VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYHLEDÁVÁNÍ DUPLICITNÍCH TEXTŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE TOMÁŠ PEKAŘ
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYHLEDÁVÁNÍ DUPLICITNÍCH TEXTŮ

DUPLICATE TEXT IDENTIFICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE TOMÁŠ PEKAŘ
AUTHOR

VEDOUCÍ PRÁCE doc. RNDr. PAVEL SMRŽ, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Cílem této práce je navrhnout a implementovat systém pro vyhledávání duplicitních textů. Výsledná aplikace by měla umět dokumenty indexovat a také je v indexu vyhledávat. V naší práci se zabýváme předzpracováním dokumentů, jejich fragmentací a indexací. Dále rozebíráme metody vyhledávání duplicit, s čímž je spojena také strategie selekce podřetězců. Práce obsahuje i popis základních datových struktur, které lze použít pro indexaci n-gramů.

Abstract

The aim of this work is to design and implement a system for duplicate text identification. The application should be able to index documents and also searching documents at index. In our work we deal with preprocessing documents, their fragmentation and indexing. Furthermore we analyze methods for duplicate text identification, that are also linked with strategies for selecting substrings. The thesis includes a description of the basic data structures that can be used to index n-grams.

Klíčová slova

vyhledávání, haš, duplikáty, indexace, n-gram, invertovaný index, datové struktury

Keywords

searching, hash, ducplicates, indexing, n-gram, inverted index, data structures

Citace

Tomáš Pekař: Vyhledávání duplicitních textů, bakalářská práce, Brno, FIT VUT v Brně, 2015

Vyhledávání duplicitních textů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. RNDr. Pavla Smrže, Ph.D. Další informace mi poskytl pan Ing. Jan Kouřil. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Tomáš Pekař
19. května 2015

Poděkování

Chtěl bych poděkovat vedoucímu panu doc. Pavlu Smržovi za odborné vedení a pomoc s prací. Dále bych chtěl poděkovat panu Ing. Kouřilovi za rady spojené s indexací textu.

Obsah

Kapitola 1

Úvod

Elektronické dokumenty jsou v dnešní době naprosto běžné. Jejich tvorba a úprava je velice snadná, ještě snazší je jejich kopírování. Některé takové dokumenty mohou ale vznikat ne zcela správným způsobem, neoprávněnou kopií původního dokumentu nebo jeho částí. Odhalování takových dokumentů, by bylo bez pomoci různých nástrojů nereálné. K jejich efektivnímu odhalování musíme proto používat počítačové programy. Nesmíme také zapomínat na to, že budeme vyhledávat ve velké kolekci dat a tomu by mělo být vše přizpůsobeno. Nemusíme se zde bavit jen o běžných elektronických dokumentech, ve výčtu se mohou objevit i internetové stránky nebo elektronická pošta.

Metod pro porovnávání elektronických dokumentů je celá škála. Můžeme hledat obsahově podobné dokumenty, jejich přesné kopie nebo jen zkopírované části. Tato práce rozebírá jednotlivé metody, přičemž cílí na hledání přesných duplicitních částí textu. S vyhledáváním je také spojena indexace textu, která je zde také popsána.

Jednotlivé kapitoly této práce se pak zabývají popisem principů využitých při návrhu a implementaci našeho systému. V druhé kapitole můžete nalézt analýzu našeho problému, od metod pro vyhledávání duplicit v textu, až po jeho zpracování. Třetí kapitola obsahuje výčet všech použitých technologií a knihoven, včetně jejich popisu. V kapitole čtyři a pět se dozvíte jak naše aplikace funguje a jaké metody jsme při jejím návrhu a implementaci využili. Na závěr je v šesté kapitole popsáno několik testů námi vytvořeného systému pro vyhledávání duplicit v textu.

Kapitola 2

Analýza problému

V případě, že musíme vyhledat duplicity ve velkém množství dat, musíme použít vhodné metody, které s tím počítají. Nejenže nás budou zajímat metody pro zpracování dokumentů, kde budeme provádět analýzu textu a jeho členění na dílčí části, ale také se musíme zajímat o to, jakým způsobem budeme tyto dílčí části dále zpracovávat, uchovávat a následně je vyhledávat.

Mluvíme o tzv. *Information Retrieval* (dále jen IR), který se zabývá právě tímto problémem. Nemůžeme ovšem využít všechny metody IR. Rozdíl je v tom, jak tyto metody definují dokumenty jako *podobné*. V mnoha z nich je použita sémantická podobnost, která by nám určila dva syntakticky rozdílné dokumenty jako relevantní. V našem případě hledáme dokumenty, které jsou syntakticky podobné, tzv. *near-duplicates*.[27]

Pokud chceme takové dokumenty vyhledávat, musíme mít určitou představu o tom, jak tyto dokumenty vznikají. Dokumenty patřící do skupiny *near-duplicates*, bychom mohli vytvořit následujícím způsobem z původního dokumentu:

- Úplnou kopií původního dokumentu
- Přidáním nebo odebráním jednoho či více odstavců
- Přidáním, odebráním nebo záměnou jednoho či více slov
- Změnou pořadí odstavců

Můžeme tedy říci, že dva dokumenty jsou syntakticky podobné, pokud jsou stejného původu. Označme takové dokumenty jako d a d_q. Ve vektorovém modelu jsou tedy syntakticky podobné, platí-li:

$$\varphi(d, d_q) \geq 1 - \varepsilon, \ kde \ 0 < \varepsilon \ll 1 \tag{2.1}$$

kde φ značí podobnostní funkci na intervalu $[0, 1]$. Mějme sadu dokumentů D, kterou chceme porovnat s dokumentem d_q. Výpočet sady D_q, kde $D_q \subset D$, obsahující všechny syntakticky podobné dokumenty d_q v D, by měl lineární časovou náročnost, která by závisela na velikosti sady D, $O(|D|)$.[20]

Pokud se pohybujeme ve vektorovém modelu, musíme brát v potaz také počet dimenzí. Vysoká dimensionalita, kde *vysoká* znamená více jak deset, způsobuje to, že některé objekty reprezentované více dimenzionálními vektory nemohou být efektivně zpracovány pomocí metod rozdělení prostoru na více podprostorů. Jedná se o metody, jako jsou například kd-trees, quad-trees nebo R-trees. Ideální nejsou ani metody sekvenčního skenování.[27]

Zde se nám nabízí možnost určit podobnost dokumentu d a d_q pomocí metody otisků (*fingerprints*) nebo fulltextového vyhledávání. Fulltextové vyhledávání není ovšem příliš vhodné, protože pracuje s onou sémantickou podobností dokumentů. Volíme proto spíše metodu otisků. Než se ale pustíme do jejího popisu, určeme si první jednotlivé kroky, které jsou potřeba k provedení celé operace odhalování duplicitních textů:

1. Předzpracování a normalizace textu
2. Selekce podřetězců
3. Hašování extrahovaných podřetězců
4. Uložení haší do indexu

Všechny tyto kroky jsou nezbytné ke správnému fungování detekce duplicit jako celku. V prvním z kroků neprovádíme jen normalizaci dokumentu, ale také odstraňujeme formátovací značky, protože v praxi nepracujeme jen s čistě textovými dokumenty, ale i HTML nebo PDF soubory. Pokud bychom pak následně chtěli vyhledat naprosto identické dokumenty, stačilo by porovnat hodnoty jejich haší, čímž bychom ale dokument důkladněji neprozkoumali. Proto ve druhém kroku dělíme dokument na menší fragmenty, které nám umožní jeho přesnější analýzu. Třetí krok je zde hlavně pro úsporu paměťových nároků a rychlejší vyhledávání. V posledním kroku zaznamenáme data, získaná z těch předchozích, do takových datových struktur, které nám umožní rychle v indexu vyhledávat.

Vyhledávání duplicit pomocí metody otisků dokumentů

Tato metoda pracuje s tzv. otisky (*fingerprints*). Otisk F_d je sada k celočíselných hodnot získaných z d. Jestliže dva otisky, F_d a F_{d_q}, sdílí κ stejných celočíselných hodnot, kde $\kappa \geq k$, je zde předpoklad, že d a d_q jsou podobné dokumenty (*near-duplicates*)[20]. Jejich podobnost můžeme vyjádřit např. použitím Jaccardova koeficientu:

$$\frac{|F_d \cap F_{d_q}|}{|F_d \cup F_{d_q}|} \geq \frac{\kappa}{k} \;\Rightarrow\; P(\varphi(d, d_q) \geq 1 - \varepsilon),\; kde\; P \rightarrow 1 \tag{2.2}$$

Můžeme definovat několik způsobů vytváření otisků dokumentů:

- Plné otisky (*full fingerprinting*)
- Překrývající se otisky (*overlapping fingerprinting*)
- Nepřekrývající se otisky (*non-overlapping fingerprinting*)

V případě plných otisků vybíráme prakticky všechny možné otisky dokumentu o zadané velikosti. Nepřekrývající se otisky si lze představit jako sekvence znaků nebo slov určité délky, které se nepřekrývají. Překrývající se otisky jsou prakticky totéž, s tím rozdílem, že ony se již vzájemně překrývají.[12]

Tolik k metodě vyhledávání duplicit pomocí otisků. Nyní již ale přejděme k detailnějšímu popisu jednotlivých kroků, kterým se zabývají následující podkapitoly.

2.1 Předzpracování a normalizace dokumentu

Předzpracování a normalizace je podstatnou částí celého procesu. Jak již bylo řečeno, v praxi nebudeme pracovat jen s čistě textovými dokumenty, ale i s dokumenty, které obsahují formátovací značky. Mějme například dva textově stejné dokumenty, jeden v HTML a druhý v PDF. Pokud bychom takové dokumenty porovnali, nedočkali bychom se pozitivního výsledku. Proto je nutné takové formátovací značky odstraňovat a snažit se vždy získat čistě textový dokument. Musíme si také dát pozor na některé značky, jejichž význam spočívá v označení konce řádku (např. v HTML je to třeba značka `<br>`). Takové značky musíme detekovat a namísto nich vkládat konce řádků.

Dalším krokem normalizace může být odstranění interpunkčních znamének, převodu velkých písmen na malá písmena nebo odstranění tzv. *stopslov*. *Stopslovo* je slovo, které nemá velkou významovou váhu a jeho výskyt je v textu častý. Příkladem takových slov jsou například spojky nebo předložky. Můžeme také využít proces *lemmatizace*, kde jsou slova převáděna zpět do jejich základního tvaru, nebo proces záměny synonym, kde jsou slova stejného významu uskupena do tzv. *synsetů*.[3]

Nicméně použití některých metod normalizace není pro vyhledávání duplicit úplně ideální. Kdybychom používali například proces odstranění *stopslov* nebo proces *lemmatizace*, nezískávali bychom pak syntaktickou podobnost dokumentů, ale spíše sémantickou.

2.2 Strategie selekce podřetězců

Metody selekce podřetězců jsou stěžejní při vyhledávání duplicit. Mezi základní patří: selekce podřetězců v závislosti na jejich pozici v textu, selekce na základě frekvence výskytu slov nebo selekce podřetězců dle předem definovaného slova[1].[27][5] Musíme si navíc uvědomit, že už samotná metoda selekce podřetězců nám určuje, jakým způsobem budeme duplicity v textu vyhledávat.

2.2.1 Selekce podřetězců v závislosti na pozici

Tato metoda výběru podřetězců je jednou z nejjednodušších. Rozdělení dokumentu může být provedeno několika způsoby:

- Rozdělením dokumentu na náhodně získané podřetězce určité délky
- Rozdělením dokumentu na věty (v případě souvětí na věty jednoduché) nebo odstavce
- Rozdělením dokumentu na sekvence znaků fixní délky
- Rozdělením dokumentu po n slovech
- Rozdělením dokumentu na n-gramy

První dvě možnosti nejsou úplně vhodné, v případě náhodného výběru nemusíme obsáhnout důležité části textu a v případě vět se nám nedostává vhodné granulity dokumentu. Mnohem lépe jsme na tom se sekvencemi znaků a výběrem n slov, kde se nám již nižší granulity dostává.[12] Nejlepší variantou, z výše jmenovaných, je využití n-gramů, kde n-gram je posloupnost n po sobě jdoucích znaků nebo slov řetězce, které se vzájemně překrývají.

[1] Někde lze najít označení jako kotevní slovo

Velikost n-gramů může být různá, od 2-gramů nebo 3-gramů[2] až po 15-gramy nebo 25-gramy.

