulee suuri määrä erilaista lisätietoa varsinaisten tiedostojen sisältöjen lisäksi.

Useiden Linux-ytimen lähdekoodiversioiden tapauksessa oma ohjelma sai hieman etua deduplikoinnista, mutta 7-Zip suoriutui vielä paremmin. Tiedostoja pakatessa oli selvästi havaittavissa jo teoriaosiossa mainittu 7-Zipin kyky esikäsitellä pakattavaa dataa. Ohjelma ei käynyt aineistoa läpi perinteisesti hakemisto kerrallaan, vaan

Taulukko 6.1: Ajankäytön mittaustulokset sekunteina. Mittaustulokset on järjestelty käytetyn ajan mukaan.

Aineisto	Zip	7-Zip	tar.gz	Oma ohjelma
Valokuvat	11,3	64,3	10,7	20,5
	11,5	66,2	10,8	20,7
	11,6	66,9	11,3	21,0
Lähdekoodi	22,1	262	25,2	805
	22,2	262	25,2	840
	22,3	263	25,3	861
Video	46,1	262	43,4	92,9
	46,5	264	45,1	94,9
	48,0	265	45,2	94,9
Lähdekoodin eri versiota	90,0	502	97,6	3 390
	90,6	598	98,9	3 710
	93,2	643	99,9	4 090
Varmuuskopiot	36 800	116 000	33 200	20 500
	37 200	119 000	33 400	20 700
	37 300	131 000	34 000	21 200

se onnistui valitsemaan identtisiä tiedostoja hakemistorakenteesta ja pakkaamaan ne sarjassa. Tämä todennäköisesti tehosti pakkausta merkittävästi. Mikäli tässä aineistossa oltaisiin käytetty 7-Zipin differentiaalisen varmuuskopioinnin toimintoa, olisivat tulokset voineet olla vieläkin parempia.

Varmuuskopioissa zip ja tar.gz eivät osanneet hyödyntää duplikaatteja tiedostoja mitenkään. 7-Zip sen sijaan selviytyi varsin hyvin, vaikkakin kulutti hyvin paljon aikaa. Tässäkin tapauksessa oli havaittavissa, että ainakin alussa 7-Zip onnistui löytämään identtisiä tiedostoja ja pakkaamaan niitä sarjassa. Tässä aineistossa 7-Zipin differentiaalisen varmuuskopioinnin toiminto olisi varmasti johtanut vielä nykyistäkin parempiin tuloksiin. Oma ohjelma suoriutui varmuuskopioista erittäin hyvin, mikä oli deduplikoinnin vuoksi odotettavissa.

6.3 Ohjelman päivittäinen käyttö

Testauksen aikana kaikkiin paketteihin tuli virheitä. Jotkin virheistä olivat melko harmittomia ja osa oli vakavia.

Eräs melko harmiton virhe oli metadatoissa sijaitsevien viittauslaskurien kasvaminen liian suuriin arvoihin. Tällaisessa tilanteessa ei menetetä mitään tietoa, mutta tiettyjä solmuja ei saada ikinä poistetuksi sillä viittauslaskurit eivät voi mennä nollaan asti.

Esimerkki vakavan virheen tilanteesta oli sellainen, missä datayksikkö oli päässyt

Taulukko 6.2: Aineistojen koot pakatussa muodossa sekä suhde alkuperäisen aineiston kokoon. Yksiköt ovat mebitavuja ja prosentteja.

Aineisto	Zip	7-Zip	tar.gz	Oma ohjelma
Valokuvat	178,0 99	178,0 99	178,0 99	178,2 100
Lähdekoodi	131,8 23	74,47 13	106,8 19	137,4 24
Video	696,5 99	697,0 99	696,5 99	697,2 99
Lähdekoodin eri versiota	514,6 23	83,78 4,8	418,0 19	302,5 14
Varmuuskopiot	315 900 85	79 700 21	315 600 85	26 450 7,1

korruptoitumaan. Tällaisessa tilanteessa menetetään tietoa ja mahdollisesti suuretkin hakemistorakenteet paketin sisällä saattavat jäädä orvoiksi, mikäli kaikki niihin viittaavat hakemistosolmut ovat korruptoituneita.

