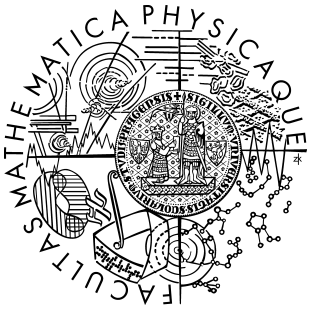
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Václav Matouš

Efektivní ukládání html stránek

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Žemlička, PhD.
Studijní program: Informatika, Datové inženýrství

2007

Rád bych poděkoval doktoru Michalu Žemličkovi, vedoucímu této práce, za všestranné rady a pomoc při jejím vypracování.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 7. srpna 2007 Václav Matouš

Obsah

Název práce: Efektivní ukládání html stránek
Autor: Václav Matouš
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Žemlička, PhD.
e-mail vedoucího: zemlicka@ksi.mff.cuni.cz

Abstrakt: V rámci této práce byl vytvořen prototyp úložiště html dokumentů. Ukládání celých dokumentů je založeno jak na stávajících kompresních technikách (gzip, bzip2), tak i na novým metodách, kdy se dokument před kompresí vhodným způsobem předzpracuje. Ukládá-li se k jednomu dokumentu více verzí, mohou některé z nich být uloženy v podobě rozdílového souboru – je-li to v daném případě výhodné. Tyto rozdílové soubory mohou být opět komprimovány. V testech se metody z předzpracováním ukázaly jako mírně účinnější než metody původní. Rozdíl mezi účinností testovaných bezeztrátových a ztrátových metod předzpracování je relativně malý. Kombinace výše uvedených technik se v testech ukázala jako vhodná náhrada za kompaktní (solidní) archivy verzí jednotlivých dokumentů, přičemž navíc poskytuje velmi rychlý přístup k poslední (aktuální) verzi dokumentu, o které předpokládáme, že bude tou nejčastěji požadovanou.
Klíčová slova: archiv, html, komprese, verze

Title: Effective storage for html documents
Author: Václav Matouš
Department: Department of Software Engineering
Supervisor: RNDr. Michal Žemlička, PhD.
Supervisor's e-mail address: zemlicka@ksi.mff.cuni.cz

Abstract: In the presented work a prototype of a storage for html documents is designed and implemented. The storing of complete documents is based on both current compression techniques (gzip, bzip2) and new methods. An input document is modified by the new methods and compressed by the current techniques. If there are stored many versions of the same document, some of them can be stored in the form of differential files – in the case that it is more useful. These differential files can be compressed as well. The compression methods with preprocessed input are a bit more effective than the original methods. The difference of efficiency of lossy and lossless compression methods is in this case relatively small. The performed tests showed that a combination of mentioned techniques could be a suitable substitution of compact archives for versioned documents. The implementation guarantees quick access to the actual version what is very important as we suppose that such versions will be the most requested ones.
Keywords: archive, compression, html, version

Kapitola 1

Úvod

1.1 Zadání práce

Vytvořte knihovnu umožňující efektivně ukládat velké množství html stránek. Jednotlivé html stránky musí být snadno (rychle) přístupné. Je třeba ukládat nejen aktuální, ale i starší verze stránek.

Důležitý je nejen dosažený kompresní poměr, ale i to, jak rychle se dají získat zejména nejnovější verze uložených stránek.

1.2 Motivace

Fulltextový systém (např. EGOTHOR [11]) stahuje z Webu velké množství dokumentů, indexuje je a umožňuje v nich vyhledávat na základě zadaných dotazů. Výsledkové listiny obsahují vedle URL nalezeného dokumentu také výstřižek z tohoto dokumentu. Aby bylo možné tento výstřižek sestavit, musí fulltextový systém znát originální dokument (typicky ve formátu html). Zde vzniká potřeba ukládat velké množství html dokumentů.

Obsah řady ukládaných dokumentů se v různě dlouhých časových intervalech mění, např. zpravodajské servery, diskuzní fóra či seznam publikovaných článků. Vzhledem k tomu, že vedle nejnovější verze dokumentu mohou být vyžadovány i verze starší, musí být fulltextový systém schopný ukládat různé verze téhož html dokumentu.

Z hlediska počtu požadavků na získání jednotlivých verzí bude nejčastějším požadavkem získání nejnovější verze. Doba potřebná pro získání nejnovější verze by proto měla být co nejkratší. Ke starším verzím dokumentu nebude přistupováno tak často, můžeme tudíž preferovat účinnost komprese před dobou přístupu k těmto verzím. Dosáhneme tak vysokého stupně komprese při zachování rychlého přístupu k nejnovějším verzím dokumentu.

1.3 Cíle a úkoly práce

Cílem této práce je navrhnout a implementovat knihovnu, která bude sloužit k ukládání velkého množství html dokumentů a k následnému přístupu k nim. Ze zadání plynou dva relativně nezávislé podcíle – nalezení účinné komprese ukládaných dokumentů pro redukci potřebného diskového prostoru a návrh vhodné

organizace ukládání souborů pro minimalizaci doby nutné k získání především nejnovějších verzí uložených dokumentů.

1.4 Výchozí stav

Abychom byli schopni zhodnotit výsledné řešení, potřebujeme vědět, jaká jsou existující řešení. Knihovna, která je předmětem této práce, má sloužit k ověření nových metod pro případné nasazení v úložišti pro fulltextový systém EGOTHOR [5]. Podívejme se tedy, jak ukládal tento systém dokumenty v době zahájení této práce.

Systém EGOTHOR z přibližně tisíce dokumentů vytvoří jeden archiv, který zkomprimuje obecnou metodou gzip. Index pak obsahuje také odkaz na archiv, ve kterém je příslušný dokument uložen. Nabízí se hned několik vylepšení existujícího řešení, která jsou v rámci této práce navržena a otestována.

Prvním vylepšením může být zmenšení velikosti komprimovaného archivu a tím úspora diskového prostoru. Důvodem pro toto vylepšení může být fakt, že nyní používaná kompresní metoda gzip je určena pro obecné dokumenty. Pro html dokumenty, typické svými značkami, nedosahuje gzip nejlepších výsledků, neboť nevyužívá struktury dokumentu, ani případné znalosti o textovém obsahu dokumentu. Vytvářený archiv může obsahovat různorodé dokumenty, tedy dokumenty s odlišnou strukturou i obsahem. Tím se zmenšuje možnost nalezení stejných řetězců, které zvyšují účinnost komprese. Tady se otevírá cesta pro případná vylepšení.

Při získání souboru z archivu komprimovaného metodou gzip je nutné dekomprimovat celý archiv. Možnost přímého přístupu k jednotlivým souborům uloženým v archivu bez nutnosti jeho celé dekomprese by mohla přinést další zkvalitnění v podobě časové úspory.

Kapitola 2

Komprese ukládaných dokumentů

2.1 Stávající techniky

Pro kompresi html dokumentů připadá v úvahu v současné době několik kompresních technik, především obecných kompresních metod a ukládání rozdílu dokumentů (diff) pro různé verze téhož dokumentu.

2.1.1 Gzip

Gzip [4] je obecná kompresní metoda založená na kompresi LZ77 [21] resp. na její variantě LZSS [19]. Kompresní algoritmus hledá duplicitní řetězce (obecně posloupnosti bajtů) ve vstupních datech. Druhý výskyt řetězce je nahrazen ukazatelem na předešlý výskyt téhož řetězce ve formě dvojice (vzdálenost, délka). Vzdálenosti jsou omezeny na 32 kB, délky na 258 bajtů. Pokud se řetězec nevyskytl nikde v předešlých 32 kB, pak jde na výstup jako posloupnost literálů (samostatných bajtů).

Literály a délky nalezených shod jsou komprimovány jedním Huffmanovým stromem [7], vzdálenosti nalezených shod jsou komprimovány jiným Huffmanovým stromem. V obyčejném Huffmanově algoritmu může jedna množina kódovaných elementů a jejich vah generovat více různých Huffmanových stromů. Algoritmus gzip používá drobnou modifikaci Huffmanových stromů v podobě dvou přidaných pravidel: elementy s kratšími kódy jsou umístěny nalevo od těch s delšími kódy a z elementů, které mají stejnou délku kódů, je vlevo element, který se v množině elementů vyskytl dříve. Tato přidaná pravidla zaručují pro každou množinu elementů a jejich vah vygenerování právě jednoho Huffmanova stromu. Z jednoznačnosti spolu se znalostí množiny elementů plyne, že jediná informace potřebná pro rekonstrukci Huffmanova stromu jsou délky kódových slov přiřazených jednotlivým elementům.

Stromy jsou pak ukládány na začátku každého bloku jako sekvence délek jednotlivých kódových slov. Bloky mohou mít libovolnou velikost (dokud se komprimovaná data vejdou do dostupné paměti). Blok končí, když kompresní algoritmus rozhodne, že by bylo užitečné začít nový blok s novými stromy.

Duplicitní řetězce se vyhledávají pomocí hašovací tabulky. Všechny vstupní řetězce délky 3 se vloží do hašovací tabulky. Hašovací index je spočítán pro ná-

sledující 3 bajty. Pokud hašovací sekvence pro tento index není prázdná, všechny řetězce v hašovací sekvenci jsou porovnány s aktuálním vstupním řetězcem a vybírá se nejdelší shoda.

Hašovací sekvence jsou vyhledávány počínaje nejnovějšími řetězci, kvůli upřednostnění menších vzdáleností nalezené shody a tím pádem využití předností Huffmanova kódování. Hašovací sekvence jsou jednoduše spojovány. Není z nich třeba mazat, algoritmus jednoduše zruší shody, které jsou příliš staré. Algoritmus ne vždy nalezne nejdelší možnou shodu, ale obecně nalezne shodu dostatečně dlouhou.

Kompresní algoritmus dále oddaluje výběr shody mechanismem líného vyhodnocování. Potom, co byla nalezena shoda délky n, hledá algoritmus delší shodu na dalším vstupním bajtu. Pokud je nalezena delší shoda, předešlá shoda je zkrácena na délku jedna (tudíž vzniká jednoduchý literál) a proces líného vyhodnocování začíná znovu. V opačném případě se původní shoda zachová a další shoda se zkouší najít pouze o n kroků později.

Při dekompresi se kvůli efektivitě nestaví znovu celý Huffmanův strom, ale používají se víceúrovňové vyhledávací tabulky. Krátká kódová slova, která se vyskytují nejčastěji a jejich dekódování by tedy mělo být co nejrychlejší, jsou dekódována pomocí tabulky první úrovně. Delší kódová slova lze dekódovat pomocí tabulek dvou, případně i více tabulek. Počet bitů (délka tabulky) pro klíče v tabulce první úrovně by měl být nižší než je počet bitů nejdelšího kódového slova. Hodnota vztažená k tomuto klíči udává jednak symbol, kterému kódové slovo patří, a také počet bitů, které tvoří příslušné slovo. Pokud je kódové slovo delší než klíč v tabulce první úrovně, obsahuje tabulka odkaz na tabulky nižších úrovní a také počet bitů z klíče, které tvoří prefix slova. Tabulky nižších úrovní se vytvářejí podobně. Dekompresní algoritmus metody gzip si vystačí se dvěma úrovněmi.

Ukázka dekódování pomocí tabulek. Mějme 10 zakódovaných symbolů, jejichž kódová slova mají délky od 1 do 6:
A: 0, B: 10, C: 1100, D: 11010, E: 11011, F: 11100, G: 11101, H: 11110, I: 111110, J: 111111
Nechť tabulka první úrovně má velikost 3 bity (8 klíčů):
000: A, 1
001: A, 1
010: A, 1
011: A, 1
100: B, 2
101: B, 2
110: -> tabulka X (použité všechny 3 bity)
111: -> tabulka Y (použité všechny 3 bity)
Tabulka X bude 2 bity velká, neboť nejdelší slovo začínající 110 má délku 5:
00: C, 1
01: C, 1
10: D, 2
11: E, 2
Tabulka Y má velikost 3 bity, neboť nejdelší kódové slovo s prefixem 111 má délku 6:
000: F, 2

001: F, 2
010: G, 2
011: G, 2
100: H, 2
101: H, 2
110: I, 3
111: J, 3

2.1.2 Bzip2

Obecná kompresní metoda bzip2 [17] komprimuje soubory použitím Burrows-Wheelerovy transformace (BWT) [1] a Huffmanova kódování [9]. Komprese metodou bzip2 je obecně pro dostatečně velká data znatelně lepší než komprese metodami založenými na algoritmu LZ77 nebo LZ78 [22].

BWT je také známa jako kompresní algoritmus třídící bloky. Pokud původní řetězec obsahoval řadu shodných podřetězců, pak po transformaci bude řetězec obsahovat několik úseků, kde se bude vyskytovat pouze jediný znak mnohokrát za sebou. To je užitečné pro kompresi, protože je jednoduché komprimovat dlouhé sekvence stejných znaků transformací move-to-front a kódováním délek sekvencí.

Samotná BWT probíhá ve dvou fázích: tvorba matice a move-to-front transformace. V první fázi se vytvoří všechny rotace komprimovaného řetězce a setřídí se podle abecedy, viz obrázek 2.1

A	N	A	N	A	S
A	N	A	S	A	N
A	S	A	N	A	N
N	A	N	A	S	A
N	A	S	A	N	A
S	A	N	A	N	A

Obrázek 2.1: Setříděné rotace komprimovaného slova

Ve druhé fázi se pomocí move-to-front tranformace zakóduje poslední sloupec vytvořené matice. Znaky se při move-to-front transformaci kódují aktuální pozicí znaku ve slovníku. Zároveň se znak přesune na začátek slovníku. Transformaci posledního sloupce matice z první fáze demonstruje obrázek 2.2. Předpokládejme, že slovník pro transformaci je již naplněný a má podobu A N S.

Kódovaný znak		S	N	N	A	A	A
Slovník	A N S	S A N	N S A	N S A	A N S	A N S	A N S
Kód		3	3	1	3	1	1

Obrázek 2.2: Move-to-front transformace posledního sloupce matice

Poslední sloupec matice S N N A A A je pomocí move-to-front transformace zakódován jako 3 3 1 3 1 1. Ještě je třeba kvůli zpětné transformaci přidat číslo

řádku v matici, na kterém byl původní komprimovaný řetězec. V našem případě kód 1.

Bzip2 poté výsledek move-to-front transformace kóduje pomocí Huffmanova kódování, které frekventovanějším entitám přiřazuje kratší kódy.

Dekomprese ale není proces inverzní ke kompresi. Při dekompresi se snadno rozkóduje výsledek move-to-front transformace. Tím se získá poslední sloupec matice. První sloupec matice se získá také poměrně snadno, stačí poslední sloupec setřídit podle abecedy, neboť ve všech sloupcích se vyskytují stejné znaky. Problém je nalezení zbývajících sloupců. Platí invariant, že řádky matice začínající stejným znakem jsou vždy ve stejném pořadí.

Označme L poslední sloupec matice, F první sloupec matice a $T[j] = i$ označuje fakt, že si odpovídají pozice $L[j]$ a $F[i]$. Pak $L[T[j]] = F[i]$. Zbývající sloupce matice se zrekonstruují opakovaným použitím této transformace, jak je ilustrováno na obrázku 2.3.

$T[j] = i$	L	F					
$T[1] = 6$	S	A	N	A	N	A	S
$T[2] = 4$	N	A	N	A	S	A	N
$T[3] = 5$	N	A	S	A	N	A	N
$T[4] = 1$	A	N	A	N	A	S	A
$T[5] = 2$	A	N	A	S	A	N	A
$T[6] = 3$	A	S	A	N	A	N	A

Obrázek 2.3: Rekonstrukce matice v BWT

Komprimovaný řetězec se v původní matici nacházel v prvním řádku, ve zrekonstruované matici je rovněž v prvním řádku, tedy ANANAS. Alternativou ke kódování čísla řádku s komprimovaným řetězcem je přidání znaku konce řetězce na konec komprimovaného řetězce a úplně stejné provedení komprese a dekomprese. Hledaný řetězec se pak ve zrekonstruované matici nachází na řádku, ve kterém je posledním znakem právě znak konce řetězce.

Metoda bzip2 používá při BWT bloky velikosti 100 kB, 200 kB, ..., 900 kB. Komprese je nejúčinnější při použití bloku velikosti 900 kB, ale zároveň je tato komprese nejnáročnější na paměťové kapacity a hlavně nejpomalejší. Větší blok také negativně ovlivňuje rychlost dekomprese, protože se musí třídit větší bloky.

2.1.3 Rozdíl souborů

Ukládají-li se různé verze téhož dokumentu, může být mnohem výhodnější neukládat celé dokumenty, ale pouze jejich rozdíl. Původní algoritmus diff [8] rozlišuje rozdíly ve dvou souborech na úrovni řádků. Výsledkem porovnání dvou dokumentů je (pokud možno) minimální množina operací (přidání, smazání nebo změna řádku), jejichž aplikací lze jeden dokument transformovat na dokument druhý, např.:

```
1 d 1
3 c 1
změněný třetí řádek
4 a 2
```

```
jeden přidaný řádek
druhý přidaný řádek
```

Výstup rozdílového porovnání dvou dokumentů lze uložit místo jednoho z dokumentů. Z pohledu komprese (úspory prostoru) mohou být problémem zvláště operace typu změna řádku, protože i když se dva řádky ve dvou dokumentech liší byť jen v jediném znaku, je jeden z těchto řádků uložen celý do rozdílu souborů. To může vést k nepříjemnému nárůstu velikosti rozdílového souboru.

Tento problém se pokouší vyřešit práce [14]. Místo diffu na úrovni řádek zavádí víceúrovňový diff. Klasický řádkový diff rychle a efektivně vygeneruje změny všech tří typů. U změny typu změna řádku se dále zkoumá velikost změny. Pokud není velká, použije se diff na úrovni slov na tento řádek. Obdobně při změně slova se zkoumá její rozsah a případně se použije diff na úrovni znaků. Tento přístup přinesl prostorovou úsporu, která je zaplacena větší časovou náročností.

Použití víceúrovňového diffu při porovnávání dvou verzí stejného html dokumentu může být smysluplné, protože například záhlaví a zápatí různých verzí téhož dokumentu se může lišit jen v několika málo slovech (datum, čas) stejně jako třeba ankety, kde se řádky mohou lišit jen počtem odpovědí na otázku.

2.2 Předzpracování

Html dokumenty obsahují semistrukturovaná data, tedy vedle běžně zobrazovaných dat (typicky textů) obsahují i data, která určují jejich strukturu. Na html dokumenty se proto můžeme dívat dvěma pohledy. Za prvé je to text, a tedy k jeho kompresi můžeme použít nějaký kompresní algoritmus založený na kompresi slov či slabik. Vzhledem k přítomnosti html značek není ale celý dokument v přirozeném jazyce, a proto tato komprese nedosáhne plné účinnosti. Dále jsou tyto metody závislé na jazyce a obsahu komprimovaného dokumentu, pro některé jazyky a pro některé dokumenty nemusí být slovní a slabikové metody vůbec vhodné.

Za druhé se na html dokument můžeme dívat jako na strukturovaný dokument. Existuje několik sofistikovaných kompresních algoritmů určených především pro xml dokumenty, které využívají právě struktury komprimovaných dokumentů a díky oddělené kompresi struktury a obsahu dokumentu dosahují lepšího kompresního poměru oproti obecným kompresním metodám. Příkladem algoritmu pro xml může být XMill [12]. Důležitým předpokladem těchto specializovaných metod je dobře formovaný (z pohledu značek dobře uzávorkovaný) dokument. Bohužel, specifikace html dokumentů [20] jsou v tomto směru dosti tolerantní, proto ani u dokumentu splňujícího specifikaci není zaručena jeho dobrá formovanost, např. `<b><i>.....</b></i>` vyhovuje specifikaci, ale není dobře formovaná část dokumentu. Dobrou formovanost vyžaduje až specifikace pro xhtml, takže na tyto dokumenty lze (pokud jsou validní) výše zmíněné kompresní metody použít. Podíváme-li se na zastoupení jednotlivých verzí html na Internetu, zjistíme, že nejčastěji se vyskytujícím typem dokumentu jsou dokumenty psané podle specifikace verze 3 a 4 (dokonce lze stále najít i dokumenty psané dle specifikace verze 2). Specializované metody využívající při kompresi struktury dokumentu nelze tedy u dokumentů stažených z Internetu obecně použít.