Tvorbu n-gramů slov si můžeme předvést na následujícím příkladu věty: „Lorem ipsum dolor sit amet“, ze které vytvoříme tři 3-gramy.

1. Lorem impsum dolor
2. ipsum dolor sit
3. dolor sit amet

Ve výsledku je celková délka všech n-gramů původního řetězce o velikosti k, dlouhá $k + 1$ n-gramů znaků/slov.[2]

Využití n-gramů má oproti ostatním typům výběrů podřetězců jednu výhodu. Onou výhodou je odolnost n-gramů proti posunu textu (např. přidáním nebo odebráním slova z věty).[5] Demonstrujme si tuto problematiku na příkladu: pokud máme originální dokument d a jeho upravenou kopii d_q, která vznikla přidáním jednoho slova, bude situace s využitím 3-gramů vypadat následovně:

- **Originální řetězec** Lorem ipsum dolor sit amet
- **Podobný řetězec** Lorem ipsum dolor at sit amet
- **Shoda** Lorem ipsum dolor | ~~ipsum dolor at~~ | ~~dolor at sit~~ | ~~at sit amet~~

Nicméně v takovém krátkém řetězci, kde je pozitivní pouze jeden n-gram ze čtyř, bychom těžko určovali podobnost. V případě, že bychom měli 25 slov v řetězci, by byla situace jiná. Doplněním jednoho slova bychom ovlivnili pouze tři 3-gramy z celkového počtu dvaceti dvou. Zde už by se nám mnohem snáze určovala podobnost.

2.2.2 Selekce podřetězců pomocí frekvence výskytu

Získávání podřetězců je závislé na jejich frekvenci výskytu. To znamená, že vyšší váhu při selekci mají ty podřetězce, kde je míra výskytu nižší. Strategii lze rozdělit na dvě metody.

První metoda se zabývá pouze zkoumaným dokumentem, kde vyhledává podřetězce s nejvyšší četností (*tf - term frequency*). Všechny získané podřetězce jsou seřazeny dle četnosti a vybrány jsou pouze ty, které splňují definovaný požadavek na četnost.[12]

Druhá metoda postupuje tak, že vyhledá všechny podřetězce a seřadí je podle toho jaká je jejich četnost v závislosti na dokumentech v celé kolekci. Jedním ze zástupců takového způsobu vybírání podřetězců je algoritmus I-Match. Ten získává hodnotu *idf* (*inverse document frequency*) pro každý výraz (*term*) rovnicí:

$$t_x = log\left(\frac{N}{n}\right) \tag{2.3}$$

kde N je počet dokumentů v kolekci a n počet dokumentů obsahující získaný výraz. Všechny podřetězce jsou pak vybrány na základě hodnoty *idf*.[7] Novější verze algoritmu I-Match již berou v potaz i hodnotu četnosti podřetězce v samotném dokumentu. Tuto metodu můžeme najít pod označením $tf.idf$.

Tyto metody selekce jsou spíše určeny pro zjištění sémantické podobnosti dokumentů.

[2] Čti bigram, trigram

2.2.3 Selekce podřetězců pomocí předem definovaných slov

Tento typ strategie využívá struktury dokumentu. To nám umožňuje detekovat dokumenty, které byly upraveny změnou pozic slov nebo změnou frekvencí slov. Zde vybíráme podřetězce začínající určitým slovem nebo posloupností znaků. Takto můžeme vybírat podřetězce ve větách, odstavcích nebo takto můžeme vybírat samotné věty jako podřetězce.

Metoda má tu nevýhodu, že ne všechna definovaná slova mohou být stejně efektivně využita pro všechny dokumenty.

2.2.4 Délka podřetězce

Délka podřetězce nám poměrně hodně ovlivňuje výsledek vyhledávání duplicitních textů. V případě, že zvolíme malou délku podřetězce, zvýší se pravděpodobnost nálezu falešné pozitivní shody (*false positive*). V textech se můžou objevovat různá, často používaná slova nebo slovní obraty, které by při výběru malé délky podřetězce, byly označeny jako pozitivní shoda (např. systém SCAM používá velice krátké podřetězce[27]). Naproti tomu, když vybíráme příliš dlouhé podřetězce, může docházet k tomu, že nenalezneme pozitivní shodu tam, kde by nalezena být měla, protože je duplicitní text obsažen pouze v části testovaného podřetězce.

Je proto vhodné volit takovou délku podřetězce, která povede k nejlepším výsledkům. Výzkumy poukazují na to, že nejvhodnější délka podřetězce je 3 až 5 slov[27]. Míru ovlivnění vyhledávání důsledkem velikosti n-gramů můžete nalézt v 6.3.

2.2.5 Počet podřetězců

Některé metody omezují počet extrahovaných podřetězců a to takovým způsobem, že indexují pouze pevně stanovený počet podřetězců, který je závislý na velikosti dokumentu. Otisky jsou takto selektovány pro ušetření paměťového prostoru. V případě, že zpracováváme malý dokument, tak vzniká problém zahození velké části podřetězců, a tím nemusí dojít k identifikaci duplicity.[11]

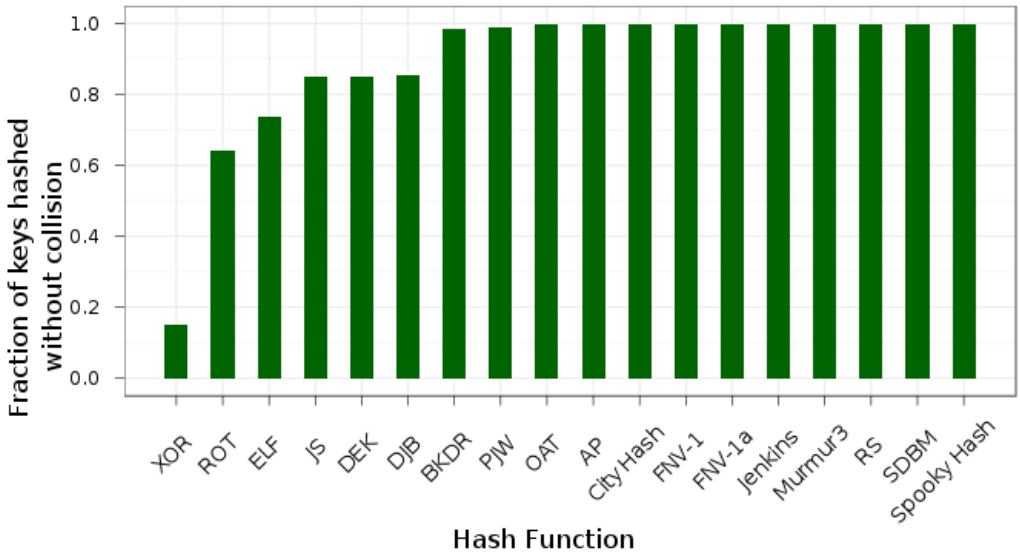
2.3 Hašování extrahovaných podřetězců

Využití hašovací funkce je nedílnou součástí extrakce otisků dokumentu. Hašovací funkce převádí vstupní hodnoty, v našem případě podřetězce vstupního dokumentu, na celočíselné nebo hexadecimální hodnoty. Výběr hašovací funkce je pro celkový výsledek důležitý. Musíme vybírat takovou hašovací funkci, která má nízkou míru kolize. To znamená, že pro dvě různé vstupní hodnoty dostaneme dvě různé celočíselné hodnoty, tedy alespoň ve většině případů. Mnoho starších hašovacích funkcí (i kryptografických) má poměrně velkou míru kolize. Příkladem může být hašovací funkce CRC32, která pro řetězce „plumless“ a „buckeroo“ generuje stejnou hodnotu[21].

Pravděpodobnost kolize ρ v závislosti na rozsahu a počtu generovaných hodnot můžeme vyjádřit takto:

$$\rho = 1 - e^{\frac{-k(k-1)}{2N}} \tag{2.4}$$

kde N značí počet všech možných hodnot a k počet generovaných hodnot. Statistiku kolizí nejznámějších hašovacích funkcí můžeme vidět na obrázku 2.1, kde bylo použito 42 milionů klíčů v rozsahu 192 nebo 256 bitů.



Obrázek 2.1: Porovnání hašovacích funkcí v počtu kolizí, převzato z [17]

Využití hašovací funkce pro snížení paměťových nároků

Hlavním důvodem využití hašování podřetězců je úspora paměťového prostoru. Podívejme se na tabulku 2.1, která říká, kolik znaků v průměru obsahovaly extrahované n-gramy z korpusu COCA[3]. Když vezmeme v úvahu, že jeden znak má velikost 1 B, tak 5-gram s průměrným počtem 25 znaků má velikost 25 B. Potom hodnota řetězce kódovaná do 64 bitové celočíselné hodnoty by měla pouhých 8 B. Tento přístup nám umožní snížit paměťové nároky.

	2-gram	3-gram	4-gram	5-gram
Průměrná délka [znak]	14.01	16.77	20.44	24.65

Tabulka 2.1: Průměrná délka n-gramu, převzato z [23]

2.4 Indexace podřetězců

Indexaci extrahovaných podřetězců si můžeme rozdělit na dvě části. V první části je nutné ukládat extrahované podřetězce ze zpracovávaného dokumentu do takových datových struktur, které nejsou příliš náročné na operační paměť (hašovací tabulky, B^+ stromy). V druhé části tvoříme úplný index všech extrahovaných podřetězců, kde index je nejčastěji tvořen dvojicí podřetězec-dokument. Musíme také zvolit takové datové struktury (popř. nástroje), které nám umožní rychlé vyhledání požadovaného podřetězce.

Při výběru vhodných datových struktur musíme hledět na několik faktorů, jimiž jsou: rychlost vkládání prvků, rychlost vyhledání a paměťová náročnost. Ideální by samozřejmě

[3] Viz http://www.ngrams.info/download_coca.asp

bylo použít takovou datovou strukturu, která splňuje všechny tři dané požadavky na výbornou, nicméně každá z níže uvedených má jisté výhody a nevýhody. Pokud jde o datové struktury pro naše účely, budeme v prvé řadě volit ty, které mají nejmenší paměťové nároky.

Hašovací tabulka

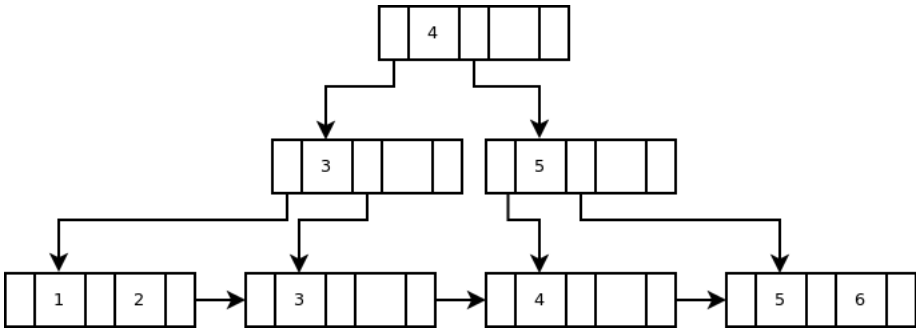
Hašovací tabulka, neboli tabulka s rozptýlenými hodnotami, je datová struktura představující pole dvojic klíč-hodnota. Klíčem je myšlena celočíselná hodnota získaná hašovací (mapovací) funkcí. Rozsah hašovací funkce nám také udává velikost naší hašovací tabulky. Máme několik způsobů implementace. První možností je dávat do tabulky přímo hodnoty a druhou je mít na dané pozici ukazatel na patřičnou hodnotu. Druhá varianta je lepší v případě, že nenaplníme celou tabulku, nebudeme tak zbytečně zabírat další místo v paměti. Hašovací tabulka je jedna z datových struktur určená pro rychlé vyhledávání. Má konstantní časovou náročnost.[1]

B^+ strom

B^+ strom je stromová datová struktura, který vychází z B-stromu a umožňuje rychlé vkládání, vyhledávání a mazání dat. Data jsou zpřístupněna pomocí klíčů a hlavní rozdíl mezi B^+ stromem a B-stromem je ten, že B^+ strom uchovává data pouze v listech. Každý uzel B^+ stromu obsahuje pole klíčů a pole ukazatelů na další uzel. B^+ strom má následující vlastnosti:

- Kořen stromu má vždy maximálně N následníků
- Každý uzel, kromě kořene, má maximálně N a minimálně $N/2$ následníků
- Data jsou uložena pouze v listech
- Všechny listy jsou uloženy na stejné úrovni

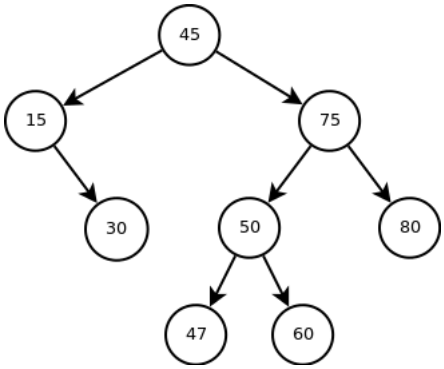
Tato datová struktura je vždy vyvážená a používá se primárně k uložení dat na disku. Vyhledání záznamu v B^+ stromu má logaritmickou složitost.[23] Příklad B^+ stromu můžete nalézt na obrázku 2.2.



