Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Meranie redundancie v dátach

Diplomová práca

2024
Bc. Matúš Hluch

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Meranie redundancie v dátach

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Ing. Dušan Bernát, PhD.

Bratislava, 2024
Bc. Matúš Hluch

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Matúš Hluch
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Meranie redundacie v dátach
Measurement of data redundancy

Anotácia: Analyzujte známe prístupy k meraniu redundancie dát v súborovom systéme s ohľadom na jej využitie pre deduplikáciu, respektíve kompresiu. Zamerajte sa na vplyv veľkosti bloku a využitie rôznych hashovacích funkcií pre maximalizovanie nájdenej zhody.

Navrhnite a implementujte nástroj umožňujúci prenos súboru medzi dvoma procesmi, pričom budú identifikované zhodné prenášané bloky dát. Pre relevantnú dátovú množinu vyhodnoťte početnosť násobného výskytu blokov a to v závislosti od veľkosti bloku, s cieľom nájsť veľkosť s najväčším potenciálom pre využitie redundancie. Porovnajte tiež rôzne hashovacie funkcie. Výsledky zhodnoťte v kontexte známych riešení.

Vedúci: Ing. Dušan Bernát, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 16.12.2022

Dátum schválenia: 28.04.2023 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem školiteľovi, Ing. Dušanovi Bernátovi, PhD., za jeho čas, ochotu, odborné ako aj technické rady a pripomienky, ktoré mi poskytol pri tvorbe tejto práce.

Abstrakt

V práci sa venujeme meraniu redundancie v dátach, konkrétne duplikátom s fixnou veľkosťou. Skúmame ich rôzne vlastnosti: vzdialenosti, počty a ďalšie, a to pri úprave rôznych parametrov deduplikovania. Pre tento problém sme vyvinuli prakticky použiteľný nástroj, ktorý umožňuje posielať cez sieť deduplikované dáta. Ako vstup pre uvedený nástroj využívame bežné používateľské dáta roztriedené podľa typov. Nástroj na prenos deduplikovaných dát poskytuje výstup, ktorý využívame na ďalšiu analýzu nami vytvorenými programami. Na základe našich experimentov a analýz a za pomoci spomenutých programov, sme na vybraných dátach zistili, že počty duplikátov výrazne závisia od typu dát a veľkosti blokov, podľa ktorých sú tieto dáta delené. Okrem toho sme preskúmali aj ďalšie vlastnosti redundancie, pričom výsledky uvádzame v texte.

Kľúčové slová: redundancia, deduplikácia, hešovanie, súborový typ

Abstract

In this thesis, we focus on measuring data redundancy, specifically, fixed-size duplicates. We investigate various properties such as distances, counts, and others, while adjusting various deduplication parameters. To address this problem, we have developed a practical tool that enables the transmission of deduplicated data over the network. This tool takes as input common user data sorted by type. The output provided by the deduplicated data transfer tool is used for further analysis with programs we have created. Based on our experiments and analyses using these programs and selected data, we discovered that the number of duplicates significantly depends on the type of data and the size of the blocks into which the data is divided. In addition, we explored other properties of redundancy, the results of which are presented in this thesis.

Keywords: redundancy, deduplication, hashing, file type

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

V posledných rokoch sme svedkami nárastu množstva dát, ktoré sa spracovávajú v počítačových systémoch, pričom vo väčšine prípadov je potrebné ich uložiť. Hardvér, ktorý s týmito dátami pracuje sa tiež zefektívňuje a zdokonaľuje, ale častokrát nestačí držať krok s potrebami dnešnej doby. S riešeniami preto musí prichádzať aj softvér, ktorý sa používa na spracovávanie a formátovanie dát pred ich uložením.

Formátovaniu dát pred uložením sme sa venovali aj v našej práci. Hlavným prístupom v tejto oblasti je využívanie kompresných algoritmov, ktoré sa snažia efektívne zmenšiť ukladané dáta. Ďalší zaužívaný spôsob, ktorý sa niekedy môže považovať aj za súčasť bezstratových kompresných algoritmov je deduplikácia. Pri deduplikácii sú časti dát, ktoré sú rovnaké minimalizované, napríklad ich nahradením zjednodušenou reprezentáciou.

Zamerali sme sa na meranie redundancie a vlastností duplikátov v dátach. V rámci našej práce sme preto vytvorili viacero nástrojov, ktoré umožňujú analyzovať vlastnosti redundancie v dátach. Tieto nástroje sme následne, s rôznymi nastaveniami, spustili na rôznych typoch dát. Namerané výsledky a ďalšie merania, ktoré boli pomocou nástrojov vykonané, môžu byť použité pri nastaveniach existujúcich systémov a nástrojov ale aj pri návrhu nových algoritmov a programov.

Textovú časť práce sme rozčlenili na 4 kapitoly. V prvej kapitole sme sa venovali aktuálnym riešeniam problematiky deduplikácie. Okrem už praxou overených deduplikačných systémov a nástrojov, sme sa venovali aj výskumu v oblasti deduplikácie za posledných pár rokov. V druhej kapitole je popísaný návrh riešenia našej práce, v ktorej navrhujeme postupy a architektúry nástrojov, programov a analyzovaných parametrov týkajúcich sa deduplikácie. Ďalšia kapitola následne vysvetľuje samotnú implementáciu častí popísaných v návrhu riešenia. Na záver, v poslednej kapitole, uvedieme a zhodnotíme výsledky, ktoré sa nám pomocou implementovaných nástrojov a programov podarilo dosiahnuť.

Kapitola 1

Analýza súčasného stavu problematiky

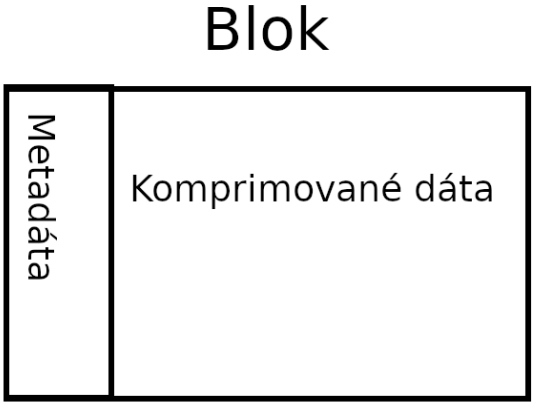
V tejto kapitole uvedieme relevantné články a práce, ktoré sa zaoberajú a riešia problémy blízke tým, ktorým sa venujeme v našej práci, najmä deduplikácii dát. Deduplikácia spočíva vo vyhľadávaní rovnakých častí dát - duplikátov a ich spájaní dokopy a nahrádzaní referenciami alebo odstraňovaní. Najbežnejším cieľom deduplikácie dát je zmenšenie ich objemu, či už v úložnom priestore alebo aj pri prenose.

1.1 ThinDedup

Deduplikácia dát prináša mnoho pozitívnych výsledkov, akými sú napríklad úspora pamäte alebo menšie opotrebovanie disku. Aby však deduplikácia mohla fungovať, je potrebné brať do úvahy aj režijné náklady - dodatočné výpočty a pomocné dáta alebo aj metadáta, ktoré deduplikáciu umožňujú. Článok s názvom ThinDedup: An I/O Deduplication Scheme that minimizes Efficiency Loss due to Metadata Writes [17] sa venuje práve spomínanej problematike metadát vznikajúcich pri deduplikácii.

Pri deduplikácii dát je potrebné uchovávať rôzne metadáta, ktoré sú určené na referenciu na duplikát alebo na dáta, ktorých funkciou je urýchlenie deduplikácie, napríklad rôzne dátové štruktúry, ktoré umožňujú porovnávať a vyhľadávať v uložených dátach. S metadátami sú spojené viaceré výzvy, jednou z nich je aj otázka, kam tieto metadáta uložiť a ako ich zapisovať na disk. Bežným postupom býva zapísanie týchto dát do osobitného bloku. To však spôsobuje neželaný zápis navyše, čo spomaľuje zapisovanie v porovnaní so zápisom bez deduplikácie a zároveň spôsobuje zaplnenie jedného bloku navyše. Druhým problémom s takýmto prístupom je, že blok s metadátami môže spomaľovať aj čítanie. Ak časť dát, ktoré čítame, je deduplikovaná, je nutné prečítať metadáta na lokalizovanie referencie na deduplikované dáta. V tomto prípade sa môže blok s metadátami nachádzať v úplne inej časti disku. To môže spôsobovať spomalenie čítania, a to najmä na pevných diskoch, pri ktorých môže byť nutné posunúť čítaciu hlavu na inú stopu. Ak po systéme požadujeme aj zachovanie konzistentnosti zapisova-

ných dát a odolnosť systému proti výpadkom, je dôležité, aby boli metadáta ukladané priamo na disk. Riešenie, v ktorom by metadáta pre rôzne bloky boli ukladané do jedného spoločného bloku určeného výhradne pre metadáta by preto nebolo vyhovujúce. V tomto prípade by sa metadáta zhromažďovali a až po určitej dobe zapísali do spoločného bloku na disk, to by ale mohlo spôsobiť spomínanú stratu dát počas výpadku.



Obr. 1.1: Zjednodušené znázornenie bloku podľa *Thindedup*.

Autori práce *ThinDedup* prišli so šetrným riešením práce s metadátami. Namiesto vytvárania osobitného, z veľkej časti prázdneho, bloku s metadátami, vytvoria jeden spoločný blok pre zapisované dáta a potrebné metadáta, ktoré sú zapisované. Ich riešenie nepotrebuje žiaden špeciálny alebo upravený hardvér a je založené iba na softvérovej implementácii.

Uvedené riešenie funguje na princípe kompresie dát, ktoré sú ukladané do bloku. Voľné miesto, ktoré kompresiou dát v bloku vznikne, je následne použité na uloženie najdôležitejších metadát určených na deduplikáciu. Ďalej sa autori spomínaného článku venujú optimalizácii a implementácii takéhoto riešenia. Jednou z tém, ktorou sa zaoberajú, je výpočtová náročnosť kompresie, na ktorú použili bezstratový kompresný algoritmus LZ4, ktorého prednosťou je práve jeho rýchlosť. Ďalšou optimalizáciou na zrýchlenie deduplikácie, ktorú autori využívajú, je rozlišovanie dátových blokov na také, ktoré sú a nie sú komprimované. Tým je umožnené bloky bez metadát nekomprimovať alebo nedekomprimovať a vyhnúť sa zbytočným výpočtom navyše.

Experimenty, ktoré autori práce vykonali, dokazujú, že deduplikácia s *Thindedup* je rýchla a efektívna. Celý softvér na deduplikáciu bol naprogramovaný, ako ovládač do jadra operačného systému Linux a je založený na platforme *Dmdedup*, ktorá je určená na tvorbu deduplikačných programov v Linuxe. Softvér *Thindedup* bol porovnávaný voči iným deduplikačným programom (*Dmdedup* a *OrderedWrites*) na syntetických a skutočných dátach. Program bol na syntetických dátach testovaný s rôznymi parametrami – veľkosť deduplikačného bloku, komprimovateľnosť dát a deduplikovateľnosť dát.

Experimenty ukázali, že *Thindedup* poskytoval väčšiu priepustnosť ako oba spomenuté konkurenčné programy, a to najmä v prípade náhodných prístupov v porovnaní so sekvenčnými prístupmi na disk. Kvalitu *Thindedup* potvrdili aj experimenty so skutočnými dátami zo systémov, ktoré spravujú e-mailové schránky Floridskej medzinárodnej univerzity zozbieraných počas 21 dní. Softvér *Thindedup* v testoch preukázal v niektorých prípadoch až 112 percentné zlepšenie priepustnosti na pevných diskoch v porovnaní s konkurenciou. Výrazné zlepšenie priepustnosti na pevných diskoch, vďaka *Thindedup*, je podľa autorov článku pravdepodobne spôsobené zníženým počtom nesekvenčných prístupov, ktoré sú na pevných diskoch pomalé.

1.2 CA-Dedupe

V práci *Thindedup* autori popísali optimalizáciu jedného z problémov deduplikácie. Existuje však mnoho ďalších oblastí, ktoré je možné a aj sú v oblasti deduplikácie skúmané a vylepšované. Na jednu z týchto oblastí sa zamerala aj práca, ktorá bola uverejnená v roku 2020. V tejto časti popíšeme článok CA-Dedupe: content-aware deduplication in SSDs od Ramin Gholami Taghizadeh a kolektívu [12].

Deduplikácia je zväčša používaná v prípade väčšieho objemu dát, ktoré majú viacero podobných častí. Práca s obrovským objemom dát je pomerne náročná, a to sa týka aj deduplikácie. S rastúcou veľkosťou dát a počtom deduplikovaných častí týchto dát je deduplikácia komplikovanejšia, a to najmä, kvôli práci s veľkým počtom alebo veľkým objemom metadát. V praxi sa to môže prejaviť pomalými zápismi, keďže pri prehľadávaní možných duplikátov so zapisovanými dátami je potrebné pracovať s väčším množstvom metadát. Pri špecifickej implementácii sa to môže prejaviť aj pri čítaní, kedy môže byť komplikovanejšie spracovať metadáta referencujúce na duplikáty, vďaka ich veľkosti alebo zložitosti.

Práca *CA-Dedupe* sa venuje najmä už skôr spomenutej komplikácii, a teda spomaleniu zapisovania. Veľký počet metadát určených na hľadanie zhody medzi blokmi, ktorými sú najčastejšie heše jednotlivých blokov, môžu byť výpočtovo náročné na prehľadávanie a efektívne porovnávanie s metadátami, zväčša tiež hešom, aktuálne zapisovaných dát. Preto existuje mnoho prác venujúcich sa vhodným dátovým štruktúram a algoritmom pracujúcich s pomocnými dátami.

V práci *CA-Dedupe* prišli s vlastnou myšlienkou v oblasti deduplikácie, a to rozdeľovať a prehľadávať heše blokov podľa typu súboru a jeho obsahu, z ktorého daný blok pochádza. To môže výrazne skvalitniť mieru deduplikácie a urýchliť zápisy, keďže v práci predpokladali, že bloky, ktoré sú duplicitné môžu byť časťami súborov rovnakého typu.

Na rozpoznanie typu alebo formátu súboru existuje viacero techník od základných

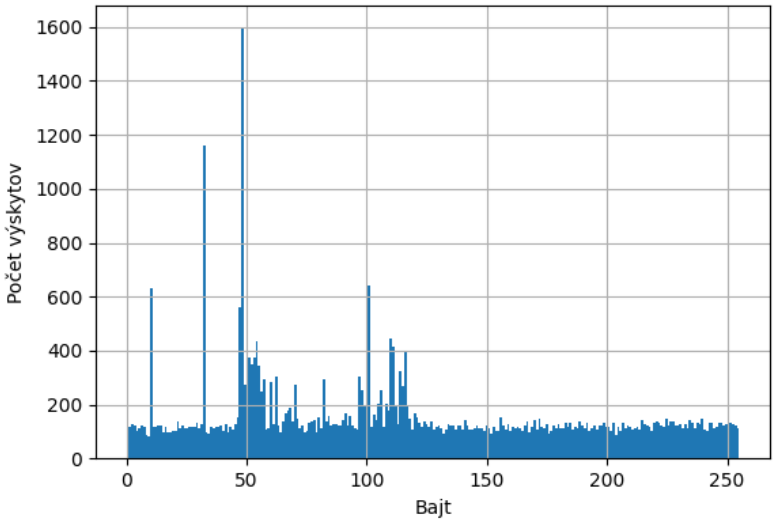
štatistických metód po strojové učenie a je samostatnou oblasťou výskumu. V prípade *CA-Dedupe* však bola veľmi dôležitým parametrom rýchlosť, ktorá je pri zápise blokov na disk potrebná a na základe toho zvolili dva pomerne jednoduché a efektívne prístupy.

Prvým spôsobom, ktorý je v práci *CA-Dedupe* používaný na rozpoznanie typu, je využitie magických čísel alebo bajtov (*magic bytes*). Magické čísla napomáhajú rozpoznávaniu typu súboru tak, že každý typ súboru má určený svoj podpis, teda zopár špecifických bajtov, ktoré sú uvedené najčastejšie na začiatku takéhoto súboru. Na rozpoznanie typu prvého bloku zapisovaného súboru preto stačí vo väčšine prípadov jednoducho prečítať prvých pár bajtov a na základe nich zaradiť blok do kategórie s daným typom. Tento prístup nefunguje pri blokoch, ktoré nie sú prvé v súbore a kvôli tomu v práci používajú aj druhý spôsob na rozpoznanie typu. Pre ilustráciu uvádzame príklady typických magických čísel, kde najprv uvádzame bajty v hexadecimálnom zápise a za nimi typ súboru oddelený dvojbodkou:

- **89 50 4E 47 0D 0A 1A 0A**: png, obrázkový formát
- **25 50 44 46 2D**: pdf, dokument
- **49 44 33**: mp3, zvukový formát
- **1F 8B**: gz, súbor komprimovaný algoritmom gzip

Druhým spôsob je používaný v prípade, že sa na začiatku bloku, ktorý je potrebné zaradiť do kategórie podľa typu, nenachádzajú žiadne magické čísla. Ide teda o bloky, ktoré nie sú prvé v súbore. Autori softvéru *CA-Dedupe* sa v tomto prípade vysporiadali s danou situáciou štatistikou. Iné práce venujúce sa rozpoznávaniu súborových typov zistili, že väčšina rôznych súborových typov má pre nich typické frekvencie výskytov bajtov. Na rozpoznanie súborového typu súboru na základe bloku, ktorý nie je prvým blokom, je teda možné použiť histogram výskytu bajtov a zaradiť ho do kategórie s podobnými frekvenciami výskytu. V článku tiež spomínajú, že aj keď je v tomto prípade na analýzu bajtov dostupná iba časť súboru, ak je dostatočne veľká (v článku je uvedených 1024 bajtov), tak je odhad súborového typu pomerne presný.

Na uloženie blokov rovnakého súborového typu v práci vyvinuli špeciálnu dátovú štruktúru stromového typu. Pre každý súborový typ, s ktorým sa softvér *CA-Dedupe* počas práce stretne, vytvorí špeciálny prehľadávací AVL strom, ktorý nazvali strom súborového typu (*File Type Tree*). Vo vrcholoch sú v tomto strome uložené súbory, ktoré boli zapisované, ako aj ďalšie potrebné metadáta, akými sú napríklad rôzne heše určené na zrýchlenie prehľadávania alebo jednotlivé počty výskytov bajtov v blokoch spracovávaného súboru.



Obr. 1.2: Príklad histogramu počtu výskytov bajtov na súbore typu pdf.

1.2.1 Experimentovanie a namerané výsledky

Nástroj bol testovaný na simulovanom SSD disku pomocou simulačného nástroja SSD-Sim prostredníctvom, ktorého je možné odsimulovať virtuálny SSD disk s parametrami, akými sú napríklad výkon SSD alebo jeho spotreba, nastaviteľnými cez konfiguračný súbor. Následne pri používaní tohto virtuálneho SSD disku sa zaznamenávajú rôzne užitočné údaje o využití disku, napríklad zoznam všetkých požiadaviek na zápis alebo čítanie.