Důsledkem diskutovaných problémů je komprese html dokumentů pomocí obecných metod. Jak již bylo uvedeno, dosažený kompresní poměr těchto metod není pro html dokumenty optimální. Rozhodli jsme se proto, zkusit vylepšit účinnost obecných metod pomocí vhodného předzpracování komprimovaného dokumentu.

Html dokumenty jsou charakteristické použitím omezené množiny elementů a jejich atributů, které jsou navíc více či méně frekventované a jejich použití je dáno příslušnou specifikací. Z tohoto důvodu si můžeme dovolit předpokládat výskyt určitých řetězců v dokumentu.

Předzpracování spočívá v náhradě řetězců reprezentujících tyto často se vyskytující elementy a atributy kratšími symboly – nepoužitými znaky. Tato myšlenka předzpracování komprimovaných dokumentů není obecně nová. Řada podobných metod pro anglické texty obsahující pouze dolních 128 znaků je popisována v [18]. V popisovaných metodách se například výskyty slov z obvykle pevného slovníku před kompresí nahradí znaky z horních 128 znaků, nebo místo slov se používají Q-gramy (Q-tice nejčastěji se vyskytujících po sobě jdoucích znaků), jejichž výskyty jsou rovněž před kompresí nahrazeny horními 128 znaky. Pro kompresi takto předzpracovaných dokumentů jsou následně použity obvyklé kompresní algoritmy.

V rámci této práce byly navrženy a vyzkoušeny celkem tři metody předzpracování. Dvě metody používají statický slovník a liší se především způsobem kódování slov ze slovníku. Třetí metoda používá slovník adaptivní. Pro kompresi na předzpracované dokumenty byly využity kompresní algoritmy gzip a bzip2. V případě algoritmu gzip by předzpracování mělo urychlit inicializaci slovníku a díky náhradě delších řetězců kratšími také prodloužit výhled. V případě algoritmu bzip2 předzpracování může zjednodušit vstup. Porovnání všech tří navržených a zkoumaných metod bude provedeno v závěru této kapitoly.

2.3 Volba statického slovníku metod předzpracování

Slovník statických metod[1] předzpracování by měl obsahovat řetězce (názvy elementů a atributů), které se vyskytují poměrně často v co největším počtu html dokumentů. Při plnění slovníku byly zohledněny statistiky, které uvádějí nejčastěji používané elementy a atributy [6] v dokumentech na Internetu, dále byly zohledněny nejčastější konstrukce využívané při tvorbě html dokumentů a také různé verze specifikace [20].

Při tvorbě slovníku byla použita tato pravidla, která tvoří jakousi heuristikou:

1. Tentýž element se může v rámci jednoho dokumentu vyskytnout jednou s atributy a podruhé bez atributů. Aby se zvýšil počet řetězců, které budou nahrazeny stejným znakem, obsahuje slovník otevírací značky vybraných elementů ve tvaru `<` a název elementu, např. `<div`. Tento řetězec bude tedy nahrazen jak při výskytu `<div>`, tak při `<div class="...">`.

[1] budeme tak nazývat metody používající pevně daný slovník, i když metody samy o sobě statické nejsou

2. Koncová značka žádného elementu nesmí dle specifikace obsahovat žádný atribut. Proto pro každý vybraný element, který má koncovou značku, obsahuje slovník tuto značku celou, např. `</div>`.

3. Pro vybrané elementy, u kterých se výskyt nějakého atributu očekává, nebo které mají alespoň jeden atribut povinný, obsahuje slovník začátek otevírací značky (viz bod 1) doplněný zprava jednou mezerou, např. `<img␣`, `<link␣`, `<meta␣`.

4. Pokud u vybraného elementu není pravděpodobný výskyt žádného atributu, je otevírací značka tohoto elementu uvedena ve slovníku bez atributů i s pravou závorkou `>`, např. `<title>`.

5. Hodnoty atributů mohou být v html dokumentech podle specifikace do verze 4 uzavřené v jednoduchých uvozovkách (`border='1'`), uvozovkách (`border="1"`) nebo nemusejí být uzavřeny vůbec (`border=1`). Aby byl slovník použitelný pro všechny tyto případy, jsou názvy atributů uváděny ve tvaru: `border=`.

6. Názvy elementů a atributů se mohou podle specifikace až do verze 4 psát jak malými písmeny, tak velkými písmeny nebo jejich libovolnou kombinací. Podle zkoumaných dokumentů jejich autoři používají názvy psané buď samými malými písmeny, nebo samými velkými písmeny. Stejně je tomu u generovaných dokumentů. Vzhledem k tomu, že při vyhledávání řetězců ze slovníku hraje velikost písmen roli, obsahuje slovník pro každý vybraný element nebo atribut obě možnosti zapsání otevírací a koncové značky elementu nebo názvu atributu, například `<div` a `</div>` i `<DIV` a `</DIV>` nebo `border=` i `BORDER=`. Pokud se vyskytne přípustný zápis elementu `<dIv>`, není tento výskyt nahrazen příslušným znakem. Podobný zápis elementů a atributů není naštěstí u autorů příliš častý.

Při testování obou statických metod předzpracování obsahoval slovník tato slova: `</a>`, `</b>`, `</body>`, `</div>`, `</font>`, `</form>`, `</head>`, `</html>`, `</input>`, `</li>`, `</ol>`, `</p>`, `</script>`, `</span>`, `</table>`, `</td>`, `</title>`, `</ul>`, `<a`, `<b>`, `<body`, `<br`, `<div`, `<font`, `<form`, `<head>`, `<html`, `<img␣`, `<input`, `<li`, `<link␣`, `<meta␣`, `<ol`, `<p`, `<script`, `<span`, `<table`, `<td`, `<title>`, `<tr`, `<ul`, `align=`, `border=`, `class=`, `content=`, `height=`, `href=`, `id=`, `language=`, `name=`, `padding=`, `src=`, `style=`, `target=`, `type=`, `width=` a samozřejmě dle bodu 6 také jejich variantu zapsanou samými velkými písmeny. Slovník tudíž obsahoval 56 řetězců ve dvou variantách, dohromady 112 řetězců.

2.4 Metoda se statickým slovníkem 1

2.4.1 Kódování slovníku

Tato metoda předzpracování používá ke kódování slov ze slovníku, která se vyskytla v komprimovaném dokumentu, ty znaky z celkových 256, které se v tomto dokumentu nevyskytují. Slova ve slovníku mají pevné pořadí. Podle tohoto pořadí

první slovo ze slovníku, které se v dokumentu vyskytlo, je kódováno prvním znakem, který nebyl v dokumentu použit. Kódování v pořadí dalších slov se provádí obdobně.

Informace o mapování vyskytnuvších se slov ze slovníku na nevyužité znaky musí být uložena v předzpracovaném dokumentu, aby bylo možné provést zpětnou rekonstrukci předzpracovaného dokumentu. Z tohoto důvodu je na začátek předzpracovaného dokumentu vkládána hlavička, kterou tvoří bitová mapa. Prvních 256 bitů odpovídá znakům a určují, který znak byl v původním dokumentu použit (na jeho pozici je v bitové mapě 1, u nepoužitých znaků 0). Druhou část mapy tvoří bity, kde každý odpovídá právě jednomu slovu ze slovníku. Pokud se příslušné slovo v původním dokumentu vyskytlo, má odpovídající bit hodnotu 1, v opačném případě 0.

Ze způsobu kódování plyne jediné omezení na použití této metody předzpracování – počet použitých znaků v původním dokumentu a slov ze slovníku, která se vyskytla v dokumentu, musí být nejvýše 256. Jedním dechem je ale třeba dodat, že u žádného z více než 75 000 dokumentů, na kterých byla tato metoda testována, nebyl limitující počet vyšší než 256. Metoda může být použita nejen pro dokumenty v anglickém jazyce, ale obecně pro dokument v libovolném jazyce používajícím 8-bitovou znakovou sadu.

2.4.2 Fáze předzpracování

Na začátku předzpracování se ze statického slovníku vytvoří vyhledávací struktura. Používá se binarizovaná TRIE, jejíž uzel[2] je v jazyce C kódován takto:

```
struct uzel{
  unsigned char znak;
  int kod;
  uzel *alternativa, *dalsi;
}
```

Položka `znak` určuje, který znak je zastoupen tímto uzlem resp. který znak je na jeho jediné vstupní hraně. Položka `kod` nabývá v uzlech první úrovně struktury TRIE hodnot $0 \leq$ `kod` ≤ 255 podle odpovídajících znaků. Tato hodnota značí, že každý znak může být koncem jednoznakového slova. V uzlech na dalších úrovních může položka `kod` nabývat hodnoty -1, nebo hodnot > 255. Hodnota -1 říká, že v tomto uzlu nekončí žádné z námi sledovaných slov. Hodnoty > 255 určují, které ze sledovaných slov končí v tomto uzlu.

Při použitém slovníku by první úroveň binarizované TRIE obsahovala 26 uzlů, tzn. že slova z použitého slovníku začínají na 26 různých znaků (pozor, nejde o 26 písmen anglické abecedy). Je tedy mnohem pravděpodobnější, že další přečtený znak není prvním znakem nějakého slova ze slovníku, a první úroveň binarizované TRIE by se musela přes ukazatele `alternativa` projít velice často celá. Z tohoto důvodu tvoří první úroveň vyhledávací struktury pole `uzel koren[256]`, které umožní přímý přístup k požadovanému uzlu.

Po inicializaci (vytvoření vyhledávací struktury ze slovníku) probíhá předzpracování ve třech fázích:

[2] ve skutečnosti se jedná o uzel struktury a jeho jedinou vstupní hranu

1. V první fázi se počítají statistiky předzpracovávaného souboru. Zjišťuje se, které znaky a také která slova ze slovníku (pomocí vyhledávací struktury) se v souboru vyskytují. Vzhledem k tomu, že není vyloučena možnost, aby jedno slovo ze slovníku bylo předponou slova jiného, používá se při hledání řetězců maximální možná shoda. Jinými slovy, pokud bude ve slovníku slovo `<h` i slovo `<html`, pak po přečtení znaků `<` a `h` není hlášen výskyt slova `<h`, ale čtou se další znaky, aby se zjistilo, zda nenásledují znaky `t`, `m`, `l`. Pokud ne, ohlásí se výskyt slova `<h` a pokračuje se v hledání v souboru od první pozice za tímto slovem a od první úrovně ve vyhledávací struktuře. Dříve než bude algoritmus demonstrován pomocí pseudokódu, nadefinujme si jednu pomocnou funkci, kterou algoritmus používá. Funkce pro daný ukazatel na uzel v TRIE a znak na vstupu vrací ukazatel na uzel v TRIE, na který se má přejít z daného uzlu na základě daného znaku:

   ```
   KROK(p_uzel, znak):

   IF p_uzel = NULL THEN
       RETURN &kořen[znak];
   ENDIF

   IF p_uzel->dalsi = NULL THEN
       RETURN NULL;
   ENDIF

   p_uzel := uzel->dalsi;
   WHILE p_uzel <> NULL AND p_uzel->znak < znak DO
       p_uzel := p_uzel->alternativa;
   DONE

   IF p_uzel = NULL OR p_uzel->znak <> znak THEN
       RETURN NULL;
   ELSE
       RETURN p_uzel;
   ENDIF
   ```

   A nyní již samotný algoritmus zapsaný pseudokódem. Vstupem algoritmu je ukazatel `p_uzel` na aktuální uzel v TRIE a ukazatel `p_znak` na aktuální znak ze vstupu. V popisu algoritmu proměnná `p` představuje ukazatel do vstupního souboru na znak, který je posledním znakem naposledy nalezeného slova. Dále proměnná `ppk` uchovává hodnotu posledního kódu koncového stavu při hledání slov ze slovníku. Algoritmus provádí opakovaně tuto činnost:

   ```
   ...
   p_uzel := KROK(p_uzel, *p_znak);
   pokud se poslední nalezené slovo už nedá prodloužit
   IF p_uzel = NULL THEN
       ...
   ```

```
    příslušná reakce na nalezené slovo – zaznamenání jeho nalezení
    v první fázi, tisk přiřazeného znaku ve druhé fázi
    ...
    p_znak := p + 1;
    p_uzel := &kořen[*p_znak];
ENDIF
pokud je dosažen koncový stav
IF p_uzel->kod > 0 THEN
    ppk := p_uzel->kod;
    p := p_znak;
ENDIF
posun ve vstupním souboru o jeden znak dál
p_znak := p_znak + 1;
...
```

2. Ve druhé fázi se mapují nalezená slova ze slovníku na nepoužité znaky. Postup je popsán ve 2.4.1. Vytvořená bitová mapa se zapíše na začátek předzpracovaného souboru.

3. Za zapsanou hlavičku (bitovou mapu) z fáze 2 se zapíše obsah souboru. V podstatě jde o původní soubor, ve kterém jsou výskyty slov ze slovníku nahrazeny příslušnými znaky. Vyhledávání slov se samozřejmě provádí podle shodného algoritmu jako ve fázi 1.

Z popisu fází předzpracování je patrné, že předzpracování vyžaduje dva průchody předzpracovávaným dokumentem. Předpokládejme, že délka dokumentu je m a délka nejdelšího hledaného slova je n. Pak v nejhorším případě vyhledání slov v dokumentu vyžaduje přečíst $O(m * n)$ znaků vstupního dokumentu[3] – hledá se vždy nejdelší vzorek tvaru $a^{n-1}b$ v dokumentu $a^{m-1}b$. Pro dokumenty v přirozeném jazyce bude průměrný počet znaků, které je třeba přečíst, $m * konst$, tedy $O(m)$. Použili jsme stejné odhady jako pro triviální algoritmus hledání jednoho vzorku v dokumentu, který se stejně jako náš algoritmus vyhledávání vrací za poslední pozici shody dokumentu se vzorkem. Odhady jsou analogické odhadům v [10].

2.4.3 Zpětná rekonstrukce

Na rozdíl od předzpracování odpadá při zpětné rekonstrukci původního souboru fáze počítání statistik. Zpětná rekonstrukce má tedy jen dvě fáze:

1. Přečte se hlavička (bitová mapa) z předzpracovaného dokumentu a na jejím základě se provede rekonstrukce mapování slov ze slovníku na nepoužité znaky. První použité slovo ze slovníku bylo v předzpracovaném dokumentu nahrazeno prvním nepoužitým znakem, druhé slovo druhým znakem atd.

2. Čte se zbytek předzpracovaného dokumentu po znacích. Pokud přečtený znak nezastupuje žádné slovo ze slovníku, tento znak se pouze zreprodukuje na výstup. V opačném případě se na výstup pošle příslušné nahrazené slovo.

[3] v naší implementaci probíhá toto čtení v paměti

Zpětná rekonstrukce si vystačí pouze s jedním průchodem předzpracovaným dokumentem, při kterém čte každý znak právě jednou.

Protože při zpětné rekonstrukci je každý znak kódující nějaký řetězec v původním dokumentu nahrazen právě tímto řetězcem, získáme po zpětné rekonstrukci dokument totožný s původním. Pokud dále předpokládáme, že kompresní metoda, která zpracovává takto upravený vstup, je bezeztrátová, můžeme prohlásit, že celá zde popsaná metoda komprese s předzpracovaným vstupem je rovněž bezeztrátová.

2.4.4 Výsledky

Metoda byla testována na více než 75 000 dokumentech, které byly stažené z Internetu pomocí fulltextového systému EGOTHOR z různých akademických domén. Každý soubor byl zkomprimován samotnými metodami gzip a bzip2 a také těmito metodami se vstupy upravenými metodou 1. U metody gzip bylo ponecháno implicitní nastavení, pro metodu bzip2 byla nastavena úroveň 9 (nejúčinnější), i když u souborů menších než 100 kB by velikost bloku na účinnost komprese neměla mít vliv. Vždy byla změřena velikost zkomprimovaného souboru. Metody se chovají na různě velkých souborech odlišně, proto jsou výsledky v tabulkách rozděleny podle velikosti komprimovaných souborů. Pro stručnější zápis jsou použity tyto zkratky:

mgz1 předzpracování souboru metodou 1 a následná komprese pomocí metody gzip

mbz1 předzpracování souboru metodou 1 a následná komprese pomocí metody bzip2

Soubory		Kolikrát byla nejúčinnější metoda			
vel. v kB	počet	gzip	bzip2	mgz1	mbz1
≤ 10	33 382	6 102	443	27 061	10
10–20	25 888	115	1 291	23 910	619
20–30	8 293	8	948	6 174	1 205
30–40	3 759	1	503	2 023	1 253
40–50	1 939	1	273	727	952
50–60	977	0	92	85	803
60–70	359	0	71	32	258
70–80	229	0	32	5	193
80–90	148	0	31	5	112
90–100	130	0	30	1	100
> 100	196	0	44	0	153

Tabulka 2.1: Počty nejlépe zkomprimovaných souborů – porovnání metody 1 upravující vstup pro gzip nebo bzip2 a použití samotných aplikací gzip a bzip2

Tabulka 2.1 uvádí, v kolika případech byla která z testovaných metod nejlepší. Pokud nejlepší zkomprimované velikosti na stejném souboru dosáhlo více metod, započítal se tento soubor všem inkriminovaným metodám. Z tabulky mimo jiné

vyplývá, že nejčastěji měly soubory velikost do 20 kB. Na těchto souborech dopadla nejlépe metoda mgz1. Pro soubory od velikosti 50 kB se dostává na první místo metoda mbz1, zanedbatelná na větších souborech není ani metoda bzip2. Naopak metoda gzip se ukázala na souborech větších než 10 kB jako nevýhodná, alespoň co do počtu nejlepších výsledků.

Soubory v kB	Celková velikost zkomprimovaných souborů v B				
	bez komp.	gzip	bzip2	mgz1	mbz1
≤ 10	195 441 248	62 201 974	66 889 194	61 406 642	67 139 789
10–20	372 419 892	93 891 488	96 931 143	92 414 381	96 837 481
20–30	205 296 289	43 353 599	43 558 168	42 678 055	43 439 597
30–40	131 980 917	24 072 347	23 666 224	23 673 127	23 597 876
40–50	88 015 810	14 836 842	14 330 609	14 552 546	14 289 687
50–60	54 676 625	9 249 960	8 635 500	8 986 686	8 596 907
60–70	23 674 719	3 863 760	3 508 452	3 788 286	3 496 536
70–80	17 466 596	2 473 924	2 230 697	2 407 189	2 222 098
80–90	12 796 748	1 741 864	1 540 899	1 699 385	1 536 227
90–100	12 634 495	1 683 799	1 463 146	1 645 555	1 458 605
> 100	22 467 477	2 740 619	2 282 637	2 667 778	2 276 837

Tabulka 2.2: Celková velikost zkomprimovaných souborů – porovnání metody 1 upravující vstup pro gzip nebo bzip2 a použití samotných aplikací gzip a bzip2

K posouzení použitelnosti metody je třeba znát nejen to, kolikrát daná metoda komprimovala nejlépe a kolik na to potřebovala času, ale i to, kolik se tak podařilo ušetřit. Tabulka 2.2 uvádí, jaký je součet velikostí všech danou metodou komprimovaných souborů dané velikosti. Navržená metoda předzpracování přináší ve spojení s kompresí gzip (pro menší soubory) nebo bzip2 (pro větší soubory) prostorovou úsporu. Úspora není sice až tak výrazná, ale pokud hraje roli velikost ukládaných dokumentů a nevadí mírné zpomalení při kompresi, může být použití tohoto předzpracování cesta ke zmenšení nároků na diskový prostor.