Obrázek 2.2: Ukázka jednoduchého B^+ stromu

AVL strom

AVL strom je výškově vyvážený Binární vyhledávací strom, kde délka levého a pravého podstromu každého uzlu je buď stejná nebo rozdílná maximálně o 1. Autoři AVL stromu Adelson-Velsky a Landis dokázali, že jeho výška je maximálně o 45% vyšší než váhově vyvážený strom. Na rozdíl od váhově vyvážených stromů, jejichž vyváženost se v důsledku vložení nebo zrušení uzlu restrukturalizuje obtížně, restrukturalizace je u AVL stromu mnohem snazší. Vzhledem k těmto vlastnostem pracuje tato datová struktura v logaritmicky omezeném čase.[1]



Obrázek 2.3: Ukázka jednoduchého AVL stromu

Srovnání vybraných datových struktur

Podívejme se na obrázek 2.4, který srovnává výše jmenované datové struktury a jejich variace. Levý graf znázorňuje rychlost vkládání prvků do struktury a pravý paměťové nároky jednotlivých datových struktur v případě ukládání n-gramů. Pokud tyto dva grafy porovnáme, můžeme říci, že ideální střední cestou je volba hašovací tabulky (práce ze které tyto grafy vycházejí obsahovala ještě porovnání rychlosti vyhledání jednotlivých n-gramů, kde nejlepších hodnocení dosahovala jednoznačně hašovací tabulka). Při porovnání jednotlivých struktur byl využit korpus COCA[4].

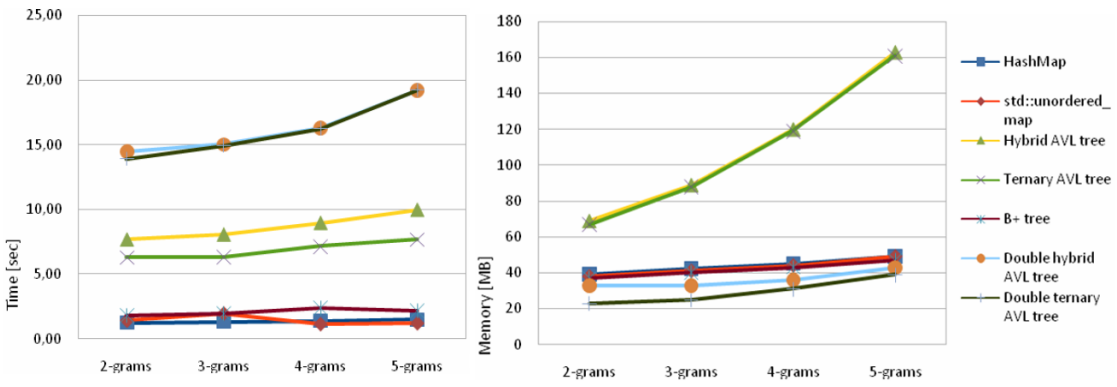
2.5 Využití detekce duplicitních textů

Pro detekci duplicit můžeme najít i další využití než v oblasti odhalování plagiátů. Zde je seznam několika z nich:

Odhalování zrcadlených webů - Takové využití vyhledávání duplicit je určeno spíše pro vyhledávače. Při vyhledávání internetových stránek a určovaní jejich hodnocení, je nutné zrcadlené stránky vyhledávat. Některé stránky mohou být postupně aktualizovány a je tak potřeba hledat tu nejnovější.

Shlukování podobných dokumentů - Zde můžeme najít příklad ve vyhledávání článků pojednávajících o stejné věci, které ovšem vycházejí z rozdílných zdrojů. Zde vycházíme spíše ze sémantické podobnosti dokumentů nežli ze syntaktické.

[4]Viz http://www.ngrams.info/download_coca.asp



Obrázek 2.4: Porovnání datových struktur pro indexaci n-gramů, převzato z [23]

Extrakce dat - V tomto případě vycházíme z podobné struktury dokumentů. Při klasickém vyhledávání dokumentů nebo jejich shlukování, ignoruje formátovací značky. Zde těchto značek využíváme a určujeme podobnost právě na jejich základě.

Odhalování plagiátů - Je další možností využití detekce duplicitních textů. Zde porovnáváme páry dokumentů nebo jejich částí, které byly zkopírovány nebo jinak neoprávněně upraveny a přivlastněny.

Detekce spamů - Jedna z možností využití detekce duplicit. Při kontrole pošty hledáme texty nebo části textu, které jsou označeny jako spam nebo nesou prvky spamu.

Detekce duplicit revidovaných dokumentů - V této oblasti se zaměřujeme na odstranění duplicitních dokumentů, které byly nějak upraveny/revidovány a uloženy zpět do databáze. Některé takovéto dokumenty již není potřeba uchovávat, proto musíme na serverech hledat jejich kopie a v rámci úspory místa je mazat.

Všechny výše uvedené možnosti využití souvisí, ať již méně nebo více, s oblastí vyhledávání duplicitních textů (*Information Retrieval*).

Kapitola 3

Použité technologie

Kapitola obsahuje přehled použitých technologií, knihoven a jejich základní popis. V některých případech je připojeno i zdůvodnění využití dané knihovny. V podkapitole 3.1 si přiblížíme důvody použití jazyka C++, v podkapitole 3.2 technologii ElasticSearch a na závěr si přiblížíme použité knihovny.

3.1 Jazyk C++

Prvotní verze aplikace byla implementována v jazyce Python, kde vznikaly problémy s velkou paměťovou náročností při zpracování větších dat a pomalou interpretací kódu. Z tohoto důvodu jsme se rozhodli přejít na jazyk C++, který nám umožní rychlejší zpracovávání vstupních dat s menšími paměťovými nároky. Další výhodou je dostupnost mnoha kvalitních knihoven poskytujících požadované datové struktury, se zaměřením na rychlost nebo paměťovou náročnost.

Přechodem na jazyk C++ se rychlost aplikace zvedla přibližně o 80% a paměťové nároky se zmenšily přibližně o 60%. Jedinou nevýhodou bylo zvýšení náročnosti implementace a horší přenositelnost.

3.2 ElasticSearch

ElasticSearch je open source vyhledávač postavený na knihovně Apache Lucene, což je knihovna pro fulltextové vyhledávače. V dnešní době je Lucene pravděpodobně nejvyspělejší, nejvýkonnější a plně funkční knihovnou pro vyhledávače. Lucene je jen knihovna a musíme ji tedy integrovat do naší aplikace. Vzhledem k tomu, že je implementována v jazyce Java, je tedy nutné také danou aplikaci vytvářet v Javě. V tomhle ohledu nám pomáhá ElasticSearch, který je také psán v Javě a všechny funkce Lucene nám zprostředkovává skrze jednoduché komunikační rozhraní. Nemusíme se tak zabývat pochopením složité Lucene, ale stačí nám jen zvládnout jednoduché API ElasticSearch.[10]

Přesněji řečeno, ElasticSearch je samostatný databázový server, který data ukládá v sofistikované podobně pro rychlé fulltextové vyhledávání. Jak jsme psali výše, práce s ním je poměrně jednoduchá, protože hlavní protokol je implementován pomocí HTTP a formátu zpráv JSON. ElasticSearch je také snadno škálovatelný, podle toho jak se zvyšuje zatížení serveru.[8]

Komunikační rozhraní pro ElasticSearch

ElasticSearch poskytuje oficiální komunikační rozhraní pro mnoho programovacích jazyků, jako jsou: Java, .NET, PHP, Python a další. Jazyk C++ mezi nimi bohužel chybí, proto musíme zvolit neoficiální rozhraní vytvořené společností Q-Hedge Technologies. Toto rozhraní ovšem obsahuje pouze základní funkce pro komunikaci s ElasticSearch, proto se tedy nevyhneme tomu, že budeme muset navíc sami implementovat některé funkce a metody. Knihovna je dostupná z [22].

3.3 Použité knihovny

Při tvorbě aplikace jsme použili několik dostupných knihoven. Jedna z nich je knihovna Boost[1], což je kolekce moderních C++ knihoven založených na standardu C++. Zdrojové kódy jsou postavené na Boost Software Licence, která dovoluje použití, úpravu a distribuci zcela zdarma.[24] My používáme Boost zejména pro zpracování textových řetězců, zpracování vstupních argumentů a práci se souborovým systémem. Další knihovnou je Pugixml[2], která je určena pro zpracování XML dokumentů. Pugixml cílí na rychlost, ale také na efektivní využití paměti.

Dále si detailněji rozebereme knihovnu pro hašování řetězců CityHash a knihovnu hašovací tabulky Sparsehash.

Hašovací funkce CityHash

Při výběru hašovací funkce nás nebude zajímat jen pravděpodobnost kolize (viz podkapitola 2.3), ale bude nás také zajímat její rychlost a její vlastnosti u tzv. lavinového efektu. *Lavinový efekt* je definován následovně: pokud změníme jeden bit původní zprávy, musí se s 50% pravděpodobností změnit každý bit výsledného haše. Tím je zajištěno, že i podobné zprávy mají zcela rozdílené hodnoty.[16] Pokud máme vzít v potaz míru kolize, vlastnost lavinového efektu a necílíme na kryptografickou haš, protože nám jde i o rychlost, tak je CityHash jasná volba.

CityHash[3] je rychlá hašovací funkce pro hašování textových řetězců, která vychází z hašovací funkce Murmur3. V současné době, jak autoři sami uvádějí, je CityHash nahrazena novou hašovací funkcí FarmHash a nepředpokládají se její další aktualizace.

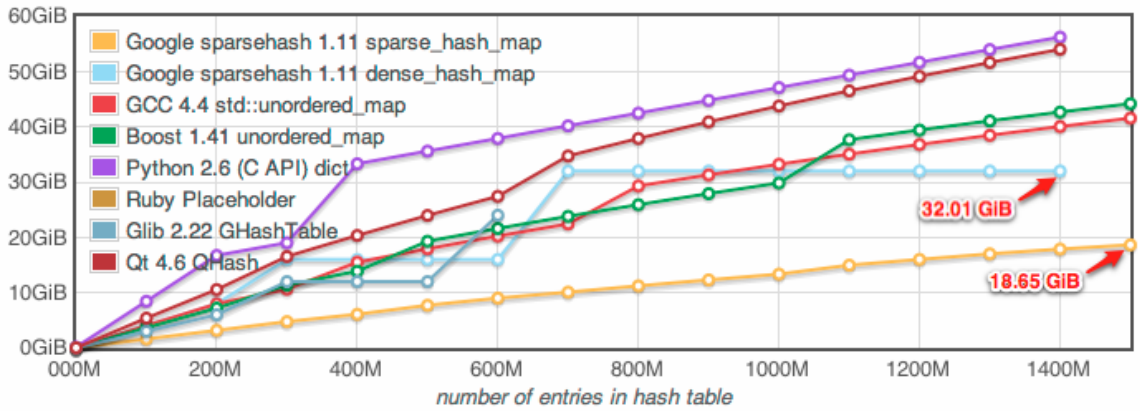
Hašovací tabulka Sparsehash

V našem zájmu je vybrat takovou hašovací tabulku, která bude mít nejmenší paměťové nároky. Na obrázku 3.1 můžeme vidět srovnání běžně dostupných hašovacích tabulek, při ukládání celočíselných hodnot. Na první pohled můžeme hned vidět, že Sparsehash dosahuje, co se týče paměťových nároků, nejlepších výsledků. S rychlostí vkládání nových prvků je to u Sparsehash horší, i když ne nejhorší. Pokud to celkově zhodnotíme, je hašovací tabulka Sparsehash jedna z nejlepších voleb.