So simulovaným SSD diskom sa nepracuje ako s obyčajným diskom, ale je potrebné, aby simulačný nástroj dostal na vstupe záznam čítaní a zápisov, ktoré je potrebné vykonať. Uvedený záznam v práci *CA-Dedupe* vytvorili pomocou druhého nástroja s názvom DiskMon. Tento nástroj umožňuje zaznamenať celú aktivitu na disku v operačnom systéme Windows. Takýto záznam vytvorený DiskMon nástrojom následne mohli v práci opakovane použiť pri experimentoch s ich deduplikačným softvérom na simulovanom SSD disku.

Experimenty boli vykonané na operačnom systéme Windows s bežnými používateľskými dátami, teda so súbormi typu mp3, mp4 alebo pdf. Softvér *CA-Dedupe* bol na týchto dátach porovnávaný s konkurenčnými deduplikačnými softvérmi *CAFTL* a *NF-Dedupe*. Výsledky experimentov boli porovnávané vo viacerých parametroch: ako dlho trvá vyhľadávanie duplikátov a zápis, ako dobre dokázal softvér deduplikovať dáta a zároveň, aká bola celková spotreba pamäte. Experimenty ukázali, že softvér *CA-Dedupe* dokáže nájsť duplikáty so 70 percent vyššou presnosťou a zároveň bol pritom o 23 percent rýchlejší pri zápise, ako konkurenčný softvér.

1.3 DmDedup

Deduplikačné softvéry sú pomerne komplikované na implementáciu, hlavne v prípade, ak sú vytvorené ako ovládač do jadra operačného systému. Takáto implementácia je potrebná pre najefektívnejšiu a najrýchlejšiu deduplikáciu, čo je zväčša cieľom väčšiny deduplikačných programov. Z veľkej časti sú preto práce, ktoré sa zameriavajú na tvorbu a skúmanie takýchto ovládačov do jadra realizované početnými tímami výskumníkov. Väčšina týchto prác však implementuje podobné časti a častokrát začínajú vytváraním ovládačov úplne od začiatku, pričom stavajú na spoločných základoch.

V práci *Dmdedup*: Device Mapper Target for Data Deduplication od Vasily Tarasova a kolektívu [20] vytvorili platformu, ktorá má tento problém riešiť, a teda uľahčiť a urýchliť výskumníkom a programátorom tvorbu deduplikačného softvéru. *DmDedup* vývojárom poskytuje API na zjednodušenie tvorby deduplikačných systémov. Platforma funguje na úrovni dátových blokov.

V práci autori popisujú rozdiel medzi implementáciou deduplikácie na blokovej vrstve a vrstve súborového systému. Úroveň blokov, ako názov naznačuje, sa nachádza nad blokovými zariadeniami, podobne ako napríklad RAID a pracuje s blokmi rovnakej veľkosti. Na úrovni súborového systému, ktorá je nad úrovňou blokov, je možné deduplikovať nad komplexnejšími štruktúrami a objektami, napríklad nad súbormi. Zároveň je pri deduplikácii možné využívať štruktúry súborového systému, akými sú napríklad rôzne stromy a tabuľky s informáciami o súborovom systéme.

Deduplikácia na vrstve súborového systému, akú využíva väčšina prác venujúcich sa deduplikácii, je podľa autorov menej výhodná z dôvodu neprenositeľnosti medzi jednotlivými súborovými systémami. Aj keď vývoj deduplikačných systémov na úrovni dátových blokov má tiež svoje nevýhody, autori sa rozhodli zamerať sa na čo najväčšiu prenositeľnosť a modularitu. *Dmdedup* teda funguje pre vrstvu súborového systému, ako akékoľvek iné blokové zariadenie.

Platforma *Dmdedup* je rozdelená na viacero častí, ktoré je možné implementovať alebo upraviť podľa potreby experimentu. Na najnižšej vrstve sú potrebné dve zariadenia – jedno na metadáta druhé na dáta. Autori odporúčajú aby zariadenie na metadáta bolo typu SSD z dôvodu rýchlejšieho čítania a zapisovania. Nie je to však nutné, môže sa použiť jedno zariadenie s dvoma partíciami.

Nad spomenutými dvoma zariadeniami sa nachádza blokové zariadenie *DmDedup*, ktoré je zložené z menších podčastí s rôznymi účelmi. Najprv je potrebné rozdeliť dáta, ktoré sú zapisované na časti – bloky vhodné na deduplikáciu. Následne sú bloky hešované hešovacou funkciou a potom sú tieto heše porovnávané pri hľadaní duplikátov. Ďalšia časť sa stará o mapovanie medzi logickými adresami blokov a fyzickými adresami, aby duplikované dáta mohli byť referencované na tú istú fyzickú adresu. Z článku je možné sa dozvedieť, ako implementovali mapovania na heše a ich úpravy. Taktiež

spomínajú aj *garbage collection* prepísaných blokov a k nim prislúchajúcich hešov.

Platforma *Dmdedup* necháva správu metadát na implementujúcich a tým platforme dodáva flexibilitu. Pre vývojárov je dostupné API – dvanásť funkcií, ktoré je možné vytvoriť a tým implementovať vlastný deduplikačný systém.

1.3.1 Dosiahnuté výsledky s rôznymi implementáciami

Keďže *DmDedup* je platforma na tvorbu deduplikačných blokových zariadení, pre experimentovanie je potrebné implementovať funkcie poskytnutého API. Jeden takto vytvorený deduplikačný systém je aj už skôr popísaný softvér *ThinDedup*. Autori *Dmdedup* na porovnanie a preukázanie možností ich platformy taktiež implementovali tri rôzne deduplikačné systémy. Celá platforma je určená pre operačný systém Linux, a teda aj jednotlivé experimenty boli robené na Linuxe.

Najjednoduchšou z ukážkových implementácií je implementácia, ktorú nazvali INRAM. Metadáta sú v tejto implementácii uložené v RAM pamäti a je možné ich uložiť a načítať zo súboru. Deduplikácia je vyriešená hešovacou tabuľkou a jedným poľom.

Druhou implementáciou, založenou z veľkej časti na INRAM, je implementácia s názvom DTB. DTB je takmer vo všetkom rovnaká, ako INRAM (pole a hešovacia tabuľka), až na miesto, kam sa ukladajú metadáta. Implementácia DTB ich ukladá na disk, aby sa však pri každom zápise dát nemusel robiť ešte jeden zápis kvôli metadátam, autori využili Linuxový subsystém *dm-bufio*. *Dm-bufio* subsystém postupne zbiera požiadavky na zápis, až kým nedosiahnu veľkosť štyroch kilobajtov a následne ich zapíše na disk.

Poslednú najzložitejšiu implementáciu, uvedenú v práci, nazvali CBT. Táto implementácia používa na ukladanie hodnôt a kľúčov B - stromy. Tak ako pri DTB implementácii, tak aj v tomto prípade využili aj subsystém *dm-bufio* na spojenie viacerých zápisov na disk do jedného.

Zjednodušené popisy jednotlivých implementácií spomenutých v práci:

- INRAM: pole a hešovacia tabuľka v pamäti
- DTB: pole a hešovacia tabuľka na disku
- CBT: B - strom na disku

Experimenty boli vykonávané na všetkých troch spomenutých implementáciách platformy *DmDedup*, pričom bol vykonaný tiež jeden kontrolný experiment – na zariadení bez deduplikačného zariadenia. Experimenty boli tiež vykonané aj na konkurenčnom deduplikačnom programe *lessfs*. Použitý bol jeden pevný disk na dáta a jeden SSD disk na metadáta.

Všetky testy boli robené na skutočnom serverovom hardvéri na operačnom systéme Linux. Na porovnanie ich troch implementácií a zariadenia bez deduplikácie použili umelo vytvorené dáta – náhodné vygenerované dáta, opakovaný 4 kilobajtový blok dát a viacero jadier Linuxu. Tieto dáta ešte zapisovali dvoma rôznymi spôsobmi – náhodný zápis a sekvenčný zápis.

Zaujímavým zistením je, že pri rôznych typoch dát sú jednotlivé implementácie lepšie alebo horšie. Napríklad pri náhodných dátach je najrýchlejším zariadenie bez deduplikácie a implementácia DTB a INRAM sú rýchlejšie ako implementácia CBT, kvôli väčšiemu počtu zápisov na disk. Na druhej strane, pri dátach s Linuxovými jadrami všetky implementácie prekonali v priepustnosti zariadenie bez deduplikácie a implementácia CBT bola rýchlejšia ako DTB.

Na záver autori porovnali implementáciu CBT na skutočných dátach s konkurenčným softvérom *Lessfs*. Dáta boli použité tie isté ako v práci *ThinDedup*, teda záznamy zo serverov z Floridskej medzinárodnej univerzity. Hlavným zámerom práce *Dmdedup* síce bola tvorba platformy so zameraním sa na možnosť rôznych implementácií a rozšírení, ale okrem toho došli k záveru, že aj v tom najlepšom nastavení pre softvér *lessfs*, je CBT implementácia 1.6-krát rýchlejšia.

1.4 Rsync

Nástroj *rsync* je už dlhodobo jedným z najpoužívanejších nástrojov pri prenose dát na Unixových ale aj iných operačných systémoch. Bol vyvinutý v roku 1996 Andrewom Tridgellom a Paulom Mackerassom a aj dodnes je tento nástroj udržiavaný a vylepšovaný. Prvýkrát popísaný je v článku The rsync algorithm [21] od samotných autorov nástroja *rsync*.

Rsync je nástroj, ktorý umožňuje prenášať a synchronizovať súbory a priečinky. Ďalším pomerne známym nástrojom na Unixových systémoch pre prenos dát je nástroj SCP (*secure copy protocol*). Aj SCP aj *rsync* vedia kopírovať súbory cez sieť v štandardnom nastavení cez sieťový protokol SSH (*secure shell protocol*). Rozdielom medzi týmito dvoma nástrojmi je, že *rsync* na rozdiel od SCP, nemusí vždy prenášať cez prenosový kanál všetky dáta a teda môže znížiť objem prenesených dát a tým aj urýchliť celý prenos. *Rsync* teda môže byť použitý ako efektívny nástroj aj na lokálne kopírovanie a synchronizáciu dát.
Príklad použitia nástroja *rsync*:

Presun lokálneho priečinka src na adresu remote používateľa user na miesto dst:

```
$ rsync -a src user@remote:dest
```

Pri prenose cez *rsync* sa existujúce cieľové súbory aktualizujú na základe zmien oproti zdroju. Ak teda cieľové miesto už obsahuje súbory, ktoré sú pomocou *rsync*

nástroja kopírované, tento nástroj prenáša už len časti, ktoré sú zmenené. Algoritmus, ktorý toto celé umožňuje vlastne hľadá rozdiely medzi zdrojom a cieľom. Tieto rozdiely sú následne prenesené prijímateľovi, ktorý si na základe nich dokáže lokálne dáta aktualizovať.

V rámci algoritmu, ktorý *rsync* používa na hľadanie rozdielov medzi dátami sú použité dva typy hešovacích funkcií. Algoritmus používa štandardne známe funkcie, ktoré vypočítajú heš na základe vstupu a špeciálne - *rolling* alebo posuvné hešovacie funkcie, ktoré sú popísané v nasledujúcej definícii.

Definícia 1 ***Posuvná hešovacia funkcia*** *je hešovacia funkcia, ktorá dokáže efektívne počítať heše s posunom po dátach. Takáto hešovacia funkcia spracováva okno fixnej veľkosti, ktoré sa presúva po dátach a vypočíta sa na základe hešu predošlého okna, pridaných dát z konca okna a odobratých dát zo začiatku predošlého hešovaného okna. Formálne, posuvná hešovacia funkcia pre podreťazec* $x_2...x_{n-1}x_n$*, reťazca* $x_1x_2...x_{n-1}x_n...x_{n+k}$ *sa vypočíta na základe predošlého hešu, teda hešu pre podreťazec* $x_1x_2...x_{n-1}$ *a symbolov* x_1 *a* x_n*.*

Nástroj *rsync* vo svojom algoritme na hľadanie rozdielov využíva posuvný heš Adler-32, ďalšou známou posuvnou hešovacou funkciou je napríklad Rabinov odtlačok (Rabin fingerprint) a je častokrát využívaná v rôznych implementáciách algoritmu Rabin-Karp [14].

V algoritme si prijímateľ na začiatku rozdelí dáta na bloky rovnakej veľkosti. Ďalej sú na blokoch počítané dve rôzne hešovacie funkcie. Prijímateľ vypočíta z každého bloku heš a posuvný heš - už spomínanou hešovacou funkciou Adler-32. Prvý spomenutý heš je náročnejší na výpočet s väčšou rezistenciou proti kolíziám. V ďalších krokoch je používaný na potvrdenie zhody v časti súboru, ktorá bola určená posuvným hešom. V poslednej verzii *rsync* sa na toto potvrdzovanie zhody používa hešovacia funkcia *MD5*. Tieto heše sú následne poslané odosielateľovi. Dáta odosielateľa sú prechádzané s posuvným hešom a s veľkosťou okna, rovnakou ako prijímateľ a hľadá zhody s hešmi, ktoré prijímateľ poslal. Pre okná, kde odosielateľ našiel zhodu, odosielateľ pošle príjemcovi miesto zhody a pre zvyšné časti samotné dáta.

Okrem porovnávania hešov, pre ďalšie zefektívnenie a zrýchlenie prenosu, nástroj pred prenosom porovnáva *mtime*, teda čas poslednej úpravy a veľkosť súborov. Ak sa zhodujú a táto kontrola nie je vypnutá cez prepínač, súbory, ktoré majú tieto parametre zhodné sa nesynchronizujú. Okrem tohto nastavenia ponúka nástroj rsync viacero nastavení cez prepínače a parametre umožňujúce komprimovať vstupné dáta, nastavovať práva pre kopírované súbory, nastavenia pre symbolické linky a mnohé ďalšie.

1.5 Súborové systémy a deduplikácia

Rozsiahlou oblasťou, kde sa bežne využíva deduplikácia sú súborové systémy. Deduplikáciu je výhodné použiť najmä na serveroch, kde sa môže nachádzať veľa podobných dát, akými napríklad môžu byť disky virtuálnych strojov. Cieľom väčšiny deduplikačných súborových systémov je deduplikovať transparentne pre používateľov, a to v takom zmysle, aby sa s deduplikovanými dátami dalo pracovať ako s ktorýmikoľvek inými.

Deduplikáciu je možné v súborových systémoch implementovať na rôznych úrovniach. Na najnižšej úrovni - hardvéri je deduplikácia zriedkavá, zväčša určená pre špecifické prípady. Ďalšou možnosťou je implementácia na úrovni blokov, nezávisle od súborového systému nad ňou, takto je implementovaný aj nástroj *Dmdedup*. Poslednou možnosťou je implementácia na úrovni samotného súborového systému.

Aj v prípade implementácie na úrovni súborového systému existuje viacero možností implementácie a viaceré súborové systémy k nej pristupujú rôzne. V prvom rade je potrebné sa pri deduplikácii v súborovom systéme rozhodnúť, kedy deduplikácia prebehne. Priamočiarym riešením je deduplikácia pri zápise alebo úprave dát. Toto riešenie môže byť v niektorých prípadoch nevyhovujúce, keďže deduplikácia môže byť výpočtovo náročná a teda môže spôsobiť spomalenie systému, práve v čase, keď sa pracuje s dátami a samotným systémom. Preto niektoré súborové systémy využívajú aj ďalší prístup - deduplikáciu na požiadanie. Dáta sú deduplikované v tomto prípade až po zadaní príkazu používateľom, či už na všetky dáta v súborovom systéme, alebo iba na časť z dát - napríklad priečinok špecifikovaný používateľom. Podobnou možnosťou je aj deduplikácia spustená v pravidelných časových intervaloch alebo v časoch, keď je najmenej vyťažený procesor.

Typy deduplikácie podľa toho, kedy prebieha:

- pri zápise dát
- na požiadanie
- v pravidelných alebo nepravidelných časových intervaloch

Ďalším parametrom podľa, ktorého je možné deduplikáciu v súborových systémoch rozdeliť je granularita delenia dát určených na deduplikovanie. Najjednoduchším spôsobom je deduplikácia na úrovni súborov. Namiesto ukladania viacerých rovnakých súborov sa v súborovom systéme uloží jeden a ostatné rovnaké súbory sú ukladané ako referencia na prvý uložený súbor. Časť súborových systémov deduplikuje po blokoch fixnej veľkosti. To znamená, že dáta sú rozdelené na časti fixnej veľkosti a následne sú tieto časti porovnávané, s cieľom nájsť také, ktoré sú rovnaké. Hlavné výhody tohto prístupu sú pomerne jednoduchá implementácia a nízke režijné náklady, aj výpočtové

a aj v objeme metadát. Nevýhodou je možná, nižšia miera deduplikácie, ktorá môže byť spôsobená tým, že duplikáty v dátach nemusia mať rovnakú veľkosť ako prednastavená veľkosť deduplikačného bloku. Táto nevýhoda môže byť riešená deduplikáciou s premenlivou veľkosťou bloku. Na druhej strane, takáto deduplikácia môže byť podľa spôsobu ukladania dát v súborovom systéme náročnejšia na implementáciu a zároveň pomalšia.

Typy deduplikácie podľa granularity:

- súbory
- fixná veľkosť bloku
- premenlivá veľkosť bloku

Doteraz sme v práci primárne popisovali spôsoby, ako môžu byť dáta deduplikované. Ak sú však dáta v systéme už deduplikované, je potrebné riešiť aj prácu s deduplikovanými dátami. Jednou z možností je, aby deduplikované dáta neboli upravovateľné a boli určené iba na čítanie. Ďalšou z možností je úprava deduplikovaných dát špeciálnym postupom. Používa sa tu metóda COW (*Copy-on-write*), ktorá je využívaná nielen v súborových systémoch, ale aj v iných oblastiach, napríklad v operačných a databázových systémoch. V systémoch implementujúcich COW sa pri pokuse o zápis do dát, ktoré boli deduplikované, vytvorí kópia a úpravy sa následne vykonávajú na tejto kópii. Všetko sa vykonáva tak, že používateľ nevidí rozdiel medzi prácou s deduplikovanými a nededuplikovanými dátami.

V nasledujúcich dvoch podčastiach popíšeme dva bežne v praxi používané súborové systémy BTRFS a ZFS a ich prístup k deduplikácii.

1.5.1 BTRFS

Súborový systém BTRFS [19] je známy vďaka veľkému počtu nastavení, efektívnosti a podpore funkcionalít, akými sú napríklad snímky, kontrolné súčty, podpora pre RAID a ďalšie. Základ BTRFS je postavený na modifikovanej dátovej štruktúre B - stromov tak, aby umožňovala techniku COW a zvyšné funkcionality.