2.4.5 Case-insensitive varianta

Podle html specifikace až do verze 4 nehraje velikost písmen v názvech html značek a jejich atributů roli. Nabízí se možnost umístit do slovníku slova pouze v jedné variantě a při hledání slov v dokumentu nerozlišovat velikost písmen, tzn. `<div`, `<DIV` i `<dIv` budou nalezena a kódována jako stejné slovo. Při zpětné rekonstrukci jsou všechna tato slova rekonstruována jako slovo jedno (ze slovníku). Tím se ovšem ztratí informace o velikosti znaků a metoda přijde o přívlastek bezeztrátová. Pro zobrazení html dokumentů nemá tato ztráta význam. Pokud by ale byla ve velikosti znaků ukryta třeba nějaká tajná informace, nebyla by tato ztráta akceptovatelná.

Původní varianta předzpracování sice používá slovník, kde slova jsou samé názvy html elementů a jejich atributů, ale teoreticky by mohlo být v jejím slovníku i jiné slovo, které se například vyskytuje poměrně často v obsahu html dokumentů. Tato možnost je pro case-insensitivní variantu vyloučena, protože by došlo ke změně zobrazovaného dokumentu.

Bohužel, ani slovník tak, jak jsme ho použili, nemusí být bez problémů. Nebezpečí číhá u dokumentů, ve kterých je obsahem ukázka html kódu. Znaky < a > ohraničující html značky by měly být v obsahu dokumentu zapsány pomocí entit `<` a `>`, ale atributy se mohou vyskytnout ve stejném tvaru jako ve struktuře dokumentu. Obsah takových dokumentů může být case-insensitivní variantou předzpracování pozměněn a to může být nepřípustné.

Pokud jsou ve zpracovávaném dokumentu všechny html značky a atributy, které se nahrazují pomocí slovníku, zapsané samými malými či velkými znaky, pak se výstupy obou variant liší pouze délkou bitové mapy v hlavičce (u senzitivní varianty je dvojnásobný slovník). Významu varianta nabývá u dokumentů, ve kterých se vyskytují smíšené formy zápisu.

Při testech byl použit stejný slovník jako u varianty rozlišující velikost znaků a testy byly provedeny nad stejnými soubory. Značení:

mgz1i předzpracování souboru metodou 1 bez rozlišení velikosti znaků při kódování vybraných řetězců a následná komprese pomocí metody gzip

mbz1i předzpracování souboru metodou 1 bez rozlišení velikosti znaků při kódování vybraných řetězců a následná komprese pomocí metody bzip2

Soubory		Kolikrát byla nejúčinnější metoda			
vel. v kB	počet	mgz1	mbz1	mgz1i	mbz1i
≤ 10	33 382	11 338	127	32 196	242
10–20	25 888	7 994	565	23 698	1 337
20–30	8 293	1 675	657	6 083	1 553
30–40	3 759	339	475	1 925	1 315
40–50	1 939	65	301	725	941
50–60	977	15	343	82	591
60–70	359	3	93	31	244
70–80	229	1	49	4	180
80–90	148	0	28	5	119
90–100	130	0	27	1	103
< 100	196	0	46	0	155

Tabulka 2.3: Počty nejlépe zkomprimovaných souborů – porovnání metody 1 s a bez rozlišení velikosti písmen při kódování vybraných řetězců upravující vstup pro gzip nebo bzip2

Dle předpokladu metoda bez rozlišení velikosti znaků kódovaných řetězců dopadla lépe než metoda rozlišující velikost znaků. Pro kolik testovaných souborů byla tato a původní varianta s rozlišením velikosti znaků nejúčinnější ze všech čtyř sledovaných metod uvádí tabulka 2.3. Tabulka neporovnává přímo účinnost předzpracování s a bez rozlišení velikosti znaků kódovaných řetězců, ale říká, pro kolik testovaných souborů byla která metoda tou nejúčinnější. Pokud nejmenší velikosti zkomprimovaného souboru dosáhlo více metod současně, je tento soubor započten všem těmto metodám. Například pro soubory o velikosti do 10 kB nastala shoda mezi metodou mgz1 a mgz1i ve více než 10 600 případech, tzn. metoda mgz1 byla účinnější než mgzi1 pouze asi v 700 případech. Velikost souboru předzpracovaného variantou bez rozlišení velikosti znaků kódovaných řetězců by

neměla být ale nikdy větší než velikost souboru předzpracovaného variantou s rozlišením velikosti znaků stejné metody. Dokonce by měla být vždy o alespoň 7 bajtů menší (zapisuje se kratší bitová mapa do hlavičky předzpracovaného souboru). Můžeme z toho usuzovat, že po kompresi předzpracovaných souborů by měla platit stejná nerovnost, protože varianta bez rozlišení velikosti znaků kódovaných řetězců kóduje všechny řetězce jako varianta s rozlišením. Navíc může kódovat i některé další řetězce. Protože ale ne vždy tomu tak bylo, věnovali jsme tomuto problému více času.

Znovu jsme ověřili všechny soubory, aby se ukázalo, že velikost souboru předzpracovaného variantou bez rozlišení velikosti znaků kódovaných řetězců je vždy o alespoň 7 bajtů menší než velikost souboru předzpracovaného variantou s rozlišením velikosti znaků. Nebyla nalezena ani žádná chyba ve fungování algoritmu předzpracování. Máme proto důvod předpokládat, že předzpracování funguje správně. Vysvětlení je tedy nutné hledat v kompresních algoritmech, které byly použity pro následnou kompresi.

Při použití komprese gzip byl průměrný rozdíl ve velikosti zkomprimovaných souborů, které se vymykaly předpokladu, zhruba od 1 bajtu pro nejmenší soubory do 10 bajtů pro větší soubory. Domníváme se, že důvodem může být rozdílná délka bitové mapy uložená na začátku každého předzpracovaného dokumentu, která způsobí rozdílnou inicializaci kompresního algoritmu. Dále mohlo být variantou bez rozlišení velikosti znaků kódovaných řetězců nahrazeno nepoužitými znaky více řetězců v původním dokumentu. Části těchto řetězců mohly být jako podřetězce jiných řetězců komprimovány efektivněji než znaky, kterými byly nahrazeny.

Při následné kompresi metodou bzip2 byl v případech vymykajících se předpokladu průměrný rozdíl od 6 bajtů pro malé soubory do 12 bajtů pro větší soubory. Na rozdíl od metody gzip bylo ale takových případů více. Důvod je opět třeba hledat přímo v algoritmu metody bzip2. Domníváme se, že variantou bez rozlišení velikosti znaků kódovaných řetězců byl při nahrazování většího počtu nalezených řetězců snížen počet běžně se vyskytujících znaků (písmen), které se vyskytují v celém dokumentu. Po předzpracování variantou s rozlišením velikosti znaků obsahoval dokument více běžně se vyskytujících znaků a méně znaků, které se v původním dokumentu nevyskytly, než po předzpracování variantou bez rozlišení velikosti znaků kódovaných řetězců. Tato druhá varianta totiž mohla nahradit více řetězců, které obsahují samozřejmě běžné znaky, a tím jejich celkový počet snížit. Z popisu algoritmu bzip2 ve 2.1.2 plyne, že využívá efektivní komprese dlouhých sekvencí stejných znaků, které vzniknou po setřídění posledních znaků všech rotací bloku. Máme proto důvod předpokládat, že delší sekvence běžných znaků a kratší sekvence doplněných znaků byla ve vymykajících se případech komprimována efektivněji než o něco kratší sekvence běžných znaků a delší sekvence doplněných znaků. Navíc bitové mapy vkládané na začátek předzpracovaného dokumentu jsou různé a mohou vnést nové znaky. Tím ovlivní i samotné třídění bloku.

V tabulce 2.4 je uvedena celková velikost zkomprimovaných souborů při použití předzpracování vstupu pro gzip a bzip2 metodou nerozlišující velikost znaků kódovaných řetězců.

Soubory v kB	Celková velikost souborů v B		
	bez komp.	mgz1i	mbz1i
≤ 10	195 441 248	61 146 153	66 686 741
10–20	372 419 892	92 113 626	96 371 478
20–30	205 296 289	42 570 638	43 280 498
30–40	131 980 917	23 610 066	23 502 399
40–50	88 015 810	14 512 087	14 231 711
50–60	54 676 625	8 973 306	8 578 679
60–70	23 674 719	3 780 463	3 485 988
70–80	17 466 596	2 400 383	2 213 438
80–90	12 796 748	1 695 379	1 531 341
90–100	12 634 495	1 642 152	1 454 363
< 100	22 467 477	2 663 256	2 271 749

Tabulka 2.4: Celková velikost zkomprimovaných souborů – metoda 1 bez rozlišení velikosti písmen při kódování vybraných řetězců upravující vstup pro gzip nebo bzip2

Z obou tabulek vyplývá, že pokud se při předzpracování přestane brát zřetel na velikost znaků kódovaných řetězců, může tato úprava přinést další prostorovou úsporu. Ovšem za cenu ztráty části informace.

2.5 Metoda se statickým slovníkem 2

Předchozí metoda využívá ke kódování všech 256 znaků. Pokud by kompresní algoritmus využíval fázování, znamenalo by snížení univerza (počtu znaků, kterými se kóduje) i zlepšení kompresního poměru. Tato metoda využívá oproti metodě 1 pouze dolních 128 znaků. To také znamená, že metodu lze použít pouze na dokumenty, které neobsahují žádný z horních 128 znaků, např. dokumenty psané v angličtině.

Zmenšení abecedy bylo zamýšleno hlavně pro kompresní metody, které dokáží menší než 8-bitové abecedy využít. V úvahu připadají kompresní metody odvozené od metody LZ78.

2.5.1 Kódování slovníku

Tato metoda se liší od předchozí metody prakticky pouze drobnou změnou v kódování slovníku. K zakódování celého dokumentu se snaží využít pouze dolních 128 znaků. Slova ze slovníku jsou v dokumentu kódována nevyužitými znaky z dolních 128 znaků. Stejně jako u první metody jsou slova ve slovníku uspořádána v pevném pořadí. Dle tohoto pořadí první slovo, které se vyskytlo ve zpracovávaném dokumentu, je kódováno prvním nevyužitým znakem z dolních 128, druhé vyskytnuvší se slovo druhým nepoužitým znakem atd.

Informace o mapování slov ze slovníku na nepoužité znaky se rovněž uchovávají v hlavičce předzpracovaného dokumentu ve formě bitové mapy. Tentokrát si ale část určená pro znaky vystačí pouze se 128 bity, kde 1 značí, že příslušný znak se v dokumentu vyskytl, 0 značí pravý opak, takže tento znak mohl být využit

ke kódování slov ze slovníku nalezených v dokumentu. Druhá část hlavičky jsou bity odpovídající právě jednomu slovu ze slovníku. Bit s hodnotou 1 znamená, že příslušné slovo bylo v dokumentu nalezeno.

Z popsaného způsobu kódování plyne jedno velice nepříjemné omezení – součet počtu slov slovníku, která se v dokumentu vyskytla, a počtu použitých znaků z dolních 128 nesmí být vyšší než 128, aby bylo možné provést kódování. Toto omezení se často ukázalo jako poměrně silné.

2.5.2 Fáze předzpracování

Při předzpracování se na začátku vytvoří ze slovníku stejná vyhledávací struktura jako u metody 1, viz 2.4.2. Také následující fáze jsou totožné s fázemi předzpracování metody 1:

1. V první fázi se počítají statistiky, tzn. zjišťují se znaky, které jsou v dokumentu použity, a také se ve vstupním dokumentu hledají slova ze slovníku. Opět se hledá maximální možné slovo.

2. Ve druhé fázi se mapují nalezená slova ze slovníku na znaky v dokumentu nepoužité. Vytvořená bitová mapa se zapíše do hlavičky předzpracovaného dokumentu.

3. Znovu se čte a kóduje vstupní dokument. Výstup se zapisuje za hlavičku. Pokud není znak součástí nalezeného slova ze slovníku, prostě se zreprodukuje na výstup. Nalezená slova jsou nahrazena příslušnými znaky, na které byla ve fázi 2 namapována.

Stejně jako u metody 1 vyžaduje předzpracování dokumentu dva průchody vstupním dokumentem. Každý z nich má stejnou složitost jako průchod dokumentem u metody 1, viz 2.4.2.

2.5.3 Zpětná rekonstrukce

Zpětná rekonstrukce je opět totožná se zpětnou rekonstrukcí metody 1, viz 2.4.3. Pouze stručně:

1. Přečte se hlavička předzpracovaného dokumentu, na jejímž základě se rekonstruuje mapování slov ze slovníku na nepoužité znaky.

2. Čte se předzpracovaný dokument. Pokud přečtený znak zastupuje slovo ze slovníku, nahradí se příslušným slovem, jinak znak zůstává.

Stejně jako v prvním případě stačí pro zpětnou rekonstrukci jeden průchod předzpracovaným dokumentem. Rovněž se jedná o bezeztrátovou metodu, zrekonstruovaný dokument je totožný s původním.

2.5.4 Výsledky

Testy byly provedeny nad stejnou sadou souborů jako pro metodu 1 a stejným způsobem. Značení:

mgz2 předzpracování souboru metodou 2, následná komprese pomocí metody gzip

mbz2 předzpracování souboru metodou 2, následná komprese pomocí metody bzip2

Soubory		Kolikrát byla nejúčinnější metoda			
vel. v kB	počet	gzip	bzip2	mgz2	mbz2
≤ 10	12 452	3 207	217	9 141	8
10–20	3 999	22	412	3 377	207
20–30	571	0	129	252	193
30–40	191	0	30	75	89
40–50	73	1	16	19	37
50–60	24	0	6	2	16
60–70	20	0	6	0	14
70–80	4	0	3	0	1
80–90	5	0	0	0	5
90–100	8	0	5	0	3
< 100	15	0	7	0	8

Tabulka 2.5: Počty nejlépe zkomprimovaných souborů – porovnání metody 2 upravující vstup pro gzip nebo bzip2 a použití samotných aplikací gzip a bzip2

Tabulka 2.5 obsahuje pouze výsledky souborů, na které bylo možné vzhledem ke kódování metodu použít. Hlavní omezení metody se nakonec ukázalo být jako poměrně silné, protože u více než 75 000 souborů bylo možné metodu použít jen pro necelou jednu čtvrtinu z nich. Přesto je patrné, že komprese, jejíž vstup byl předzpracován, byla účinnější než komprese bez předzpracování.

Soubory	Celková velikost zkomprimovaných souborů v B				
v kB	bez komp.	gzip	bzip2	mgz2	mbz2
≤ 10	62 739 109	19 874 372	21 431 639	19 669 956	21 485 656
10–20	54 419 163	13 566 330	13 811 893	13 388 914	13 791 498
20–30	14 247 976	2 653 836	2 598 176	2 615 983	2 590 116
30–40	6 853 762	1 002 942	956 219	982 381	951 238
40–50	3 299 090	479 361	446 038	469 373	444 387
50–60	1 350 512	177 344	157 809	173 291	157 363
60–70	1 334 194	199 773	176 472	196 492	176 015
70–80	303 095	49 895	41 689	48 898	41 710
80–90	433 105	30 931	25 623	29 338	25 289
90–100	773 277	112 170	93 901	110 904	93 800
< 100	1 726 686	208 429	168 872	203 106	168 845

Tabulka 2.6: Celková velikost zkomprimovaných souborů – porovnání metody 2 upravující vstup pro gzip nebo bzip2 a použití samotných aplikací gzip a bzip2

Tabulka 2.6 obsahuje součty velikostí jednotlivých souborů zkomprimovaných danou metodou. Předzpracování opět přineslo drobné prostorové zlepšení oproti původním kompresím. V případě aplikace gzip je toto zlepšení výraznější než pro bzip2.

2.5.5 Case-insensitive varianta

Byla vyzkoušena také varianta, která nerozlišuje velikost písmen při vyhledávání slov ze slovníku. Oproti původní variantě je slovník poloviční, proto je tato varianta použitelná pro více dokumentů (při testech o něco málo více než pro jednu čtvrtinu testovaných souborů), neboť bude třeba méně volných znaků v původním dokumentu pro kódování slov ze slovníku. Na druhou stranu, jak ukázaly testy, vyskytlo se několik souborů, na kterých byla varianta s rozlišením velikosti znaků použitelná, ale varianta bez rozlišení nikoli. Pravděpodobně se jedná o dokumenty, ve kterých byly html značky a atributy zapsané jinak než samými malými či velkými písmeny. Varianta bez rozlišení tím pádem nalezla více řetězců, které bylo třeba zakódovat a nestačil jí počet volných znaků pro kódování. Značení:

mgz2i předzpracování souboru metodou 2 bez rozlišení velikosti znaků při kódování vybraných řetězců a následná komprese pomocí metody gzip

mbz2i předzpracování souboru metodou 2 bez rozlišení velikosti znaků při kódování vybraných řetězců a následná komprese pomocí metody bzip2

Soubory		Kolikrát byla nejúčinnější metoda			
vel. v kB	počet	mgz2	mbz2	mgz2i	mbz2i
≤ 10	12 448	68	59	12 173	165
10–20	3 997	16	163	3 431	416
20–30	571	0	108	258	217
30–40	191	0	41	75	80
40–50	73	0	13	22	39
50–60	24	0	3	2	20
60–70	20	0	6	0	14
70–80	4	0	0	0	4
80–90	5	0	2	0	3
90–100	8	0	3	0	5
< 100	15	0	2	0	13

Tabulka 2.7: Počty nejlépe zkomprimovaných souborů – porovnání metody 2 s a bez rozlišení velikosti písmen při kódování vybraných řetězců upravující vstup pro gzip nebo bzip2

Tabulka 2.7 porovnává varianty metody 2 rozlišující a nerozlišující velikost znaků kódovaných řetězců na souborech, na kterých jsou obě varianty použitelné. Ze zjištěných údajů plyne, že varianta bez rozlišení velikosti znaků kódovaných řetězců může kompresní poměr ještě dále vylepšit. Varianta bez rozlišení velikosti znaků ale ztrácí informaci právě o velikosti znaků v původním dokumentu.

Také u této metody byla v některých případech komprese dokumentu předzpracovaného variantou s rozlišením velikosti znaků účinnější než komprese dokumentu předzpracovaného variantou metody bez rozlišení velikosti znaků kódovaných řetězců. Procento těchto případů bylo obdobné jako u metody 1. Také průměrný rozdíl velikosti souborů v těchto případech byl obdobný. Nabízí se stejné vysvětlení jako při diskuzi nad výsledky metody 1.

Soubory v kB	Celková velikost souborů v B		
	bez komp.	mgz2i	mbz2i
≤ 10	69 370 116	21 715 768	23 668 090
10–20	62 698 775	15 475 316	15 952 980
20–30	17 480 249	3 208 098	3 190 540
30–40	7 810 789	1 117 112	1 077 352
40–50	3 611 197	519 049	491 327
50–60	1 791 979	222 541	204 028
60–70	1 402 078	210 133	187 182
70–80	453 327	68 558	58 886
80–90	602 749	59 622	53 527
90–100	773 277	110 851	93 730
< 100	1 726 686	202 985	168 664

Tabulka 2.8: Celková velikost zkomprimovaných souborů – metoda 2 bez rozlišení velikosti písmen při kódování vybraných řetězců upravující vstup pro gzip nebo bzip2. Zkoumaly se pouze soubory, na kterých je použitelná varianta bez rozlišení velikosti znaků

Tabulka 2.8 obsahuje součty velikostí zkomprimovaných dokumentů po předzpracování metodou 2 – variantou bez rozlišení velikosti znaků kódovaných řetězců, na které byla použitelná.

2.6 Metoda s adaptivním slovníkem

Předchozí dvě metody předzpracování používaly pevný slovník, jehož prvky ve vstupním dokumentu nahrazovaly kratšími kódy (znaky). Pevný slovník musí pokrývat vlastnosti co nejširšího spektra dokumentů. To klade velký důraz na výběr slov, která budou do slovníku vložena. Výběr proto musí být omezen na slova, která jsou v dokumentech nějakým způsobem predikovatelná (html značky elementů a jejich atributy). V některých dokumentech se může často opakovat poměrně dlouhý řetězec obsahující například hodnotu URL. Takové řetězce nelze zahrnout do statického slovníku, protože v době jeho návrhu nebývají obvykle známé.