Sparsehash[4] je paměťově efektivní implementace hašovací tabulky. Knihovna obsahuje několik implementací hašovací tabulky, včetně implementací, které jsou optimalizovány pro

[1] Knihovna Boost je dostupná z: http://www.boost.org/
[2] Knihovna Pugixml je dostupná z: http://pugixml.org/
[3] Knihovna CityHash je dostupná z: https://code.google.com/p/cityhash/
[4] Knihovna Sparsehash je dostupná z: https://code.google.com/p/sparsehash/

rychlost (`dense_hash_map`) nebo paměťovou efektivitu (`sparse_hash_map`). Tato implementace má podobné rozhraní jako SGI `hash_map` a tr1 `unordered_map`, ale s rozdílným výkonem. Je proto jednoduché nahradit `hash_map` nebo `unordered_map` za `sparse_hash_map` nebo `dense_hash_map`. Knihovna obsahuje také metody pro serializaci a deserializaci hašovací tabulky z disku.[25]



Obrázek 3.1: Porovnání paměťové náročnosti hašovacích tabulek, převzato z [18]

Kapitola 4

Návrh

Cílem práce je vytvořit systém, který dokáže identifikovat duplicitní texty v sadě dokumentů. K tomu patří i tvorba indexu všech dokumentů, který umožní rychlé a efektivní vyhledávání. Výstupem aplikace by pak měl být soupis dokumentů, které se do jistý míry shodují (jsou buď totožné nebo obsahují části stejných textových dat).

Přejděme nyní k samotnému návrhu systému. V našem návrhu budeme vycházet z kapitoly 2, ze které budeme využívat některé metody a postupy. Náš systém by měl zvládat dvě věci:

1. Indexaci dokumentů
2. Vyhledávání dokumentů v indexu

Pro lepší představu se podívejme na obrázek 4.1, kde je vše znázorněno. Levá část obrázku představuje indexaci dokumentu a pravá vyhledávání dokumentů v indexu. Prostřední část, společná jak pro indexaci, tak vyhledávání, představuje zpracování daného dokumentu. Pokud se nad problémem jednoduše zamyslíme, zjistíme, že pro indexované a vyhledávané dokumenty musí platit stejné podmínky. Pojďme si nyní popsat krok po kroku, dle toho, jak po sobě budou následovat.

4.1 Předzpracování a normalizace dokumentu

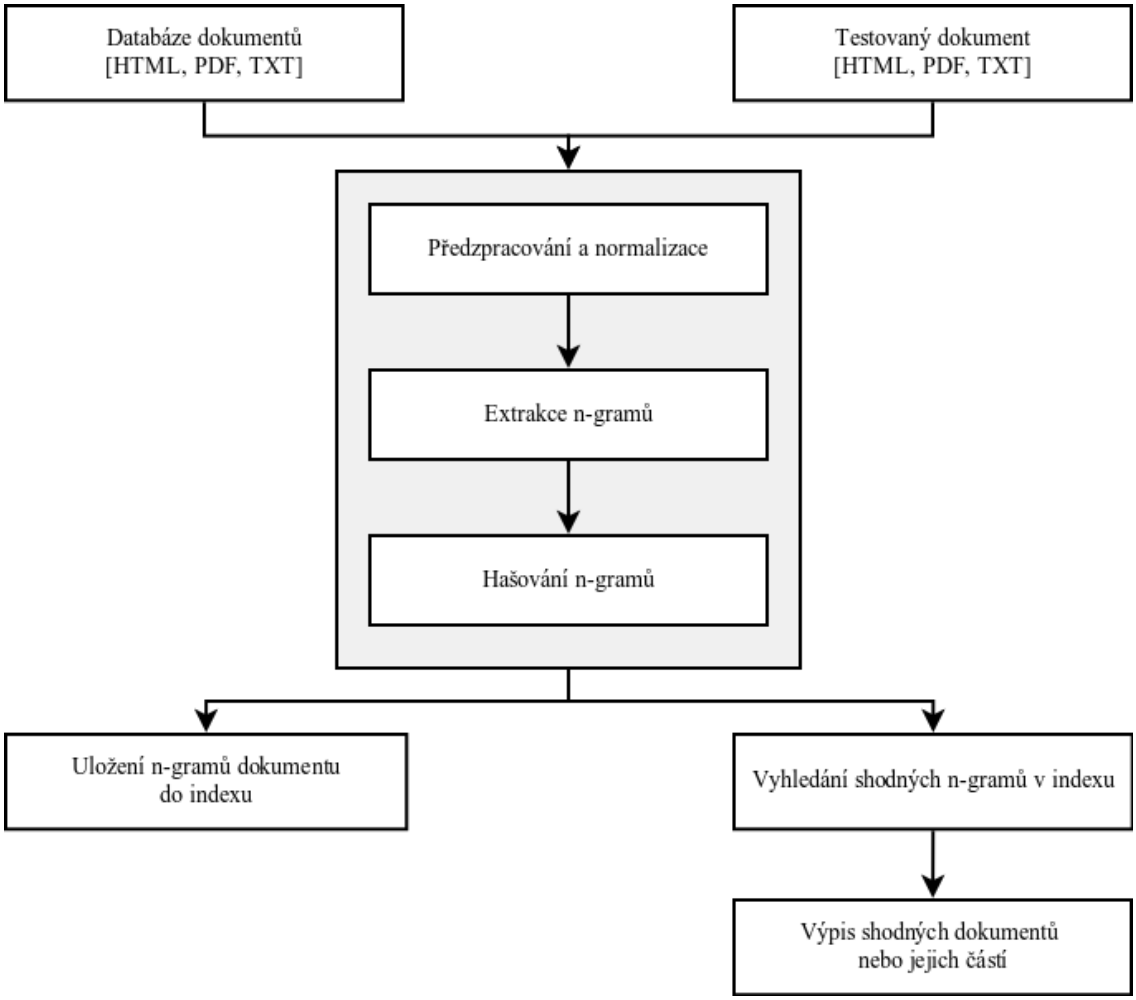
Cílem této fáze je dostat vstupní dokument do čistě textové podoby, abychom dosahovali co nejlepších výsledků při porovnávání. Když se nad tím zamyslíme, zjistíme, že se můžeme setkat s více typy souborů, a tudíž je nutná jistá možnost se přizpůsobit. Toho můžeme docílit přidáním konfiguračního souboru, který bude obsahovat různá pravidla pro zacházení s danými typy souborů.

Pokud jsme vyřešili otázku odstranění formátovacích značek, zbývá ještě odstranění například interpunkčních znamének, které by nám mohly negativně ovlivnit výsledek.

Jako poslední by měla být možnost odstranění stopslov. Z hlediska vyhledávání syntakticky stejných dokumentů to sice není úplně ideální, ale tato možnost by měla být zachována minimálně pro experimentální účely.

4.2 Extrakce a hašování n-gramů

V předchozí analýze problému v kapitole 2 se ukázalo, že pro naše účely bude nejvhodnější využít jako extrahované podřetězce n-gramy slov. Hledáme totiž syntakticky podobné



Obrázek 4.1: Návrh indexace dokumentu a vyhledávání v indexu

dokumenty a potřebujeme získávat otisky celého dokumentu, ne jen vybraných částí na základě definovaných slov nebo četnosti výskytu.

Otázkou nyní je, jak tyto n-gramy extrahovat. Můžeme postupovat následujícími způsoby:

Maximální délka - Vybíráme n-gramy na základě maximální délky jednoho n-gramu. To znamená, že pokud budeme vybírat n-gramy z řetězce (nebo odstavce), který nesplňuje daný maximální požadavek na počet slov/znaků, budeme je i tak ukládat.

Doplnění pozic - Postupujeme stejně jako při předchozí metodě s tím rozdílem, že při extrakci prvního n-gramu využijeme první pozici pro prázdný znak, což nám bude značit začátek řetězce. Na konci zase budeme chybějící znaky doplňovat prázdným znakem. Výhodou této metody je právě označení začátku a konce řetězce, vidět ji můžeme spíše u extrakce n-gramů znaků nežli slov.

Fixní délka - Vybíráme n-gramy pouze stanovené délky. Pokud n-gram nedosáhne požadované délky, je ignorován.

Pro naše účely volíme spíše variantu fixní délky n-gramů. Vzhledem k tomu, že nemůžeme odstraňovat *stopslova* a tím trochu omezit výsledný počet n-gramů, můžeme ho alespoň snížit tímto způsobem. Jednoduše řekneme, že požadovaný řetězec nemá minimální stanovenou délku pro určení shody. Nesmíme zapomínat na to, že budeme zpracovávat velké množství dat, a proto je částečné omezení počtů n-gramů vítáno.

O délce extrahovaných n-gramů necháme rozhodnout uživatele, protože nevíme jaká pro něj bude vhodná délka. Malá velikost n-gramů může být citlivá na detekci změn v textu (přidání nebo odebrání slov) a nemusí nám vznikat tolik unikátních n-gramů, čímž se nám omezí jejich celkový počet. Nevýhoda ovšem spočívá v detekci falešné pozitivní shody. V případě velkých n-gramů již k falešné pozitivní shodě docházet nebude a výsledný počet bude také menší. Nemůžeme již, ale detekovat tolik drobných změn v textu a počet unikátních n-gramů bude také vyšší. Z tohoto důvodu je nutné nechat rozhodnutí o velikosti n-gramů na uživateli.

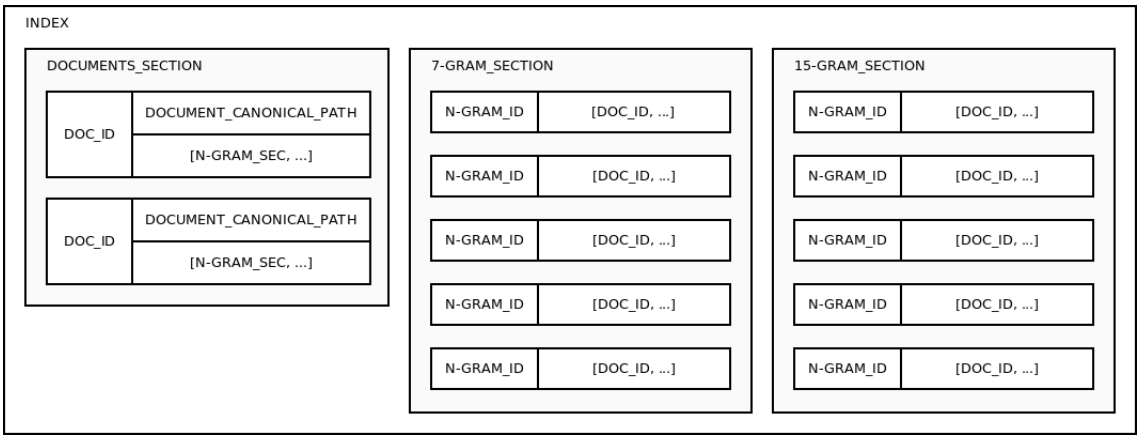
Následně vytváříme haš takto extrahovaných n-gramů. Důvod, proč provádíme hašovaní n-gramů, můžete najít v kapitole 2.3.

4.3 Indexace n-gramů

K extrakci n-gramů z dokumentu budeme využívat hašovací tabulku. Z kapitoly 2.4 je patrné, že hašování a použití hašovací tabulky je pro nás jedním z nejvhodnějších způsobů uchování extrahovaných n-gramů. Vzhledem k tomu, že nepotřebujeme ukládat další informace o n-gramech (hodnotu, četnost a jiné), nemusíme využívat hašovací tabulku, ale stačí nám využít pouze *hašovací sadu*. Tím se nám opět zmenší paměťové nároky na chod aplikace.

Pokud máme takto připravené n-gramy k indexaci, můžeme je uložit do indexu. K vytvoření indexu budeme používat ElasticSearch (viz 3.2), který nám umožní rychlé prohledávání indexu.

Podobu indexu můžeme demonstrovat na obrázku 4.2. Náš index se bude skládat ze dvou nebo více částí, pokud budeme využívat terminologii ElasticSearch, nazvěme tyto části jako *typy*. Typ nazvaný *dokumenty* bude obsahovat informace o zaindexovaných dokumentech. Pro naše účely postačí, když budeme vědět kanonickou URL dokumentu pro jeho jednoznačnou identifikaci a velikosti n-gramů na které byl dokument rozdělen.