Aby bola umožnená flexibilita pri deduplikácii, samotné hľadanie duplikátov je riešené v BTRFS externými nástrojmi. Duplikáty v dátach nájdené externými nástrojmi sú odovzdané špeciálnymi *ioctl* volaniami súborovému systému BTRFS, ktorý ich následne spracuje, deduplikuje a uvoľní priestor v úložisku. Deduplikácia v BTRFS je na požiadanie, nástroj na deduplikáciu je potrebné spustiť a až následne sú aktuálne dáta deduplikované.

Podľa dokumentácie BTRFS existujú dva rôzne udržiavané externé deduplikčné nástroje BEES a *Duperemove*. V dokumentácii sú uvádzané základné vlastnosti týchto

deduplikačných nástrojov. Oba nástroje sú inkrementálne, teda po spustení nástroja sa deduplikačné metadáta ukladajú a po ďalších spusteniach aktualizujú. Zatiaľčo nástroj *Duperemove* umožňuje špecifikovať, ktoré súbory majú byť deduplikované, nástroj BEES vždy deduplikuje celý súborový systém.

Bližšie sme sa pozreli na nástroj *Duperemove* [9]. Deduplikáciu pomocou nástroja *Duperemove* je možné spustiť pomocou jednoduchého príkazu.

Deduplikácia pomocou *Duperemove* súborov *subor1.img*, *subor2.img* a rekurzívna deduplikácia priečinka *priecinok*

```
$ duperemove -r subor1.img subor2.img priecinok/
```

Dáta sú deduplikované po blokoch s fixnou veľkosťou, ktorá je nastaviteľná od 4096 bajtov. Metadáta z deduplikácie, z najväčšej časti tvorené hešmi blokov, sú následne ukladané do SQL databázy, ktorá je po deduplikácii uložená na disk. Funkciu na hešovanie blokov je možné nastaviť a vybrať z dvojice *MurMur* a *xxHash*.

Výpis 1.5.1: Ukážkový výstup z nástroja *Duperemove* po deduplikácii

```
Found 8 identical extents.
Simple read and compare of file data found 4 instances
of extents that might benefit from deduplication.

Showing 2 identical extents of length 6967 with id 20964dc7
Start           Filename
0       "/mnt/loopback/html/005890.html"
0       "/mnt/loopback/html/005900.html"
Showing 2 identical extents of length 15499 with id 10416fb1
...
```

Vo výpise 1.5.1 je možné si prezrieť počty a dĺžky extentov (viacero za sebou idúcich blokov), ktoré boli nájdené, ako duplicitné a následne odovzdané súborovému systému na deduplikáciu. Ku každej duplicite sú pripísané aj ich súbory a začiatky v počte bajtov od začiatku súboru, kde sa dané duplicitné extenty začínajú.

1.5.2 ZFS

Podobne ako BTRFS, tak aj ZFS [4] ponúka veľké množstvo funkcionalít od snímok súborového systému až po podporu RAID a využíva aj techniku COW. Populárny je najmä vďaka kvalitnej kontrole integrity a schopnosti opravy poškodených dát. V niektorých oblastiach vyniká ZFS a v iných BTRFS a záleží na konkrétnom prípade, ktorý z týchto dvoch súborových systémov je lepší.

Tabuľka 1.1: Ukážkový výstup stavu deduplikácie v ZFS.

bucket	**allocated**				**referenced**			
refcnt	**blocks**	**LSIZE**	**PSIZE**	**DSIZE**	**blocks**	**LSIZE**	**PSIZE**	**DSIZE**
1	68.7K	8.59G	4.69G	4.69G	68.7K	8.59G	4.69G	4.69G
2	551	68.9M	30.3M	30.3M	1.10K	141M	63.0M	63.0M
4	7	896K	724K	724K	30	3.75M	2.83M	2.83M
8	8	1M	8K	8K	94	11.8M	94K	94K
16	2	256K	2K	2K	44	5.50M	44K	44K
32	1	128K	1K	1K	43	5.38M	43K	43K
Total	69.2K	8.65G	4.72G	4.72G	70.0K	8.75G	4.76G	4.76G

Na rozdiel od deduplikácie v BTRFS je deduplikácia v ZFS implementovaná priamo, bez potreby externých nástrojov. Táto funkcia súborového systému je nastaviteľná a je možné ju zapnúť alebo vypnúť a upraviť iné vlastnosti deduplikácie. Po jej zapnutí, sú každé zapisované a upravované dáta automaticky deduplikované, bez potreby spúšťania iných príkazov. Deduplikácia v ZFS je vykonávaná na úrovni blokov s maximálnou veľkosťou nastavenou podľa parametra recordsize a v základnom nastavení využíva na deduplikáciu hešovaciu funkciu *sha256*.

Výpis 1.5.2: Ukážkový výstup stavu deduplikácie v ZFS

```
    $ zpool status -D mypool
pool: mypool
state: ONLINE
  errors: No known data errors

dedup: DDT entries 70901, size 223B on disk, 120B in core
```

Výstup stavu deduplikácie v ZFS je možné skontrolovať zadaním príkazu **zpool status**. Ukážkový výstup je uvedený v tabuľke 1.1 a výpise 1.5.2. Prvá, textová časť, popisuje základné údaje o deduplikačnej tabuľke. V prípade uvedeného príkladu tabuľka obsahuje informácie o 70901 blokoch, využíva 223 bajtov na disku a 120 bajtov v pamäti.

Druhá časť obsahuje samotnú tabuľku, ktorá je rozdelená na 3 sekcie: **bucket**, **allocated** a **referenced**. V stĺpci **bucket**, sú popísané počty referencií na bloky, ktoré sú popísané v ďalších stĺpcoch. Počty referencií sú zoskupené po mocninách dvojky, napríklad v riadku s hodnotou **refcnt** 4 sú bloky, ktoré sú referencované 4, 5 a 6-krát a v riadku s hodnotou **refcnt** 8, sú bloky, ktoré sú referencované 8 až 17-krát.

Sekcie tabuľky **allocated** a **referenced** majú stĺpce s rovnakými názvami podstĺpcov. Sekcia **allocated** označuje miesto, ktoré je alokované, fyzickú veľkosť dát. Sekcia **referenced** popisuje logickú veľkosť dát, teda celý objem bez deduplikácie.

Stĺpce v oboch sekciách potom majú nasledovný význam:

- **blocks** Počet blokov, ktoré boli referencované **refcnt** krát.
- **LSIZE** Logická veľkosť, pred kompresiou a inými optimalizáciami veľkosti.
- **PSIZE** Fyzická veľkosť na disku.
- **DSIZE** Skutočná veľkosť na disku, môže sa líšiť s **PSIZE**, napríklad, ak sa používa RAID.

Najzaujímavejším riadkom je riadok **Total**. Mieru deduplikácie je možné odpozorovať porovnaním stĺpcov **DSIZE** medzi sekciami **allocated** a **referenced** v tomto sumárnom riadku. Výsledná hodnota popisuje, aké veľké miesto sa podarilo deduplikáciou ušetriť. V tomto príklade sa podarilo ušetriť okolo 40 megabajtov.

Kapitola 2

Návrh riešenia

V tejto kapitole popíšeme náš prítup k deduplikácii dát a ich analýze. Vysvetlíme a zdôvodníme naše rozohodnutia pre jednotlivé postupy.

2.1 Analýza existujúcich hešovacích funkcií

V tejto kapitole spomenieme rôzne hešovacie funkcie, ktoré majú špeciálne zameranie. Niektoré z nich porovnáme a využijeme aj v našej práci. Okrem nižšie uvedených funkcií využívame aj ďalšie hešovacie funkcie *sha256* a *MD5*, ktoré považujeme za štandardné a v tejto časti ich nebudeme bližšie popisovať.

2.1.1 SimHash

Väčšina bežne známych hešovacích funkcií, akými sú napríklad *MD5* alebo *SHA-512* sú kryptograficky bezpečné funkcie. Hešovacie funkcie tohto typu sú využívané viacerými spôsobmi: utajovanie informácie, digitálne podpisovanie a ďalšie. Sú teda od nich vyžadované rôzne vlastnosti ako napríklad odolnosť voči kolíziám alebo ireverzibilnosť.

Alternatívnou hešovacou funkciou, nespĺňajúcou všetky vlastnosti, ktoré bežne od hešovacích funkcií očakávame, napríklad kryptografickú bezpečnosť, je aj funkcia *SimHash*. V článku Similarity Estimation Techniques from Rounding Algorithms [5] ju prezentoval Moses S. Charikar.

Táto hešovacia funkcia, ako jej názov naznačuje zaznamenáva podobnosť hešovaných dát. Dáta, ktoré sú podobné, majú podobný výstup hešovacej funkcie *SimHash*.

V článku je definovaná podobnostná miera:

Definícia 2 *Podobnostnú mieru definujeme na dvoch množinách X a Y nasledovne* $sim(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$

Takáto podobnostná miera sa nazýva aj Jaccardova funkcia. Ako si je možné z definície všimnúť, nadobúda hodnoty z intervalu $[0, 1]$. Hodnotu 1 dostaneme na výstupe

v prípade, ak na vstupe zadáme rovnakú množinu dvakrát.

Funkcia *SimHash* z článku aproximuje zadefinovanú podobnostnú mieru. Ak máme dva výstupy funkcie *SimHash*, miera ich podobnosti je daná ich Hammingovou vzdialenosťou, ktorá je definovaná nižšie. Teda, čím väčšia je ich Hammingova vzdialenosť, tým menej by mali byť podobné zadané dáta na vstupe.

Definícia 3 *Dva bitové reťazce rovnakej dĺžky majú Hammingovu vzdialenosť rovnú počtu pozícií, na ktorých sú reťazce odlišné [15].*

Napríklad reťazce **0001010** a **0000011** majú Hammingovu vzdialenosť dva, keďže na štvrtej a poslednej pozícii v reťazci sú bity odlišné.

Hešovacia funkcia *SimHash* a jej podobné funkcie, ktoré sú nazývané aj ako hešovacie funkcie citlivé na lokalitu (*locality-sensitive hashing*), teda hešovacie funkcie, ktoré majú pre podobný vstup podobný alebo rovnaký výstup, majú využitie vo viacerých oblastiach. Dajú sa použiť na deduplikáciu pri hľadaní duplicitných dát, antiplagiátorských systémoch, hľadaní podobností v obrázkoch a ďalších. Konkrétne funkcia *SimHash* bola používaná napríklad aj firmou Google na detekciu podobných stránok [16].

2.1.2 MurMurHash

Podobne ako pri hešovacom algoritme *SimHash* je aj hešovacia funkcia *MurMurHash* [2] špecifická a tiež nie je kryptograficky bezpečná. Už skôr spomínané vlastnosti kryptografických hešovacích funkcií, akými sú napríklad odolnosť voči kolíziám alebo ireverzibilnosť nesú so sebou určité nevýhody. Jednou z najzásadnejších je ich náročnosť na ich výpočet a rýchlosť.

Hešovacia funkcia *MurMurHash* v tomto smere vyniká oproti ostatným hešovacím funkciám. Je rýchla - v porovnaní priepustnosti na veľkých dátach napríklad s kryptografickou funkciou *MD5* mala päťnásobnú priepustnosť. Na menších dátach, krátkych reťazcoch, bola dokonca približne sedemkrát rýchlejšia ako *MD5* [6]. Toto je možné vďaka špeciálnej implementácii, ktorá minimalizuje počet operácií na procesore. Skompilovaná funkcia má na procesore typu x86 iba okolo 50 inštrukcií. Aj tieto inštrukcie sú vyberané tak, aby boli rýchle. Názov *MurMurHash* vznikol z anglických slov *multiply* a *rotate* (vynásob a urob cyklický posun), ktoré sú v programe veľakrát opakované. Na druhej strane, keďže nie je kryptografická, pre túto hešovaciu funkciu sa podarilo nájsť viacero kolízií a aj spôsob ich systematického generovania [3].

Zároveň však spĺňa viacero užitočných vlastností, vďaka, ktorým má rôzne aplikácie. Autori na stránke, ktorá popisuje alogritmus *MurMurHash* tvrdia, že je odolná na náhodné kolízie a podľa experimentov má rovnomerné rozdelenie. Funkciu *MurMurHash* je možné použiť na deduplikáciu, je používaná v hešovacích tabuľkách a podobne.

2.1.3 xxHash

Algoritmus *xxHash* [6] je zameraný na rýchlosť výpočtu. Tak ako aj zvyšné, doteraz spomenuté hešovacie funkcie, nie je kryptograficky bezpečná a tým dokáže byť výrazne rýchlejšia. *xxHash* zastrešuje viacero rýchlych hešovacích funkcií, každý variant má rôznu veľkosť výstupu: 32, 64 alebo 128 bitov. Je využívaná v rôznych známych nástrojoch, knižniciach, databázach a ďalších aplikáciách aj vďaka implementáciám vo väčšine bežne využívaných programovacích jazykoch.

Okrem toho, že patrí medzi jedny z najrýchlejších hešovacích funkcií, je dokonca rýchlejšia ako *MurMurHash* a v niektorých experimentoch až takmer 50-krát rýchlejšia ako *MD5*. Zároveň však podobne ako funkcia *MurMurHash* spĺňa vlastnosti, akými sú minimalizovanie náhodných kolízií a rovnomerné rozdelenie.

Autori algoritmu xxHash implementovali hešovacie funkcie *xxHash32* a *xxHash64* s 32 a 64 bitovými výstupmi tak, aby boli, čo najrýchlejšie. Prvá funkcia je určená pre 32 bitové a druhá pre 64 bitové architektúry, využívajúc 32 bitové respektíve 64 bitové operácie. Všetky implementácie používajú iba rýchle operácie: násobenie prednastavenými konštantami - prvočíslami, sčítavanie, bitový posun a bitové logické operácie. Najnovší algoritmus *xxHash3* s 64 a 128 bitovými výstupmi, optimalizuje rýchlosť výpočtu aj na základe veľkosti vstupných dát. Algoritmus delí dáta na 3 veľkosti, malé od 0 do 16 bajtov, stredné od 17 do 240 bajtov a veľké od 241 bajtov. Finálna implementácia teda obsahuje tri rôzne podimplementácie, ktoré sa spustia podľa veľkosti vstupu.

2.2 Návrh nástroja Dedupnetgo

Jednou z motivácií práce je deduplikácia v súborových systémov a preskúmanie redundancie dát v systémoch a ich vlastností. Ako sme už v prvej kapitole spomínali, implementácia modulov do súborového systému vyžaduje množstvo práce aj pre väčšie tímy odborníkov. Preto sme sa rozhodli vytvoriť nástroj na deduplikáciu dát pri prenose cez sieť, ktorý pri návrhu umožňuje väčšiu flexibilitu oproti obmedzeniam, ktoré môžu vyplývať z architektúry súborového systému.

Rozhodli sme sa, že tento nástroj pomenujeme *Dedupnetgo*. Názov vznikol ako spojenie častí slov *Dedup*, ktorý používa v názvoch svojich deduplikačných softvérov veľa prác venujúcich sa deduplikácii, časti *net*, keďže sa cez nástroj posielajú dáta cez sieť a časti *go* v spojitosti s implementáciou v programovacom jazyku Go. Toto pomenovanie budeme používať aj v ďalších častiach textu.

Aby sme sa priblížili a skúmali aj vlastnosti deduplikácie v súborových systémoch, ktoré sú často obmedzené fixnou veľkosťou bloku, navrhujeme v práci nástroj s deduplikáciou s fixnou veľkosťou deduplikačných blokov. Vstupné dáta teda budeme deliť na

rovnako veľké časti a hľadať v nich duplikáty. Cieľom je, po nájdení duplikovaného bloku na vstupe prenosu, daný blok neprenášať celý odznova, ale na výstup prenosu posielať iba informácie potrebné na jeho rekonštrukciu.

Nástroj je teda rozdelený na dve podčasti: odosielateľa dát a prijímateľa. Obidve strany, aj prijímateľ a aj odosielateľ majú svoje špecifiká. Na základe zadaných vstupných parametrov je u odosielateľa nastavená adresa, na ktorú sa majú dáta poslať a prijímateľ na tejto adrese prijíma dáta.

V práci sa zameriavame aj na hešovacie funkcie a ich vplyv na deduplikáciu. Navrhujeme nástroj na prenos tak, aby bolo možné hešovaciu funkciu jednoducho zmeniť nastavením. Implementácia teda umožňuje jednoduchú výmenu hešovacej funkcie, bez zmeny hlavnej funkcionality nástroja a jednoduché pridanie nových hešovacích funkcií. Keďže hešovacie funkcie môžu mať rôzne veľkosti výstupov, ktoré potom ovplyvňujú pravdepodobnosť možných kolízií, nástroj dokáže pracovať s rôzne veľkými výstupmi hešovacích funkcií.

Hlavnou dátovou štruktúrou, na ktorej je nástroj *Dedupnetgo* postavený je hešovacia tabuľka. Ako kľúče do hešovacej tabuľky sa používajú heše blokov. Pre lokalizáciu bloku a ďalšiu analýzu duplikátov zapisujeme do tejto tabuľky, ako hodnotu posunutie (*offset*) od začiatku súboru, určujúce miesto kde sa blok s daným hešom - kľúčom začína vo vstupnom súbore. Aby sme vedeli analyzovať všetky výskyty duplikátov, do hešovacej tabuľky ukladáme ich všetky príslušné posunutia, teda hodnoty v tabuľke sú jednorozmerné zoznamy posunutí v súbore.

V *Dedupnetgo* je používaná jedna hešovacia tabuľka na strane odosielateľa. Odosielateľ postupne číta vstupné dáta a delí ich na bloky. Pre každý blok je vypočítaný heš, ktorý je použitý ako kľúč do hešovacej tabuľky. Ak sa ešte v hešovacej tabuľke tento kľúč nenachádza, pridá sa do nej s hodnotou - posunutím na mieste, kde daný blok začína a celé dáta sa pošlú prijímateľovi. Prijímateľ prijatý blok dát spracuje a uloží si ho do výstupného súboru. V opačnom prípade, ak už sa v hešovacej tabuľke nachádza hodnota s kľúčom - hešom práve spracovávaného bloku, prijímateľovi sa odošle iba posunutie - hodnota z hešovacej tabuľky. Keďže sa heš bloku u odosielateľa v hešovacej tabuľke nachádzal, znamená to, že už raz bol blok s takýmito dátami prijímateľovi odoslaný. Teda prijímateľ má tento blok taktiež aspoň raz zapísaný a to na mieste začínajúcim posunutím odoslaným odosielateľom. Prijímateľ skopíruje dáta, určené od miesta určeného prijatým posunutím až po miesto určené posunutím plus veľkosťou bloku, na ktoré sa dáta delia a zapíše ich na koniec aktuálne zapisovaných dát. Znázornenie hešovacej tabuľky popisovanej v tejto časti sa nachádza v tabuľke 2.1

Tabuľka 2.1: Príklad hešovacej tabuľky používanej na strane odosielateľa.