Navržená metoda má tedy v rámci dokumentu najít frekventované řetězce dostatečné délky a tyto kódovat nepoužitými znaky. Z tohoto důvodu jsme tuto metodu nazvali metoda frekventovaných slov. Metoda pro každý dokument vytváří vlastní slovník, slovník je tedy adaptivní.

2.6.1 Fáze předzpracování

Předzpracování se skládá ze tří fází. V první fázi se vytvoří slovník pro zpracovávaný dokument, zároveň s tvorbou slovníku se počítají četnosti výskytu nalezených slov. Slovo je v této metodě chápáno jako posloupnost znaků ukončená separátorem, který se již do slova nepočítá. Jako separátor slouží znaky (číslované podle ASCII): 0–47, 58, 59, 61, 63, 91–93 a 123–125.

Dále se zjistí, které znaky byly v dokumentu použity. Po provedení první fáze je k dispozici seznam slov setříděný podle jejich četnosti výskytu v dokumentu,

slova o stejné četnosti jsou dále tříděna dle jejich délky. Do seznamu slov se dostanou pouze slova, která dosahují minimální požadované délky. Jako seznam slov slouží obousměrný spojový seznam, kde každý jeho uzel je v jazyce C kódován jako:

```
struct st_slovo{
  unsigned char *retezec;
  slovo *pred, *nasl;
  int frek;
  int delka;
  uzel *list;
}
```

Položka `retezec` je řetězec znaků reprezentující dané slovo, `pred` resp. `nasl` je ukazatel na předcházející resp. následující uzel seznamu, `frek` počet výskytů daného slova v dokumentu, `delka` určuje délku daného slova a `list` je ukazatel na uzel v TRIE, ve kterém končí dané slovo.

Z nalezených slov se zároveň staví vyhledávací struktura – binarizovaná TRIE. Každý jeho uzel má následující strukturu:

```
struct uzel{
  unsigned char znak;
  int kod;
  uzel *alternativa, *dalsi, *otec;
  st_slovo *slovo;
}
```

Položka `znak` je znak, který daný uzel zastupuje, `kod` určuje, zda v tomto uzlu končí nějaké slovo ($0 \leq$ `kod` ≤ 255), nebo nikoli (`kod` = -1). Pokud v tomto uzlu nějaké slovo končí, pak položka `slovo` je ukazatelem na příslušné slovo v seznamu slov. Položky `alternativa`, `dalsi`, `otec` představují po řadě ukazatel na sourozence, syna, otce daného uzlu v TRIE.

Ve druhé fázi se zjistí, kolik znaků nebylo v dokumentu použito. Nechť jich je n. V seznamu slov se ponechá pouze prvních n nejfrekventovanějších slov, přičemž lze stanovit minimální požadovanou frekvenci. Pokud by takových slov bylo méně, vezme se tento menší počet. Slova se dále mapují na nepoužité znaky způsobem první slovo na první nepoužitý znak atd. Výsledné mapování je třeba uložit do výsledného souboru, proto se na jeho začátek zapíše bitová mapa velikosti 256, kde hodnota 1 na i-té pozici znamená, že znak s číslem i nebyl v původním dokumentu použit a zároveň že tento znak zastupuje ve výsledném dokumentu nějaké slovo (pozor, význam hodnot v bitové mapě je odlišný od jejich významu u předchozích dvou metod). Mapování je dále nutné promítnou do vyhledávací struktury vytvořené v první fázi. Hodnota položky `kod` uzlu TRIE, ve kterém dané slovo končí, je nastavena na číslo volného znaku, který bude zastupovat příslušné slovo.

Protože tato metoda vytváří slovník pro každý dokument zvlášť, musí být slovník součástí výsledného dokumentu. Za zapsanou bitovou mapu se ukládají slova ze slovníku ve stejném pořadí, ve kterém byla namapována na volné znaky. Slova se oddělují jedním ze separátorů použitých v první fázi.

Ve třetí fázi se znovu čte předzpracovávaný dokument, přičemž slova ze slovníku se nahrazují příslušnými znaky. Výsledek je ukládán za bitovou mapu a slovník do výstupního souboru. Vzhledem k tomu, že jedno slovo ze slovníku může být předponou jiného slova ze slovníku, hledá se maximální možná shoda, tzn. s oznámením nalezeného slova se čeká tak dlouho, dokud není pochyb, že nemůže být slovo prodlouženo na nějaké jiné. Hledání potom začíná na první pozici ve vstupním dokumentu za právě nalezeným slovem.

Z hlediska rychlosti hraje roli počet průchodů předzpracovávaným dokumentem. Předzpracování vyžaduje jeden průchod vstupním dokumentem v první fázi a jeden průchod ve třetí fázi. Předpokládejme, že délka vstupního dokumentu je m. V první fázi se vytváří slovník, každý znak se čte právě jednou, složitost první fáze měřená počtem čtených znaků vstupního dokumentu je proto $\theta(m)$. Ve třetí fázi probíhá vyhledávání slov, která tvoří slovník. Pro třetí fázi platí stejný odhad jako pro metodu 1 a 2 se statickým slovníkem, viz 2.4.2, protože vyhledávání slov ze slovníku probíhá podle stejného algoritmu. Pro texty v přirozeném jazyce je složitost měřená počtem čtených znaků vstupního dokumentu v průměrném případě $m * konst$. Celková složitost předzpracování je tedy i pro tuto metodu $O(m)$.

2.6.2 Zpětná rekonstrukce

Při transformaci předzpracovaného dokumentu v dokument původní stačí jeden průchod předzpracovaným dokumentem. Předzpracování lze popsat ve třech krocích:

1. Ze vstupního dokumentu se přečte bitová mapa o délce 256 bitů určující, kolika a kterými znaky byla nahrazena slova ze slovníku. Pokud na i-té pozici bitové mapy je 1, pak znak s číslem i nebyl v původním dokumentu použit a zastupuje nějaké slovo ze slovníku.

2. Pokud bitová mapa přečtená v kroku 1 obsahuje n bitů s hodnotou jedna, pak se čte ze vstupního dokumentu n slov (řetězec znaků do prvního separátoru). Dále se rekonstruuje mapování slov ze slovníku na volné znaky. První znak, který měl v bitové mapě hodnotu 1, zastupuje první slovo přečtené v tomto kroku, druhý znak druhé slovo atd.

3. Čte se zbytek předzpracovaného dokumentu a vytváří se dokument původní. Znaky, které zastupují nějaké slovo jsou tímto slovem nahrazeny.

Tato metoda předzpracování je opět bezeztrátová úprava, rekonstruovaný dokument je totožný s dokumentem původním.

2.6.3 Výsledky

Metoda předzpracování s adaptivním slovníkem byla testována na stejné sadě html souborů jako obě metody se statickým slovníkem. Minimální délka slov byla stanovena na 3, minimální četnost výskytu slova byla 2.

Na testovaných souborech měla slova ve slovníku nejčastější délku od 3 znaků do 22 znaků. Vůbec nejčastější byla slova délky 6 znaků, která se ve slovníku vyskytla 210 636krát. Nejdelší slovo, které se během testování vyskytlo, mělo délku 543 znaky. Značení:

fsg předzpracování metodou frekventovaných slov, komprese pomocí gzip

fsb předzpracování metodou frekventovaných slov, komprese pomocí bzip2

Soubory		Kolikrát byla nejúčinnější metoda			
vel. v kB	počet	gzip	bzip2	fsg	fsb
≤ 10	33 382	32 835	537	5	3
10–20	25 888	22 844	3 084	0	5
20–30	8 293	5 296	3 003	0	3
30–40	3 759	1 600	2 163	5	0
40–50	1 939	440	1 369	124	7
50–60	977	34	931	12	0
60–70	359	14	345	0	0
70–80	229	3	225	1	0
80–90	148	2	146	0	0
90–100	130	0	130	0	0
> 100	196	0	193	0	3

Tabulka 2.9: Počty nejlépe zkomprimovaných souborů – porovnání metody frekventovaných slov upravující vstup pro gzip nebo bzip2 a použití samotných aplikací gzip a bzip2

Tabulka 2.9 srovnává kompresi pomocí metod gzip a bzip2 na původních souborech a na souborech upravených pomocí metody frekventovaných slov. Z naměřených dat jasně plyne, že pro drtivou většinu souborů byla nejlepší jedna z původních metod (gzip, bzip2). Tento způsob předzpracování nebudeme tedy dále uvažovat.

Soubory	Celková velikost komprimovaných souborů v B				
v kB	bez komp.	gzip	bzip2	fsg	fsb
≤ 10	195 441 248	62 201 974	66 889 194	65 470 627	71 002 797
10–20	372 419 892	93 891 488	96 931 143	98 386 623	102 945 401
20–30	205 296 289	43 353 599	43 558 168	45 005 777	45 785 297
30–40	131 980 917	24 072 347	23 666 224	24 789 638	24 699 694
40–50	88 015 810	14 836 842	14 330 609	15 060 885	14 863 699
50–60	54 676 625	9 249 960	8 635 500	9 260 703	8 906 561
60–70	23 674 719	3 863 760	3 508 452	3 883 708	3 621 396
70–80	17 466 596	2 473 924	2 230 697	2 477 591	2 301 971
80–90	12 796 748	1 741 864	1 540 899	1 735 019	1 585 510
90–100	12 634 495	1 683 799	1 463 146	1 683 780	1 506 596
> 100	22 467 477	2 740 619	2 282 637	2 727 049	2 351 884

Tabulka 2.10: Celková velikost zkomprimovaných souborů – porovnání metody frekventovaných slov upravující vstup pro gzip nebo bzip2 a použití samotných aplikací gzip a bzip2

Také srovnání celkové velikosti souborů komprimovaných jednotlivými metodami uvedené v tabulce 2.10 vyznívá jasně lépe pro kompresi původními metodami gzip a bzip2.

Předzpracování metodou frekventovaných slov je použitelné pro všechny soubory s výskytem libovolných znaků a bez nutné znalosti struktury souborů, protože slovník, jehož prvky jsou v předzpracovávaném souboru nahrazovány nepoužitými znaky, se vytváří pro každý soubor zvlášť. Adaptivní slovník ale musí být uložen spolu s každým předzpracovaným souborem. Domníváme se, že právě slovník uložený v předzpracovaném dokumentu brání tomu, aby komprese takto upraveného souboru vycházela lépe než komprese souboru původního.

2.7 Shrnutí

Pro zvýšení účinnosti komprese ukládaných html dokumentů jsme pro předzpracování komprimovaných dokumentů navrhli a otestovali dvě metody (pro každou dvě varianty) používající statický slovník a jednu metodu, která vytváří nový slovník pro každý dokument. V předcházející části textu byly všechny tři metody porovnány s metodami gzip a bzip2 – běžnými kompresními metodami, které připadají v úvahu pro kompresi ukládaných html dokumentů. Aby bylo možné rozhodnout, která z nich je lepší, je třeba porovnat je vzájemně. Porovnání bylo provedeno na souborech, na které jdou všechny metody použít (limitující je metoda 2 rozlišující velikost znaků). Zároveň bylo provedeno porovnání všech metod také na všech souborech, přičemž metoda předzpracování 2 i se svou variantou byly vyhodnocovány pouze na souborech, na kterých šly použít. Jen připomeňme, že pokud nejlepšího výsledku pro nějaký soubor dosáhne více metod současně, je prvenství započteno všem těmto metodám.

Soubory		Kolikrát byla nejúčinnější metoda											
v kB	počet	gzip	bzip2	mgz1	mbz1	mgz1i	mbz1i	mgz2	mbz2	mgz2i	mbz2i	fsg	fsb
≤ 10	12 448	2 571	196	20	4	48	4	46	2	**9 737**	27	3	3
10–20	3 997	7	269	11	48	13	81	16	43	**3 370**	206	0	2
20–30	571	0	80	0	46	0	62	0	39	**253**	115	0	0
30–40	191	0	16	0	15	0	27	0	16	**75**	45	1	0
40–50	73	0	11	0	8	0	11	0	8	20	**21**	0	0
50–60	24	0	3	0	2	0	4	0	0	2	**14**	0	0
60–70	20	0	4	0	1	0	4	0	3	0	**8**	0	0
70–80	4	0	1	0	0	0	**2**	0	0	0	1	0	0
80–90	5	0	0	0	0	0	0	0	2	0	**3**	0	0
90–100	8	0	**4**	0	1	0	1	0	0	0	2	0	0
< 100	15	0	**5**	0	2	0	3	0	0	0	**5**	0	0

Tabulka 2.11: Porovnání všech metod na souborech obsahujících pouze dolních 128 znaků, na kterých byly použitelné obě varianty metody 2

Jak můžeme nahlédnout z tabulky 2.11 a 2.12, nelze říci, že by některá z metod byla jasně nejlepší a měla by se používat pouze ta. Na druhou stranu metoda frekventovaných slov může být z dalšího vývoje vyřazena, neboť ze všech metod dosáhla nejlepšího výsledku pouze na několika málo souborech. Dle očekávání se nejvíce prosazovaly metody předzpracování bez rozlišení velikosti znaků kódovaných řetězců. Tyto metody jsou ovšem ztrátové.

Z pohledu účinnosti komprese se vstupem upraveným některou z navržených metod bude kromě počtu případů, ve kterých byla účinnější, zajímavá také velikost prostorové úspory. Tabulka 2.13 uvádí průměrnou úsporu místa jednotlivých

Soubory		Kolikrát byla nejúčinnější metoda											
v kB	počet	gzip	bzip2	mgz1	mbz1	mgz1i	mbz1i	mgz2	mbz2	mgz2i	mbz2i	fsg	fsb
≤ 10	33 382	4 790	395	4 964	4	**18 417**	23	46	2	9 742	27	3	3
10–20	25 888	84	848	5 937	197	**20 201**	732	16	43	3 370	207	0	2
20–30	8 293	7	512	1 558	378	**5 789**	1 200	0	39	254	116	0	0
30–40	3 759	0	296	318	348	**1 836**	1 103	0	16	75	45	1	0
40–50	1 939	0	108	65	242	697	**868**	0	8	20	21	0	0
50–60	977	0	58	15	310	79	**552**	0	0	2	14	0	0
60–70	359	0	42	3	64	31	**217**	0	3	0	8	0	0
70–80	229	0	21	1	39	4	**168**	0	0	0	1	0	0
80–90	148	0	16	0	21	5	**105**	0	2	0	3	0	0
90–100	130	0	22	0	18	1	**88**	0	0	0	2	0	0
< 100	196	0	29	0	31	0	**135**	0	0	0	5	0	0

Tabulka 2.12: Porovnání všech metod na všech souborech. Metoda 2 a její varianty se vyhodnocují pouze na souborech, na kterých jdou použít

Soubory	Průměrná úspora v B na jeden soubor proti původní metodě									
v kB	mgz1	mbz1	mgz1i	mbz1i	mgz2	mbz2	mgz2i	mbz2i	fsg	fsb
≤ 10	32	17	40	28	27	18	35	26	8	7
10–20	58	22	69	39	45	24	56	34	34	14
20–30	82	29	95	46	67	29	83	44	68	25
30–40	106	33	123	54	108	37	117	43	79	0
40–50	147	35	168	60	139	39	150	53	92	25
50–60	270	48	284	65	169	34	182	56	123	0
60–70	210	50	232	77	164	43	177	52	113	0
70–80	291	48	321	88	249	65	287	66	130	0
80–90	289	50	316	79	319	67	332	90	162	0
90–100	297	53	323	87	183	60	190	66	134	0
< 100	379	50	400	74	412	47	390	55	227	63

Tabulka 2.13: Průměrný zisk jednotlivých metod oproti původní metodě v případech, kdy byla komprese s předzpracováním účinnější

metod s předzpracovaným vstupem oproti původním metodám, ze kterých byly upravené metody odvozeny. Sledovaly se pouze případy, kdy byla upravená metoda účinnější. Tabulka 2.14 uvádí průměrnou úsporu místa upravených metod oproti nejlepší metodě z původních (včetně možnosti bez komprese velmi malých souborů). Opět jsme uvažovali pouze případy, kdy byla upravená metoda účinnější. Tyto průměrné zisky nejsou tak velké jako průměrné zisky oproti metodě, ze které byla nová odvozena. Tyto zisky se totiž počítají vůči nejúčinnější ze všech původních metod a mnohdy metoda, ze které je nová odvozena, není tou nejúčinnější na zkoumaném souboru. Proto jsou průměrné zisky menší.

Při pohledu na průměrné zisky upravených metod je jasné, že úspora není příliš velká. Význam ale může mít pro samostatně ukládané soubory, kdy velikost komprimovaného souboru je těsně nad násobkem velikosti alokační jednotky disku. Pak i drobné snížení velikosti může přinést větší úsporu v podobě jedné alokační jednotky. Pokud se ukládá více takto komprimovaných souborů do jednoho archivu, získává uspořený prostor opět na významu, neboť tyto drobné zisky se sčítají.

Metoda 1 a metoda 2 zřejmě oproti metodě frekventovaných slov těží ze statického slovníku, který je přímo součástí dané aplikace. O nejlepší kompresní metodě se bude muset rozhodnout pro každý soubor zvlášť. Stojí za zvážení,

Soubory v kB	Průměrná úspora v B na jeden soubor proti původní metodě									
	mgz1	mbz1	mgz1i	mbz1i	mgz2	mbz2	mgz2i	mbz2i	fsg	fsb
≤ 10	32	11	40	23	27	9	35	16	5	24
10–20	57	19	69	34	45	20	55	26	0	5
20–30	79	26	91	40	68	28	88	37	0	32
30–40	93	33	110	50	108	35	111	41	56	0
40–50	111	34	133	57	115	38	119	51	63	26
50–60	106	48	122	65	82	35	89	58	153	0
60–70	102	50	122	76	0	43	0	52	0	0
70–80	80	49	87	88	0	65	0	66	6	0
80–90	127	50	153	78	0	67	0	90	0	0
90–100	110	53	156	87	0	60	0	66	0	0
< 100	0	50	23	74	0	47	0	55	0	63

Tabulka 2.14: Průměrný zisk jednotlivých metod oproti nejlepší původní metodě v případech, kdy byla komprese s předzpracováním účinnější

zda do rozhodování zařadit i metodu 2, která se ukázala použitelná pouze na přibližně jedné čtvrtině testovaných souborů. Pokud faktor velikosti ukládaného dokumentu hraje hlavní roli a mírné zdržení při kompresi nevadí, rozhodně lze metodu 2 do rozhodování zahrnout. V případě, že faktor rychlosti při uložení je přednější, mělo by se rozhodování omezit na metodu 1 a původní metody gzip a bzip2.

Pokud navíc nevadí ztráta informace o velikosti znaků použitých v názvech html elementů a jejich atributů, lze hledat nejlepší kompresi i mezi výsledky metod se statickým slovníkem bez rozlišení velikosti znaků.

Použitá metoda	Celková velikost v B	Kompresní poměr
původní soubory	1 136 870 816	100,0 %
uložené systémem EGOTHOR	317 049 259	27,9 %
pouze gzip	260 110 176	22,9 %
pouze bzip2	265 036 669	23,3 %
pouze mgz1	255 919 630	22,5 %
pouze mbz1	264 891 640	23,3 %
pouze mgz1i	255 107 509	22,4 %
pouze mbz1i	263 608 385	23,2 %
pouze fsg	270 481 400	23,8 %
pouze fsb	279 570 806	24,6 %
pro každý soubor nejlepší	251 769 011	22,1 %

Tabulka 2.15: Celková velikost všech souborů zkomprimovaných pomocí zkoumaných metod a pro daný soubor vždy nejlepší metodou. Kompresní poměr je počítán jako $\frac{\text{velikost zkomprimovaného souboru}}{\text{velikost původního souboru}}$

Jistě je také zajímavé podívat se na celkovou velikost uspořeného prostoru. Tabulka 2.15 uvádí srovnání celkové velikosti souborů, pokud by se každý z nich komprimoval pomocí zkoumaných metod (vyjma obou variant metody 2 vzhledem k jejich omezené použitelnosti) nebo pokud by se každý z nich komprimoval tou metodou, která je pro něj nejlepší. Pro srovnání je také uvedena velikost archivů vytvořených systémem EGOTHOR, ze kterých se testovací soubory zís-

kaly. Archivy byly ale komprimovány pro více souborů najednou a obsahovaly pro každý z nich identifikační údaje.