Obrázek 4.2: Návrh indexu

Nyní máme dvě možnosti. Můžeme vytvářet klasický index (*forward index*) nebo invertovaný index (*inverted index*). Rozdíl jednotlivých přístupů můžete vidět v tabulce 4.1. Klasický index vytváří seznam slov nebo n-gramů, které se v daném dokumentu vyskytují. Invertovaný index pracuje přesně obráceně. Ten k daným slovům v indexu přiřazuje dokumenty, ve kterých se vyskytly.

Forward index	
Dokument 1	lorem, ipsum, dolor, sit, amet
Dokument 2	dolor, sit, amet
Dokument 3	lorem, sit, amet

Inverted index	
lorem	Dokument 1, Dokument 3
ipsum	Dokument 1
dolor	Dokument 1, Dokument 2

Tabulka 4.1: Příklad klasického a invertovaného indexu

My budeme využívat invertovaný index, který nám umožní rychleji vyhledávat stejné n-gramy. Nebudeme tak muset procházet dokument po dokumentu a zjišťovat, zda neobsahuje hledaný n-gram. Rychlejší varianta je najít n-gram a okamžitě zjistit, ve kterých dokumentech se vyskytuje. Varianta invertovaného indexu má jednu výhodu. Vždy budeme vyhledávat maximálně v $2^{64} - 1$ záznamech, v případě 64 bitového hašování n-gramů. Podobu našeho invertovaného indexu můžete vidět na obrázku 4.2.

4.4 Vyhledávání n-gramů v indexu

Vyhledávání v indexu bude zajišťovat ElasticSearch, naším úkolem v tomto případě bude zpracovat výsledky hledání a vytvořit podrobný výpis výsledků. Výpis by měl obsahovat srovnání jednotlivých odstavců textu, tzn. kolik n-gramů bylo v daném odstavci identifikováno jako duplicitní a ke kterému dokumentu patří. Dále bychom měli zkoumat celkový počet n-gramů, abychom vyhodnotili celkovou míru shody kontrolovaných dokumentů. Výsledkem tak bude seznam dokumentů, které obsahují části podezřelého dokumentu.

Pokud budeme chtít výsledná data nějak zpracovávat, bude vhodné je nějakým způsobem strukturovat. Zde je ideální použít XML, protože můžeme výsledky jednoduše strukturovaně vypsat a přitom zachováme pro uživatele rozumnou formu výpisu.

Kapitola 5

Implementace

Program je vytvořen jako konzolová aplikace, kde se hlavní program stará o zpracování vstupních argumentů, načítání dokumentů a komunikaci s ElasticSearch. Dále využívá třídu *Indexor*, která má na starosti normalizaci textu a extrakci n-gramů.

5.1 Hlavní program

Nyní si popíšeme hlavní program, kde budeme logicky postupovat podle toho, jaký kód se bude provádět během spuštění programu. Začínáme zpracováním argumentů, které je v režii knihovny `Boost`. Jakmile máme argumenty bezchybně načteny provedeme načtení konfiguračního souboru. V případě, že nemůžeme konfigurační soubor načíst, tak se program ukončí. Všechna data z konfiguračního souboru jsou po úspěšném načtení uložena do patřičných globálních proměnných a do struktury `Settings`, která se předává třídě `Indexor` a obsahuje všechny důležité informace k normalizaci textu. Program automaticky načítá soubor `config.xml`, pokud není přepínačem `--config` definováno jinak. Následně provádí jednotlivé kroky, dle toho jaké byly zvoleny argumenty.

Indexace dokumentu

Při indexaci využíváme dva přepínače programu. Jedním volíme délku extrahovaných n-gramů slov a druhým přidáváme prefix před kanonickou cestu k dokumentu. Právě prefix zde umožňuje rozlišovat dva různé dokumenty se stejnou cestou, které pochází například z různých serverů. Prefix nám také umožňuje lepší orientaci v dokumentech, protože je můžeme takto kategorizovat. V dalším kroku, pokud je zadaná cesta k dokumentu správná, kontrolujeme zda se dokument již v indexu nevyskytuje. Jak můžeme vidět na obrázku 4.2, který nám ukazuje strukturu indexu, pro kontrolu nám bude stačit přístup k typu `documents`. Identifikátor každého záznamu v `documents` je hašovaná hodnota kanonické cesty dokumentu, pro základní určení, zda byl dokument vůbec indexován, nám stačí kontrola záznamu s daným identifikátorem. Jestliže se dokument v indexu vyskytl, musíme navíc zkontrolovat zda je indexován právě pod požadovanou hodnotou n-gramu. Pokud ano, je uživatel dotázán zda chce takový dokument přepsat. V případě přepisu, jsou z indexu patřičného typu n-gramu upraveny všechny záznamy. Pokud záznam obsahuje další dokumenty, je z nich odstraněn pouze tento indexovaný dokument, pokud další dokumenty neobsahuje, odstraníme celý záznam. Pokud se dokument v indexu nevyskytuje, vytváříme nový záznam s tímto dokumentem. Jak již bylo psáno, nový záznam má jako identifikátor hašovanou hodnotu

kanonické cesty, uložena je navíc také nehašovaná podoba cesty. Dále se ještě k záznamu přidává ve kterém typu n-gramu je indexován.

Všechny tyto záznamy, ať již dokumentů nebo následně n-gramů, přidáváme pomocí hromadných příkazů (v terminologii ElasticSearch lze nalézt pod názvem *bulk*). Ty nám umožňují využívat skripty, díky kterým jsme schopni vytvářet v ElasticSearch datové struktury typu pole a přidávat do něj jednotlivé položky. Hromadný příkaz může mít teoreticky neomezenou velikost, prakticky je to ale jinak. Velikost hromadných příkazů se nám odvíjí od rychlosti našeho internetu, protože námi využívané komunikační rozhraní čeká na odpověď jen omezenou dobu. V praxi je maximální velikost hromadného příkazu asi 100 příkazů. Může se také stát, že je síť chvílemi přetížená, a proto zkoušíme odeslat hromadný příkaz několikrát. Celkem se o to pokoušíme pětkrát, třikrát hned po sobě a posléze dvakrát po tří sekundových pauzách. O odeslání hromadných příkazů se stará funkce `send_bulk`.

Pokud máme připravený index, můžeme začít zaznamenávat jednotlivé n-gramy. V prvé řadě provádíme instanciaci objektu `Indexor` s jeho potřebnými atributy (délka n-gramu a nastavení normalizace). Následně voláme jednotlivé metody určené ke zpracování textu. První provádíme metodu `Normalize`, která nám dokument normalizuje. Jako druhá přichází na řadu metoda `Ngramize`, která extrahuje n-gramy a vytváří jejich haš. Následně jsou všechny extrahované n-gramy přístupné v proměnné `allngrams` třídy `Indexor`.

Nyní již zaznamenáme n-gramy do indexu. K tomu používáme již zmíněné hromadné příkazy a skripty. Náš vytvořený záznam obsahuje pouze pole dokumentů, ve kterých se vyskytuje. Jeho identifikátor je hašovaná hodnota n-gramu. K vytvoření záznamu využíváme tzv. příkaz `upsert`, který říká, že pokud záznam ještě není vytvořen, nemá se aktualizovat, ale vytvořit. To provádíme následovně:

```
{ "update" : {
    "_id" : "123",
    "_type" : "7gram",
    "_index" : "xpekar10",
    "_retry_on_conflict" : 3
}}
{ "script" : {
    "ctx._source.docs += new_doc",
    "params" : { "new_doc" },
    "upsert" : { "docs" : [ "321" ] }
}}
```

Po indexaci všech n-gramů již jen zavoláme destruktor třídy `Indexor`, uvolníme použitou paměť a můžeme načíst další dokument určený pro indexaci.

Vyhledávání dokumentu

Při vyhledávání postupujeme z počátku podobně jako u indexace. První zpracováváme požadované přepínače, kterých je celkem pět. Jeden z nich volí délku n-gramů, přesně tak jako to bylo u indexace. Druhým volíme přesnost vyhledávání, kde zadaná hodnota v rozsahu $[0, 100]$ nám určuje procentuální přesnost, resp. požadovanou míru shody. To znamená, že vypisujeme jen takové dokumenty, které dosáhly zadané shody, ať již pouze v odstavci nebo celém dokumentu. Třetím přepínačem lze vypsat více informací o nalezeném dokumentu. Předposlední přepínač udává, při jaké minimální sekvenci po sobě jdoucích slov, je odstavec označen za shodný (je přidán do výpisu). Poslední přepínač slouží k určení adresáře pro výstupní soubory v případě, že zpracováváme více jak jeden soubor.

Nyní opět postupujeme stejně jako u indexace. Provádíme instanciaci objektu `Indexor` a normalizujeme dokument. Poté již ale nevoláme metodu `Ngramize`, ale používáme metodu `NgramizeParsOnly`. Tato metoda nám neukládá jen unikátní n-gramy, jak to bylo u `Ngramize`, ale ukládá je všechny a zachovává je rozdělené v jednotlivých odstavcích. To nám pak umožňuje určit míru shody jednotlivých odstavců.

Po zpracování dokumentu můžeme přejít na vyhledávání shodných n-gramů v indexu. Pro zaznamenávání shody používáme dvě hašovací tabulky, `all_duplicates` pro seznam všech shod v celém dokumentu a `par_duplicates` pro seznam všech shod v odstavci. Nyní pomocí dvou do sebe vnořených cyklů `for` procházíme pole obsahující n-gramy. Pomocí metody `getDocument` komunikačního rozhraní pro ElasticSearch získáváme jeden dokument za druhým a procházíme jeho pole dokumentů. Identifikátor záznamu nám slouží jako klíč pro naše hašovací tabulky `all_duplicates` a `par_duplicates`. Pozici definovanou klíčem inkrementujeme. Proměnná `par_duplicates` je na začátku každého vnitřního cyklu inicializována a je tak připravena pro uložení n-gramů nového odstavce.

Všechny nalezené shody, které procentuálně odpovídají zadané hodnotě přesnosti, jsou uloženy do XML. Na tvorbu XML dokumentu zde využíváme knihovnu `PugiXML`. Na konci zpracování je XML dokument vypsán na standardní výstup. Pokud jsme vyhledávali ve více dokumentech, jsou tyto dokumenty uloženy jako soubory s příponou `*.out.xml`.

Vyčištění indexu

Odstranění záznamů z indexu, neboli vyčistění indexu, provádíme rychle a jednoduše. Není potřeba mazat jeden záznam po druhém, to by trvalo příliš dlouho. Jednoduše provedeme odstranění celého indexu a znovu jej vytvoříme. Tím nám vznikne nový prázdný index.

Odstranění dokumentu z indexu

Pokud máme možnost dokumenty indexovat, musíme mít možnost i dokumenty mazat, aniž bychom smazali celý index. Nejdříve kontrolujeme zda je dokument vůbec indexován, pokud ano, získáme všechny záznamy z celého indexu, které tento dokument obsahují. Následně pomocí hromadných příkazů a skriptů odstraňujeme dokumenty z jednotlivých záznamů. Pokud záznam obsahuje jediný dokument, a to ten náš hledaný, mažeme celý záznam. Poté co smažeme všechny záznamy obsahující daný dokument, mažeme ho také z typu `documents`.

Seznam indexovaných dokumentů a typů n-gramů

Vypsání seznamu indexovaných dokumentů je nezbytnou součástí systému. Uživatel musí mít možnost zjistit, které dokumenty byly v indexu zaznamenány a pod jakou velikostí n-gramů byly zpracovány. Všechny tyto informace jsou vypsány na standardní výstup ve formátu XML.

Seznam typů n-gramů se do jisté míry může také hodit, navíc u každého typu vypisujeme i celkový počet záznamů, které obsahuje.

5.2 Třída Indexor

Tato třída je určena pro zpracování vstupních dat, konkrétně je jedná o normalizaci a extrakci n-gramů z textu. Třída obsahuje několik metod, z nichž nejdůležitější jsou metody `Normalize`, `Ngramize` a `NgramizeParsOnly`. Tyto tři metody jsou popsány níže. Dále

`Indexor` obsahuje metodu `GetPar`, která vrací text odstavce normalizovaného textu. Pokud budeme chtít uživateli zpětně ukázat, jaký odstavec byl vyhodnocen jako duplicitní, tak budeme potřebovat právě tuto metodu. Další dvě metody jsou `City_64`, která používá knihovnu CityHash a `IsResChar`. Na závěr zde máme ještě konstruktor, který vyžaduje dva argumenty. Jedním je délka n-gramu a druhým je nastavení normalizace, předávané strukturou `Settings`.