Kľúč (heše blokov)	**Hodnota (posunutia začiatkov blokov v súbore)**
heš1	posunutie1, posunutie2, posunutie3
heš2	posunutie4, posunutie5, posunutie6, posunutie7, posunutie8
heš3	posunutie9

2.2.1 Nastavenia nástroja

Viacero častí prenosu dát je v nástroji nastaviteľných, pričom odosielateľ a prijímateľ majú parametre odlišné. Začneme popisom parametrov a prepínačov odosielateľa. Prvým a zrejmým parametrom je cieľová adresa, kam sa majú dáta posielať. Ďalším parametrom je samotný zdroj dát, ktorý sa bude presúvať. Keďže sa v práci venujeme aj vplyvu veľkosti blokov, podľa ktorých sa deduplikuje, *Dedupnetgo* má aj celočíselný parameter nastavujúci túto vlastnosť. Pri prenose cez *Dedupnetgo* sa heše blokov ukladajú do pamäte, ktorej veľkosť je tiež možné obmedziť celočíselným parametrom. Posledným parametrom, taktiež s obmedzeným počtom možných vstupov, je parameter na nastavenie hešovacej funkcie určenej na hešovanie blokov používaných v deduplikačnej hešovacej tabuľke.

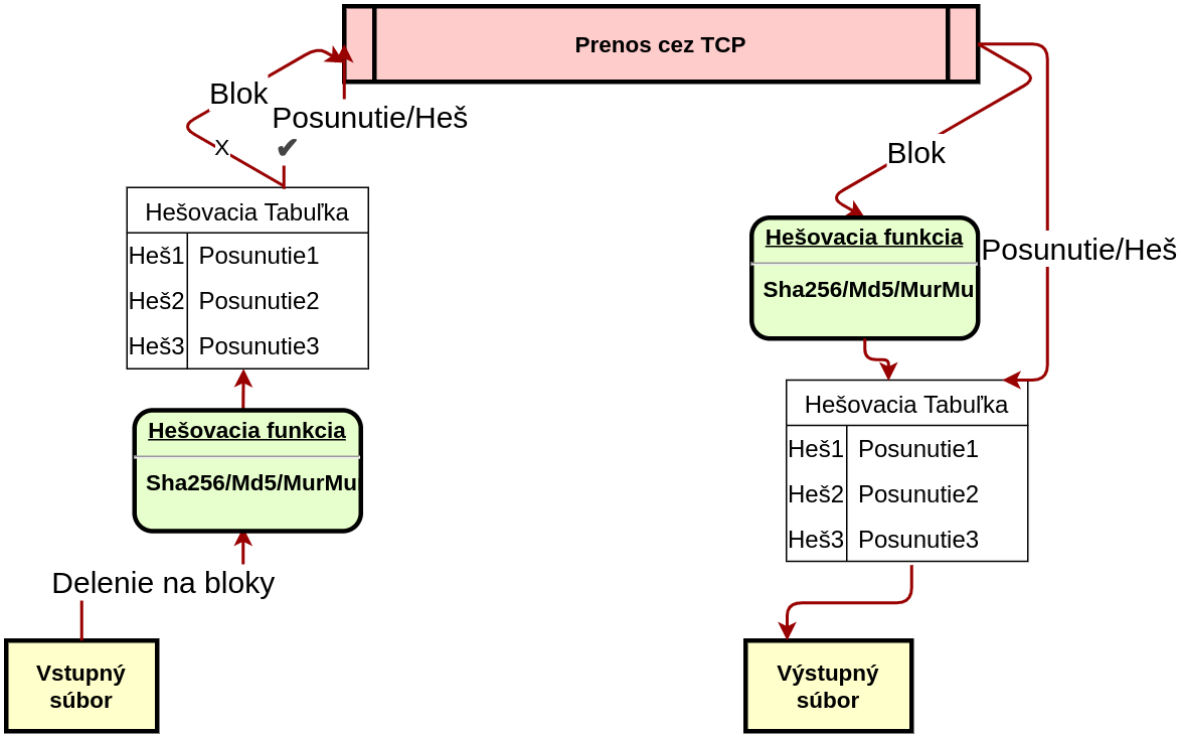
Okrem parametrov má strana odosielateľa v *Dedupnetgo* ďalšie prepínače, ktoré upravujú prenos. Prepínač pre štandardný vstup umožňuje nastaviť ako vstupné dáta štandardný vstup, následne nie je potrebné pridávať parameter so zdrojom dát. Ďalší prepínač umožňuje detegovať kolízie hešovacích funkcií pri deduplikácii. Detekcii kolízií sa budeme venovať ešte v ďalšej časti textu. Prepínač **sendHash** upravuje, čo sa prijímateľovi posiela v prípade duplikovaného bloku, buď posunutie duplikovaného bloku alebo jeho heš. Nakoniec, prepínač **output_seeks** umožňuje na výstup vypísať údaje o prenose, ktoré následne budeme analyzovať a popisovať v podčasti Analýza výstupných dát.

Prijímateľ má menej parametrov a prepínačov ako odosielateľ. Okrem základného parametra s adresou, kde prijímateľ bude prijímať dáta, má ešte parameter na určenie miesta, kam sa budú dáta zapisovať. Zvyšné parametre sú prenesené na začiatku prenosu od odosielateľa, keďže kvôli korektnosti prenosu a deduplikácie musia byť zhodné, a preto sú automaticky nastavené aj u prijímateľa.

Špeciálnym nastavením je možnosť spustiť detekciu kolízií. Keďže niektoré heše, ktoré v *Dedupnetgo* používame nie sú kryptograficky bezpečné, môžu pri hešovaní blokov nastať kolízie hešov. V prípade prenosu by to znamenalo chybný prenos dát, prijímateľ by deduplikované dáta nesprávne zrekonštruoval a ani by sa o tom nedozvedel. Po spustení nastavenia detekcie kolízií, sa tomuto problému predchádza porovnávaním dát blokov. Ak je heš bloku zhodný s hešom iného bloku, tak sú dáta týchto dvoch blokov porovnané bajt po bajte na zhodu. Ak sa nezhodujú, ide teda o kolíziu hešov, prijíma-

teľovi nie je poslané posunutie v súbore alebo heš, ale dáta samotné. Vďaka tomu sa dosiahne to, že aj napriek kolízii hešov blokov prijímateľ obdrží dáta korektne.

Ďalšie informácie ako nástroj nainštalovať, spustiť a nápovedy k prepínačom a používaniu nástroja sú uvedené v Prílohe A 4.2.5.



Obr. 2.1: Vizualizácia implementácie nástroja *Dedupnetgo*.

2.2.2 Komunikačný protokol

V rámci nástroja *Dedupnetgo* sme navrhli aj špecializovaný komunikačný protokol určený na prenos deduplikovaných dát. Jedným z cieľov nástroja je aj minimalizácia objemu prenesených dát od odosielateľa k prijímateľovi. Na základe toho je navrhovaný aj samotný prenosový protokol, ktorý by mal minimalizovať objem prenesených dát cez sieť ale zároveň by mal byť rýchly a neprerušovaný.

Prenášané dáta sú delené na nami navrhnuté bloky. Každý blok má svoj typ, podľa ktorého prijímateľ určuje, čo s daným blokom vykonať. Bloky majú premenlivú veľkosť určenú metadátami a predošlými prijatými blokmi. Bloky sú rozdelené na 2 časti: hlavičku s metadátami a telo s dátovým obsahom. Hlavička je na začiatku bloku a je tvorená práve 1 bajtom, za ktorým nasleduje telo bloku s typom určeným podľa typu hlavičky.

Na základe hlavičky, rozdeľujeme bloky na nasledujúce typy:

- Blok typu posunutie v súbore, pri duplikovaných blokoch sa odosiela posunutie tohto duplikovaného bloku.

- Blok typu blok dát, v prípade, že blok nie je duplikovaný, alebo bol prvýkrát prečítaný, odosielateľ pošle prijímateľovi celý blok dát.

- Blok typu posledný blok dát, keďže veľkosť prenášaných dát nemusí byť násobkom veľkosti bloku, pri prenášaní poslednej časti dát môže byť prenesený blok dát s menšou veľkosťou ako ostatné. Takýto blok dát musí byť špeciálne označený, aby ho prijímateľ vedel spracovať.

- Blok typu koniec prenosu. Na konci prenosu je prijímateľovi tento koniec oznámený špeciálnou hlavičkou.

Komunikácia odosielateľa a prijímateľa prebieha podľa nasledujúceho opisu. Na začiatku komunikácie odosielateľ pošle prijímateľovi nastavenia o prenose, veľkosť bloku a ďalšie parametre, ktoré sa nastavujú iba u odosielateľa. Následne sa začne samotný prenos. Výstupné dáta sa u odosielateľa postupne čítajú a delia na bloky s veľkosťou nastavenou podľa parametra. V prípade prvýkrát videného bloku sa na druhú stranu pošle, v závislosti od nastavenia, posunutie alebo heš. Ak sa ide poslať posledný blok, pošle sa blok s hlavičkou posledný blok. V tele tohto bloku je na začiatku dĺžka posledného bloku dát, aby prijímateľ vedel, koľko dát má očakávať a nakoniec samotné dáta. Po prijatí posledného bloku prijímateľ odošle odosielateľovi blok s hlavičkou koniec prenosu, aby mohla byť s odosielateľom korektne ukončená komunikácia.

2.3 Analýza výstupných dát

Ako sme už v tejto kapitole spomenuli, *Dedupnetgo* okrem prenosu dát s deduplikáciou poskytuje aj výpisy o deduplikácii. V našej práci následne tieto dáta spracovávame a analyzujeme, už mimo prenosu dát a mimo nástroja *Dedupnetgo*.

Nástroj poskytuje 2 druhy výstupov. Prvý, kratší, je určený pre bežného používateľa a zhrnie základné štatistiky o dokončenom prenose. Popisuje, koľko bolo celkom poslaných dátových blokov a koľko bolo poslaných posunutí v súbore a ich percentuálne vyjadrenie. Druhý, dlhší a konkrétnejší výstup je určený na ďalšiu analýzu. Tento výstup je tvorený dvojrozmerným zoznamom kladných celých čísel. Každý zoznam reprezentuje jeden blok duplikovaných dát. Hodnoty v týchto zoznamoch sú posunutia - miesta v súbore, kde sa daný blok nachádza.

Okrem nástroja *Dedupnetgo* sme v práci navrhli aj nástroje na analýzu spomínaných výstupov z *Dedupnetgo*. Keďže porovnávame hešovacie funkcie pri deduplikácii, veľkosti deduplikačných blokov a ďalšie parametre, porovnávame ich tak, že nástroj *Dedupnetgo* automatizovane spúšťame viackrát na rôznych typoch dát s rôznymi nastaveniami.

Primárne sme sa zamerali na porovnanie týchto parametrov:

- Vplyv hešovacej funkcie. Zaujíma nás, aký veľký vplyv má na rýchlosť prenosu použitá hešovacia funkcia, ktorá z nich je najrýchlejšia alebo najpomalšia.

- Početnosti duplikátov. Zaujímajú nás, početnosti duplikátov, ktoré sa opakovali v rovnakom počte. Aký vplyv na tieto početnosti má typ dát alebo veľkosť bloku, na ktoré sa dáta delia.

- Vzdialenosť medzi dvoma nasledujúcimi duplikátmi. Koľko blokov sa nachádza medzi dvoma blokmi, ktoré sú duplikátmi niektorých iných blokov a podobne ako pri početnosti duplikátov, aký vplyv má na tieto vzdialenosti typ dát a veľkosť deduplikačných blokov.

- Rozloženie duplikátov v dátach. Zaujíma nás, ako sú duplikované dáta rozložené po dátach, či sú v niektorých častiach dát duplikáty častejšie, alebo sú rovnomerne rozložené cez celé dáta.

- Ušetrený prenos. Na záver zisťujeme, ako efektívny je prenos dát cez nástroj *Dedupnetgo* a aký objem dát sa pri ich prenose podarilo deduplikáciou s fixnou veľkosťou deduplikačného bloku ušetriť, ak budeme brať do úvahy aj metadáta využívané pri prenose cez *Dedupnetgo*.

Kapitola 3

Implementácia riešenia

V tejto časti sa budeme venovať konkrétnej implementácii deduplikácie a analýzy duplikovaných dát, ktoré sme zvolili v našom riešení. Spomenieme využité nástroje, programovacie jazyky a knižnice. Budeme sa venovať aj komplikáciám a problémom, ktoré počas implementácie vznikli a ako sme ich riešili.

3.1 Implementácia nástroja Dedupnetgo

Na implementáciu nástroja *Dedupnetgo* sme využili programovací jazyk Go [18], niekedy označovaný aj *Golang*. Ide o pomerne nový programovací jazyk, vyvinutý v roku 2007 firmou Google. Poslednou dobou sa stáva čoraz populárnejším a je využívaný na vývoj webových aplikácií, cloudovej infraštruktúry a ďalších aplikácii z rôznych oblastí.

Go je kompilovaný jazyk, v porovnaní s inými jazykmi, s nadpriemerne rýchlym kompilovaním. Zároveň je to jazyk, ktorý má *garbage collection*. Syntaxou je veľmi podobný jazyku C a jedným z hlavných cieľov dizajnu tohto jazyka je dosiahnuť jednoduchosť, minimalizmus a ľahkú čitateľnosť. Zo známych projektov, ktoré sú naprogramované, z veľkej časti v jazyku Go, sú napríklad projekty a aplikácie *Docker*, *Kubernetes* alebo *Grafana*.

V porovnaní s v najbežnejšie používanými programovacími jazykmi súčasnosti, je Go neštandardný svojím prístupom k objektovo orientovanému programovaniu. Go nemá triedy a s nimi spojenú dedičnosť. Namiesto toho využíva štruktúry a rozhrania a ich skladanie. Na rozdiel od jazyka C, používa silné typovanie a využíva statickú typovú kontrolu. Go je tiež povestný svojou dobre použiteľnou implementáciou súbežnosti (*concurrency*). Využíva špeciálne konštrukty *goroutiny* - funkcie, ktoré sa konkurentne vykonávajú s inými funkciami a kanály, ktoré umožňujú komunikáciu medzi *goroutinami*.

Okrem vyššie spomenutých detailov sme sa pre implementáciu nástroja *Dedupnetgo* v jazyku Go rozhodli pre dobré možnosti jazyka Go v oblasti vývoja aplikácii pracu-

júcich so sieťami. Súčasťou Go je aj štandardná interná knižnica **net**. Knižnica **net** poskytuje viacero konštrukcií a funkcií vhodných pre prácu so sieťami. Od základnej práce s IP adresami až po komunikáciu cez TCP a UDP protokoly a DNS.

Go sme vybrali aj pre množstvo dostupných externých knižníc, z ktorých sme viaceré, v rámci *Dedupnetgo*, použili. Rozširovanie a zverejňovanie nových knižníc je v Go celkom jednoduché, priamo cez gitové repozitáre. Ďalším pozitívom je jednoduchá práca s externými závislosťami, ktorých správa je tiež základnou súčasťou jazyka.

V rámci implementácie sme využili viacero hešovacích funkcií, z ktorých niektoré sme už opísali v časti Návrh riešenia 2. Podobne, ako konštrukty pre sieťovú komunikáciu z knižnice **net**, aj niektoré hešovacie funkcie, ktoré sú v *Dedupnetgo* dostupné, pochádzajú priamo z interných knižníc jazyka. Konkrétne sú poskytnuté knižnicou **crypto** a používame z nej hešovacie funkcie *md5* a *sha256*. Hešovacia funkcia *xxhash* sa v knižniciach, ktoré sú súčasťou Go nenachádza, a preto sme vybrali externú knižnicu odporúčanú samotnými autormi algoritmu *xxhash*, implementovanú čiastočne v Go a čiastočne v jazyku symbolických inštrukcií. Rovnako sme vybrali externú knižnicu pre implementáciu algoritmu *murmurhash*. Hešovacie funkcie, ktoré boli spomenuté, sa dajú, tak ako sme požadovali v kapitole Návrh riešenia, jednoducho pridať a vymeniť cez nastavenia, keďže všetky implementujú štandardné Go rozhranie pre heše.

Pre nami navrhnutý komunikačný protokol v časti Návrh riešenia, sme sa na prenos cez sieť, rozhodli využiť protokol TCP. Vyberali sme z protokolov UDP a TCP. Nakoľko potrebujeme, aby boli prenesené všetky bloky dát a nami navrhnutý komunikačný protokol očakáva, že sú v poradí v akom boli odoslané, vybrali sme komunikačný protokol TCP. Prijímateľ bude teda adresovaný IP adresou a portom, na ktorom musí byť predom spustené prijímanie cez nástroj *Dedupnetgo*.

Vstupom aj výstupom pre nástroj je jeden súbor. Nástroj neakceptuje na vstupe priečinky a viaceré súbory naraz. Jedným z dôvodov je priamočiarejšia implementácia a možnosť takéto dáta spojiť do jedného súboru pomocou archívu. Spojený súbor sa dá vytvoriť, napríklad Linuxovým príkazom *tar* alebo iným archivovacím nástrojom v niektorých prípadoch aj s možnosťou kompresie. Podobný prístup sme zvolili aj pri šifrovaní prenášanej komunikácie. Prenos dát pomocou nástroja nie je šifrovaný z dôvodu väčšej flexibility. Používateľ v prípade potreby môže prenos cez TCP šifrovať, napríklad pomocou TLS alebo VPN.

3.1.1 Cobra

Nástroj *Dedupnetgo* bol implementovaný, ako nástroj pre príkazový riadok pre jednoduché a pohodlné použitie. Aj prijímateľ aj odosielateľ používajú ten istý príkaz **dedupnetgo**, avšak obe strany s inými nastaveniami.

Keďže rozsiahle časti vývoja aplikácií pre príkazový riadok sú totožné a bolo by to

zrejme aj v našom prípade, využili sme knižnicu *Cobra* [10], ktorá tieto spoločné časti implementuje. *Cobra* je knižnica v jazyku Go, ktorá poskytuje rozhranie umožňujúce vývoj aplikácií pre príkazový riadok. Používaná je aj populárnymi aplikáciami, akými sú napríklad Kubernetes alebo *Github CLI*.

Pomocou nástroja *cobra-cli* *Cobra* umožňuje vytvorenie kostry pre aplikáciu pre príkazový riadok v jazyku Go so základnými nastaveniami. Aplikácie pre príkazový riadok v *Cobre* sú štruktúrované pomocou príkazov a podpríkazov. Nástroj *cobra-cli* dokáže pre tieto časti vytvoriť predpripravené súbory v Go, kde každý príkaz a podpríkaz sú rozdelené do osobitných súborov.

Vo vytvorených súboroch pre príkazy a podpríkazy je možné pridávať argumenty a prepínače cez volania funkcií knižnice *Cobra*. Cez parametre týchto funkcií je možné nastaviť meno a skrátené meno argumentu alebo prepínača a tiež nastaviť, aký typ vstupu tieto prepínače prijímajú. Môže ísť napríklad o logickú hodnota alebo reťazec znakov, je tiež možné nastaviť predvolenú hodnotu a ďalšie detaily. Prepínače a argumenty, ktoré je možné cez *Dedupnetgo* nastavovať, sa dajú načítať do programu taktiež cez volania knižničných funkcií, na základe názvu a typu parametra alebo prepínača.