Virheiden korjaaminen tiedostoista oli usein liian työlästä, joten paketit poistettiin ja varmuuskopiointi aloitettiin uudestaan. Tässä menetettiin historiallista tietoa, tosin sitä ei katsottu kovin tarpeelliseksi.

Virheet olivat todennäköisimmin peräisin katkenneesta verkkoyhteydestä, jonka aikana tiedostoon ei ole saatu kirjoitettua kaikkea vaadittua tietoa eikä kirjanpitokaan ole pelastanut tilannetta.

Ainakin yksi tätä oletusta tukeva ohjelmointivirhe löydettiin ja korjattiin. Virhe tuli mahdolliseksi eräässä kirjoitustilanteessa, kun kirjanpito oli tehty ja valmistauduttiin varsinaiseen tiedostoon kirjoittamiseen. Tässä kirjoitustilanteessa tiedostosta unohdettiin päivittää erästä laskuria, eli tiedostoon suunnitellut kirjoitukset eivät sisältäneetkään loogista tilaa, mitä vaadittiin kohdassa 3.2.6. Tätä kyseistä laskuria kyllä päivitettiin myöhemmin, mutta mikäli verkkoyhteys katkesi ennenkuin ohjelma ehti tähän kohtaan, jäi paketti virheelliseen tilaan. Vaikka kirjanpito olisikin tehnyt mahdolliseksi kirjoittaa kesken jääneet kirjoitukset loppuun, olisi mainittu laskuri edelleen väärässä tilassa.

Virheistä huolimatta esimerkiksi varmuuskopioinnit Saksasta selvisivät koko reilun puolivuotisen testijakson ajan. Niistä löytyi vain edellä mainittuja viitelaskurien virheellisiä kasvamisia.

Kohdassa 4.2.6 esitelty tyhjän tilan täyttäminen ei toiminut kovin hyvin. Algoritmi sai joitain täytedatayksiköitä siirrettyä data-alueen loppuun ja siten lyhennettyä tiedostoa, mutta tämä oli liian tehotonta jotta data-alueella oleva tyhjän tilan suhteellinen määrä olisi vähentynyt poistoja tehdessä.

7. PÄÄTELMÄT

Työn tuloksena syntyi puurakenteeseen perustuva pakkausohjelma ja -formaatti. Puuta käytetään kuvaamaan pakettitiedoston sisältämää hakemistorakennetta. Ohjelma on komentorivipohjainen ja tukee salausta sekä toipumista suorituksen yllättävästä keskeytymisestä.

Deduplikointi toteutetaan solmujen tasolla siten, että identtisiä solmuja ei todellisuudessa ole olemassa kuin yksi kappale. Koska solmuilla kuvataan esimerkiksi hakemistoja, voidaan tällä tavoin deduplikoida tiedostojen lisäksi myös hakemistoja eli myös hakemistorakenteita, olivatpa ne kuinka suuria tahansa.

Tiedostoformaatti koostuu neljästä osasta: otsakkeesta, metadata-alueesta, data-alueesta ja kirjanpidosta. Puun solmut on tallennettu osittain metadata- ja osittain data-alueelle.

7.1 Mittaustulokset ja käyttökokemus

Mittauksissa omaa ohjelmaa vertailtiin zip-, tar.gz- ja 7-Zip -ohjelmiin. Aineistoina olivat neljä erilaista tapausta jotka sisälsivät isoja ja pieniä tiedostoja. Viidentenä tapauksena oli johdannossa esitetty oma varmuuskopiointitapaus. Selvisi, että zip ja tar.gz ovat yleensä nopeimpia, mutta eivät onnistu pakkaamaan yhtä tehokkaasti. 7-Zip oli hitain lukuunottamatta tilanteita, joissa oli paljon pieniä tiedostoja. Näissä oma ohjelma oli selvästi hitain ja kärsi myös huonosta pakkaustehokkuudesta, ellei pakattavassa aineistossa ollut paljon identtisiä tiedostoja.