Metoda Normalize

Metoda má jediný argument a tím je vstupní text, který se má zpracovávat. Vstupní text předáváme jako adresu, takže nedochází ke kopii dat a tím šetříme místo v paměti. Vstupní text přemisťujeme pomocí funkce `move` do třídní lokální proměnné `source`, se kterou dále pracujeme. Ke zpracování textu využíváme dvě metody knihovny `Boost`, jsou to metody `replace_all` a `erase_all`. V textu tak provádíme změny definované konfiguračním souborem. Konkrétně je to odstranění definovaných značek, změna konců řádků, změna tabulátorů na mezery a odstranění definovaných znaků, nahrazení stopslov mezerami.

Metody Ngramize a NgramizeParsOnly

Obě metody dělají prakticky totéž, postupně vybírají jeden řádek za druhým z normalizovaného textu a nechávají ho zpracovat metodou `GetParNgrams`. Rozdíl je v tom do jaké struktury následně extrahované n-gramy ukládají a co dělají se zdrojovým textem. Metoda `Ngramize` ukládá n-gramy získané pomocí `GetParsNgrams` do hašovací sady, která uchovává jen unikátní hodnoty. Naproti tomu metoda `NgramizeParsOnly` ukládá n-gramy do dvourozměrného pole tak, aby bylo jasné v jakém odstavci se jaký n-gram vyskytuje a kolikrát. Proměnná hašovací sady je nazvaná `allngrams` a proměnná dvourozměrného pole `allparsngrams`. `Ngramize` navíc ještě v určitých intervalech postupně s tím, jak se zvětšuje hašovací sada, zdrojový text promazává.

Metody takto získávají n-gramy jednotlivých odstavců, což někomu nemusí vyhovovat, protože potřebuje např. vytvořit n-gramy z celého textu neděleného odstavci. V takovém případě stačí jen upravit nastavení normalizace tak, aby se ignorovaly veškeré konce řádků.

Metoda GetParNgrams

Tato metoda převádí vstupní řetězec na n-gramy zadané délky. Algoritmus extrakce je rozdělen na dvě části, v první části testujeme, zda řetězec vůbec obsahuje minimální počet slov na tvorbu n-gramu zadané délky. Pokud je tato podmínka splněna, je vytvořen první n-gram a přecházíme do druhé části, kde pokračujeme s tvorbou dalších. Jak jsme uvedli v 4.2, tímto způsobem tak omezujeme počty n-gramů.

Samotný algoritmus extrakce je poměrně jednoduchý. Postupně procházíme normalizovaný textový řetězec a získáváme jednotlivá slova, která jsou dělena mezerami. Tato slova postupně přidáváme do proměnné `ngram` a zároveň s každým novým přidaným slovem, zaznamenáme jeho pozici v řetězci do pole `ngindex`. Toto pole tak obsahuje všechny počáteční indexy n-gramů obsažených v proměnné `ngram`. Pokud máme vytvořený první n-gram, který splnil požadovanou délku, můžeme se pustit do extrakce dalších. Jako první z proměnné `ngram` odstraníme první slovo, které je označené indexem v `ngindex`, přepočítáme indexy a dekrementujeme proměnnou `words` sloužící k počítání slov v proměnné `ngram`. Následně provedeme proces získání slova z textu a přiřadíme ho na konec k proměnné `ngram`. Jakmile máme stanovený počet slov na daný n-gram, vytváříme haš proměnné `ngram` a přidáváme ji

do pole `ngrams`. Celý proces následně opakujeme dokud se nedostaneme na konec textového řetězce. Na závěr vracíme celé pole extrahovaných a hašovaných n-gramů.

Při extrakci n-gramů byly v minulosti odstraňovány mezery mezi slovy (některé algoritmy tak opravdu činí), což se nakonec neukázalo jako vhodné. V některých případech, kde máme dva podobné, ale rozdílné n-gramy, může dojít při odstranění mezer k tomu, že budou naprosto totožné. Příklad si můžeme ukázat na dvou 3-gramech: „to ma ema“ a „tom a ema“. Zde při odstranění mezer vzniknou dva stejné n-gramy, což je nežádoucí, i když by takto vzniklá chyba byla velice nepravděpodobná a minimální.

5.3 ElasticSearch API

Při implementaci aplikace bylo potřeba několik metod, které API v základu neobsahovalo. Konkrétně se jednalo o metodu `fullScan`, která nám získává jednotlivé záznamy. V základu byla metoda schopna poskytnout záznamy jen při definovaném indexu a typu. My jsme potřebovali, ale získat všechny záznamy ze všech typů na základě daného požadavku. Proto byla navíc implementována další metoda `fullScan`, která nevyžaduje typ.

V některých případech jsme potřebovali také kontrolovat, zda se v indexu vyskytuje určitý typ n-gramů. Původní API obsahovalo pouze metodu `exists` pro kontrolu existence indexu, nikoli typu. Proto jsme přidali navíc i tuto metodu.

API jsme doplnili také o metodu `getMapping` pro získání mapování indexu, což využíváme při získávání dat o indexu a přehledu všech typů n-gramů.

Poslední úprava spočívala ve výpisu některých informací. Při nenalezení požadovaného záznamu, např. při jeho získávání metodou `getDocument`, vrací ElasticSearch chybové hlášení s kódem 404, nenalezeno. V tomto případě se nic zásadního neděje, ovšem knihovna na standardní výstup vypisuje varovné hlášení, což není ideální. Proto byl implementován přepínač na zrušení jejich výpisu.

Kapitola 6

Vyhodnocení systému

V této kapitole se zaměříme na vyhodnocení našeho systému. Budeme provádět několik druhů testů, od porovnání běžně dostupných nástrojů pro extrakci n-gramů, přes testy vyhodnocující míru úspěšností vyhledávání duplicitních textů, až po testy výkonu systému. Dále si také popíše datové sady, na kterých jsme systém testovali a vyhodnocovali.

6.1 Datové sady pro vyhodnocení systému

Pro vyhodnocení systému bylo zvoleno několik datových sad, přitom každá z nich byla využita pro jiný druh testu. Konkrétně jsme využívali PAN korpus, CommonCrawl korpus, Wikipedia dump a vlastní testovací dokumenty.

PAN korpus

Pan korpus[1] představuje vhodný prostředek pro otestování našeho systému v oblasti odhalování plagiátů. Korpus obsahuje přibližně 5000 originálních i podezřelých dokumentů, což celkem činí asi 30 MB textových dat. Součástí korpusu jsou také seznamy dvojic dokumentů, které byly označeny za duplicitní (plagiát).

CommonCrawl korpus

CommonCrawl korpus obsahuje peta bajty dat sesbíraných za 7 let procházení internetu. Korpus obsahuje webové stránky, metadata a textová data. Tato data jsou zdarma ke stažení například na Amazonu nebo na univerzitních serverech. Pro naše účely můžeme použít CommonCrawl z hlediska velkých reálných dat.

Wikipedia dump

Wikipedia[2] nabízí zdarma ke stažení celý její obsah v několika jazycích. Pokud budeme cílit pouze na jednotlivé články, jedná se o 40 GB nekomprimovaných dat. Tyto data nám pomohou otestovat systém v jistém reálném prostředí.

[1] Dostupné z: http://www.uni-weimar.de/medien/webis/events/pan-15/pan15-web/plagiarism-detection.html

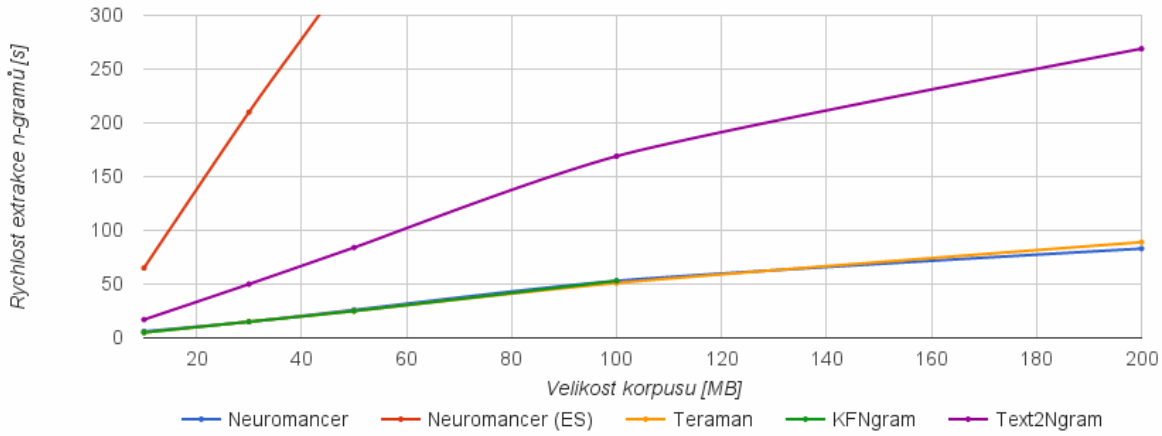
[2] Dostupné z: http://en.wikipedia.org/wiki/Wikipedia:Database_download

6.2 Porovnání rychlosti a paměťové náročnosti

Při vyhodnocování systému jsme provedli dva testy, které měly ověřit úspěšnost extrakce n-gramů v rámci rychlosti a paměťové náročnosti v porovnání s ostatními běžně dostupnými nástroji. Při tomto testu jsme využili část dat korpusu z Wikipedie, kde jsme postupně testovali extrakci n-gramů z dokumentů o velikosti 10, 30, 50, 100 a 200 MB, které byly získány nástrojem Wikipedia Extractor[3]. Oba testy byly provedeny v závislosti na velikosti vstupního dokumentu.

Náš systém můžete v grafech najít pod označením Neuromancer a Neuromancer (ES), rozdíl mezi nimi je v ukládání dat do indexu. Neuromancer ukládá data do klasického indexu (forward index), který vytváří exportem dat do textového souboru. Neuromancer (ES) používá k vytvoření indexu ElasticSearch. Náš systém byl porovnáván s nástroji Teraman (vlastní specifický algoritmus) [6], fkNgram (sufixové pole) [26] a Text2Ngram (sufixový strom) [15].

V prvním grafu na obrázku 6.1 můžete vidět porovnání rychlosti extrakci n-gramů jednotlivých programů. Na první pohled lze vidět, že je náš systém, který využívá ElasticSearch pro indexaci n-gramů, velice pomalý, což je způsobeno pomalejší komunikací se serverem ElasticSearch. Nástroj Text2Ngram také trochu zaostává. Naopak náš systém s klasickou indexací, Teraman a kfNgram indexují n-gramy poměrně rychle, jejich křivka je téměř totožná.

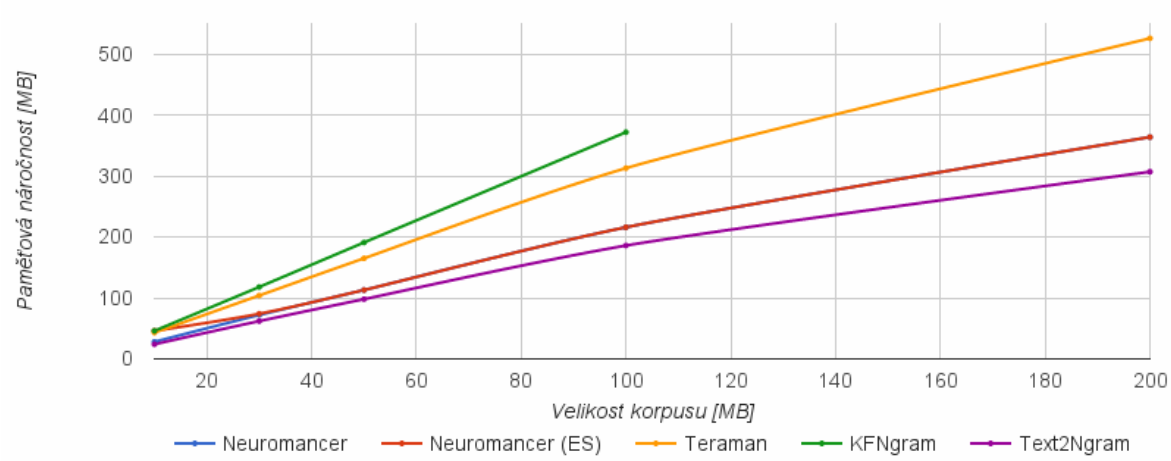


Obrázek 6.1: Rychlost extrakce n-gramů

Druhý graf na obrázku 6.2 ukazuje paměťovou náročnost. V tomto testu nejlépe obstál program Text2Ngram, který vykazuje nejmenší paměťové nároky. Z grafu to není moc patrné, ale náš systém získává od 30 MB stejné paměťové nároky pro oba typy indexace. Hůře už je na tom program Teraman, který vyžaduje skoro o 200 MB více paměti pro 180 MB vstupní soubor než náš systém.