Nástroj *Dedupnetgo* je implementovaný pomocou *Cobra* ako jeden príkaz **dedupnetgo** a odosielanie je možné spustiť pomocou podpríkazu **send**.

Príkaz na odoslanie súboru *input.txt* cez *Dedupnetgo* s delením na bloky po 64 bajtoch na adresu 192.168.0.1:1234

```
$ dedupnetgo send 64 192.168.0.1 1234 input.txt
```

Prijímanie súboru je možné spustiť cez podpríkaz **receive**.

Príkaz na prijatie súboru cez *Dedupnetgo* 192.168.0.1:1234 so zápisom do súboru *output.txt*

```
$ dedupnetgo receive 192.168.0.1 1234 output.txt
```

3.1.2 Efektívnosť Dedupnetgo

Keďže jedným z cieľov práce je, aby bol nástroj použiteľný v praxi, zamerali sme sa aj na to, ako efektívny je nástroj *Dedupnetgo* pri prenose dát. Efektívnosť sme overovali aj pomocou profilovania (*profiling*). Pre Go bol Googlom vyvinutý nástroj a knižnica *pprof*, cez ktorú je možné vytvárať profilovacie záznamy a vizualizácie týchto záznamov.

Nástroj dokáže profilovať využívanie pamäte aj procesora. Profilovanie do profilovaného programu je umožnené pridaním volaní knižničných funkcií *pprof*, ktoré zbierajú a zapisujú dáta o využití procesora alebo pamäte do súboru. Nástroj *pprof* následne umožňuje tieto dáta zo súborov analyzovať a vizualizovať v grafoch a iných vizualizáciach.

Jednou z vizualizácií, ktoré nástroj *pprof* ponúka je graf volaní, ktorý je aj na obrázku 3.1. Do vrcholov tohto grafu sú okrem názvu zavolanej funkcie pridané dodatočné informácie, výsledky z profilovania.

V prípade obrázku 3.1 sú do vrcholov pridané informácie o názve funkcie a balíka z ktorej pochádza a z profilovania pamäte. V každom vrchole, reprezentujúcom vykonanú funkciu, z ktorej boli volané ďalšie iné funkcie sú napísané 2 čísla. Prvé určuje, koľko pamäte alokovala samotná funkcia reprezentovaná vrcholom. Druhé číslo určuje, koľko pamäte alokovali, všetky vrcholy v grafe pod ňou, teda funkcie, ktoré z nej boli zavolané. Pre pridanie dôrazu, funkcie, ktoré mali najväčšiu spotrebu pamäte sú vykreslené ako väčšie vrcholy s červeným pozadím.

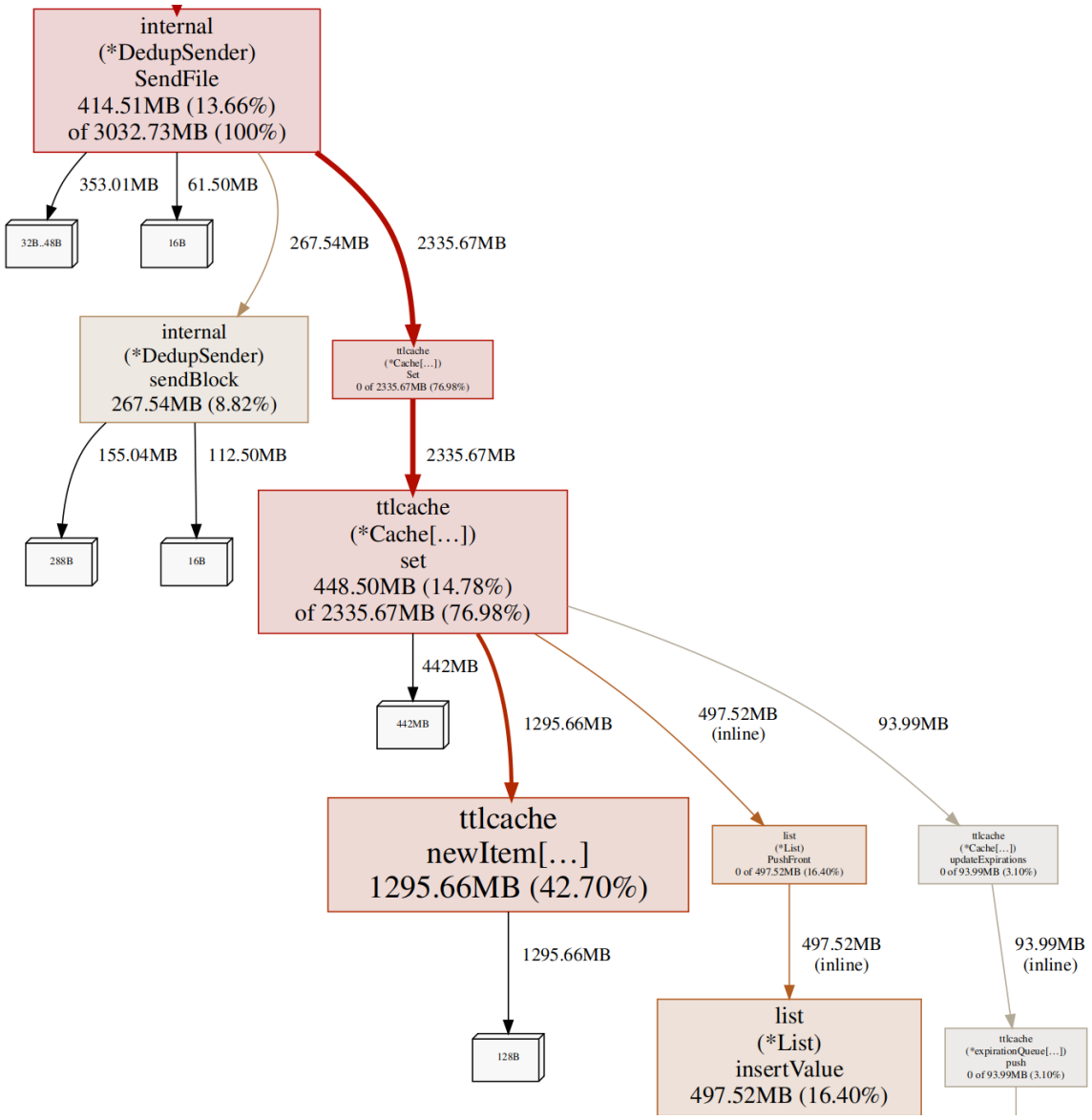
Prvý problém, s ktorým sme sa pri vývoji *Dedupnetgo* stretli, bol veľmi pomalý prenos, evidovaný najmä pri prenose väčších súborov. Na základe profilovania využívania procesoru *Dedupnetgo* počas prenosu dát, sme zistili, že veľká časť programu čaká na systémové volania, konkrétne na zápis a čítanie súborov a prenos cez TCP.

V jazyku Go bola pre tento problém vytvorená interná knižnica bufio. Táto knižnica rozširuje objekty, ktoré zapisujú alebo čítajú, napríklad zo soketu alebo súboru tak, že k zápisu alebo čítaniu pridáva vyrovnávaciu pamäť (*buffering*). S takto rozšíreným objektom sa pracuje ako s bežným objektom určeným na prácu so vstupno výstupnými operáciami s pridanými funkciami, určenými na prácu s vyrovnávacou pamäťou. Po pridaní *bufia* na objekty používajúce vstupno výstupné operácie, sa problém s dlhým čakaním na systémové volania minimalizoval a vyriešil.

Druhým problémom v efektívnosti nástroja bola vysoká spotreba pamäte odosielateľa. Na odhalenie príčiny, tak ako aj pri pomalom prenose, sme využili profilovanie, avšak tentoraz profilovanie využitia pamäte. Na obrázku 3.1 je uvedený výstup profilovania pamäte odosielateľa v *Dedupnetgo* pri prenose 4 gigabajtového videa. Na základe výstupov profilovania sme zistili, že príčinou veľkej spotreby pamäte bola samotná hešovacia tabuľka, v ktorej sú ukladané posunutia v súbore a heše blokov.

Objem využitej pamäte pri prenose sa na strane odosielateľa zväčšoval so zväčšujúcou sa veľkosťou prenášaného súboru a zmenšujúcou sa veľkosťou blokov, podľa ktorých sú dáta deduplikované. Keďže ide o základnú funkcionalita *Dedupnetgo*, nie je možné ju upraviť tak, aby spotrebovávala menej pamäte. Preto sme prišli s iným riešením.

Namiesto štandardnej hešovacej tabuľky z jazyka Go sme použili tabuľku z knižnice *jellydator/ttlcache* [8]. Na rozdiel od štandardnej hešovacej tabuľky, má táto špeciálna tabuľka možnosť nastaviť vymazávanie obsahu. Kľúč s hodnotou sa z tabuľky vymaže podľa nastavenia pri vytváraní tabuľky. Na toto nastavenie sú v *ttlcache* dve možnosti. Prvé, podľa času, ako dlho sa kľúč s hodnotou v tabuľke nachádzal. Je možné napríklad nastaviť, že kľúč s hodnotou bude z tabuľky odstránený po 30 sekundách od ich vloženia. Druhá možnosť, ktorú sme využili v implementácii nástroja, je nastavenie in-



Obr. 3.1: Výrez z grafovej vizualizácie profilovania využitia pamäte odosielateľa v *Dedupnetgo*.

formácie maximálne možnom počte kľúčov v tabuľke. Po prekročení tohto limitu sú z *ttlcache* automaticky odstránené kľúč a hodnota, ktoré sú najstaršie, teda boli vložené do *ttlcache* medzi prvými.

Využitie *ttlcache* a priebežné vymazávanie hešov a posunutí z tabuľky nenaruší samotný prenos. Po prenose dáta na strane odosielateľa aj prijímateľa budú rovnaké aj napriek priebežnému vymazávaniu hešovacej tabuľky. V prípade, že sa odosiela blok dát s hešom, ktorý bol z hešovacej tabuľky vymazaný, je poslaný znova ako celé dáta a uloží sa opätovne do hešovacej tabuľky. Prijímateľ preto prijme rovnaké dáta, ako bez priebežného vymazávania.

Jednou z nevýhod tohto riešenia je možný väčší objem prenesených dát, keďže namiesto posunutia, ktoré bolo z tabuľky vymazané sa prijímateľovi pošle celý blok dát.

Podľa experimentov na vybraných dátach, ktoré sme vykonali a bližšie sme ich popisovali v časti Overenie riešenia, vymazávanie z hešovacej tabuľky nemusí výrazne zväčšiť objem prenesených dát oproti, deduplikácii bez priebežného vymazávania. Tento rozdiel môže byť výrazne ovplyvnený aj algoritmom určujúcim, ktoré hodnoty z hešovacej tabuľky budú vymazané ako prvé.

3.1.3 Testovanie

Aby sme zabezpečili korektnosť prenosu dát a správania *Dedupnetgo*, tento nástroj sme počas jeho vývoja zároveň testovali a to dvoma spôsobmi.

Prvý spôsob sme využívali v priebehu implementácie jednotlivých častí nástroja. Tento typ testovania sa nazýva aj jednotkové testy (*unit testy*). Jednotkové testy nám v priebehu implementácie pomáhali priebežne odhaľovať a opravovať chyby v častiach implementácie.

Jazyk Go má v sebe zabudované časti, ktoré umožňujú a zjednodušujú tvorbu, spúšťanie a vyhodnocovanie jednotkových testov. Testy sa tvoria jednoducho, vytvárajú sa ako funkcie v súboroch končiacich *_test.go*. Všetky testy je potom možné spustiť jediným príkazom **go test**. Štandardná interná knižnica v Go, ktorá umožňuje automatizované testovanie, sa nazýva *testing*. Keďže jednou z filozofií Go je jednoduchosť a minimalizmus, viacero zložitejších testovacích konštruktov v Go chýba. Na testovanie preto využívame aj externú knižnicu *testify/assert* [7]. Túto knižnicu sme následne využívali na kontroly rovnosti, nerovnosti, chýb a ďalších testovacích faktorov.

Druhý spôsob testovania sme využívali na testovanie celého nástroja. Niekedy sa tento typ testovania nazýva aj *end-to-end* testovanie. Nástroj sme spúšťali a prenášali sme pomocou neho testovacie dáta rôznych veľkostí, rôznych typov a s rôznymi nastaveniami nástroja od veľkosti deduplikačného bloku až po rôzne nastavenia prepínačov. Aby testovanie bolo čo najjednoduchšie, obidve strany aj prijímateľa aj odosielateľa sme spúšťali na jednom počítači a v rámci jedného operačného systému. Dáta tak boli iba kopírované z jedného miesta na druhé. Dáta na testovanie sme používali tie, ktoré sú spomenuté v časti Overenie riešenia.

V prípade, že sa podarilo uskutočniť prenos, nástroj nevypísal, alebo neskončil chybou a zároveň na konci prenosu boli dáta prenesené korektne, test prebehol úspešne. Keďže prijímateľ aj odosielateľ boli na rovnakom počítači, prenesené dáta a odoslané dáta bolo možné jednoducho porovnať. Na toto porovnanie sme využili štandardný Linuxový príkaz **diff**, ktorý hľadá rozdiely v súboroch bajt po bajte. V prípade, že nebol žiaden rozdiel medzi odoslanými a prijatými dátami, môžeme predpokladať, že prenos prebehol korektne.

3.2 Analyzovanie výstupu

Ako sme popisovali v návrhu riešenia, informačné výstupy sme rozdelili na dva typy. Prvý, zjednodušený výstup pre používateľa, sme implementovali ako výpis obyčajného textu do štandardného výstupu *stdout* v príkazovom riadku. Nakoľko druhý, zložitejší výstup, môže byť pomerne veľký a nie je vhodný na čítanie používateľom bez použitia iných nástrojov. Z tohto dôvodu bol implementovaný, ako výpis do chybového výstupu *stderr*.

Keďže druhý výstup je určený na spracovanie inými nástrojmi, prispôsobili sme tomu aj jeho štruktúru. Zároveň cieľom tohto výstupu nie je len skutočnosť, aby bol výstup analyzovaný iba nami vytvorenými nástrojmi, ale aby bol jednoducho použiteľný aj pre iné nástroje. Na reprezentáciu tohto výstupu sme preto použili jeden z najznámejších štandardných formátov na reprezentáciu dát JSON. Výstup je teda tvorený jedným JSON objektom, ktorý obsahuje zoznamy posunutí, pričom každé posunutie v zozname označuje miesto, kde sa začínali bloky s rovnakými dátami.

Jazyk Go nie je v praxi bežne používaný na analýzu dát. Vďaka tomu, že výstupné dáta z nástroja sú v štandardnom formáte, na analýzu sme mohli využiť aj iný vhodnejší jazyk. Na analyzovanie sme využili programovací jazyk Python. Aj keď je v porovnaní s jazykom Go pomalší, najmä tým, že ide o interpretovaný jazyk, na druhej strane je vhodnejší na analyzovanie dát, a to pre lepšiu dostupnosť knižníc pre dátovú analýzu a jednoduchšiu manipuláciu s dátami.

V Pythone sme vytvorili aj program, ktorý viackrát spúšťa nástroj *Dedupnetgo* s rôznymi nastaveniami. Nastavenia a vstupné súbory tohto programu sú načítavané z konfiguračného súboru. Dajú sa v ňom nastaviť: vstupný a výstupný súbor, veľkosť deduplikačného bloku pri prvom spustení *Dedupnetgo*, o koľko a do akej veľkosti sa má v ďalších spusteniach zväčšovať a ďalšie nastavenia. Po jeho spustení, program postupne podľa konfigurácie spúšťa prenos dát, prijímateľa a odosielateľa, oboch v samostatných procesoch použitím knižnice jazyka Python - *subprocess*.

Ukážkový konfiguračný súbor pre program určený na analýzu duplikátov

```
[DEFAULT]
PathToDedupnetgo = ../dedupnetgo/dedupnetgo
InputFile = ubuntu.img
OutputFile = /dev/null
Host = localhost
Port = 12345
[WINDOW]
StartSize = 128
EndSize = 8194
StepSize = 128
```

```
PowerOfTwo = false
[OUTPUT]
Distributions = true
Graphs = true
```

Po dokončení prenosu medzi odosielateľom a prijímateľom, prijímateľ vypíše dáta o duplikátoch v spomínanom formáte JSON. Tento výstup je načítaný a spracovaný programom na analyzovanie duplikátov a ďalej reprezentovaný, ako zoznam zoznamov celých čísel.

Pre každý analyzovaný parameter popísaný v časti 2 Návrh riešenia, sme vytvorili osobitnú funkciu, ktorá sa mu venuje. Výstupom väčšiny týchto funkcií je grafické znázornenie, na ktoré využívame knižnicu *Matplotlib* [13]. Pri niektorých parametroch sme sa pokúšali odhadnúť, ku ktorej distribúcii sa dáta najviac približujú, a to pomocou knižnice *SciPy* [22]. Namerané dáta a výstupy z tejto analýzy popisujeme v časti 4 Overenie riešenia.

Kapitola 4

Overenie riešenia

Táto časť práce obsahuje nami namerané výsledky o redundancii dát vytvorené pomocou nástroja *Dedupnetgo*. Na vybraných vstupných dátach zhodnotíme význam jednotlivých vlastností duplikátov. Uvedieme najmä najvýznamnejšie výsledky našej práce, pričom ďalšie výsledky zverejňujeme v elektronickej prílohe, ku ktorej uvádzame informácie v Prílohe A 4.2.5.

4.1 Vstupné dáta

Jedným zo zámerov práce je preskúmať duplikáty v bežne používaných dátach a analyzovať ich vlastnosti. Zamerali sme sa na dáta, s ktorými sa bežní používatelia stretávajú pri typických operáciách pri práci s elektronickými technológiami.

Medzi najbežnejšie používané typy dát patria obrázky, texty alebo dokumenty a videá. Navyše, z každého zo spomenutých typov dát existujú viaceré formáty, ktorými môžu byť reprezentované. V rámci práce sme sa pokúšali nájsť vhodnú verejnú množinu dát, ktorá by obsahovala väčšie množstvo takýchto rôznorodých súborov.

Jednou z prvých ťažkostí s takýmto typom dát je, že by malo ísť o používateľské dáta, ktoré však je komplikované, z hľadiska rôznych licenčných a právnych dôvodov, využívať korektne na experimentovanie. Náročnosť vyhľadať takúto dátovú množinu sa zvyšuje aj vďaka tomu, že by mala byť rôznorodá. Väčšina voľne dostupných dátových množín je rovnakého formátu a najčastejšie je nejakým spôsobom štandardizovaná, napríklad v prípade obrázkov je použité rovnaké rozlíšenie. Dôvodom je, že použitie takýchto dátových množín je určené zväčša pre trénovanie umelej inteligencie a iných algoritmov.