7-Zip oli tehokkain pakkaaja kaikissa tapauksissa lukuunottamatta johdannossa esitettyä varmuuskopiointitapausta. Siinä paketti sisältää suuren määrän identtisiä tiedostoja ja hakemistorakenteita. Oma ohjelma selviytyi tästä tapauksesta todella hyvin. Se oli sekä nopein että pakkasi selvästi tehokkaimmin. Toisaalta 7-Zipillä olisi hieman soveltaen voinut myös pakata differentiaalisella varmuuskopioinnilla. Tämä olisi kuitenkin ollut hankalaa ja tuottanut useita pakettitiedostoja yhden sijasta, joten sitä ei käytetty. Käytettäessä olisi kuitenkin ollut mahdollista, että 7-Zip olisi pakannut kaikista tehokkaimmin myös tässä tapauksessa.

Ohjelmaa käytettiin päivittäin reilu puoli vuotta verkkopohjaisen tiedostojärjestelmän kautta johdannossa esitettyyn varmuuskopiointitapaukseen. Tänä aikana paketteihin tuli lukuisia virheitä, tosin käytön aikana saatiin myös useita ohjelmointivirheitä korjatuksi. Vaikka paketit eivät menneet kovinkaan usein täysin käyttö-

kelvottomiksi, osoittaa tämä että parannettavaa olisi selkeästi. Todennäköisin syy virheille ovat keskenjääneet kirjoitusoperaatiot jotka ovat syntyneet verkkoyhteyden katketessa. Toipuminen yllättävästä keskeytymisestä ei siis toimi täydellisesti.

7.2 Jatkokehitys

Ohjelmassa on paljon jatkokehitettävää. Koska ohjelman on tarkoitus olla ilmainen ja avointa lähdekoodia, on teoreettinen mahdollisuus että ohjelman käyttäjäkunnalta voisi saada tässä apua, mikäli ohjelma tulee koskaan nauttimaan minkäänlaisesta suosiosta. Tärkeimpiä jatkokehitysehdotuksia on lueteltu alla.

1. Ohjelma vaatisi kattavaa testausta ja korjailua, jotta virheiden päätyminen pakettitiedostoon käytön aikana saataisiin estetyksi.

2. Ohjelmassa käytetyt salaustekniikat eivät ole minkään salausasiantuntijan tarkastamia, vaan ne on toteutettu lähinnä kevyen turvallisuuden saamiseksi. Todennäköisesti tällaisessa tarkastuksessa löytyisi paljonkin paranneltavaa.

3. Poistettaessa tiedostoja ja hakemistoja paketista, ei pakettitiedoston koko välttämättä pienene ollenkaan. Tämä johtuu siitä, että poistettujen kohtien luomia rakoja on vaikea saada siirrettyä tiedoston loppuun. Tämän varalle kehiteltiin toimintatapa, mutta se osoittautui melko heikosti toimivaksi. Sitä pitäisi siis kehittää tehokkaammaksi.

LÄHTEET

[1] Apple Inc.: *Mac 101: Time Machine*, Viitattu 19.10.2012, Saatavilla www-muodossa: http://support.apple.com/kb/HT1427

[2] Bonwick, Jeff: Jeff Bonwick's Blog, *ZFS Deduplication*, Viitattu 11.10.2012, Saatavilla www-muodossa: https://blogs.oracle.com/bonwick/entry/zfs_dedup

[3] Bosneaga, Constantin: A32.Blog, *Automated differential backup using 7zip for linux/windows*, Viitattu 1.7.2013, Saatavilla www-muodossa: http://a32.me/2010/08/7zip-differential-backup-linux-windows/