Jestliže se tedy před kompresí každého dokumentu vyzkouší, která z metod je pro něj z hlediska prostoru nejlepší (včetně metody 2, pokud jde použít), dá se při ukládání většího počtu dokumentů uspořit významnější diskový prostor.

Na testovaných souborech činila průměrná délka dokumentu předzpracovaného metodami se statickým slovníkem 85 % délky původního dokumentu. Doba potřebná na předzpracování dokumentu a následnou kompresi metodou gzip byla v průměru o 5 % delší než doba komprese původního dokumentu metodou gzip. Doba předzpracování dokumentu a následné komprese metodou bzip2 byla průměrně o 12 % kratší než doba komprese původního dokumentu metodou bzip2. Na zkrácení doby se zřejmě podepsala menší velikost vstupu pro bzip2.

Pro následnou kompresi předzpracovaných dokumentů byly vyzkoušeny i jiné kompresní metody než jen gzip a bzip2. Tyto metody však nebyly použitelné pro jejich vysoké časové nároky (7zip) nebo pro jejich nedostatečnou účinnost (WinRar, LHA).

Na závěr se ještě zmíníme, že metody 1 a 2 ve variantách rozlišujících velikost znaků byly přijaty k publikaci jako [13].

Kapitola 3

Tvorba úložiště

V této kapitole jsou popsány principy fungování a struktura úložiště html dokumentů. V úvodu nejprve připomeňme, jaké požadavky by mělo úložiště splňovat:

1. Minimalizovat diskový prostor potřebný pro uložení dokumentu.
2. Umožňovat ukládání více verzí téhož dokumentu, tzn. umožnit uložení nejen aktuální, ale i starších verzí.
3. Jednotlivé html dokumenty musejí být rychle přístupné, zejména jejich nejnovější verze. Starší verze mohou být přístupné s větší prodlevou, neboť jsou tyto verze dotazovány velice zřídka.

Z požadavků plynou operace, které musí úložiště podporovat:

1. Uložit novou verzi dokumentu do archivu.
2. Vrátit nejnovější verzi dokumentu.
3. Vrátit požadovanou verzi dokumentu.

Při vkládání nové verze dokumentu se zároveň kontroluje, zda naposledy uložená verze není shodná s právě ukládanou. Pokud shodné jsou, musí to být zřejmé z výsledku dané operace, ať už se uložení shodné verze ignoruje, nebo se shodná verze do archivu ukládá. Informace o shodných verzích je velice důležitá pro fulltextový systém, který dané úložiště bude používat, neboť systém musí u každého dokumentu vědět, s jakou periodou se obsah příslušného dokumentu mění a tedy s jakou periodou je třeba dokument aktualizovat (stahovat novou verzi).

Vzhledem k tomu, že nejčastější operací v rámci jednoho dokumentu je vrácení aktuální verze, lze tuto operaci provádět samostatně bez nutnosti znát číslo nejnovější verze dokumentu. Pokud je známo číslo nejnovější verze, lze tuto verzi získat rovněž pomocí operace pro získání konkrétní verze. Touto operací lze dále získat libovolnou starší verzi uloženého dokumentu.

Z těchto požadavků na vlastnosti a funkčnost jsme také vycházeli při návrhu struktury archivu pro ukládání jednotlivých html dokumentů.

3.1 Výběr nejlepší komprese

V předcházející kapitole bylo předvedeno, že účinnost komprese běžnými metodami gzip a bzip2 může být o něco málo vylepšena vhodným předzpracováním vstupu těchto kompresních metod za cenu mírného zpomalení komprese. Která z původních a upravených metod dosáhne nejlepšího výsledku (nejmenší velikosti zkomprimovaného dokumentu), se nedá dopředu dost dobře odhadnout. Výsledek závisí na velikosti souboru, jazyce jeho obsahu, ale hlavně na jeho struktuře (použitých html elementech a jejich atributech).

Problém obtížné predikce nejlepší komprese pro ukládaný dokument je při ukládání řešen provedením všech kompresních metod na daný soubor. Jde o jakousi soutěž, kdy se ukládaný dokument zkomprimuje všemi metodami a ukládá se výsledek té komprese, která dosáhla nejvyšší účinnosti. Pro některé velmi malé soubory (o velikosti řádově desítky až stovky bajtů) může být původní soubor menší než soubor komprimovaný. Pak se jako nejlepší komprese bere samozřejmě žádná komprese a soubor se ukládá v původní podobě.

Daní za vyzkoušení všech kompresních metod při hledání nejlepší komprese je delší doba potřebná při ukládání nové verze dokumentu. Doba výběru kompresní metody pro danou verzi je ale kratší než doba potřebná na porovnání ukládané verze s verzí předchozí. Doufáme, že doba výběru nejlepší kompresní metody nebude úzkým hrdlem systému.

Dobu je možné částečně zkrátit snížením počtu zkoušených metod. Výsledné řešení dává uživateli možnost nastavit, které metody se budou zkoušet. Metody gzip a bzip2 jsou základem a provádějí se vždy, dále lze provádět obě metody předzpracování se statickým slovníkem. Uživatel také může umožnit vyzkoušení komprese metodami předzpracování ve variantě bez rozlišování velikosti znaků kódovaných řetězců. V minimálním případě se vybírá nejlepší komprese pouze z původních metod gzip a bzip2 (a samozřejmě možnosti bez komprese), v maximálním případě se vedle varianty bez komprese vybírá z původních metod gzip a bzip2, dále z těchto metod se vstupem upravovaným metodami předzpracování 1 a 2 včetně obou jejich variant s rozlišením a bez rozlišení velikosti znaků kódovaných řetězců.

Metoda předzpracování používající dynamický slovník nebyla do výsledného řešení zahrnuta, protože při testování nedosáhla očekávaných výsledků.

3.2 Struktura archivu

Dokumenty jsou ukládány do archivů, pro každý dokument se vytváří vlastní archiv. Pro tvorbu archivů jsou využívány soubory běžného souborového systému. Testování probíhalo na souborovém systému Ext2. Každý archiv je tvořen několika soubory, které mají stejnou část jména bez přípon, která odpovídá jednoznačnému identifikátoru ukládaného dokumentu. Soubory tvořící archiv se dají rozdělit na dva typy:

Datové soubory jsou soubory obsahující jednotlivé zkomprimované verze daného dokumentu.

Index jsou soubory obsahující metadata, tzn. informace o ukládaných verzích dokumentu, použitou kompresní metodu atd.

Dále archiv využívá další dva typy pomocných souborů, které přímo neslouží k ukládání dat:

Zámek je soubor sloužící k zamykání archivu při práci s ním.

Kontrola konzistence je soubor, jehož výskyt indikuje, že archiv se nenachází v konzistentním stavu.

Úložiště k ukládání souborů tvořících jednotlivé archivy má tuto adresářovou strukturu:

data – adresář pro ukládání datových souborů

index1 – adresář pro ukládání souborů indexu první úrovně

index2 – adresář pro ukládání souborů indexu druhé úrovně

locks – adresář pro ukládání zámků

consist – adresář pro ukládání souborů na kontrolu konzistence

tmp – adresář, který slouží pro ukládání nových souborů tvořících archiv do doby, než budou všechny nové soubory hotové

3.2.1 Datové soubory

Archiv příslušející nějakému dokumentu obsahuje jeden či více datových souborů (důvod bude vysvětlen dále), které jsou uloženy v adresáři *data*. Název každého datového souboru má tvar `IdentifikátorDokumentu.dt.ČísloSouboru`. Organizace datového souboru do jisté míry připomíná sekvenční soubor [15, str. 20] – uspořádaný homogenní soubor záznamů proměnné délky, kde klíčem pro třídění je číslo verze dokumentu. Záznamy jsou v našem případě jednotlivé zkomprimované verze uloženého dokumentu. Pro oddělení jednotlivých záznamů se nepoužívají žádné oddělovače, ale informace z indexu. Poloha každého záznamu v datovém souboru je jednoznačně určena jeho vzdáleností od začátku souboru a jeho délkou.

Nabízejí se dva způsoby, kterými lze jednotlivé verze ukládat v datovém souboru – od nejnovější verze na začátku souboru po nejstarší verzi na jeho konci, nebo opačně. Pokud bude nejnovější verze na začátku datového souboru, bude při vkládání nové verze třeba vytvořit nový soubor (zvětšený o novou verzi), novou verzi dokumentu uložit na začátek nového souboru a za ní překopírovat starý datový soubor. Vytvořením nového souboru bude zajištěno, že soubor bude z pohledu disku uložen souvisle[1] – v blocích za sebou – a bude se rychleji číst jeho obsah. Variantě s nejnovější verzí na konci bude při ukládání nové verze stačit zvětšení souboru a uložení nové verze na jeho konec. Při tomto přístupu ale hrozí fragmentace souboru na více místech disku a případné pomalejší čtení obsahu souboru, neboť se může stát, že jedna verze dokumentu bude uložena ve více blocích na různých místech disku.

Pro uložení nejnovější verze na začátku datového souboru dále hovoří způsob ukládání souborů v systému souborů Ext2 [2], na kterém bylo úložiště vyvíjeno

[1] to nemusí platit, pokud není na disku dostatek souvislého místa

a testováno. U jiných systémů souborů mohou nastat obdobné problémy. Každému souboru v Ext2 je přiřazen i-uzel, který kromě jiných informací o souboru obsahuje 10 přímých odkazů na datové bloky, ve kterých je soubor uložen, a 3 odkazy na nepřímé bloky, z nichž jeden odkazuje na datové bloky, jeden znovu na nepřímé bloky a poslední se odkazuje na nepřímé bloky, které se znovu odkazují na nepřímé bloky. Pokud je soubor dostatečně velký, je při přístupu na jeho konec potřeba procházet seznamem odkazů z nepřímých bloků do datových bloků, což samozřejmě stojí čas.

Dle požadavku 3 má být důraz kladen především na rychlost získání nejnovější verze. U starších verzí je tolerována delší doba pro získání příslušné verze dokumentu. Proto jsou verze dokumentu do datového souboru ukládány právě v pořadí od nejnovější verze na začátku souboru po nejstarší verzi na jeho konci.

Druhou výhodou zvoleného pořadí ukládání jednotlivých verzí dokumentů je fakt, že pokud je to z hlediska prostoru výhodné, ukládá se místo verze pouze její rozdíl od verze novější. Při získání starší verze uložené jako rozdíl je pak nutné číst soubor postupně od novějších verzí a požadovanou verzi rekonstruovat z uložených rozdílů. Organizace datového souboru dává proto možnost číst datový soubor po blocích uložených na disku za sebou, což by u druhé diskutované organizace datového souboru nešlo.

Jak už bylo v úvodu zmíněno, datové soubory jsou implementovány jako běžné soubory souborového systému. Jak známo, takový soubor má omezenou velikost (např. při velikosti alokační jednotky 512 B činí maximální možná velikost jednoho souboru přibližně 1 GB). V archivu se proto počítá, že pokud by velikost datového souboru po uložení další verze přesáhla maximální možnou velikost, začne se v rámci archivu vytvářet další datový soubor. Jednotlivé datové soubory stejného archivu jsou odlišeny druhou příponou. Pro lepší představu: předpokládejme, že dokument má identifikátor `abcd` a jemu odpovídající archiv obsahuje dva datové soubory. Pak datový soubor se staršími verzemi bude mít název `abcd.dt.1` a datový soubor s novějšími verzemi `abcd.dt.2`.

Lehce jsme se zmínili, že při organizaci datového souboru s nejnovější verzí na jeho začátku je třeba při vkládání každé nové verze překopírovávat původní datový soubor do datového souboru nového. Představme si, že datový soubor se svojí velikostí bude blížit jeho maximální velikosti. Pak překopírování řádově stovek MB při každém ukládání nové verze je časově náročné. I když je při ukládání nové verze tolerována delší doba, není tato prodleva přípustná. Řešení přináší zavedení tzv. primárního datového souboru. Primární datový soubor se svojí organizací neliší od dříve zavedených datových souborů. Jeho název je také odlišován druhou příponou, která má hodnotu 0, tedy v kontextu předchozího příkladu bude název primárního datového souboru `abcd.dt.0`.

Do primárního datového souboru se ukládají nejnovější verze stejným způsobem jako do ostatních datových souborů, tedy nejnovější verze vždy na začátku. Nejnovější verze se vždy ukládá do tohoto souboru. Zároveň má primární datový soubor definovaný nejmenší a největší možný počet verzí v něm uložených. Nejmenší možný počet by měl odpovídat počtu nejnovějších verzí, na které se nejčastěji dotazuje, aby při vyhodnocování těchto dotazů nebylo nutné pracovat s více datovými soubory. Horní hranice počtu verzí částečně určuje, jak velká část bude pravidelně kopírována při vkládání každé nové verze, a zároveň určuje, jak často budou nejstarší verze z primárního datového souboru přesouvány do

běžného datového souboru. Tento přesun už samozřejmě znamená kopírování potenciálně velkého původního souboru.

Důvod zavedení primárního datového souboru si ještě ukažme na jednoduchém příkladu. Nechť nejmenší počet verzí v primárním datovém souboru je 5 a nejvyšší počet verzí 25. Pak při ukládání nové verze bude při velikosti primárního datového souboru 6, 7, ..., 20 verzí nutné kopírovat maximálně řádově desítky kB (aktualizovat se bude pouze primární datový soubor) a teprve při ukládání další verze bude třeba kopírovat až řádově stovky MB. Pro náš příklad tato situace nastává pouze při ukládání každé jedenadvacáté verze. Průměrná doba na uložení jedné verze se díky tomu podstatně sníží.

Díky tomu, že každá verze je samostatně zkomprimována (dokonce různé verze mohou být komprimovány různými metodami) a až potom uložena do datového souboru, lze přímo přistupovat do datového souboru k jednotlivým uloženým verzím bez nutnosti dekomprese celého datového souboru. Tím se urychluje operace získání libovolné verze (požadavek 3). To však může negativně ovlivnit celkový dosažený kompresní poměr.

Podrobný popis algoritmu uložení nové verze dokumentu bude popsán později, zde byly zmíněny jen některé, pro popis struktury datových souborů nezbytné, záležitosti.

3.2.2 Index

Kvůli uchování informací o uložených verzích dokumentu se skládá archiv vedle datových souborů také ze souborů indexu. Informace z indexu pomáhají lokalizovat začátek a konec úseku v datovém souboru, který odpovídá jedné uložené verzi dokumentu. V rámci jednoho archivu může existovat více datových souborů, proto musí mít index dvě úrovně.

Začněme popisem indexu druhé úrovně. Právě jeden soubor indexu druhé úrovně obsahuje nezbytné informace o verzích uložených v právě jednom datovém souboru. Názvy souborů jsou ve tvaru
`IdentifikátorDokumentu.id.ČísloSouboru`, kde číslo souboru odpovídá druhé příponě názvu datového souboru, který je tímto souborem indexován. Soubory indexu druhé úrovně jsou textové soubory ve formátu CSV, kde každý řádek je záznam odpovídající jedné uložené verzi a dodržuje tento formát:

```
KomprimovanáVelikost,PůvodníVelikost,MetodaKomprese,ČasovéRazítko
```

Jednotlivá pole řádku rozeberme trochu podrobněji:

Komprimovaná velikost je velikost komprimované verze uváděná v bajtech. Udává počet bajtů, které odpovídají uložené verzi.

Původní velikost je velikost v bajtech původního souboru.

Metoda komprese říká, jakou metodou byla uložená verze dokumentu komprimována a samozřejmě jakou metodou má být dekomprimována při jejím získávání. Může nabývat těchto hodnot:

1 žádná komprese, uložen původní soubor

2 komprese pomocí původní metody gzip

3 komprese pomocí původní metody bzip2

4 komprese pomocí metody gzip se vstupem upraveným metodou 1

5 komprese pomocí metody bzip2 se vstupem upraveným metodou 1

6 komprese pomocí metody gzip se vstupem upraveným metodou 1 bez rozlišení velikosti znaků kódovaných řetězců

7 komprese pomocí metody bzip2 se vstupem upraveným metodou 1 bez rozlišení velikosti znaků kódovaných řetězců

8 komprese pomocí metody gzip se vstupem upraveným metodou 2

9 komprese pomocí metody bzip2 se vstupem upraveným metodou 2

10 komprese pomocí metody gzip se vstupem upraveným metodou 2 bez rozlišení velikosti znaků kódovaných řetězců

11 komprese pomocí metody bzip2 se vstupem upraveným metodou 2 bez rozlišení velikosti znaků kódovaných řetězců

Stejné hodnoty se mohou vyskytnout také se záporným znaménkem. Význam takové hodnoty je uložení pouze rozdílu této verze od novější verze a komprese pomocí metody odpovídající kladné hodnotě.

Časové razítko odpovídá času uložení této verze dokumentu do archivu. Udává se v počtu sekund od začátku unixové éry (1. 1. 1970), což umožňuje jeho transformaci do libovolného formátu.

Ukažme si obsah souboru indexu druhé úrovně na krátkém příkladu:

```
12143,89008,4,1182797275
4934,88989,-3,1182797274
4077,88604,-3,1182797272
...
```

Pokud přeložíme do běžné řeči například první řádek, tak se dozvíme, že uložená verze dokumentu má původní velikost 89 008 bajtů. Jako nejúčinnější komprese byla vybrána metoda identifikovaná číslem 4, tedy gzip se vstupem upraveným metodou předzpracování 1. Touto metodou byl dokument komprimován na velikost 12 143 bajtů a uložen do datového souboru. Poslední položka v řádku je časové razítko uložení v počtu sekund od začátku unixové éry.

Záznamy jsou v souboru uloženy ve stejném pořadí, jako příslušné verze v datovém souboru – záznam popisující nejnovější verzi v datovém souboru uloženou na začátku souboru, záznam popisující nejstarší verzi na jeho konci. Při vkládání nové verze se opět nový záznam vkládá na začátek souboru indexu a zbytek starého souboru indexu se překopíruje za nový záznam. Velikost souboru indexu je nesrovnatelně menší[2] než datový soubor, proto nebyla zavedena obdoba primárního datového souboru pro zmírnění časové náročnosti při kopírování zbytku souboru.

První úroveň indexu je tvořena souborem se jménem ve tvaru `IdentifikátorDokumentu.id`. Soubory indexu první úrovně všech dokumentů se

[2] až na triviální případy ukládání pouze několikabajtových dokumentů nebo opakované ukládání shodných verzí

ukládají do adresáře *index1*. Jedná se o textový soubor ve formátu CSV, jehož každý řádek odpovídá jednomu souboru indexu druhé úrovně a má následující formát:

```
OdVerze,DoVerze,Číslo,Velikost,PočetDiffů
```

A opět malá ukázka obsahu souboru indexu první úrovně (maximální velikost datového souboru byla nastavena na 100 kB):

```
32,40,0,47158,19
24,31,2,32850,-1
0,23,1,99514,-1
```

Význam jednotlivých polí je následující:

Od verze je číslo nejstarší verze uložené v indexovaném datovém souboru.

Do verze je číslo nejnovější verze uložené v indexovaném datovém souboru.

Číslo jednoznačně identifikuje soubor indexu druhé úrovně a odpovídající datový soubor. Je to číslo použité jako druhá přípona v jejich názvech.

Velikost je celková velikost datového souboru, který je indexován tímto souborem indexu druhé úrovně. Uvádí se v bajtech.