Väčšinu spomenutých ťažkostí rieši dátová množina *Govdocs1* [11] od organizácie Digital Corpora. Tieto dáta sú primárne určené pre forenzný výskum a analyzovanie súborov. Dátová množina *Govdocs1* obsahuje 1 000 000 súborov rôznych typov, ktoré sú voľne distribuovateľné. Tieto dáta boli získané prehľadávaním .gov domén pomo-

cou vyhľadávačov Google a Yahoo. Vyhľadávané boli náhodné čísla a slová z Unixového slovníka. Následne boli z nájdených výsledkov stiahnuté súbory a boli pridané do dátovej množiny. Keďže ide o súbory z .gov domén nemali by podliehať uvedeným problémom s licenčnými a právnymi komplikáciami.

V dátovej množine sa nachádzajú najmä súbory typu jpg, doc, html, txt, pdf, csv a menší počet ďalších, teda dáta, ktoré sa najčastejšie používajú bežnými používateľmi. Množina dát je voľne dostupná na stiahnutie zo stránky organizácie Digital Corpora. Stiahnutie dát je umožnené z viacerých formátov: 1 000 priečinkov s 1 000 súbormi, 1 000 zip súborov s 1 000 súbormi alebo dáta rozdelené podľa vybraných typov súborov. V každom stiahnuteľnom formáte sú súbory pomenované číslom, napríklad 0000001.jpg. Ku každému súboru sú zároveň v ďalšom stiahnuteľnom zdroji poskytnuté informácie, kedy, odkiaľ, z akej adresy bol stiahnutý a ktorým vyhľadávačom bol vyhľadaný.

Dáta z Govdocs1 sme si, pomocou jednoduchého programu v Pythone (nachádza sa medzi pomocnými programami, ktoré sú spomenuté v Prílohe A 4.2.5 v priečinku *helpers*), rozdelili podľa nami vybraných typov do osobitných priečinkov: doc, html, jpg, ppt, ps, txt a xls, ktoré považujeme za jedny z najpoužívanejších v bežných kontextoch. Zároveň sme ich prerozdelili tak, aby každá podmnožina s rovnakým typom obsahovala podobný objem dát v bajtoch, aby veľkosť dát neskresľovala výsledky experimentov.

Dátová sada Govdocs1 neobsahuje videá, ktoré tiež tvoria veľkú časť bežne používaných dát. Preto sme dátovú podmnožinu s videami vytvorili z videí poskytovaných organizáciou Pixabay [1]. Pixabay zbiera a poskytuje médiá, najmä obrázky a videá, vytvorené používateľmi a zverejňované bez licenčných poplatkov a zmlúv, teda je ich možné použiť aj na experimentovanie v našej práci.

Všetky vytvorené podmnožiny pred experimentovaním, teda pred prenosom cez nástroj *Dedupnetgo*, zjednotíme do jedného súboru. Na spájanie súborov do jedného sme pracovali s viacerými alternatívami. Prvou bolo spojiť súbory pomocou Linuxového archivovacieho príkazu *tar*. *Tar* spája súbory do jedného súboru so špeciálnou štruktúrou určenou pre zaznamenávanie dát na magnetické pásky, napríklad na video a audio kazety alebo zálohovacie pásky.

Aj keď existuje viacero tar formátov, základnú štruktúru ukladania dát majú spoločnú: dáta zo súborov sú ukladané jeden za druhým a pred každým súborom sú vložené metadáta o danom súbore. Metadáta tvoria takmer vo všetkých formátoch informácie o názve súboru, vlastníkoch, veľkosti a ďalšie informácie, ktoré sú všetky zapisované ASCII znakmi. Tieto metadáta sú zarovnané na 512 bajtov null ASCII znakmi alebo medzerami, podľa používaného tar formátu. Za metadátami nasledujú dáta súborov, ktoré sú tiež zarovnané na násobok 512 bajtov, null ASCII znakmi alebo medzerami.

Ďalšou možnosťou ako spojiť súbory do jedného, je uložiť ich do súborového systému. Súborový systém môže byť následne reprezentovaný ako súbor na disku. V práci

sme používali najčastejšie využívaný súborový systém na Linuxe ext, konkrétne verziu 4. Aby redundancia v dátach bola čo najmenej ovplyvnená dodatočnými dátami súborového systému, súborový systém sme vytvárali s vypnutým žurnálovaním. To by tvorilo časť súborového systému a mohlo by skresliť výsledky analýz duplikátov.

Ďalším spôsobom, ktorým sa dá spojiť súbory do jedného, je zapísať ich bajt po bajte za sebou do jedného súboru. Pri použití tohto spôsobu, narozdiel od ostatných, by nemuselo byť možné dáta rozdeliť naspäť na pôvodné súbory. Zároveň, oproti zvyšným dvom spôsobom, nie sú dáta v súboroch zarovnané podľa rovnako veľkých blokov (v tar sú to 512 bajtové bloky), čo pri deduplikácii s fixnou veľkosťou blokov pri štruktúrovaných dátach môže ovplyvniť deduplikáciu. To je možné vyriešiť zarovnaním veľkosti súborov na násobok veľkosti deduplikačného bloku, napríklad nulovými bajtami. To by zároveň malo zarovnať dáta tak, aby sa zachovali možné duplikáty v štruktúrovaných dátach a zároveň nepridať navyše veľa duplikátov do dát.

Pre tento postup sme vytvorili jednoduchý program v Pythone (nachádza sa medzi pomocnými programami, ktoré sú spomenuté v Prílohe A 4.2.5 v priečinku *helpers*), ktorý všetky súbory v priečinku na koncoch doplní nulovými bajtami na najmenší násobok podľa prednastavenej veľkosti. Následne všetky doplnené súbory v priečinku spojíme do jedného pomocou príkazu *cat*.

Posledný typ dát, ktorému sme sa venovali, boli obrazy virtuálnych strojov (*virtual machine*), presnejšie, obrazy ich virtuálnych diskov. Na tieto experimenty sme využili novú inštaláciu Linuxového operačného systému Lubuntu založenom na Ubuntu, ktorý patrí medzi najpoužívanejšie Linuxové distribúcie. Nová inštalácia nie je prázdna, obsahuje základný predinštalovaný softvér pre základnú prácu s počítačom: kancelársky balík na tvorbu a úpravu dokumentov, základný softvér na pozeranie videí a fotografií a ďalšie najčastejšie používané programy.

Pri experimentovaní s virtuálnym diskom sme pracovali s viacerými formátmi tohto disku, ktorým sa budeme bližšie venovať priamo pri analýze týchto dát v nasledujúcich častiach.

Pre zhrnutie, na experimentovanie sme použili typy dát so súborovými príponami: doc, hmtl, jpg, pdf, ppt, ps, txt, xls z dátovej množiny *Govdocs1*, avi, mov, mp4 od organizácie Pixabay a súbory virtuálnych diskov s operačným systémom Lubuntu.

4.2 Namerané dáta

So vstupnými dátami popísanými v predošlej časti sme vykonávali experimenty pomocou nástroja *Dedupnetgo*. Analyzovali sme ich pomocou programu, ktorý sme vyvinuli na analýzu výstupu z *Dedupnetgo* a je popísaný v kapitole Implementácia riešenia. V nasledujúcich častiach popíšeme, aké výsledky sa dosiahli na jednotlivých nami navr-

hnutých parametroch uvedených v časti Analýza výstupných dát 2.3 z kapitoly Návrh riešenia.

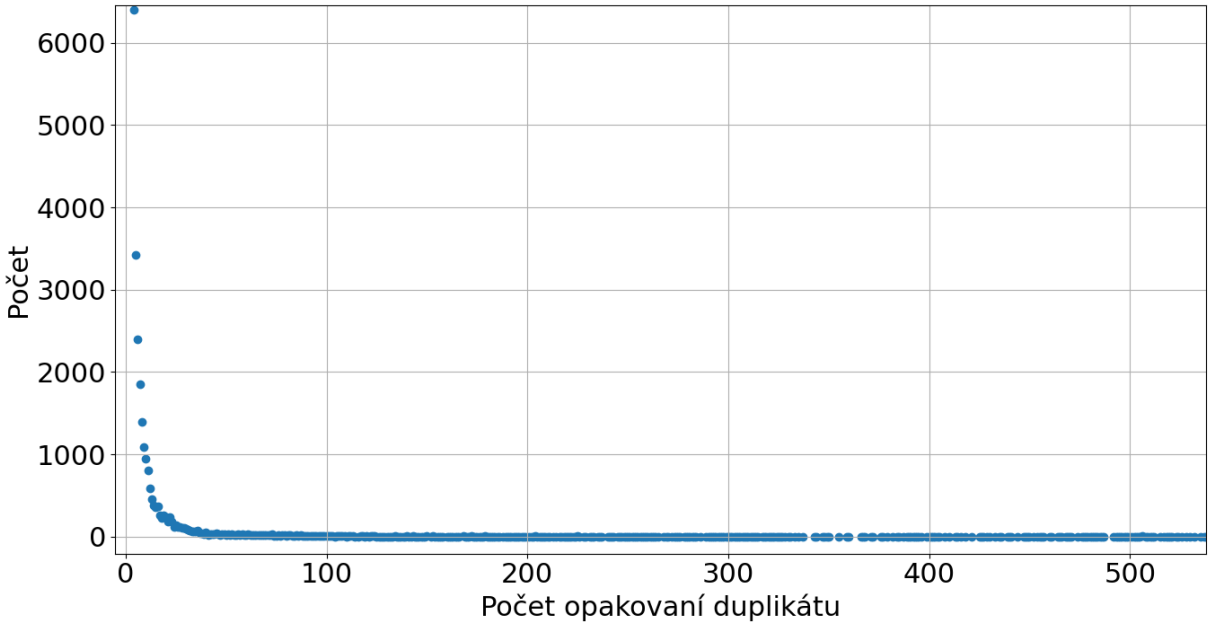
4.2.1 Početnosti duplikátov

Prvým parametrom, ktorému sme sa venovali, boli početnosti opakovaní rovnakých duplikátov. Pri tomto parametri rátame počty identických blokov, ktoré sa opakovali v rovnakom počte. Pre upresnenie, čo tento parameter znamená, uvedieme zjednodušený príklad. Symboly gréckej abecedy budú reprezentovať bloky, každý rovnaký symbol reprezentuje blok s rovnakými dátami. Ak teda máme vymyslené dáta rozdelené na bloky $\alpha, \alpha, \beta, \alpha, \gamma, \beta, \delta, \delta, \psi, \xi$, tak početnosti duplikátov, ktoré sa opakovali 1-krát boli 3, a to boli: γ, ψ, ξ, teda hovoríme, že ich početnosť je 3, duplikáty, ktoré sa opakovali 2-krát boli δ, β, teda ich početnosť bola 2 a duplikáty, ktoré sa opakovali 3-krát boli iba α teda ich početnosť bola 1 a duplikáty, ktoré by sa opakovali 4 a viackrát neboli žiadne, teda ich početnosť bola 0.

Okrem vyjadrenia týchto početností pre všetky nami vybrané súborové typy roztriedené z dátovej sady pre rôzne veľkosti blokov, sme tieto výsledné dáta porovnávali s niektorými známymi distribučnými funkciami. Využiteľnosť týchto výsledkov a získanej distribučnej funkcie môže napríklad pomôcť pri návrhu efektívnejšieho ukladania metadát o duplikátoch, či už v rámci súborového systému alebo rôznych deduplikačných nástrojov.

Na odhadnutie distribučnej funkcie, ktorá by mohla reprezentovať početnosti v rovnakom počte opakovaných duplikátov, sme využili funkciu fit z Python knižnice *stats/SciPy*. Funkcia *fit* sa pokúsi čo najbližšie odhadnúť rozdelenie zadaných dát pomocou metódy maximálnej vierohodnosti. Funkciu fit sme využili pre všetky dostupné distribučné funkcie v spomínanej knižnici, pričom v nami použitej verzii tejto knižnice išlo o 106 rôznych distribučných funkcií. Nakoniec sú všetky odhadnuté distribučné funkcie ešte porovnávané pomocou výpočtu reziduálneho súčtu štvorcov (*Sum of Square Errors*) od vstupných dát pre každú distribučnú funkciu. Distribučná funkcia prispôsobená dátam, ktorá má túto metriku najmenšiu je vybraná ako tá, ktorá podľa tejto metódy najlepšie odhaduje distribučnú funkciu dát.

Z nameraných dát vyplýva, že početnosti duplikátov s rovnakým počtom opakovaní sú najväčšie pri malom počte opakovaní a rýchlo klesajú pri vyšších počtoch, príklad vizualizácie tohto experimentu je možné vidieť na grafe 4.1. Tomu zodpovedajú aj distribučné funkcie, ktoré sa najviac blížili nameraným dátam. Pri experimentovaní sme používali najprv 32 bajtové bloky, podľa ktorých sa deduplikovalo. V prípade 32 bajtových blokov sa medzi najvhodnejšími odhadmi vyskytovali rozdelenia Lomax, polovica Cauchyho rozdelenia (*Half-Cauchy*), polovica zovšeobecneného normálneho rozdelenia (*Generalized normal distribution*) a exponenciálne rozdelenie. Výsledky pri



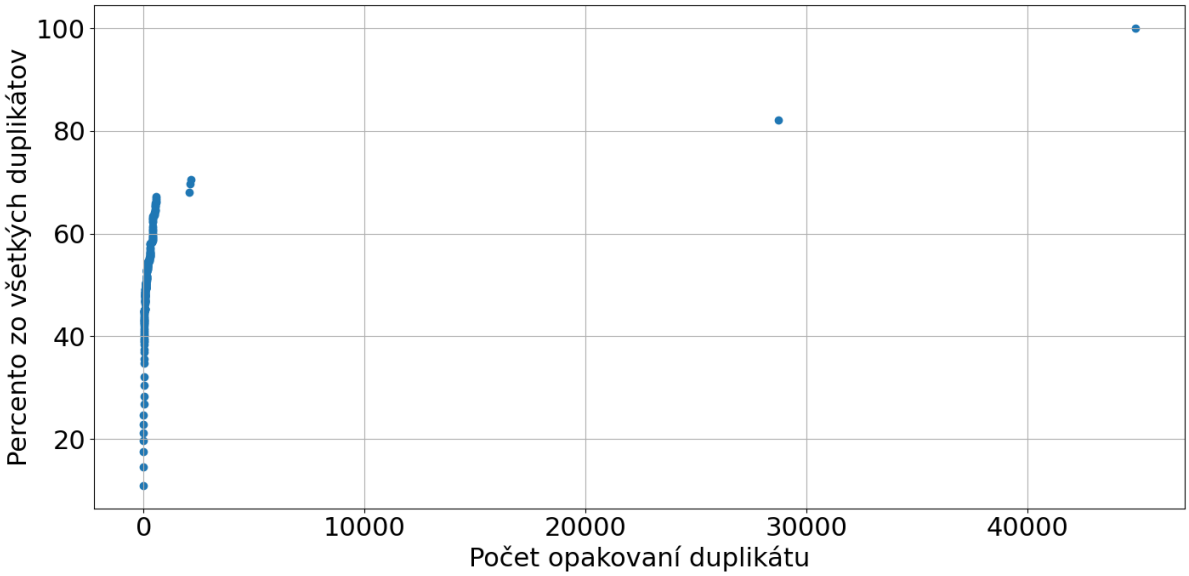
Obr. 4.1: Početnosti rovnako veľakrát opakovaných duplikátov na množine súborov txt s delením dát po 32 bajtoch, formát spájania súborov s doplnením nulovými bajtami. Uvedený obrázok je iba výrez z celého grafu.

128 bajtovom bloku boli podobné. Ďalšie výsledky a vizualizácie sú popísané v Prílohe A 4.2.5.

Ako príklad tiež uvádzame vizualizáciu kumulatívneho počtu duplikátov na množine obrázkov typu jpg 4.2. V tomto grafe sú na x-ovej osi znázornené počty opakovaní duplikátu a na y-ovej osi je znázornené percento všetkých duplikátov, ktoré tvoria duplikáty s menším alebo rovným počtom opakovaní, ako je hodnota na x-ovej osi. V tomto prípade sme na spájanie súborov do jedného využili formát tar.

Na grafe je možné vidieť, že okolo 70 percent všetkých duplikátov sa opakuje menej ako 1 000-krát, čo je pravdepodobne spôsobené kompresiou, ktorá sa v jpg súboroch využíva. Najčastejšie opakovaný duplikát, ktorý sa opakoval až 50 000-krát je tvorený bajtami v hexadecimálnej reprezentácii **\x00**. Ide o bajty s najväčšou pravdepodobnosťou s významom null, ktoré z väčšej časti pochádzajú z hlavičiek a zarovnaní na násobok 512 bajtov. Preto bol duplikát s takýmto obsahom bajtov pomerne častý takmer vo všetkých nami skúmaných dátových podmnožinách, ktoré boli spájané nástrojom tar. Jeden z duplikátov s najväčším počtom opakovaní tiež obsahoval text **ustar**, čo je názov jedného z formátov archívu tar. Druhý najčastejšie sa vyskytujúci duplikát bol v ASCII reprezentácii tvorený medzerami.

V porovnaní s formátom spájania súborov do jedného pomocou súborového systému ext4, bol rozdiel najmä v počte duplikátov tvorených nulovými bajtami. Súbor



Obr. 4.2: Početnosti rovnako veľakrát opakovaných duplikátov na množine súborov jpg s delením dát po 32 bajtoch, formát spájania súborov tar, kumulatívny graf.

s obrazom súborového systému bol o približne 100 megabajtov väčší oproti dátam v tar formáte, ktoré mali okolo 300 megabajtov. Väčšina z týchto dát, ktoré boli navyše, boli tvorené práve spomínanými nulovými bajtami.

Toto zväčšenie súboru môže mať viacero príčin. Jednou z príčin môže byť, že ext4 nepodporuje spájanie čiastočne využitých blokov (*tail merging*), a teda časť blokov zostáva nevyužitých a zaplnených nulovými bajtami. Ďalšou z možných príčin je, že časť z ext4 súborového systému je vyhradená pre správcu (štandardne 5%), ktorá tiež môže byť tvorená nulovými bajtami.

Výsledky použitia formátu, ktorý zarovnával dáta na 32 bajtov nulami a následne ich spájal, neboli veľmi odlišné od výsledkov za použitia tar. V tomto formáte sa vyskytovalo o niečo menej nulových duplikátov a duplikát s najväčším počtom opakovaní bol v ASCII reprezentácii tvorený medzerami.