Pokud bychom měli zhodnotit výsledky této fáze testování, tak můžeme říci, že náš systém v testech v porovnání s ostatními programy obstál.

[3]Dostupné z: https://github.com/bwbaugh/wikipedia-extractor



Obrázek 6.2: Paměťová náročnost během procesu extrakce n-gramů

6.3 Testování a vyhodnocení systému

V této kapitole se zaměříme na několik testů. Bude se jednat o test úspěšnosti vyhledání duplicitních textů, vlivu délky n-gramů na výsledek vyhledávání a vlivu délky n-gramů na paměťovou náročnost.

Úspěšnost systému při vyhledávání duplicitních textů

Při tomto testu nám posloužil čistě textový korpus PAN, který je určený k identifikaci duplicitních částí textů. Testovali jsme úspěšnost vyhledávání duplicit v daném korpusu, kde jsme jako měřítko úspěšnosti použili data, která jsou u zdrojových dokumentů přímo uvedena.

Pro test jsme vybrali 30 podezřelých dokumentů, které byly porovnány s 3230 zdrojovými dokumenty. Mezi 30 vybranými dokumenty byly dokumenty, které kopírovali větší části textu, dokumenty snažící se nějakým způsobem kopírovaný text zakrýt a dokumenty, které nijak nekopírovaly zdrojové texty. Velikost n-gramů během testu byla 7 slov a během normalizace jsme odstranili všechna interpunkční znaménka. Během testu jsme vyhledávali dokumenty s přesností 70% shody, dále jsme hledali shodu 10 a 7 postupně po sobě jdoucích slov.

	a = 70%	m = 10	m = 7
Originální	100%	100%	-
Kopie větších částí textu	47%	93%	-
Kopie menších částí textu	0%	29%	69%

Tabulka 6.1: Test úspěšnosti vyhledávání duplicit v textu

Výsledky v tabulce 6.1 nám jasně ukazují, že žádný z originálních dokumentů nebyl označen za plagiát. Při odhalování větších částí textů jsme již nedosahovali tak dobrých výsledků. Je patrné, že podmínka 70% shody je nedostačující, neboť se nám tak podařilo odhalit pouze necelou polovinu dokumentů. Zato podmínka s délkou shody minimálně 10

po sobě jdoucích slov, je již dostačující. V případě menších částí textu 70% shoda vůbec neobstála. Testované dokumenty obsahovaly poměrně dlouhé odstavce, takže pro nás není tento výsledek moc velké překvapení. Ani shoda 10 po sobě jdoucích slov nedosáhla dobrých výsledků, proto jsme se rozhodli vyzkoušet použít shodu 7 slov, kde 7 slov je pro nás minimum vzhledem k velikosti extrahovaných n-gramů. Zde je již výsledek dostačující.

Provedli jsme také test, u kterého jsme se snažili přiblížit reálné situaci při odhalování duplicit v textu. Použili jsme k tomu data z Wikipedie a dalších internetových stránek, přičemž jsme se snažili vytvořit dokument, který by částečně kopíroval některá data z Wikipedie. I zde jsme byli schopni bezpečně určit odstavce nebo jejich části, které pocházely z Wikipedie.

Délka n-gramů a její vliv na detekci duplicit

V tabulce 6.2 můžeme vidět výsledky našeho testu. Při testu jsme použili text o 5 odstavcích do kterých jsme vložili čtyři kopie částí textů. Na první pohled můžeme vidět, že n-gram o velikosti 25 slov nám neodhalil žádnou duplicitu. Zde se stalo to, čemu říkáme falešná negativní shoda (*false negative*). Vložené duplicitní texty nedosahovaly délky 25 slov, proto jsme nebyli schopni je odhalit. U ostatních délek n-gramů byly výsledky více méně stejné.

Můžeme si všimnout, že při délce n-gramů o 3 slovech, jsme nedostali žádnou falešnou pozitivní shodu (*false positive*). Zde se ale může jednat jen o náhodu a špatně zvolený referenční text.

	a = 70%	m = 5	m = 7	m = 15	m = 25
3-gram	1	3	3	1	0
5-gram	1	3	3	1	0
7-gram	1	-	3	1	0
15-gram	1	-	-	1	0
25-gram	0	-	-	-	0

Tabulka 6.2: Test délky n-gramů a jejího vlivu na detekci duplicit

Délka n-gramů a její vliv na paměťové nároky

Již z počátku musí být jasné, že se paměťové nároky budou odvíjet od počtu n-gramů a počet n-gramů od jejich velikosti. V tabulce 6.3 jsou přesně tyto závislosti znázorněny. Můžeme si všimnout, že nejnižší hodnoty počtů n-gramů jsou u velikosti o 3 a 25 slovech. V případě 3-gramů je to způsobeno tím, že sada neobsahuje tolik unikátních hodnot. Naopak u 25-gramů je to tím, že jsme se rozhodli neindexovat data, které nesplňují požadavek na minimální počet slov.

Můžeme zde sledovat zajímavý skok v počtu n-gramů zejména mezi 3-gramem a 5-gramem, kde je rozdíl něco málo přes 8 miliónů n-gramů.

	3-gram	5-gram	7-gram	15-gram	25-gram
Celkový počet n-gramů	17,393,318	25,494,977	25,588,323	22,855,456	19,552,064
Paměťová náročnost [MB]	140	204	205	183	156

Tabulka 6.3: Test délky n-gramů a jejího vlivu na detekci duplicit

V tomto testu jsme opět využili část dat z Wikipedie. Testovaný dokument byl normalizován, byla odstraněna interpunkční znaménka. Paměťová náročnost byla vypočtena s předpokladem využití 64 bitových celočíselných hodnot haší n-gramů.

6.4 Celkové zhodnocení systému

Náš systém jsme hodnotili dvěma způsoby. V prvním případě jsme ho porovnávali s ostatními systémy určenými k extrakci n-ngramů a v druhém případě jsme ho hodnotili z hlediska úspěšnosti vyhledávání duplicit v dokumentech.

Při extrakci n-gramů si náš systém nevedl špatně, dosahoval v tomto testu přibližně stejných výsledků jako ostatní nástroje určené pro extrakci n-gramů. Nedosahoval tak dobrých výsledků pouze v případě využití ElasticSearch pro vytvoření indexu n-gramů. Pokud se na to ale podíváme z druhé strany, tak zrovna ElasticSearch nám naopak umožňuje data rychle vyhledávat.

Pokud bychom měli zhodnotit úspěšnost systému v rámci vyhledávání duplicitních textů, mohli bychom říci, že se správných nastavením normalizace a argumentů programu, bychom byli schopni poměrně bezpečně odhalovat duplicitní části textu dokumentů.

Kapitola 7

Závěr

Zadáním této bakalářské práce bylo analyzovat problematiku detekce duplicit v textu a navrhnout, implementovat a vyhodnotit systém k tomu určený. Výsledný systém je schopen schopen zpracovávat a indexovat dokumenty dle zadaných kritérií a také umožňuje porovnávání zkoumaného dokumentu s indexem. Výstupem je pak XML dokument obsahující výsledky porovnávání.

Součástí systému je také algoritmus pro extrakci n-gramů. Proto jsme porovnali náš program s běžně dostupnými nástroji k tomu určenými. Výsledek porovnání lze považovat za uspokojující, neboť rychlost a paměťová náročnost se výrazně nelišila od ostatních systémů a dosahovala poměrně dobrých výsledků. Bylo provedeno také testování schopnosti odhalit duplicitní texty v dokumentech. Zde jsme při správné konfiguraci dosáhli více než devadesátiprocentní úspěšnosti.

Aplikaci je možné v budoucnu rozšířit o grafické uživatelské rozhraní, které umožní uživateli zobrazit části kopírovaného textu přímo v podezřelém dokumentu v původní nenormalizované podobě.

Literatura

[1] Allen, W. M.; aj.: *Data Structures and Algorithm Analysis in C++*. Pearson Education India, 2007, iSBN 978-0-13-284737-7.

[2] Cavnar, W. B.; Trenkle, J. M.; aj.: N-gram-based text categorization. *Ann Arbor MI*, ročník 48113, č. 2, 1994: s. 161–175.

[3] Češka, Z.: Porovnání technik předzpracování textu pro detekci plagiátů. *Znalosti 2009*: s. 293–296.

[4] Ceska, Z.: The future of copy detection techniques. *Proc. YRCAS*, 2007: s. 5–10.

[5] Češka, Z.: Využití n-gramů pro odhalování plagiátů. In *Proceedings of ITAT 2007*, 2007.

[6] Češka, Z.; Hanák, I.; Tesař, R.: Extrakce N-gramů z rozsáhlých textů. 2007: s. 209–216, iSBN 978-1-4244-1491-8.

[7] Chowdhury, A.; Frieder, O.; Grossman, D.; aj.: Collection statistics for fast duplicate document detection. *ACM Transactions on Information Systems (TOIS)*, ročník 20, č. 2, 2002: s. 171–191.

[8] ElasticSearch: Elasticsearch - Search & Analyze Data in Real Time [online]. https://www.elastic.co/products/elasticsearch, [cit. 2015-05-11].

[9] Fan, J.; Huang, T.: A fusion of algorithms in near duplicate document detection. In *New Frontiers in Applied Data Mining*, Springer, 2012, s. 234–242.

[10] Gormley, C.; Tong, Z.: *Elasticsearch: The Definitive Guide*. O'Reilly & Associates, 2014, iSBN 978-1-449-35854-9.

[11] Heintze, N.; aj.: Scalable document fingerprinting. In *1996 USENIX workshop on electronic commerce*, ročník 3, 1996.

[12] Hoad, T. C.; Zobel, J.: Methods for identifying versioned and plagiarized documents. *Journal of the American society for information science and technology*, ročník 54, č. 3, 2003: s. 203–215.

[13] Kasprzak, J.; Brandejs, M.; Kripac, M.: Finding plagiarism by evaluating document similarities. In *Proc. SEPLN*, ročník 9, 2009, s. 24–28.

[14] Kasprzak, J.; Brandejs, M.; aj.: Improving the reliability of the plagiarism detection system. *Notebook Papers of CLEF*, 2010.

[15] Le Zhang: Text2Ngram [online]. http://homepages.inf.ed.ac.uk/lzhang10/ngram.html, [cit. 2015-05-13].

[16] Neustar Research: Choosing a Good Hash Function, Part 3 [online]. http://research.neustar.biz/2011/12/29/choosing-a-good-hash-function-part-2/, 2011-11-27 [cit. 2015-05-11].

[17] Neustar Research: Choosing a Good Hash Function, Part 2 [online]. http://research.neustar.biz/2011/12/29/choosing-a-good-hash-function-part-2/, 2011-12-29 [cit. 2015-04-30].

[18] Neustar Research: Big Memory, Part 3.5: Google sparsehash! [online]. http://research.neustar.biz/2011/11/27/big-memory-part-3-5-google-sparsehash/, 2012-02-02 [cit. 2015-05-11].

[19] Pomikálek, J.: Removing boilerplate and duplicate content from web corpora. *Disertačnı práce, Masarykova univerzita, Fakulta informatiky*, 2011.

[20] Potthast, M.; Stein, B.: New issues in near-duplicate detection. In *Data Analysis, Machine Learning and Applications*, Springer, 2008, s. 601–609, iSBN 978-3-540-78239-1.

[21] Preshing on Programming: Hash Collision Probabilities [online]. http://research.neustar.biz/2011/11/27/big-memory-part-3-5-google-sparsehash/, 2011-05-04 [cit. 2015-04-30].

[22] Q-Hedge Technologies: C++ Client for ElasticSearch [online]. https://github.com/QHedgeTech/cpp-elasticsearch, 2014-12-15 [cit. 2015-05-04].

[23] Robenek, D.; Platos, J.; Snásel, V.: Efficient In-memory Data Structures for n-grams Indexing. In *DATESO*, Citeseer, 2013, s. 48–58, iSBN 978-80-248-3968-5.

[24] Schäling, B.: *The boost C++ libraries*. XML Press, 2014, iSBN 978-1-937-43436-6.