Počet diffů udává počet verzí dokumentu bezprostředně uložených před (časově, nikoli prostorově) nejnovější verzí, které byly uloženy jako rozdíl od novější (následující) verze. Tato položka je obsažena proto, aby tento počet mohl být z důvodu časové efektivity při získávání starších verzí omezen. Významu nabývá pouze na začátku celého archivu, kam se přidávají nové verze, tedy pro primární datový soubor. Pro ostatní datové soubory nabývá hodnoty -1 značící skutečnost, že pro ně již nemá význam.

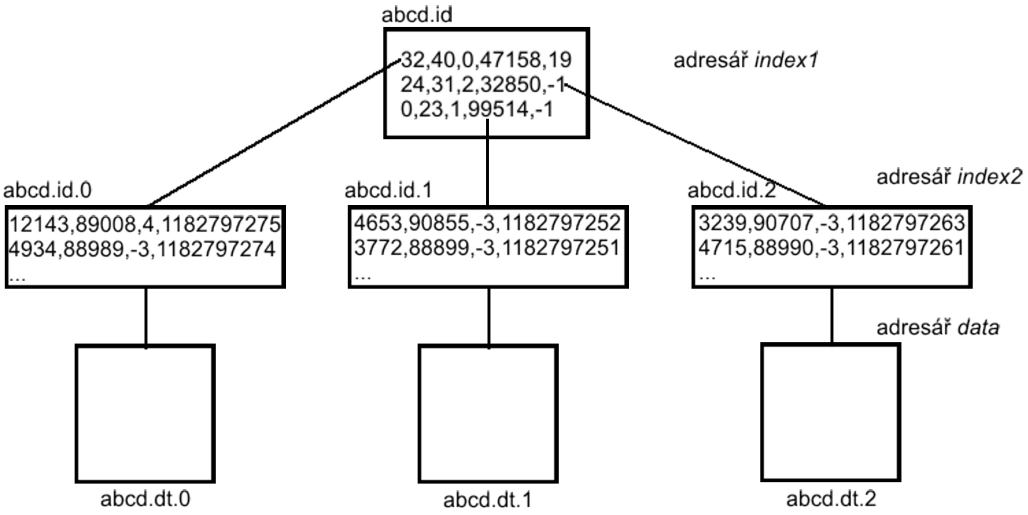
Z ukázky je patrné, že je-li v daném archivu celkem n datových souborů, pak nejnovější verze dokumentu jsou uložené v datovém souboru s číslem 0 (primární datový soubor), verze starší jsou v datovém souboru s číslem $n - 1$ až konečně verze nejstarší jsou uloženy v datovém souboru s číslem 1.

Na obrázku 3.1 je vztah mezi datovými soubory a soubory indexu první a druhé úrovně jednoho archivu vyjádřen také graficky.

3.2.3 Kontrola paralelního přístupu

S jedním archivem může zároveň pracovat více procesů. Je proto nutné zajistit, aby každý proces pracoval s archivem v konzistentním stavu. Jinými slovy to znamená, že se nesmí v žádném případě stát, že jeden proces změní soubory v rámci jednoho archivu druhému procesu, který s tímto archivem právě pracuje.

Archiv se může dostat do nekonzistentního stavu pouze při ukládání nové verze. Operace získání nejnovější nebo jakékoli jiné verze soubory archivu nemění. Musí být tedy zajištěno, že pokud některý proces ukládá novou verzi, žádný jiný proces nesmí žádnou verzi získávat nebo také ukládat novou verzi. Dále musí být zajištěno, že pokud některý proces získává libovolnou verzi, pak žádný jiný proces nesmí ukládat novou verzi. Více procesů ale může zároveň získávat uložené verze.



Obrázek 3.1: Schéma vztahů mezi soubory v archivu

Řešením tohoto problému je použití zámku pro archiv. Proces, který bude ukládat novou verzi, zamkne archiv pomocí exkluzivního zámku – žádný jiný proces nesmí s archivem pracovat (ani číst uloženou verzi, ani zapisovat novou). Proces, který bude číst libovolnou verzi z archivu, zamkne archiv pomocí sdíleného zámku – více procesů může zároveň číst, ale žádný nesmí zapisovat. Archiv je tvořen více samostatnými soubory, které se při vkládání nové verze dokumentu navíc mění a případně vznikají další. Aby nebylo nutné zamykat každý tento soubor zvlášť, existuje pro každý archiv soubor v adresáři *locks*. Zámek na celý archiv se realizuje pomocí zámku na tento soubor (přímo zamčení souboru, ne jeho vytvoření). Soubor neobsahuje žádná data, jeho název je identifikátor dokumentu.

Zamykání celého archivu exkluzivním zámkem při ukládání nové verze na celou dobu ukládání může zbytečně dlouho omezovat procesy, které chtějí získat uložené verze, což může být v rozporu s požadavkem 3. Pokud by aplikace používající knihovnu byla schopna zajistit, že v jednom okamžiku bude spuštěn v každém archivu nejvýše jeden proces provádějící ukládání nové verze, lze zamknout archiv pouze sdíleným zámkem a exkluzivní zámek použít pouze na překopírování nových souborů archivu.

3.2.4 Zajištění konzistence archivu

Při ukládání nové verze dokumentu se nové soubory archivu vytvářejí v adresáři *tmp*. Pokud jsou všechny nové soubory vytvořené, nahradí se jimi odpovídající staré soubory v adresářích *data*, *index1* a *index2*. Selže-li z jakéhokoliv důvodu proces uložení nové verze dříve, než se začnou staré soubory nahrazovat novými, nedojde k porušení konzistence archivu. K tomu dojde, je-li proces ukončen během nahrazování souborů, kdy některé soubory jsou již v příslušných adresářích nahrazeny novými a některé ještě ne, případně některé nově vytvořené datové soubory a soubory indexu druhé úrovně ještě nejsou přesunuty do příslušných adresářů.

Proces má sice archiv uzamčený pomocí zámku na souboru v adresáři *locks*, ale tento zámek je v systému Unix při skončení procesu automaticky uvolněn.

Další procesy by tedy mohly přistupovat k archivu v nekonzistentním stavu. Tomu zamezí použití tzv. souboru kontroly konzistence umístěného v adresáři *consist*. Postup je celkem jednoduchý. Proces ukládající novou verzi dokumentu do archivu těsně před zahájením nahrazování starých souborů archivu novými vytvoří v adresáři *consist* soubor, jako jehož název se použije identifikátor dokumentu. Po dokončení nahrazení všech souborů se soubor kontroly konzistence opět smaže. Pokud by byl ukládající proces ukončen během nahrazování souborů, nebude soubor kontroly konzistence smazán.

Každý proces, který chce pracovat s archivem, po získání zámku na archiv nejprve ověří, zda pro požadovaný archiv neexistuje soubor kontroly konzistence. Pokud ne, je archiv v konzistentním stavu a proces může normálně pokračovat ve své činnosti. Pokud soubor existuje, před zahájením své činnosti musí uvést archiv do konzistentního stavu. K tomu stačí z adresáře *tmp* všechny soubory příslušející danému archivu (podle identifikátoru dokumentu v jejich názvu) přesunout do patřičných adresářů (poznají se podle přípon souborů). Z popsaného postupu je jasné, že první proces, který zjistí před prací s archivem jeho nekonzistenci, zjedná nápravu jeho stavu.

3.3 Operace s archivem

3.3.1 Uložení nové verze

Způsob, jakým je nová verze dokumentu ukládána do příslušného archivu bude popsán pomocí zápisu algoritmu pseudokódem, který bude přerušován odstavci vysvětlujícími principy fungování a důvody pro zvolení příslušného řešení.

Vstup algoritmu:
identifikátor dokumentu, jehož nová verze má být uložena
cesta k souboru s novou verzí

Výstup algoritmu:
návratový kód
číslo nově uložené verze
časové razítko určující dobu uložení

Algoritmus:

```
časové razítko := NOW;
zamknout archiv pro zápis;
namapovat soubor s ukládanou verzí do paměti;
zjistit nejlepší kompresi nové verze;
```

Vstupní soubor s novou verzí dokumentu se namapuje do paměti, aby se s ním dalo zacházet jako s běžnou pamětí bez nutnosti jeho načítání. Nejlepší metoda se vybere provedením všech nastavených kompresních metod na vstupním souboru a výběrem té, která dosáhla nejmenší velikosti zkomprimovaného souboru. Pokud je nastaveno i použití variant předzpracování bez rozlišení velikosti znaků, je toto ignorováno. Při ukládání nejnovější verze se vždy bere zřetel na velikost znaků

kvůli prostorové efektivitě. Může se totiž stát, že nejnovější verze bude uložena po předzpracování jednou z metod ve variantě bez rozlišení velikosti znaků ve slovech ze slovníku. Tím může být při dekompresi řada značek a atributů nahrazena jejich ekvivalenty s rozdílnou velikostí znaků. Při ukládání další verze se vytváří rozdíl obou verzí a do této doby nejnovější verze se uloží jako rozdílový soubor, pokud je po kompresi menší než zkomprimovaný původní dokument. Takový rozdílový soubor může být ale mnohem větší, než rozdílový soubor dvou verzí se zachováním velikosti znaků. Například u testovaného dokumentu s id 6 byl při použití metod předzpracování bez rozlišení velikosti znaků kódovaných řetězců i na aktuální verze zaznamenán nárůst velikosti zkomprimovaného archivu o 7 %.

```
IF archiv pro tento dokument ještě neexistuje THEN
    číslo revize := 0;
    vytvořit primární datový soubor obsahující pouze zkomprimovaný
        vstupní soubor;
    vytvořit soubor indexu 1. úrovně obsahující pouze záznam
        odkazující na soubor indexu 2. úrovně s příponou 0;
    vytvořit soubor indexu 2. úrovně obsahující pouze záznam
        o uložené verzi;
    vytvořit soubor kontroly konzistence;
    přesunout soubory z adresáře tmp do příslušných adresářů;
    odstranit soubor kontroly konzistence;
    odemknout zámek;
    návratový kód := "OK";
    RETURN;
ENDIF
```

Pokud se ukládá první verze dokumentu, bude mít číslo 0. Vytváří se nový primární datový soubor, soubor indexu první úrovně a druhé úrovně. Nic se nemusí kopírovat. Pokud již archiv existuje, pokračuje algoritmus takto:

```
otevřít soubor indexu 1. úrovně;
načíst první řádek;
číslo verze := číslo naposledy uložené verze + 1;
získat informace z načteného řádku;
otevřít soubor indexu 2. úrovně pro primární datový soubor;
načíst první řádek;
získat informace z načteného řádku;
otevřít primární datový soubor;
IF počet verzí v primárním datovém souboru = maximální počet THEN
    přesunout verze z primárního datového souboru;
ENDIF
```

Pokud již archiv nějaké verze obsahuje, načtou se informace o primárním datovém souboru z indexu první a druhé úrovně. Pokud je aktuální počet verzí uložených v primárním datovém souboru roven maximálnímu počtu, je třeba část verzí přesunout do datového souboru, který obsahuje předcházející verze, tedy do datového souboru s nejvyšším číslem. Přesun verzí probíhá takto:

```
dle indexu 2. úrovně pro primární datový soubor vytvoř seznam max-(min+1)
    nejstarších verzí a spočti jejich celkovou velikost;
vel := velikost datového souboru s nejvyšším číslem;
IF vel+celková vel. přesouvaných verzí > max. vel. souboru THEN
    IF jiný datový soubor než primární zatím neexistuje THEN
        vytvořit datový soubor s číslem 1;
        uložit do něj přesouvané verze;
        vytvořit soubor indexu 2. úrovně pro nový datový soubor;
    ELSE
        na začátek datového souboru vložit tolik verzí z konce
            seznamu, kolik se do něj vejde;
        aktualizovat index druhé úrovně pro tento datový soubor;
        překopírovat datový soubor za uložené verze;
        vytvořit nový datový soubor s číslem o jedna větší a
            uložit do něj zbylé verze;
        vytvořit soubor indexu 2. úrovně pro nový datový soubor;
    ENDIF
ELSE
    přesouvané verze uložit na začátek datového souboru
        s nejvyšším číslem;
    aktualizovat index druhé úrovně pro tento datový soubor;
ENDIF
```

Všechny změny souborů se provádějí na kopiích ukládaných do adresáře *tmp*. Po přesunu jsou v adresáři *tmp* uloženy všechny datové soubory, které se měnily nebo které byly vytvořeny nově, a také soubory indexu druhé úrovně, které patří k těmto změněným nebo nově vytvořeným datovým souborům. Primární datový soubor obsahuje v tuto chvíli o jedna méně verzí, než kolik je minimální počet. Minimálního počtu se dosáhne po uložení nové verze.

Vložením verzí na začátek datového souboru rozumějme vytvoření nového souboru stejného jména v adresáři *tmp*, uložení verzí na jeho začátek a překopírování původního datového souboru za tyto uložené verze. Obdobně funguje aktualizace souborů indexu druhé úrovně. Vytvoří se nový soubor v adresáři *tmp*, na jehož začátek se zapíší záznamy nově uložených verzí v odpovídajícím datovém souboru a za tyto záznamy se překopírují záznamy z původního souboru.

Ať už se verze z primárního datového souboru přesouvaly, nebo nikoli, pokračuje algoritmus následovně:

```
přečíst komprimovaná data odpovídající naposledy uložené verzi;
dekomprimovat data příslušnou metodou;
rozdíl := DIFF(nově ukládaná verze, naposledy uložená verze);
IF velikost(rozdíl) = 0 AND neukládat shodné verze THEN
    návratový kód := "CHYBA_SHODNÉ_VERZE";
    RETURN;
ENDIF
```

Nejnovější verze je vždy uložena na začátku primárního datového souboru. Jelikož bývá komprimována, je zapotřebí ji nejdříve dekomprimovat. Poté se provede

její rozdíl oproti nově ukládané verzi. Pokud se verze neliší a shodné verze s verzí předchozí se nemají ukládat, indikuje návratový kód výskyt této situace a ukládání končí.

Pokud se verze nelišily, nebo pokud se mají stejné verze ukládat, pokračuje algoritmus takto:

```
IF počet diffů < maximální počet OR velikost(rozdíl) = 0 THEN
    IF velikost(rozdíl) <> 0 THEN
        zjistit nejlepší kompresi rozdílu verzí;
        uložit na začátek primárního datového souboru
            komprimovanou novou verzi;
        IF komprim. rozdíl < komprim. naposledy uložená verze THEN
            za nově uloženou verzi uložit komprim. rozdíl;
        ELSE
            za nově uloženou verzi uložit kompr. verzi, která
                byla před zahájením ukládání nejnovější;
        ENDIF
        dokopírovat do primárního datového souboru zbytek
            původního;
        aktualizovat soubor indexu 2. úrovně pro primární
            datový soubor;
        aktualizovat soubor indexu 1. úrovně;
    ELSE
        aktualizovat soubor indexu 2. úrovně pro primární
            datový soubor;
        aktualizovat soubor indexu 1. úrovně;
        návratový kód := "SHODNÉ_VERZE";
    ENDIF
ELSE
    uložit na začátek primárního datového souboru
        komprimovanou novou verzi;
    celý původní primární datový soubor překopírovat za ní;
    aktualizovat soubor indexu 2. úrovně pro primární datový soubor;
    aktualizovat soubor indexu 1. úrovně;
ENDIF
```

Pokud počet verzí, které jsou bezprostředně za nejnovější verzí uloženy jako rozdíl od předchozí verze, nedosáhl povoleného maxima, může se uložit rozdíl, pokud je jeho nejlepší nalezená komprese (včetně komprese žádné) menší než nejlepší komprese celé verze. Uloží-li se skutečně rozdíl, zvýší se o jedna čítač diffů v záznamu primárního datového souboru v indexu první úrovně. Pokud se verze neliší a má se uložit, přidá se pouze záznam o nové verzi do souboru indexu druhé úrovně, kde velikost komprimovaných dat bude 0. S primárním datovým souborem není třeba provádět žádnou změnu. Uložení nulového rozdílu se provede i v případě, že počet verzí uložených jako rozdíl dosáhl maxima, protože rekonstrukce této verze na základě předchozí verze a jejich rozdílu nestojí žádný čas, neboť verze jsou samozřejmě shodné.

Nyní již stačí pouze aktualizovat soubor indexu první úrovně a přesunout soubory z adresáře *tmp* do odpovídajících adresářů:

aktualizovat soubor indexu 1. úrovně;
vytvořit soubor kontroly konzistence;
soubory z adresáře tmp přesunout do odpovídajících adresářů;
odstranit soubor kontroly konzistence;
odemknout zámek;
návratový kód := ”OK”;
RETURN;

3.3.2 Získání nejnovější verze

Získání nejnovější verze dokumentu je vůbec nejčastější operací. Proto byl návrh úložiště přizpůsoben tak, aby tato operace byla co nejjednodušší (nejrychlejší).

Vstup algoritmu:
    identifikátor dokumentu, jehož nejnovější verze má být získána
    cesta k souboru pro uložení získané verze

Výstup algoritmu:
    návratový kód
    časové razítko určující dobu uložení verze

Algoritmus:

IF archiv pro daný dokument neexistuje THEN
    návratový kód := ”NENALEZEN”;
    RETURN;
ENDIF

zamknout archiv pro čtení;
otevřít soubor indexu druhé úrovně s číslem 0;
přečíst první řádek z otevřeného souboru;
zavřít soubor indexu druhé úrovně;
z přečteného řádku zjistit velikost zkomprimované verze;
z přečteného řádku zjistit velikost původní verze;
z přečteného řádku zjistit kompresní metodu;
z přečteného řádku zjistit časové razítko;
otevřít primární datový soubor;

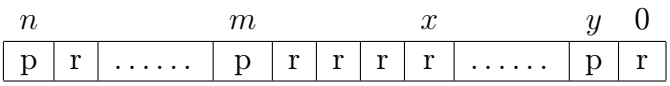
přečíst odpovídající počet bajtů ze začátku souboru;
zavřít primární datový soubor;
dekomprimovat přečtená data příslušnou metodou;
uložit získanou verzi do zadaného souboru;

vrátit časové razítko uložení verze do archivu;
návratový kód := ”OK”;
RETURN;

Z popisu algoritmu lze snadno nahlédnout, že získání nejnovější verze dokumentu z archivu vyžaduje pouze přečtení jednoho záznamu (řádku) souboru indexu druhé úrovně, přečtení známého počtu bajtů ze začátku primárního datového souboru a jejich následnou dekompresi příslušnou metodou. Operace tedy vyhovuje požadavku 3, aby nejnovější verze byla snadno (rychle) přístupná.

3.3.3 Získání libovolné verze

Jednotlivé verze dokumentu s výjimkou nejnovější mohou být uložené buď jako zkomprimovaný původní soubor, nebo jako zkomprimovaný rozdílový soubor od novější verze. Pokud si odmyslíme, že datových souborů může být více, pak jednotlivé verze dokumentu jsou uloženy způsobem, který je ilustrován na obrázku 3.2. Políčka s písmenem p představují verze, které jsou uloženy jako zkomprimovaný celý dokument, a políčka s písmenem r představují verze, které jsou uloženy jako zkomprimovaný rozdíl od předchozí verze.



Obrázek 3.2: Struktura datového souboru, p značí plnou verzi, r verzi uloženou jako rozdíl

Obrázek nám poslouží pro lepší představu o fungování algoritmu. Nechť je požadována verze číslo x. Tato verze je podle obrázku uložena jako rozdíl od předchozí verze, nelze ji proto získat přímou dekompresí. Nejprve je třeba nalézt nejbližší novější (na obrázku nalevo od požadované) verzi, která je uložena jako celý (komprimovaný) dokument. V našem modelovém případě má číslo m. Tuto verzi dekomprimujeme, abychom získali původní dokument. Dále musíme načíst a dekomprimovat verzi dokumentu následující za touto verzí, tedy verzi s číslem $m-1$. Tím získáme rozdíl verze $m-1$ od verze číslo m. Aplikací rozdílu na verzi m získáme původní dokument verze $m-1$. Obdobně se musí načíst a dekomprimovat rozdíl verze $m-2$ od verze číslo $m-1$ a jeho aplikací na dokument verze $m-1$ se získá původní dokument verze číslo $m-2$. Tímto způsobem se postupuje tak dlouho, dokud není získána požadovaná verze dokumentu.