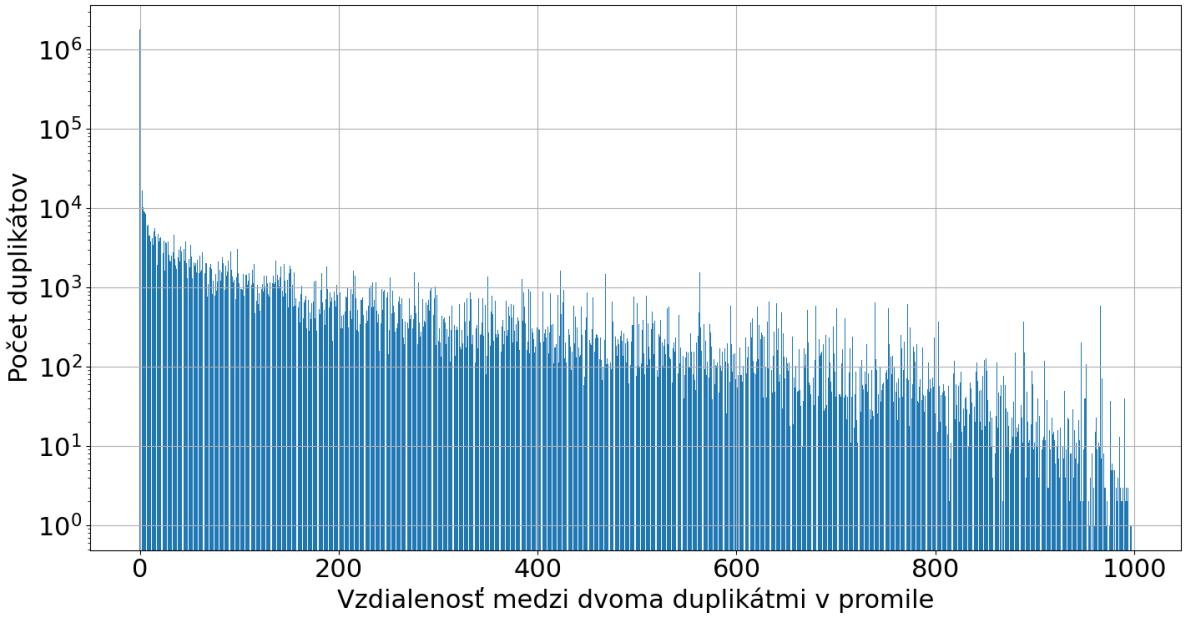
4.2.2 Vzdialenosti medzi duplikátmi

Ako sme spomínali v časti Návrh riešenia, ďalší parameter, ktorému sme sa venovali, boli vzdialenosti medzi duplikátmi. Túto problematiku sme analyzovali z viacerých pohľadov.

Prvým spôsobom bolo meranie vzdialeností medzi dvoma nasledujúcimi duplikátmi, teda blokmi s rovnakým obsahom. Tento výpočet vykonávame prechádzaním cez všetky duplikáty, kde pre každý duplikát prechádzame posunutia, inak povedané všetky miesta, kde sa bloky s rovnakou hodnotou začínajú. Dve nasledujúce posunutia pre jeden dup-

likát odčítame od seba a tým získame ich vzdialenosť v bajtoch. Následne rátame početnosti jednotlivých vzdialeností.

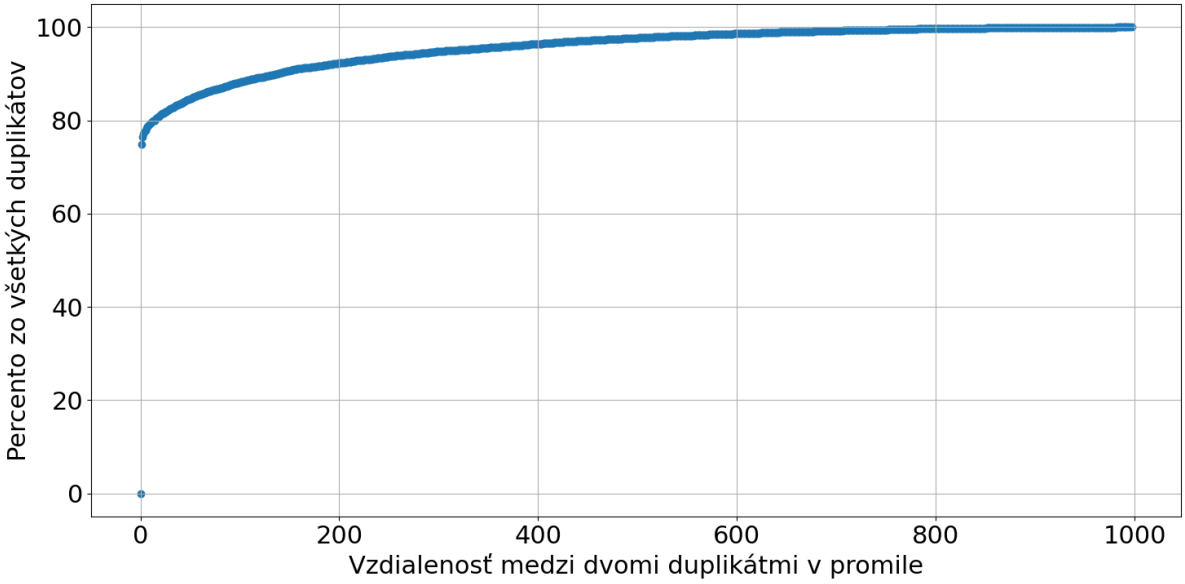
Pri druhom spôsobe merania vzdialeností medzi duplikátmi sme rátali priemernú vzdialenosť medzi nimi, a to pre všetky, ktoré sa opakovali rovnaký počet krát. Pre každý duplikát, ktorý sa opakoval n-krát, sme zrátali všetky vzdialenosti medzi ich počiatočnými pozíciami, teda posunutiami začiatkov, ako v predošlom spôsobe. Takto sme sčítali všetky vzdialenosti, ktoré boli medzi blokmi duplikátov, ktoré sa opakovali n-krát. Nakoniec sme tento súčet predelili počtom všetkých vzdialeností, ktoré do tejto sumy vstupovali.



Obr. 4.3: Vzdialenosti medzi dvoma nasledujúcimi duplikátmi na množine súborov html s delením dát po 32 bajtoch, formát spájania s doplnením nulami na násobok 32 bajtov.

Pre prvú analýzu uvádzame príklad jej vizualizácie na obrázku 4.3 na súboroch typu html s delením dát po 32 bajtoch. Na x-ovej osi je zobrazená vzdialenosť v promile veľkosti celých vstupných dát. Na y-ovej osi, ktorá je zlogaritmovaná desiatkovým logaritmom, je zobrazený počet duplikátov, ktoré majú takúto vzdialenosť. Vďaka logaritmovaniu je možné vidieť, že vzdialenosti medzi duplikátmi klesajú exponenciálne. Toto platilo pri duplikátoch takmer vo všetkých dátových podmnožinách.

Pre lepšie znázornenie sme vytvorili aj graf kumulatívne zobrazujúci percento všetkých duplikátov, ktoré má medzi sebou vzdialenosť menšiu alebo rovnú vzdialenosti v promile zobrazenej na x-ovej osi. Na tomto grafe je možné vidieť, že väčšina duplikátov je blízko pri sebe a veľmi málo je takých, ktoré sú rozložené cez celé dáta. Spôsobené to môže byť aj tým, že ide o množinu súborov (okolo 4 500) a nie o jeden súbor. Duplikáty



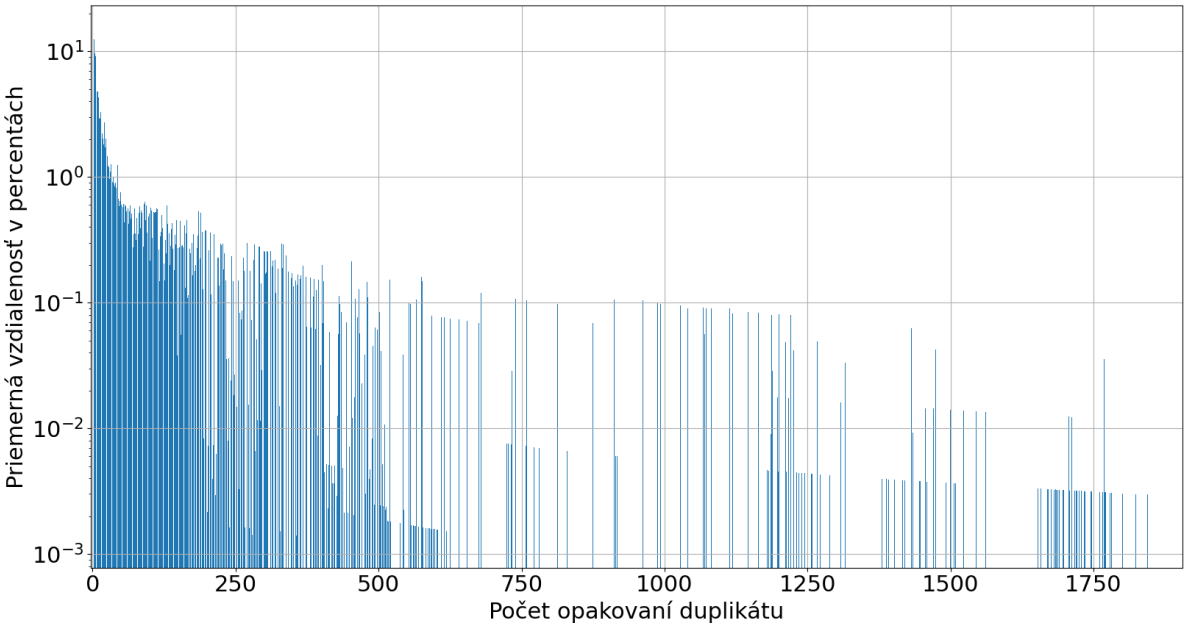
Obr. 4.4: Vzdialenosti medzi dvoma nasledujúcimi duplikátmi na množine súborov html s delením dát po 32 bajtoch, formát spájania s doplnením nulami na násobok 32 bajtov, kumulatívne.

sa teda môžu častejšie nachádzať v rámci jedného súboru. Zároveň je pomerne časté, že viacero rovnakých blokov, najčastejšie tvorených rovnakým bajtom, nasleduje jeden za druhým.

Aj pre druhý spôsob merania vzdialenosti medzi duplikátmi sme vytovrili graf. Na obrázku 4.5 je na x-ovej osi je zobrazený počet opakovaní duplikátu a na y-ovej je percentuálne vyjadrenie, priemernej vzdialenosti medzi týmito duplikátmi, k veľkosti vstupných dát. Obrázok zobrazuje iba časť grafu, duplikáty, ktoré sa opakovali v najmenšom počte, mali medzi sebou priemernú vzdialenosť cez 10 percent. Podobne ako pri predošlých experimentoch, typ dát na toto správanie nemal veľký vplyv. Pokles priemernej vzdialenosti je z časti aj prirodzený, jej veľkosť klesá, keďže so zväčšujúcim počtom duplikátov sa priemerná vzdialenosť musí zmenšovať.

Keďže pri oboch analýzach ide o viacero súborov spojených dokopy, vzdialenosti medzi duplikátmi môžu byť ovplyvnené aj ich usporiadaním. V dátových podmnožinách tvorených väčším počtom súborov (1 000 a viac súborov) sa síce jednotlivé vzdialenosti po preusporiadaní súborov mierne líšili, avšak nami zistený exponenciálny tvar zobrazenia vzdialeností sa nemenil a aj po viacerých preusporiadaniach zostával rovnaký.

Okrem spomenutých dvoch spôsobov analýzy vzdialenosti, sme analyzovali aj rozloženie duplikátov. Jednu z takýchto vizualizácií je možné vidieť aj na obrázku 4.6. Dáta sme rozdelili na tisíc rovnako veľkých častí, teda promile. Následne sme sčítavali počty

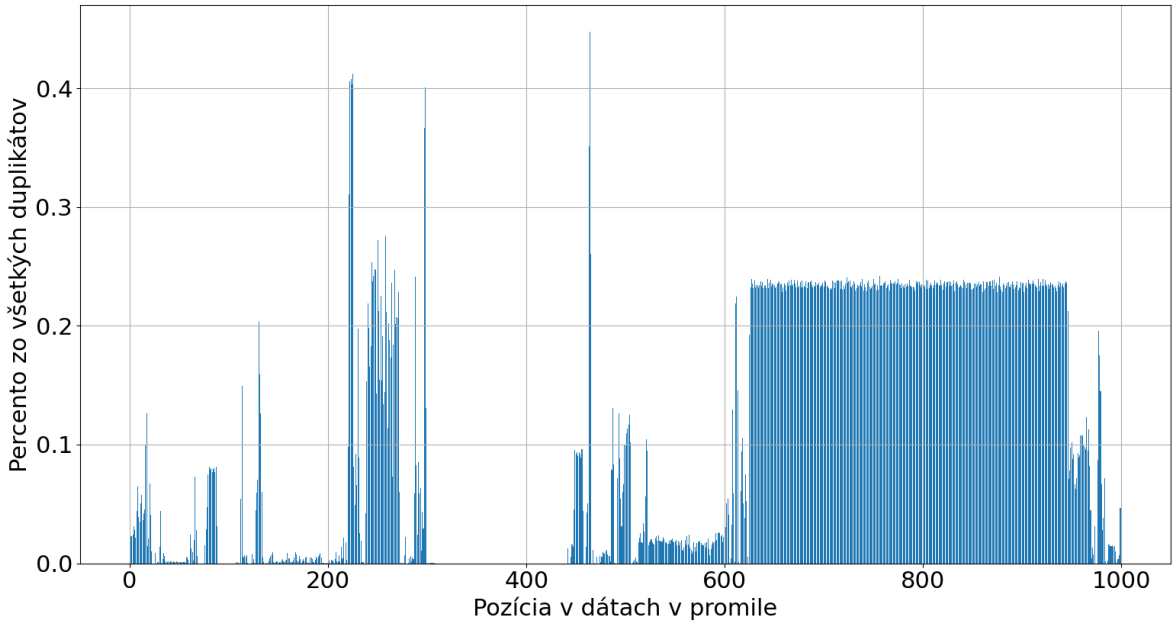


Obr. 4.5: Vzdialenosti medzi dvomi nasledujúcimi duplikátmi podľa počtu ich opakovania na množine súborov html s delením dát po 32 bajtoch, formát spájania súborov s doplnením nulami na násobok 32 bajtov.

duplikátov na každom promile dát. V histograme na obrázku 4.6 sú teda na x-ovej osi zobrazené jednotlivé tisíciny dát a na y-ovej osi je percentuálny podiel zo všetkých duplikátov, ktoré sa na danej tisícine dát nachádzali. Po preusporiadaní súborov, ktoré boli spájané, vyzerá tento histogram podobne až na umiestnenia a mierne rozdiely vo výškach stĺpcov. Táto vizualizácia vlastne popisuje, ako rovnomerne sú duplikáty po dátach rozložené.

Graf, ktorý uvádzame ako príklad, bol vytvorený na súboroch typu txt. Časť grafu od 600 do 1000 promile poukazuje na veľké množstvo duplicity v jednom z txt súborov. Išlo o súbor s veľkým počtom núl oddelených čiarkami a medzerami, s najväčšou pravdepodobnosťou sú to dáta vo formáte csv s nesprávne nastavenou príponou txt v názve.

Pri všetkých typoch dát vo všeobecnosti platilo, že duplikáty neboli rozložené rovnomerne cez celé dáta, ale v niektorých častiach dát alebo súboroch z dát boli duplikáty rozmiestnené častejšie ako v iných častiach. Táto vlastnosť by sa mohla využiť na efektívnejšiu reprezentáciu duplikátov v metadátach, napríklad by sa mohli informácie o duplikátoch, ktoré sú zoskupené blízko pri sebe, zjednotiť do jedného záznamu.



Obr. 4.6: Rozloženie duplikátov po súboroch txt, formát spájania súborov s doplnením nulami na násobok 32 bajtov.

4.2.3 Ušetrený prenos

V práci nás tiež zaujímalo, ako efektívny je prenos cez nástroj *Dedupnetgo*, aký vplyv na počty duplikátov má rôzna veľkosť blokov, podľa ktorých sa deduplikuje a typ dát, ktoré sú deduplikované.

Podľa experimentov, ktoré sme vykonali, sa zdá, že čím menšie deduplikačné bloky sa používajú, tým viac je možné dáta deduplikovať je možné dosiahnuť, a to platilo pre všetky typy dát. Problémom je, že so zmenšujúcim sa blokom sa zvyšuje podiel metadát na prenášaných dátach a zároveň celý prenos trvá dlhšie a má väčšiu spotrebu pamäte.

V tabuľke 4.1 uvádzame percento duplikovaných blokov voči všetkým blokom. Stĺpce určujú veľkosť bloku v bajtoch, riadky typ súboru, presnejšie súborovú príponu. Podľa nameraných dát sa dá usúdiť, že pre každý súborový typ platí, že so zväčšujúcou sa veľkosťou bloku klesá počet duplikátov. Ako sme očakávali, súborové typy, ktoré sú komprimované avi, jpg, mov, mp4 obsahujú výrazne menej redundancie, ako nekomprimované súborové typy doc, html, pdf, ppt, txt, xls (niektoré môžu byť tiež komprimované).

Graf 4.7 porovnáva percentuálne vyjadrenie podielu duplikátov v dátach voči ušetreným dátam vďaka prenosu cez *Dedupnetgo*, oproti prenosu dát bez *Dedupnetgo*, s meniacou sa veľkosťou bloku v bajtoch. S blokom veľkosti 16 bajtov, aj napriek tomu,

	16	32	48	64	80	96	112	128	256	512
avi	0,46%	0,44%	0,43%	0,42%	0,4%	0,39%	0,38%	0,36%	0,25%	0,1%
doc	27,43%	22,3%	18,9%	17,92%	15,06%	14,65%	13,58%	13,98%	10,53%	6,79%
html	41,76%	24,97%	17,79%	13,49%	10,76%	8,90%	6,35%	6,49%	3,07%	2,02%
jpg	2,36%	1,97%	1,75%	1,55%	1,45%	1,4%	1,23%	1,26%	0,93%	0,7%
mov	0,02%	0,02%	0,02%	0,02%	0,02%	0,01%	0,01%	0,01%	0,01%	0%
mp4	0,53%	0,52%	0,51%	0,49%	0,48%	0,47%	0,45%	0,44%	0,32%	0,18%
pdf	11,65%	7,6%	6,13%	5,21%	4,82%	4,49%	4,31%	4,07%	3,35%	2,75%
ppt	10,9%	7,6%	5,96%	5,04%	4,24%	4,09%	3,49%	3,45%	2,57%	2,02%
ps	41,58%	27,99%	23,34%	20,32%	18,36%	17,49%	16,18%	15,52%	11,66%	7,7%
txt	33,16%	19,87%	15%	12,91%	11,31%	9,88%	8,56%	7,3%	2,15%	0,27%
xls	19,37%	12,5%	10,12%	8,92%	7,63%	6,96%	6,07%	5,76%	1,93%	1,29%

Tabuľka 4.1: Vzťah veľkosti deduplikačného bloku a typu dát k percentu duplikovaných blokov. Súbory spájané s doplnením nulami na násobok veľkosti bloku.

že v tomto prípade bolo okolo 12 percent blokov tvorených duplikátmi, bol prenos cez *Dedupnetgo* menej efektívny ako obyčajný prenos (percento ušetreného prenosu je záporné). Príčinou sú metadáta, ktoré sú pri takto malých blokoch veľmi veľké. Najviac ušetreného prenosu bolo pri použití blokov s veľkosťou 112 bajtov.