[4] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford: *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[5] *eXdupe – Data deduplication*, Viitattu 8.10.2012, Saatavilla www-muodossa: http://www.exdupe.com/

[6] *FlyBack – Apple's Time Machine for Linux*, Viitattu 1.10.2012, Saatavilla www-muodossa: http://code.google.com/p/flyback/

[7] Gailly, Jean-loup, and Adler, Mark: *The gzip home page*, Viitattu 1.7.2013, Saatavilla www-muodossa: http://www.gzip.org/

[8] Gailly, Jean-loup, and Adler, Mark: *zlib*, Viitattu 18.12.2012, Saatavilla www-muodossa: http://www.zlib.net/

[9] Haikala, Ilkka ja Järvinen, Hannu-Matti: *Käyttöjärjestelmät*, Toinen painos, Talentum, 2004.

[10] Huffman, David A.: A Method for the Construction of Minimum-Redundancy Codes, *Proceedings of the IRE*, 1098–1101, 1952, Viitattu 15.10.2012, Saatavilla www-muodossa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4051119

[11] Konecki, M., Kudelić, R., Lovrenčić, A.: Efficiency of lossless data compression, *MIPRO, Proceedings of the 34th International Convention*, 810–815, 2011, Viitattu 19.10.2012, Saatavilla www-muodossa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5967166

[12] Linux Kernel Organization, Inc.: *btrfs Wiki*, Viitattu 1.7.2013, Saatavilla www-muodossa: https://btrfs.wiki.kernel.org/

[13] Mahoney, Matt: *The ZPAQ Open Standard Format for Highly Compressed Data - Level 2*, Versio 2.02, Viitattu 1.7.2013, Saatavilla www-muodossa: http://mattmahoney.net/dc/zpaq202.pdf

[14] Meyer, Dutch T. and Bolosky, William J.: A study of practical deduplication, *ACM Transactions on Storage*, vol. 7, no. 4, Article 14, 2012, Viitattu 17.10.2012, Saatavilla www-muodossa: http://dl.acm.org/citation.cfm?id=2078864

[15] Oracle Corporation: *Oracle Solaris ZFS Administration Guide*, Viitattu 1.7.2013, Saatavilla www-muodossa: http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/

[16] Pavlov, Igor: *7z Format*, Viitattu 29.6.2013, Saatavilla www-muodossa: http://www.7-zip.org/7z.html

[17] PKWARE Inc.: *.ZIP File Format Specification*, versio 6.3.3, Viitattu 29.6.2013, Saatavilla www-muodossa: http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[18] Rarlab: *RAR file format*, Viitattu 29.6.2013, Saatavilla www-muodossa: http://www.rarlab.com/rar_file.htm

[19] Reasonable Software House Limited: *Reasonable Archiver*, Viitattu 8.10.2012, Saatavilla www-muodossa: http://archiver.reasonables.com/

[20] Seward, Julian: *bzip2*, Viitattu 1.7.2013, Saatavilla www-muodossa: http://bzip.org/

[21] Silverberg, Sam: *SDFS Overview*, Viitattu 15.10.2012, Saatavilla www-muodossa: http://opendedup.googlecode.com/files/SDFS

[22] Szeredi, Miklos: *SSH Filesystem*, Viitattu 25.4.2013, Saatavilla www-muodossa: http://fuse.sourceforge.net/sshfs.html

[23] Vekkilä, Tiina: Digitaalisen tiedon pakkaaminen, Lahden ammattikorkeakoulu, Tietojenkäsittelyn koulutusohjelma, Viitattu 19.10.2012, Saatavilla www-muodossa: http://theseus17-kk.lib.helsinki.fi/bitstream/handle/10024/11929/2007-12-03-19.pdf

[24] Viestintävirasto: *Symmetrinen salaus*, Viitattu 4.10.2012, Saatavilla www-muodossa: http://www.ficora.fi/index/palvelut/palvelutaiheittain/tietoturva/salausmenetelmat/symmetrinensalaus.html