Vstup algoritmu:
identifikátor dokumentu, jehož verze má být získána
číslo požadované verze
cesta k souboru pro uložení získané verze

Výstup algoritmu:
návratový kód
časové razítko uložení požadované verze

Algoritmus:

IF archiv pro daný dokument neexistuje THEN
návratový kód := "NENALEZEN";
RETURN;

```
ENDIF
otevřít soubor indexu 1. úrovně;
načíst první záznam a získat informace o primárním datovém souboru;
IF požadovaná verze > číslo nejnovější verze THEN
    návratový kód := "NENALEZENA";
    RETURN;
ENDIF
```

Nejprve se otevře soubor indexu první úrovně a načte se první záznam – záznam o primárním datovém souboru. Podle něj se zjistí, jestli požadovaná verze je vůbec v archivu uložena. Dále informace slouží ke znalosti počtu verzí uložených v primárním datovém souboru.

```
otevřít soubor indexu 2. úrovně primárního datového souboru;
verze := nejnovější verze;
seznam := NULL;
WHILE verze ≥ požadovaná verze DO
    IF konec souboru indexu 2. úrovně THEN
        načíst další záznam ze souboru indexu 1. úrovně;
        zavřít otevřený soubor indexu 2. úrovně;
        otevřít soubor indexu 2. úrovně, na který se odkazuje
            přečtený záznam;
    ENDIF
    načíst záznam z otevřeného souboru indexu 2. úrovně;
    IF příslušná verze je uložena celá THEN
        seznam := NULL;
    ENDIF
    vložit záznam o verzi na konec seznamu;
DONE
```

Nejbližší celá verze se hledá procházením souborů indexu druhé úrovně počínaje nejnovější verzí. Informace o verzích, jak jdou za sebou, se ukládají do seznamu. Pokud je nalezena verze, která je uložena celá (nikoliv pouze rozdíl), pak je možné celý seznam zrušit a na jeho začátek vložit tuto verzi. Seznam bude tedy vždy obsahovat na svém začátku záznam o celé verzi a dále záznamy o verzích uložených jako rozdíl od předchozí verze. Pomocí tohoto seznamu bude možné provést získání požadované verze stejným způsobem, jaký byl popsán v úvodu.

Teď už jen stačí podle vytvořeného seznamu provést rekonstrukci požadované verze dokumentu:

```
přečíst a odstranit první prvek seznamu s informací o verzi;
otevřít datový soubor obsahující tuto verzi;
načíst data příslušející dané verzi;
aktuální dokument := dekomprimovat dokument;
WHILE seznam není prázdný DO
    přečíst a odstranit první prvek seznamu s informací o další verzi;
    IF další verze je v dalším datovém souboru THEN
        zavřít otevřený datový soubor;
```

```
        otevřít příslušný datový soubor;
    ENDIF
    načíst data odpovídající dané verzi;
    rozdíl := dekomprimovat data;
    aktuální dokument := aplikovat rozdíl na aktuální dokument;
DONE
uložit aktuální dokument do zadaného souboru;
vrátit časové razítko uložení verze do archivu;
návratový kód := "OK";
RETURN;
```

Vyjdeme-li z předpokladu, že procházení souborů indexu druhé úrovně je mnohem rychlejší než čtení dat verzí z datových souborů, jejich dekomprese a aplikace rozdílů, pak můžeme prohlásit, že doba získání verze dokumentu je úměrná spíše počtu verzí, které jsou bezprostředně před ní uloženy jako rozdíl, než jejímu stáří. Z tohoto tvrzení plyne zajímavý paradox: mějme dvě verze x a y téhož dokumentu (viz obrázek 3.2), verze y je mnohem starší než verze x. Dále předpokládejme, že verze y je uložena jako celý dokument, zatímco verze x je uložena jako rozdíl oproti předchozí verzi. Pak doba na získání verze y může být mnohem kratší než doba na získání verze x, i když ta je mnohem novější.

3.4 Výsledky testů

3.4.1 Testovací sada a prostředí

Úložiště má sloužit k ukládání verzovaných html dokumentů. Pro testování bylo tedy třeba získat různé html dokumenty v dostatečném počtu verzí. Spousta dokumentů se mění ve velmi dlouhých intervalech, proto se pro účely těchto testů nejlépe hodily různé zpravodajské servery nebo diskuze vzhledem ke snadné dostupnosti a hlavně častým změnám.

Na navržené úložiště se budeme v dalším textu odvolávat jako na html-repository.

Testy byly prováděny na počítači s konfigurací:

- operační systém: Mandrake Linux 10.2
- procesor: AMD Athlon XP 2200+, frekvence 1,8GHz
- pevný disk: Western Digital, 7 200 otáček/min
- operační paměť: 512 MB

Dále byly nejzásadnější parametry knihovny nastaveny na tyto hodnoty:

- minimální počet verzí v primárním datovém souboru: 3
- maximální počet verzí v primárním datovém souboru: 10
- maximální počet verzí, které lze po sobě uložit jako jejich rozdíl: 20

- nejpomalejší varianta: používá tříúrovňový diff, při hledání nejúčinnější komprese zkouší všechny metody včetně variant předzpracování bez rozlišení znaků při kódování vybraných řetězců
- nejrychlejší varianta: používá pouze jednoúrovňový diff, při hledání nejúčinnější komprese se zkouší pouze samotné metody gzip a bzip2

Tabulka 3.1 uvádí, ze kterých serverů a jak často byly testovací soubory stahovány. Tabulka 3.2 dává přehled o tom, jak velké byly testovací soubory a jak kolísala velikost souborů v rámci jednoho dokumentu. Z minimální a maximální velikosti souborů jednoho verzovaného dokumentu lze usuzovat na míru proměnlivosti jednotlivých verzí, např. pro dokument s id 8.

Id	Adresa	Popis	Stahováno
1	http://neviditelnypes.lidovky.cz	zpravodajský server	1x denně
2	http://www.pdasoft.cz/	články o PDA	1x denně
3	http://www.root.cz/diskuse/	diskuze	1x denně
4	http://www.root.cz/zpravicky/	články	1x denně
5	http://ftipky.cz/vtipy.php?ko=00	zábavné stránky	1x denně
6	http://www.databasejournal.com/	články o DB	1x denně
7	http://www.meteopress.cz/meteoblog/	předpověd počasí	1x denně
8	http://www.mvcr.cz/doprava/index.html	dopravní informace	1x denně
9	http://www.czechcomputer.cz/	internetový obchod	2x denně
10	http://www.zive.cz/default.aspx	články o počítačích	2x denně
11	http://www.ct24.cz/	zpravodajský server	2x denně
12	http://www.lidovky.cz/	zpravodajský server	2x denně
13	http://www.novinky.cz/	zpravodajský server	až 6x denně
14	http://news.bbc.co.uk/2/hi/	zpravodajský server	až 6x denně
15	http://www.novinky.cz/	diskuze ke článku	celkem 50x

Tabulka 3.1: Servery, ze kterých byly stahovány testovací html dokumenty

3.4.2 Prostorové vlastnosti

Nejprve jsme zkoumali, jaké jsou prostorové nároky na uložení archivů vytvořených pomocí vyvíjené knihovny. Pro každý dokument, jehož verze sloužily jako testovací soubory, jsme pomocí knihovny vytvořili samostatný archiv. Jednou pro nejpomalejší (a teoreticky nejúčinnější) variantu, při které se použil tříúrovňový diff a výběr nejúčinnější komprese ze všech používaných metod včetně variant předzpracování bez rozlišení velikosti znaků u kódovaných řetězců, a podruhé pro nejrychlejší variantu, která používá pouze jednoúrovňový diff a nejúčinnější kompresi vybírá pouze z metod gzip a bzip2. Velikosti takto vytvořených archivů jsou uvedené v tabulce 3.3. Sloupec poměr v této i v následujících tabulkách obsahuje kompresní poměr, který se počítá jako $\frac{\text{velikost zkomprimovaného archivu}}{\text{celková velikost původních souborů}}$.

Aby bylo možné dosažené výsledky s něčím porovnat, zkusili jsme pro stejné dokumenty vytvořit archiv pomocí programu tar a tento archiv následně zkomprimovat metodou gzip a bzip2. Účinnost a rychlost metody bzip2 se liší podle velikosti bloku, se kterým metoda pracuje. Pro větší velikost bloku je komprese

Soubory		Velikost v B			
id	počet	celková	prům.	min.	max
1	30	1 141 556	38 052	35 492	39 862
2	30	2 105 421	70 181	69 087	70 953
3	30	1 250 481	41 683	36 741	47 031
4	30	1 334 415	44 481	42 946	45 760
5	30	1 501 460	50 049	48 593	51 256
6	30	2 145 973	71 532	68 386	77 489
7	30	1 129 514	37 650	34 710	40 769
8	30	1 377 587	45 920	22 076	85 771
9	60	2 875 736	47 929	47 055	48 666
10	60	4 042 053	67 367	62 756	69 846
11	60	2 508 360	41 806	40 932	42 925
12	60	2 584 969	43 083	41 524	44 028
13	230	9 856 622	42 855	39 737	45 702
14	231	20 618 045	89 256	85 869	91 028
15	50	2 967 092	59 342	55 974	67 971

Tabulka 3.2: Prostorová charakteristika testovacích souborů

	Nejpomalejší		Nejrychlejší	
Id	vel. v B	poměr	vel. v B	poměr
1	174 773	15,3 %	199 555	17,5 %
2	97 523	4,6 %	137 697	6,5 %
3	50 294	4,0 %	51 385	4,1 %
4	142 370	10,7 %	143 242	10,7 %
5	65 934	4,4 %	116 526	7,8 %
6	178 064	8,3 %	177 463	8,3 %
7	97 728	8,7 %	105 891	9,4 %
8	139 596	10,1 %	122 348	8,9 %
9	217 870	7,6 %	240 581	8,4 %
10	374 256	9,3 %	344 451	8,5 %
11	191 625	7,6 %	203 301	8,1 %
12	274 046	10,6 %	347 163	13,4 %
13	1 154 597	11,7 %	1 196 156	12,1 %
14	797 094	3,7 %	884 446	4,3 %
15	298 381	10,1 %	331 119	11,2 %

Tabulka 3.3: Velikosti zkomprimovaných archivů pomalou a rychlou variantou knihovny html-repository

a také dekomprese pomalejší, ale prostorově účinnější. V tabulce 3.4 jsou výsledky pro metodu bzip2 používající velikost bloku 900 kB (sloupec bzip2 -9), 500 kB (sloupec bzip2 -5) a 100 kB (sloupec bzip2 -1).

	bzip2 -9		bzip2 -5		bzip2 -1	
Id	vel. v B	poměr	vel. v B	poměr	vel. v B	poměr
1	101 488	8,9 %	114 729	10,1 %	203 118	17,8 %
2	89 237	4,2 %	119 759	5,7 %	318 751	15,1 %
3	66 097	5,3 %	81 500	6,5 %	199 204	16,0 %
4	95 754	7,2 %	114 637	8,6 %	234 363	17,6 %
5	50 530	3,4 %	65 031	4,3 %	194 733	13,0 %
6	103 714	4,8 %	131 933	6,1 %	310 525	14,5 %
7	71 916	6,4 %	83 177	7,4 %	167 015	14,8 %
8	74 319	5,4 %	85 792	6,2 %	168 094	12,2 %
9	180 401	6,3 %	217 369	7,6 %	485 189	16,9 %
10	229 743	5,7 %	301 965	7,5 %	740 239	18,3 %
11	130 376	5,2 %	158 519	6,3 %	341 227	13,6 %
12	183 959	7,1 %	226 509	8,8 %	439 061	17,0 %
13	742 779	7,5 %	862 699	8,8 %	1 610 359	16,3 %
14	632 430	3,1 %	867 966	4,2 %	2 388 400	11,6 %
15	152 763	5,1 %	171 503	5,8 %	326 563	11,0 %

Tabulka 3.4: Velikosti zkomprimovaných archivů metodou bzip2 s blokem velikosti 900 kB, 500 kB a 100 kB

Metoda gzip nabízí dvě varianty – pomalou (značení gzip -9) a rychlou (značení gzip -1). Velikosti zkomprimovaných archivů jsou uvedeny v tabulce 3.5.

Srovnáme-li velikosti vytvořených archivů všemi testovanými metodami, zjistíme, že vyvíjená knihovna dosáhla znatelně lepších výsledků než gzip ve všech testovaných případech. Pro metodu bzip2 je situace zcela jiná. Nejrychlejší varianta knihovny byla sice až na jeden případ lepší než nejrychlejší varianta bzip2. Nejpomalejší varianta knihovny byla ve dvou případech lepší než střední varianta bzip2 a v jednom případě dokonce lepší i než nejpomalejší varianta bzip2. Jinak nejpomalejší varianta bzip2 samozřejmě těží z komprese velkého bloku a na větších souborech, kterými je i archiv vytvořený pomocí programu tar, dosahuje mimořádné účinnosti. Je také třeba dodat, že archiv vytvořený programem tar se komprimuje celý najednou, zatímco v archivu vytvářeném knihovnou je každá verze komprimována zvlášť.

Zmínili jsme se, že nejpomalejší varianta použití knihovny by měla být teoreticky také nejúčinnější, neboť vybírá nejúčinnější kompresi ze všech nabízených metod. Dále používá tříúrovňový diff, který produkuje menší rozdílové soubory než pouhý jednoúrovňový (řádkový) diff. Při pohledu do tabulky 3.3 zjistíme, že například pro dokumenty identifikované číslem 8 a 10 to prakticky neplatí. Tento rozpor s předpokladem si vysvětlujeme použitím diffu. Pokud bychom se podívali na velikosti verzí, které byly uloženy jako rozdílový soubor, zjistíme, že odpovídající si verze jsou u pomalejší varianty po zkomprimování větší. Jednoúrovňový diff produkuje pouze řádkové změny, zatímco tříúrovňový diff produkuje i změny slovní a znakové, jsou-li prostorově úspornější. Sám o sobě by tedy tříúrovňový diff neměl vyprodukovat větší výstup než diff řádkový. Předpokládáme, že po jejich

	gzip -9		gzip -1	
Id	vel. v B	poměr	vel. v B	poměr
1	292 322	25,6 %	339 441	29,7 %
2	384 774	18,3 %	454 081	21,6 %
3	320 077	25,6 %	370 208	29,6 %
4	334 996	25,1 %	392 082	29,4 %
5	279 956	18,6 %	343 638	22,9 %
6	376 383	17,5 %	457 436	21,3 %
7	216 648	19,2 %	281 273	24,9 %
8	200 816	14,6 %	251 238	18,2 %
9	713 903	24,8 %	807 813	28,1 %
10	866 878	21,4 %	1 036 645	25,6 %
11	448 921	17,9 %	564 114	22,5 %
12	637 466	24,7 %	757 243	29,3 %
13	2 378 084	24,1 %	2 847 344	28,9 %
14	2 915 501	14,1 %	3 831 351	18,6 %
15	423 052	14,3 %	500 559	16,9 %

Tabulka 3.5: Velikosti zkomprimovaných archivů pomalou a rychlou variantou metody gzip

zkomprimování nastala situace zcela opačná. Každý z výstupů má svou vlastní strukturu, na které budou použité kompresní metody různě účinné. Pomalejší varianta sice vybírala nejúčinnější kompresi výstupu tříúrovňového diffu ze všech metod, ale pokud i ta nejúčinnější byla horší než komprese výstupu řádkového diffu, došlo k nárůstu celkové velikosti archivu. Podle rozdílu ve velikosti oběma variantami vytvářeného archivu lze soudit, že se tak muselo stát opakovaně.

Aby se tomuto nechtěnému jevu předešlo, musela by se nejprve najít nejúčinnější komprese zvlášť pro výstup řádkového, dvouúrovňového i tříúrovňového diffu a teprve z nejúčinnějších kompresí pro všechny tři výstupy vybrat tu opravdu nejlepší. Z pohledu časových nároků by takový krok znamenal zhruba trojnásobné zpomalení, protože konstrukce rozdílu dvou verzí představuje největší položku v celkové době potřebné k uložení nové verze dokumentu a dále by bylo třeba na výstupech všech tří diffů vyzkoušet všechny kompresní metody.

3.4.3 Čas potřebný k uložení nové verze

Abychom zjistili dobu potřebnou pro uložení nové verze, vkládali jsme postupně všechny verze dokumentu od nejstarší po nejnovější do příslušného archivu. U každé operace vložení jsme měřili dobu jejího trvání od odeslání požadavku na její vykonání až po obdržení kladné odpovědi.

Celková doba pro vytvoření jednotlivých archivů postupným vkládáním novějších verzí a průměrná doba připadající na vložení jedné verze jsou uvedeny v tabulce 3.6 jak pro nejpomalejší variantu, tak pro nejrychlejší variantu. Výsledky jsou částečně ovlivněny poměrně malým počtem verzí v archivech. Větší počet verzí může znamenat zpomalení kvůli občasnému překopírování datového souboru, ale vzhledem k dominanci konstrukce rozdílu verzí z pohledu časových nároků, je doba uložení především závislá na velikosti ukládaných souborů.

Id	Počet verzí	Nejpomalejší v ms		Nejrychlejší v ms	
		celkem	průměr	celkem	průměr
1	30	4 934,0	164,5	1 447,6	48,3
2	30	3 762,1	125,4	2 119,2	70,6
3	30	2 050,3	68,3	1 144,9	38,2
4	30	3 808,8	127,0	1 402,6	46,8
5	30	3 397,9	113,3	1 559,8	52,0
6	30	7 000,2	233,3	2 210,2	73,7
7	30	2 488,4	82,9	1 279,2	42,6
8	30	4 892,0	163,1	1 670,5	55,7
9	60	6 966,2	116,1	3 102,9	51,7
10	60	15 216,0	253,6	5 035,4	83,9
11	60	6 666,6	111,1	2 574,6	42,9
12	60	8 121,1	135,4	2 826,5	47,1
13	230	35 657,6	155,0	13 687,6	59,5
14	231	103 192,4	448,7	82 013,7	356,6
15	50	14 984,3	299,7	3 981,4	79,6

Tabulka 3.6: Doba potřebná pro vytvoření archivu

Je také nutné podotknout, že nejčastější velikost html souborů, které se vyskytují na Internetu je zhruba 10–20 kB. Testovací soubory, které jsme používali, měly mnohdy až několikanásobně větší velikost.

Původně jsme chtěli srovnat dobu potřebnou pro vytvoření archivu pomocí knihovny a pomocí archivátoru tar v kombinaci s kompresní metodou bzip2 nebo gzip. Jak se ukázalo při testech, chtěli jsme srovnávat neporovnatelné. Archiv se opět vytvářel postupným vkládáním verzí dokumentu, protože všechny verze samozřejmě nejsou dostupné najednou. Každé takové přidání verze znamená dekomprimovat archiv, přidat verzi a archiv opět komprimovat. U metody bzip2 vyžadovala tato operace pro archiv o velikosti přibližně 30 verzí zhruba 1–2 sekundy, ale pro archiv obsahující asi 200 verzí už operace přidání jedné verze trvala několik desítek sekund.

3.4.4 Čas potřebný na získání uložených verzí

Důležitým kritériem pro fungování knihovny je doba potřebná pro získání uložené verze dokumentu. Čas potřebný na získání uložené verze jsme testovali pro archivy vytvořené pomalejší variantou knihovny. Z pohledu času při získání verzí může toto znamenat navíc pouze čas rekonstrukce dokumentu, byl-li tento před kompresí předzpracován jednou z našich metod.

Pro každý archiv se měřilo celkem 3 000 požadavků na získání uložené verze. Poměr tří nejnovějších verzí ku ostatním verzím činil 1:1, 5:1 a 10:1. Ve skutečnosti mohou nové verze převažovat ještě výraznějším způsobem. Pořadí verzí v požadavcích bylo zcela náhodné.