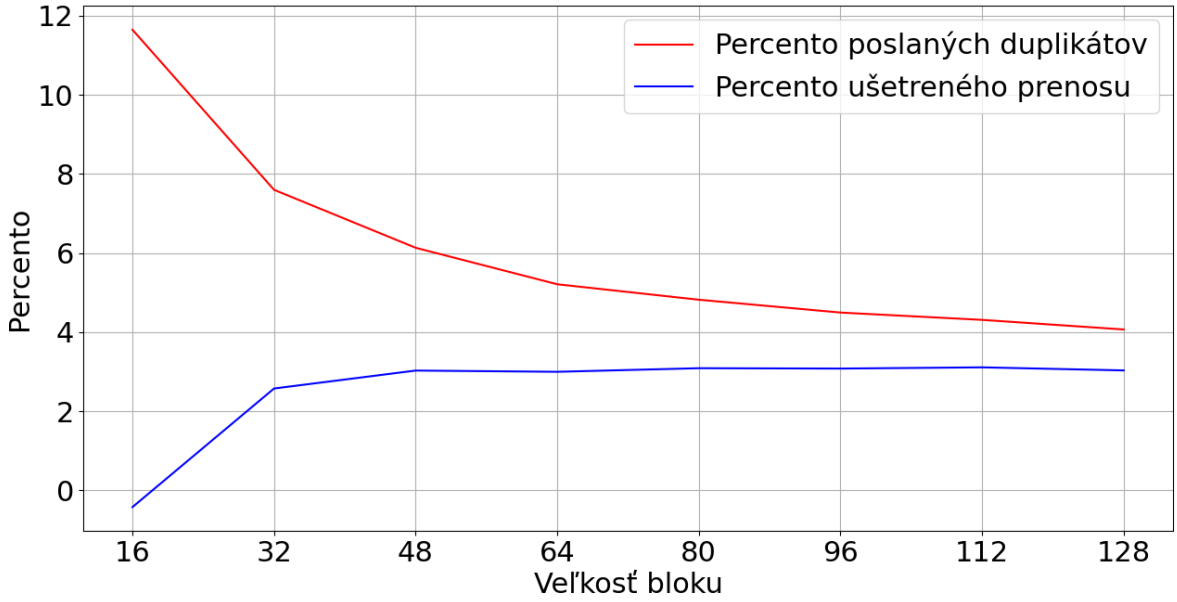
V tomto experimente sme na prenos duplikovaných dát použili 64 bitové čísla reprezentujúce posunutie. Zároveň, s každým blokom, duplikovaným alebo nie, sa na jeho začiatku posiela hlavička s veľkosťou jeden bajt. To znamená, že so zväčšujúcim sa blokom sa zmenšuje veľkosť metadát v pomere k veľkosti celých prenesených dát a zároveň sa ušetrí väčší počet bajtov pri prenose duplikovaného bloku oproti neduplikovanému bloku.

4.2.4 Porovnanie hešovacích funkcií

V práci sme skúmali, aký vplyv na deduplikáciu majú rôzne hešovacie funkcie, ktoré sa pri nej používajú. Zamerali sme sa na efektívnosť a kolízie, ktoré môžu pri hešovaní vzniknúť.

Dáta nemali veľký vplyv na efektívnosť hešovacej funkcie. Dôležitejším parametrom v tomto prípade bola veľkosť bloku, podľa ktorého sa dáta deduplikovali. Platí, že so zmenšujúcou sa veľkosťou deduplikačného bloku sa zmenšovala rýchlosť prenosu a zväčšovala spotreba operačnej pamäte.

Na obrázku 4.8 sú zobrazené hešovacie funkcie použité v nástroji *Dedupnetgo* a ich vplyv na rýchlosť prenosu. Hešovaciu funkciu simple sme vytvorili ako ukážkovú funkciu, určenú na porovnávanie s ostatnými hešovacími funkciami. Táto funkcia iba vráti prvých 10 bajtov zo vstupu. Z grafu sa dá zistiť, že hešovacia funkcia sha256 bola najpomalšia zo všetkých testovaných, aj keď najväčší rozdiel medzi najpomalšou



Obr. 4.7: Porovnanie percenta duplikátov v dátach typu pdf, voči percentu prenesených dát, ktoré boli ušetrené cez *Dedupnetgo*.

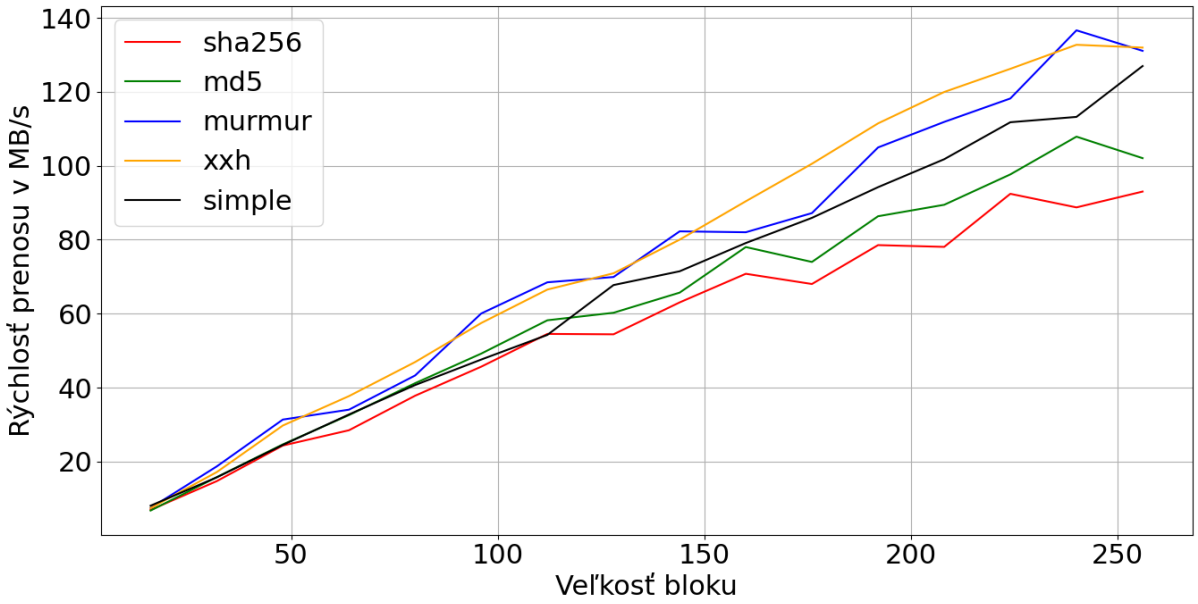
a najrýchlejšou hešovacou funkciou bol iba okolo 50 megabajtov za sekundu. Tento výsledok je však očakávateľný vzhľadom na jej kryptografickú bezpečnosť a veľkosť výstupu. Najrýchlejšími hešovacími funkciami boli, pri väčšine veľkostí deduplikačných blokov, hešovacie funkcie *Xxhash* a *Murmur*.

Vo všetkých experimentoch venovaným deduplikácii, sme nenašli kolíziu pri žiadnej hešovacej funkcii použitej v nástroji *Dedupnetgo*. Dôvodmi sú pravdepodobne menšia vzorka vstupných dát a kvalita využitých aj nekryptografických hešovacích funkcií. Pre niektoré z nich síce existujú známe kolízie, ale tie sú špeciálne vytvorené pre konkrétnu hešovaciu funkciu.

4.2.5 Obrazy virtuálnych diskov

Na záver sme vykonali experimenty aj na obraze virtuálneho stroja. Najprv sme obraz formátu ISO, stiahnutého zo stránky distribúcie Lubuntu, spustili pomocou QEMU emulátora využívajúc KVM. ISO obraz sme po spustení nainštalovali na obraz virtuálneho disku formátu qcow2.

Po inštalácii sme sa pokúsili odstrániť prebytočný a nepoužitý priestor z virtuálneho disku qcow2. Tento priestor bol z veľkej časti tvorený nulovými bajtami a tie by mohli, ako aj pri formáte tar, skresliť nami navrhnuté experimenty. Na spustenom virtuálnom stroji sme zmenšili partíciu disku, na ktorej bolo nainštalované Lubuntu, na minimum. Následne sme qcow2 obraz disku zmenšili pomocou nástroja virt-sparsify, ktorý voľné



Obr. 4.8: Vplyv hešovacej funkcie na rýchlosť prenosu na dátach typu pdf, formát spájania súborov tar.

miesto na virtuálnom disku zmenší a uvoľní ho pre hostiteľský operačný systém.

Pred spustením nástrojov na analyzovanie sme obraz qcow2 skonvertovali na obraz disku typu raw. Keďže z nástroja *Dedupnetgo* máme k dispozícii posunutia začiatkov duplikátov, vďaka súborovému systému *DebugFS* a formátu raw sme vedeli zistiť, v ktorom súbore sa duplikát s daným posunutím nachádzal. *DebugFS* je špeciálny súborový systém, ktorý je určený na ladenie súborových systémov. Jedny z funkcií, ktoré poskytuje sú aj funkcie *icheck* a *ncheck*, ktoré sme nad raw obrazom disku využili. Funkcia *icheck* vráti na základe zadaného čísla bloku v súbrovom systéme číslo i-uzla, do ktorého blok patrí. Funkcia *ncheck* na základe čísla i-uzla vráti cestu k súboru, ktorý je v danom i-uzle uložený.

Tieto dve funkcie sme využili nasledovne: funkcii *icheck* zadáme blok, ktorý získame predelením posunutia začiatku duplikátu (v raw obraze disku) veľkosťou bloku súborového systému. Jej výstupom je číslo i-uzla, ktoré zadáme funkcii *ncheck*. Tak sa nám podarilo zistiť, do ktorých súborov patria duplikáty, ktoré sa vyskytovali najčastejšie.

Ako aj pri veľkej časti ostatných typov dát, jedným z najčastejších duplikátov bol blok tvorený nulovými bajtami. Na základe postupu, ktorý sme práve popísali, sme zistili, že väčšina z duplikátov patrila do blokov, ktoré nemali pridelený i-uzol, teda boli prázdne. Niektoré patrili aj súborom, a to hlavne ovládačom a kodekom s príponou ko. Ďalším zaujímavým, pomerne často vyskytujúcim sa duplikátom bol blok tvorený rovnakým bajtom 255. Ten sa tiež vyskytoval v blokoch, ktorým neboli pridelené i-uzly. Okrem nich sa často tieto duplikáty vyskytovali vo firmvéri a ovládačoch s príponou

mbn. Ďalší duplikát tvorený rovnakým bajtom pozostával z bajtov 126, v ASCII reprezentácii je to znak tilda. Tie sa vyskytovali vo firmvérových súboroch s príponou ucode, teda mikrokód zariadení.

Záver

Navrhli a vytvorili sme viacero analýz určených na skúmanie vlastností redundancie v dátach. K týmto analýzam sme implementovali nástroje. Vytvorili sme nástroj *Dedupnetgo*, ktorý umožňuje posielať dáta deduplikovane cez sieť. Nástroj tiež poskytuje výstup o redundancii, teda duplikátoch, pričom tento výstup je vhodný na analýzu pomocou ďalších programamov.

Ako vstupné dáta sme pre experimentovanie a analyzovanie výstupu nástroja využili bežné, používateľské dáta, ktoré boli roztriedené podľa súborových typov. Výstupy z nástroja sme následne analyzovali pomocou programov, ktoré sme navrhli. Zistili sme, že na nami skúmaných dátach sa so zväčšujúcou veľkosťou blokov, podľa ktorých sa deduplikuje, výrazne klesá množstvo nájdených duplikátov. Ďalším užitočným poznatkom je, že vzdialenosti medzi dvomi nasledujúcimi duplikátmi sú v takomto type dát zväčša veľmi blízko pri sebe.

Nástroj *Dedupnetgo*, ktorý je aplikáciou pre príkazový riadok, je prakticky použiteľný na bežný prenos dát v praxi. V spojení s ďalšími programami na analýzu duplikátov, je možné tieto programy využiť vďaka konfiguračnému súboru aj na iných dátach, ako sme použili v našej práci. Užitočné sú aj výsledky, ktoré sa nám podarilo pomocou týchto nástrojov popísať. Tie sa môžu použiť pri konfigurácii existujúcich systémov a nástrojov a poskytujú dáta pre vývoj nových systémov a nástrojov pracujúcich s deduplikáciou. Niektoré z nameraných vlastnosti by tiež mohli byť použité vo výučbe, pri vysvetľovaní vlastností súborových systémov pre lepšie pochopenie problematiky.

Ďalšie smerovanie v oblasti, ktorej sme sa venovali, by mohlo viesť k rozširovaniu analyzovaných vlastností duplikátov a ďalších typov dát. V práci sme sa zamerali na deduplikáciu s fixnou veľkosťou bloku. Nevenovali sme sa však deduplikácii s premenlivou veľkosťou bloku, o túto časť by mohol byť náš výskum rozšírený. Ďalší výskum by sa mohol tiež zamerať na aplikáciu nami nameraných výsledkov do súborových systémov alebo kompresných algoritmov. Na základe týchto výsledkov by mohol byť rozšírený a vylepšený aj náš nástroj *Dedupnetgo*.

Literatúra

[1] Pixabay. https://pixabay.com. Accessed: 2024-04-12.

[2] Austin Appleby. Murmurhash. https://sites.google.com/site/murmurhash, 2008. Accessed: 2023-15-11.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. Breaking murmur: Hash-flooding dos reloaded. https://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/, 2012. Accessed: 2023-18-11.

[4] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[5] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 380–388, New York, NY, USA, 2002. Association for Computing Machinery.

[6] Yann Collet. xxhash. https://github.com/Cyan4973/xxHash/blob/dev/README.md, 2020. Accessed: 2023-14-11.

[7] Testify contributors. Testify: A sacred extension to the standard go testing package. https://github.com/stretchr/testify. Accessed: 2023-07-10.

[8] TTLCache contributors. Ttlcache: A simple key-value store with ttl for golang. https://github.com/jellydator/ttlcache. Accessed: 2023-05-05.

[9] Mark Fasheh and Duperemove contributors. Duperemove. https://github.com/markfasheh/duperemove/wiki. Accessed: 2024-03-07.

[10] Steve Francia and Cobra contributors. Cobra: A commander for modern go cli interactions. https://github.com/spf13/cobra. Accessed: 2023-11-09.

[11] Garfinkel, Farrell, Roussev, and Dinolt. Bringing science to digital forensics with standardized forensic corpora, govdocs1. https://digitalcorpora.org/corpora/file-corpora/files/, 2009. Accessed: 2024-04-12.

[12] Ramin Gholami Taghizadeh, Reza Gholami Taghizadeh, Fahimeh Khakpash, Mohammadreza Binesh Marvasti, and Seyyed Amir Asghari. Ca-dedupe: content-aware deduplication in ssds. *The Journal of Supercomputing*, 76(11):8901–8921, Nov 2020.

[13] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[14] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[15] Stan Z. Li and Anil Jain, editors. *Encyclopedia of Biometrics*, pages 668–668. Springer US, Boston, MA, 2009.

[16] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW 2007 (16th International Conference on the World Wide Web)*, pages 141–150, Banff, 2007.

[17] Fan Ni, Xingbo Wu, Weijun Li, and Song Jiang. Thindedup: An i/o deduplication scheme that minimizes efficiency loss due to metadata writes. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

[18] Rob Pike, Ken Thompson, and Robert Griesemer. The go programming language. `https://golang.org/`, 2007. Accessed: 2024-03-05.

[19] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), aug 2013.

[20] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*. Citeseer, 2014.

[21] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[22] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

Príloha A: obsah elektronickej prílohy

Všetky časti implementácie a dodatočné výsledky práce a vizualizácie sme zverejnili v gitovom repozitári na GitLabe. Repozitár je na stránke `https://gitlab.com/matu9s/master-thesis`.

Repozitár je rozdelený na 4 podpriečinky:

- **analyses**

  V tomto priečinku sa nachádza program, ktorý v práci využívame na analýzu duplikátov a deduplikácie.

- **dedupnetgo**

  V tomto priečinku sa nachádza nástroj *Dedupnetgo*. Je rozdelený na priečinky **cmd** s implementáciou venujúcou sa primárne častiam pracujúcim s príkazovým riadkom a **internal**, v ktorom je implementácia hlavnej logiky nástroja.

- **helpers**

  V tomto priečinku sa nachádzajú pomocné programy, ktoré sme používali počas experimentov.

- **results**

  V tomto priečinku sa nachádzajú dodatočné výsledky, vizualizácie a namerané dáta. V priečinku sa nachádzajú podpriečinky s názvami podľa súborovej prípony, podľa dát, ktorých výsledky sú v týchto podpriečinkoch uvedené. V každom priečinku, pre každý typ dát, sú ďalšie podpriečinky s názvom obsahujúcim veľkosť bloku, ktorý bol pri experimentovaní použitý.

Spustenie a inštalácia Dedupnetgo

Dedupnetgo je naprogramovaný v Go vo verzii 1.19. Nástroj bol spúšťaný a testovaný na Linuxe. Pre spustenie nástroja *Dedupnetgo* je potrebné nástroj najprv skompilovať. Na skompilovanie nástroja stačí v priečinku *dedupnetgo* spustiť príkaz **go build**.

Príkaz na skompilovanie nástroja *Dedupnetgo*

```
$ go build .
```

Následne je možné nástroj používať na základe nápovedy pre odosielateľa a prijímateľa.

Príkaz na skompilovanie nástroja dedupnetgo

```
$ ./dedupnetgo -h # napoveda pre nastroj
$ ./dedupnetgo send -h # napoveda pre odosielatela
$ ./dedupnetgo receive -h # napoveda pre prijimatela
```

Spustenie a inštalácia nástroja na analýzu duplikátov

Program na analyzovanie duplikátov je naprogramovaný v Pythone vo verzii 3.10.12. Pre spustenie programu na analýzu, je najprv potrebné nainštalovať použité knižnice. Tie sú v súbore *requirements.txt*. Ich nainštalovanie je možné (odporúčame inštalovanie vo virtuálnom prostredí) pomocou príkazu **pip install**.

Príkaz na inštaláciu knižníc pre program na analýzu duplikátov

```
$ pip install -r requirements.txt
```

Po vyplnení konfiguračného súboru, je možné spustiť program na analýzu duplikátov z viacerých podčastí v priečinku *analyses*. Súbor *analyse_duplicates.py* vytvorí analýzy vlastností duplikátov. Súbor *analyse_hash_functions.py* vytvorí analýzy vlastností deduplikácie pri rôznych hešovacích funkciách. Posledný súbor *analyse_window_size.py* vytvorí analýzu efektívnosti deduplikácie podľa veľkosti deduplikačného bloku.

Príkazy na spustenie jednotlivých podčastí nástroja

```
$ python3 analyse_duplicates.py
$ python3 analyse_hash_functions.py
$ python3 analyse_window_size.py
```