[25] SparseHash: An extremely memory-efficient hash_map implementation [online]. https://code.google.com/p/sparsehash/, 2012-02-23 [cit. 2015-05-11].

[26] William H. Fletcher: kfNgram [online]. http://www.kwicfinder.com/kfNgram/kfNgramHelp.html, 2012-08-29 [cit. 2015-05-13].

[27] Yang, H.; Callan, J.: Near-duplicate detection by instance-level constrained clustering. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2006, s. 421–428.

Dodatek A

Obsah DVD

Práce obsahuje přiložené DVD, na kterém naleznete všechny dokumenty tykající se naší práce. V kořenovém adresáři též najdete text technické zprávy k programu ve formátu PDF. Adresářová struktura DVD je následující:

- **bin** - spustitelné binární soubory
- **config** - konfigurační soubory
- **examples** - ukázkové soubory a pomocné skripty
- **lib** - knihovny potřebné pro překlad
- **obj** - objektové soubory
- **src** - zdrojové soubory programu
- **tex** - zdrojové soubory technické zprávy v LaTeXu
- **tools** - nástroje potřebné pro chod aplikace
- **LICENSE.txt**
- **README.txt**
- **Makefile**
- **Poster.png**
- **Duplicate_text_identification.pdf**

Dodatek B

Manuál

Aplikace je určena k vyhledávání duplicitních textů v zadaných dokumentech. Lze ji využít dvěma způsoby, pro indexaci dokumentů a porovnávání dokumentů s indexem. K indexaci využívá principu dělení textu na menší fragmenty, které zaznamenává do indexu a následně je porovnává se stejně získanými částmi podezřelého dokumentu. Ke svému správnému chodu využívá program ElasticSearch, který nám vytváří celý index.

Jedná se o konzolovou aplikaci, kde na vstupu očekáváme soubor učený pro indexaci nebo podezřelý soubor určený k porovnání s indexem. Výstupem je pak XML dokument označující duplicitní odstavce, resp. dokumenty. V manuálu můžete najít návod na instalaci a spouštění aplikace, dále také popis některých přepínačů.

Tato aplikace byla vytvořena jako součást bakalářské práce na téma Vyhledávání duplicitních textů.

B.1 Instalace

Aplikace je určena pro systému Linux, kde k instalaci potřebujete většinou běžně dostupné nástroje, které tyto systémy obsahují. Jedná se o nástroje:

- GNU Make
- GNU GCC 4.8.2 nebo vyšší
- Knihovna Boost 1.57.0 nebo vyšší

Ostatní knihovny nutné pro překlad jsou přibalené ve složce `lib`, složka také obsahuje knihovnu Boost verze 1.57.0. Knihovny přibalené k aplikaci není potřeba nijak nastavovat, výjimkou je knihovna Boost. Pokud máte již v systému nainstalovanou knihovnu Boost, ujistěte se, že používáte verzi 1.57.0 nebo vyšší. Teoreticky je možné použít Boost od verze 1.50.0, kde byla ve třídě `filesystem` zavedena metoda `Canonical`. Nicméně použití nižší verze Boost než je 1.57.0 nebylo testováno.

Pokud se rozhodnete využívat vlastní knihovnu Boost, je potřeba ze souboru `Makefile` odstranit text z proměnných `BOOSTIP` a `BOOSTLD`. Sestavovací program si následně najde knihovnu Boost nainstalovanou ve vašem systému.

V případě, že budete využívat knihovnu Boost přibalenou k programu, nemusíte v Makefile nic měnit. Je ovšem nutné kvůli některým funkcím knihovnu Boost přeložit. Při překladu knihovny je potřeba použít následující příkazy:

```
./bootstrap.sh
./b2
```

V adresáři `lib/boost_1_57_0` jsou již knihovny předkompilované, takže je pravděpodobně nebudete muset instalovat výše uvedeným způsobem. V případě problémů je doporučujeme smazat a znovu zkompilovat.

Nás program pak stačí zkompilovat pomocí `Makefile`, umístěném v kořenovém adresáři pomocí následujícího příkazu:

```
make
```

Na závěr je potřeba nakonfigurovat ElasticSearch[1]. Verzi 1.4.4 můžete najít na DVD v adresáři `tools`. Pro správný chod aplikace je potřeba, aby měl ElasticSearch v konfiguraci povolené tzv. dynamické skripty. Hotový konfigurační soubor je ve složce `config` v kořenovém adresáři, stačí ho jen nahradit za ten, který je ve výchozím nastavení uveden v adresáři ElasticSearch ve složce `config`. Soubory nemusíme nahrazovat, ale stačí přidat do konfigurace následující řádek:

```
script.disable_dynamic: false
```

Program byl testován pro ElasticSearch verze 1.4.4. Další informace o konfiguraci ElasticSearch a jeho instalaci naleznete na [8].

B.2 Spuštění

Ze všeho nejdříve je potřeba spustit ElasticSearch. Jeho spustitelný soubor můžete nalézt ve složce `bin` jeho adresáře. Po spuštění je nutné také vytvořit index, se kterým budeme pracovat, neboť naše aplikace sama neumí index vytvořit. Vytvoření indexu může být provedeno následovně pokud spouštíme ElasticSearch na `localhost`:

```
curl -XPOST 'localhost:9200/xpekar10'
```

V případě, že nespouštíme ElasticSearch na `localhost`, použijeme jinou adresu se stejným portem (pokud není v nastavení specifikováno jinak).

Jestliže jsme vytvořili index `xpekar10` na `localhost` na portu `9200`, musíme upravit konfigurační soubor. Upravíme následující části konfiguračního souboru:

```
<host>localhost:9200</host>
<index>xpekar10</index>
```

Pokud bychom chtěli měnit další konfiguraci týkající se normalizace dokumentu, museli bychom upravit i další části:

```
<new_line>true</new_line>
<new_line_tags><nlt><![CDATA[</p>]]></nlt></new_line_tags>
<delete_tags><dt><![CDATA[<p>]]></dt></delete_tags>
<delete_chars>.,:;?!-_-/\"(){}'</delete_chars>
<stopwords><sw>a</sw></stopwords>
```

Naši aplikaci spustíme v adresáři `bin` následujícím způsobem:

```
./neuromancer
```

[1] Dostupné z: https://www.elastic.co/products/elasticsearch

Pokud spustíme aplikaci bez jakýchkoli parametrů, dostane se nám nápovědy k programu. Pojďme si nyní ukázat jednoduchou indexaci souborů umístěných v adresáři `examples`:

```
./neuromancer -i ../examples/simple-test
```

Výstupem programu by pak měl být výpis zpracovaných dokumentů. V našem případě jsme zpracovávali celou složku, která obsahuje dva dokumenty. Jako výchozí se volí délka n-gramů 7 slov. Pokud bychom chtěli změnit délku n-gramů, např. na 5 slov, použili bychom přepínač `-n`. Jako druhý příklad si uveďme indexaci dokumentů s jinou délkou n-gramů a výpisem obsahujícím celkový počet indexovaných n-gramů a využití paměti:

```
./neuromancer -i ../examples/simple-test -n5 -v
```

Pro kontrolu zaindexovaných dat můžeme využít přepínače, které nám vypíší seznam zaindexovaných dokumentů a celkový počet n-gramů jednotlivých délek:

```
./neuromancer --get-docs
./neuromancer --get-ngram-list
```

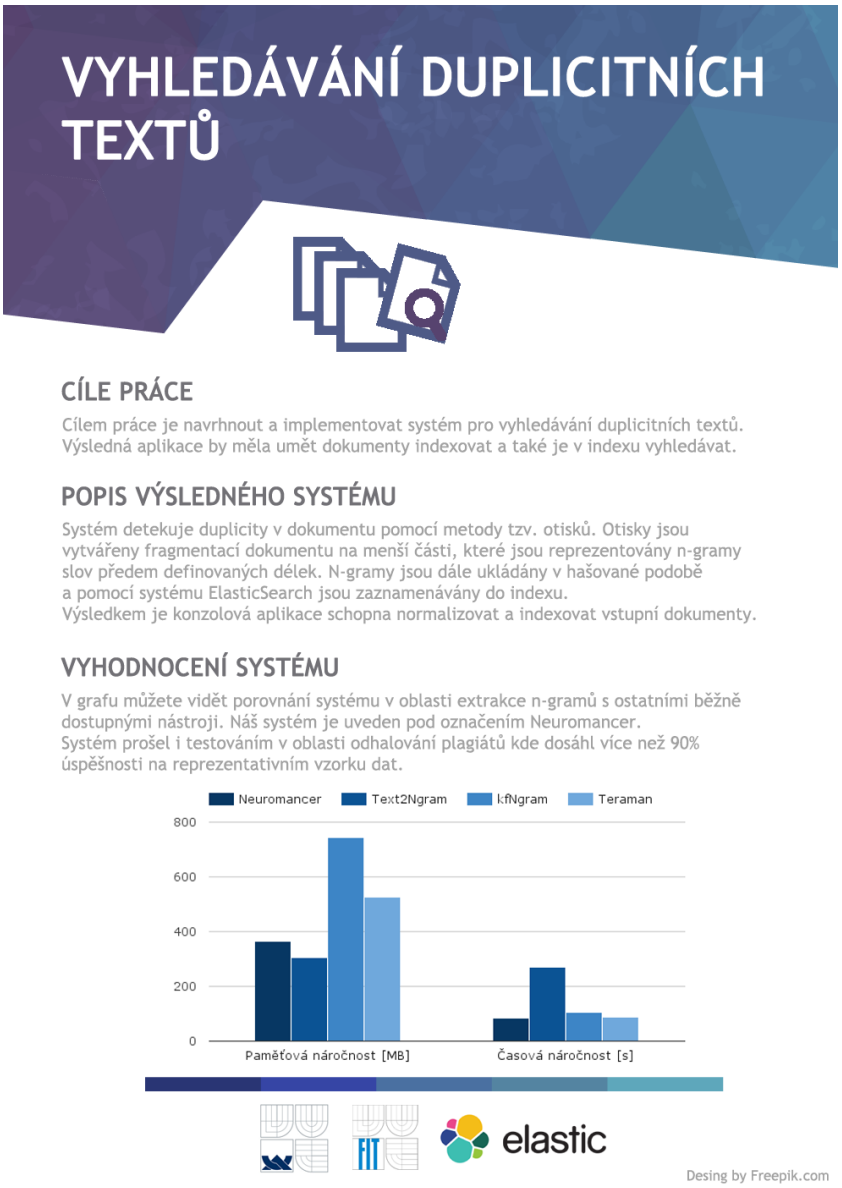
Nyní si ukážeme, jak porovnávat dokumenty uložené v indexu. Pro tento příklad použijeme dokumenty ze složky lorem-ipsum, která obsahuje jeden zdrojový a jeden podezřelý dokument:

```
./neuromancer -i ../examples/lorem-ipsum/source.txt
./neuromancer -f ../examples/lorem-ipsum/suspect.txt -a50 -m7
```

Příkazem pro vyhledávání jsme určili, že budeme vypisovat odstavce nebo dokumenty, které dosahují minimálně 50% shody nebo takové odstavce či dokumenty, které se shodují v sekvenci 7 slov. Výstupem je pak XML dokument, který uvádí, ve kterých dokumentech či odstavcích byla nalezena shoda. Pro detailnější výstup můžeme použít parametr `-w`.

Dodatek C

Plakát



VYHLEDÁVÁNÍ DUPLICITNÍCH TEXTŮ

CÍLE PRÁCE

Cílem práce je navrhnout a implementovat systém pro vyhledávání duplicitních textů. Výsledná aplikace by měla umět dokumenty indexovat a také je v indexu vyhledávat.

POPIS VÝSLEDNÉHO SYSTÉMU

Systém detekuje duplicity v dokumentu pomocí metody tzv. otisků. Otisky jsou vytvářeny fragmentací dokumentu na menší části, které jsou reprezentovány n-gramy slov předem definovaných délek. N-gramy jsou dále ukládány v hašované podobě a pomocí systému ElasticSearch jsou zaznamenávány do indexu.
Výsledkem je konzolová aplikace schopna normalizovat a indexovat vstupní dokumenty.

VYHODNOCENÍ SYSTÉMU

V grafu můžete vidět porovnání systému v oblasti extrakce n-gramů s ostatními běžně dostupnými nástroji. Náš systém je uveden pod označením Neuromancer.
Systém prošel i testováním v oblasti odhalování plagiátů kde dosáhl více než 90% úspěšnosti na reprezentativním vzorku dat.