Průměrná doba pro získání uložené verze spolu se statistickým rozptylem je pro každé testované zastoupení nových a starých verzí uvedena v tabulce 3.7.

Id	Tři nejnovější verze : Ostatní verze					
	1:1 (údaje v ms)		5:1 (údaje v ms)		10:1 (údaje v ms)	
	průměr	rozptyl	průměr	rozptyl	průměr	rozptyl
1	66,0	5,5	31,5	2,5	23,6	1,5
2	53,6	2,9	29,0	1,3	23,5	0,8
3	20,3	0,4	11,1	0,2	9,1	0,1
4	36,6	1,5	17,5	0,7	13,1	0,4
5	25,7	0,6	14,6	0,3	12,0	0,2
6	129,0	17,2	67,3	8,0	54,0	4,9
7	26,2	0,7	15,6	0,3	13,0	0,2
8	101,2	12,9	48,7	6,1	36,5	3,6
9	44,7	1,7	25,4	0,8	20,6	0,5
10	100,7	11,0	50,2	5,2	38,0	3,1
11	48,2	2,4	24,8	1,1	19,1	0,7
12	65,4	4,4	33,7	2,1	26,0	1,2
13	62,8	4,2	31,2	1,8	23,9	1,1
14	115,1	14,0	56,8	6,3	44,9	3,9
15	132,1	21,9	63,8	9,5	46,9	5,6

Tabulka 3.7: Doby získání uložených verzí jednotlivých dokumentů

Z výsledků je patrné, že průměrná doba na získání uložené verze je nižší pro požadavky s vyšším zastoupením požadavků na nové verze, což je plně v souladu s požadavkem 3.

Pokud bychom chtěli porovnat dobu získání verzí u archivu vytvořeného pomocí archivátoru tar spolu s kompresní metodou bzip2 či gzip, musíme si uvědomit, že získání každé uložené verze znamená dekompresi celého archivu. Ta může zabrat až několik sekund.

Kapitola 4

Knihovna html-repository

Tato kapitola představuje programátorskou dokumentaci vytvořené knihovny. Zároveň dává návod na použití knihovny v samostatné aplikaci.

Knihovna je napsána v jazyce C pro platformu Unix. Ke kompresi pomocí metody gzip je používána knihovna zlib [3] verze 1.2.3, komprese metodou bzip2 je zajišťována knihovnou libbzip2 [16] verze 1.0.4. Pro víceúrovňový diff je používána knihovna DiffLib [14] napsána v jazyce C++.

4.1 Struktura souborů

Adresář se zdrojovými soubory knihovny má následující strukturu:

- adresář *bzip2* obsahuje soubory se zdrojovými kódy použité knihovny libbzip2
- adresář *DiffLib* obsahuje potřebné soubory se zdrojovými kódy knihovny DiffLib
- adresář *zlib* obsahuje soubory se zdrojovými kódy použité knihovny zlib
- *client.c* není přímo součástí knihovny, ale obsahuje zdrojový kód aplikace, která sloužila při testování knihovny
- *diff.h*
- *diff3.cc*
- *example.c* není přímo součástí knihovny, ale obsahuje zdrojový kód ukázkové aplikace, která demonstruje použití knihovny
- *methods.c*
- *methods.h*
- *msg.h* není přímo součástí knihovny, ale ukázkové aplikace
- *mz.h*
- *mz1.c*

- *mz2.c*
- *mz_utils.c*
- *mz_utils.h*
- *store.c*
- *store.h*

4.2 Ukázka použití knihovny

Aby bylo možné knihovnu použít v jiném modulu, stačí do příslušného souboru zahrnout hlavičkový soubor *store.h* a při překladu zadat linkeru také zahrnutí knihoven *librepository.a*, *libz.a*, *libbzip2.a* a *libdiff.a*.

Konkrétní programátorské využití knihovny demonstruje soubor *example.c*, který implementuje jednoduchou službu. Služba pomocí fronty zpráv přijímá požadavky na uložení nové verze dokumentu, získání určité verze dokumentu nebo získání informace o čísle nejnovější verze dokumentu. Odpověď na konkrétní požadavek s výsledkem operace odesílá opět pomocí fronty zpráv.

Struktura používaná pro zasílané zprávy je definovaná v souboru *msg.h*, stejně jako konstanty pro rozlišení požadovaných operací.

Po spuštění programu se nejprve volá funkce `parse_cmdline()`, které se jako argumenty předají počet a pole argumentů tohoto programu. Funkce zpracuje přepínače a argumenty a nastaví příslušné hodnoty do globální struktury `args`, která určuje chování knihovny.

Dále se volá funkce `init()`, která zajistí alokaci globálních bufferů a ostatních struktur. Tyto zdroje jsou alokovány předem pro všechny soubory do zadané velikosti, aby z důvodu časové efektivity nebylo nutné tyto zdroje alokovat znovu pro každý požadavek na uložení či přístup k dokumentu.

Následně služba běží až do zaslání signálu ukončení. Každý požadavek se provádí v novém podprocesu. Když totiž dojde k neplánovanému ukončení procesu, neznamená to ukončení celé služby. Dále se automaticky uvolní všechny vytvořené zámky a tím nezůstane archiv blokován pro další požadavky. Archiv se sice může nacházet v nekonzistentním stavu, ale jeho nápravu zajistí první proces, který bude s daným archivem pracovat. V neposlední řadě dojde i při neplánovaném ukončení procesu k uvolnění procesem alokované paměti.

Při požadavku na uložení nové verze dokumentu je volána v podprocesu funkce `store_new_revision()`, při požadavku na získání určité verze dokumentu je podprocesem volána funkce `get_revision()` a při požadavku na získání čísla nejnovější verze dokumentu volá podproces funkci `last_revision_nr()`.

Ukázka služby vždy spouští pouze jeden podproces. Tato ukázková aplikace byla také použita při testování časových a prostorových vlastností vytvořené knihovny.

4.3 diff.h

Hlavičkový soubor *diff.h* obsahuje deklarace dvou funkcí, které tvoří rozhraní pro použití knihovny DiffLib.

4.4 diff3.cc

Tento modul je napsán v jazyce C++ a tvoří rozhraní pro použití knihovny DiffLib, která je rovněž celá napsaná v C++ a nabízí použití víceúrovňového diffu. Pokud by byla třeba použít jiná knihovna či modul pro diff, stačí upravit obě funkce, které modul obsahuje.

diff()

```
int diff(
  char *a,
  size_t size_a,
  char *b,
  size_t size_b,
  char **diff_str,
  size_t *size_diff,
  int level
)
```

Funkce `diff()` konstruuje rozdíl zadaného řetězce `a` délky `size_a` a řetězce `b` délky `size_b`. K tomu využívá metody `diff()` a `fillDiffTree()` třídy `MemLineDiff` a metodu `saveToString` třídy `DiffTree` z knihovny DiffLib. Podle parametru `level` se konstruuje diff řádkový (`level`=1), slovní (`level`=2) nebo znakový (`level`=3).

Výsledek se ukládá do řetězce, na jehož ukazatel směřuje parametr `diff_str`. Hodnota parametru `size_diff` určuje velikost paměti alokované pro `diff_str`. Pokud by výsledný rozdíl byl větší než tato paměť, je uvolněna a alokována nová paměť požadovné velikosti. V každém případě je parametru `diff_size` nastavena hodnota odpovídající velikosti rozdílu.

Pokud funkce proběhla úspěšně, vrací hodnotu 0, vyskytla-li se chyba, vrací hodnotu -1.

merge()

```
merge(
  char *a,
  size_t size_a,
  char *diff_str,
  size_t diff_size,
  char *b,
  size_t size_b
)
```

Funkce aplikací rozdílu řetězců zadaného pomocí řetězce `diff_str` délky `diff_size` na řetězec `a` délky `size_a` rekonstruuje původní řetězec, který uloží do `b` délky `size_b`. K tomu používá metody `loadFromString()` a `apply()` třídy `DiffTree` z knihovny DiffLib. Délka řetězce `b` zadaná parametrem `size_b` musí být shodná s délkou původního řetězce.

Pokud funkce proběhla úspěšně, vrací hodnotu 0, v případě chyby vrací -1.

4.5 methods.c

Soubor *methods.c* tvoří modul, který zajišťuje kompresi a dekompresi souborů. Tvoří rozhraní pro použití knihoven pro kompresi zlib a libbzip2 a modulů pro použití metody 1 a metody 2 předzpracování. Pokud by se přidávaly další kompresní metody, je tento modul místem, kde by se tak mělo učinit.

Modul definuje dvě funkce, které se dají použít po zahrnutí souboru *methods.h* a příslušných knihoven (pro kompresi metodou gzip a bzip2).

best_compression()

```
int best_compression(
  char *input,
  size_t input_size,
  char *output,
  size_t *output_size,
  int *method
)
```

Funkce `best_decompression()` slouží k nalezení nejúčinnější komprese řetězce `input` délky `input_size`. Zkomprimovaný řetězec ukládá do `output`, jehož délka je zadána parametrem `output_size`. Po uložení zkomprimovaného řetězce je hodnota `output_size` nastavena na velikost zkomprimovaného řetězce. Číslo nejlepší nalezené metody ukládá do proměnné zadané ukazatelem `method`. Konstanty používané pro identifikaci metod jsou definovány v souboru *methods.h*.

Na začátku je nejlépe zkomprimovaný řetězec původní řetězec bez komprese. Funkce postupně zkouší kompresi pomocí gzip a bzip2 (vždy), předzpracování metodami 1 a 2 ve variantách rozlišujících i ignorujících velikost znaků. Které metody kromě gzip a bzip2 se použijí určuje globální struktura `args` typu `cmdline_t`, který je definovaný v souboru *methods.h*. Pokud je řetězec zkomprimován nějakou metodou lépe než byl dosud (zkomprimovaný řetězec má menší velikost), bude tato metoda aktuálně nejlepší. Která metoda je nejlepší na konci, ta bude použita.

Pro kompresi metodou gzip používá funkci `compress()` z knihovny zlib, pro kompresi metodou bzip2 funkci `BZ2_bzBuffToBuffCompress()` z knihovny libbzip2. Dále pro předzpracování používá funkce `mz1_process()` resp. `mz2_process()` z modulu *mz1.c* resp. *mz2.c*.

Pro ukládání výstupů jednotlivých kompresí se používají externí proměnné definované v modulu *store.c*.

V případě úspěchu vrací funkce hodnotu 0. Pokud paměť pro zkomprimovaný řetězec nemá potřebnou velikost, vrací funkce hodnotu -1.

decompression()

```
decompression(
  char *compressed,
  size_t compressed_size,
  char *output,
  size_t *output_size,
```

```
  int method
)
```

Funkce dekomprimuje řetězec zadaný pomocí parametru `compressed` délky `compressed_size` metodou s číslem `method`. Dekomprimovaný řetězec ukládá do `output`, jejíž velikost je zadána pomocí `output_size`. Velikost proměnné `output` musí být dostatečná. Hodnota proměnné `output_size` je na konci nastavena na délku dekomprimovaného řetězce. Číslo metody, kterou byl vstupní řetězec komprimován, nabývá hodnot definovaných v souboru *methods.h*.

Pro dekompresi řetězců komprimovaných metodu gzip resp. bzip2 jsou použity funkce `uncompress()` resp. `BZ2_bzBuffToBuffDecompress()` z knihovny zlib resp. libbzip2. Pro zpětnou rekonstrukci předzpracování se používají funkce `mz1_reconstruct()` resp. `mz2_reconstruct()` z modulu *mz1.c* resp. *mz2.c*.

Pro ukládání výstupů jednotlivých kompresí se používají externí proměnné definované v modulu *store.c*.

V případě úspěšného provedení vrací hodnotu 0, v případě chyby během dekomprese vrací hodnotu<0 odpovídající příslušnému chybovému kódu dekompresní metody.

4.6 methods.h

Tento hlavičkový soubor obsahuje deklarace funkcí modulu *methods.c*, které tvoří jeho rozhraní. Dále obsahuje definice identifikátorů jednotlivých používaných metod a definice dvou struktur.

4.6.1 Definice konstant

Soubor obsahuje definice konstant, které slouží jako identifikátory jednotlivých kompresních metod:

ID_NONE pro původní řetězec bez komprese

ID_GZIP pro kompresi pomocí gzip

ID_BZIP2 pro kompresi pomocí bzip2

ID_MGZ1 pro kompresi pomocí gzip se vstupem předzpracovaným metodou 1 se statickým slovníkem s rozlišením velikosti znaků kódovaných řetězců

ID_MBZ1 pro kompresi pomocí bzip2 se vstupem předzpracovaným metodou 2 se statickým slovníkem s rozlišením velikosti znaků kódovaných řetězců

ID_MGZ1I pro kompresi pomocí gzip se vstupem předzpracovaným metodou 1 se statickým slovníkem bez rozlišení velikosti znaků kódovaných řetězců

ID_MBZ1I pro kompresi pomocí bzip2 se vstupem předzpracovaným metodou 1 se statickým slovníkem bez rozlišení velikosti znaků kódovaných řetězců

ID_MGZ2 pro kompresi pomocí gzip se vstupem předzpracovaným metodou 2 se statickým slovníkem s rozlišením velikosti znaků kódovaných řetězců

ID_MBZ2 pro kompresi pomocí bzip2 se vstupem předzpracovaným metodou 2 se statickým slovníkem s rozlišením velikosti znaků kódovaných řetězců

ID_MGZ2I pro kompresi pomocí gzip se vstupem předzpracovaným metodou 2 se statickým slovníkem bez rozlišení velikosti znaků kódovaných řetězců

ID_MBZ2I pro kompresi pomocí bzip2 se vstupem předzpracovaným metodou 2 se statickým slovníkem bez rozlišení velikosti znaků kódovaných řetězců

4.6.2 Definice datových typů

struct arg_t

Struktura se používá pro uchování informací o řetězci, který je třeba zkomprimovat, nebo naopak dekomprimovat.

```
typedef struct arg_t{
  char *input;
  size_t input_size;
  char *output;
  size_t output_size;
  int method;
};
```

Význam jednotlivých položek struktury:

input ukazatel na řetězec, který má být komprimován, nebo naopak ukazatel na buffer, do kterého má být zkomprimovaný řetězec dekomprimován

input_size velikost původního řetězce resp. velikost bufferu pro dekompresi

output ukazatel na řetězec, který má být dekomprimován, nebo naopak ukazatel na buffer, do kterého má být původní řetězec zkomprimován

output_size velikost zkomprimovaného řetězce resp. velikost bufferu pro kompresi

method číslo metody, kterou má být řetězec (de)komprimován

struct cmdline_t

Struktura se používá pro globální proměnnou uchovávající parametry knihovny, které ovlivňují její činnost a vlastnosti.

```
typedef struct cmdline_t{
  int init_min_revision;
  int init_max_revision;
  int init_max_diff;
  int init_store_same_revision;
  int init_diff_level;
  int init_case_insensitive;
  int init_try_mz1;
  int init_try_mz2;
```

```
  size_t init_max_archive_size;
  size_t init_sz_row;
  size_t init_sz_file_id;
  size_t init_sz_path;
  size_t init_sz_buff;
  size_t init_sz_index_block;
  size_t init_sz_preallocated;
  const char *init_dir_tmp;
  const char *init_dir_index_l1;
  const char *init_dir_index_l2;
  const char *init_dir_data;
  const char *init_dir_locks;
  const char *init_dir_consist;
};
```

Význam jednotlivých položek struktury:

init_min_revision minimální počet verzí, které má obsahovat primární datový soubor

init_max_revision maximální počet verzí, které může primární datový soubor obsahovat

init_max_diff nejvyšší počet verzí dokumentů, které mohou být za sebou uloženy jako rozdíl od následující verze

init_store_same_revision příznak, zda ukládat verzi, pokud je shodná s předchozí (=1) či nikoliv (=0)

init_diff_level příznak, zda použít řádkový diff (=1), slovní diff (=2), nebo znakový diff (=3)

init_case_insensitive příznak, zda při použití metod předzpracování vstupu pro kompresi použít rovněž variantu nerozlišující velikost znaků kódovaných řetězců

init_try_mz1 příznak, zda při hledání nejlepší komprese zkusit (=1) také předzpracování vstupu pomocí metody 1 se statickým slovníkem

init_try_mz2 příznak, zda při hledání nejlepší komprese zkusit (=1) také předzpracování vstupu pomocí metody 2 se statickým slovníkem

init_max_archive_size maximální velikost datového souboru vyjma primárního datového souboru

init_sz_row velikost proměnné pro zpracování jednoho řádku ze souborů indexu

init_sz_file_id maximální velikost identifikátoru souboru

init_sz_path maximální velikost cesty k souborům

init_sz_buff maximální velikost bufferu, pomocí kterého se kopírují datové soubory při vytváření nových

init_sz_index_block udává velikost bloků, po kterých se čtou soubory indexů

init_sz_preallocated udává maximální velikost souborů, pro které jsou předalokovány všechny buffery používané při kompresi a dekompresi

init_dir_tmp ukazatel na řetězec určující cestu k adresáři *tmp*

init_dir_index_l1 ukazatel na řetězec určující cestu k adresáři *index1*

init_dir_index_l2 ukazatel na řetězec určující cestu k adresáři *index2*

init_dir_data ukazatel na řetězec určující cestu k adresáři *data*

init_dir_locks ukazatel na řetězec určující cestu k adresáři *locks*

init_dir_consist ukazatel na řetězec určující cestu k adresáři *consist*

4.7 mz.h

Hlavičkový soubor obsahující definice konstant, které určují délky bitových map, které se používají v modulech *mz1.c* a *mz2.c*. Dále obsahuje definice chybových kódů pro stejné moduly, definici struktury a deklarace funkcí, které tvoří rozhraní pro použití těchto modulů.

4.7.1 Definice konstant

Soubor definuje několik konstant s tímto významem:

MZ_MAP_LEN určuje délku bitové mapy, kterou používají metody předzpracování rozlišující velikost znaků kódovaných řetězců

MZI_MAP_LEN určuje délku bitové mapy, kterou používají metody předzpracování nerozlišující velikost znaků kódovaných řetězců

MAP_LEN_LOWER_CHARS určuje délku bitové mapy, odpovídající dolní polovině znaků

Dále definuje konstanty, které slouží jako návratové kódy pro volání funkcí modulů *mz1.c* a *mz2.c*:

MZ_OK úspěšné provedení

MZ_ERROR výskyt blíže nespecifikované chyby

MZ_TOKENS počet volných znaků je menší než počet slov ze slovníku, které je třeba těmito znaky kódovat

MZ_MEM malá paměť pro kompresi resp. dekompresi

4.7.2 Definice datových typů

struct node

Struktura se používá jako uzel v binarizované TRIE, která se používá v modulech *mz1.c* a *mz2.c* při předpracování a zpětné rekonstrukci.

```
typedef struct node{
  unsigned char chr;
  int code;
  struct node *alternative, *next;
};
```

Význam jednotlivých položek struktury:

chr znak, který je v uzlu

code kód značící koncový stav (≥ 0), jinak (-1)

alternative ukazatel na bezprostředního sourozence

next ukazatel na prvního syna

4.8 mz_utils.c

Soubor obsahuje společné funkce, které se používají v modulech *mz1.c* a *mz2.c*.

is_equal()

```
is_equal(
  unsigned char a,
  unsigned char b,
  int ci
)
```

Funkce porovná znaky zadané parametry `a` a `b`. Pokud jsou si rovny, vrací 1, jinak 0. Pokud je parametr `ci`> 0, pak jsou znaky porovnány bez ohledu na jejich velikost, tedy a=A bude bráno jako rovnost.

is_less()

```
is_less(
  unsigned char a,
  unsigned char b,
  int ci
)
```

Funkce porovná znaky zadané parametry `a` a `b`. Pokud je číslo znaku `a` menší než číslo znaku `b`, vrací funkce 1, jinak 0. Pokud je parametr `ci`>0, pak jsou znaky porovnány bez ohledu na jejich velikost, tedy například A<b bude vyhodnoceno jako pravda.

4.9 mz_utils.h

Tento hlavičkový soubor obsahuje deklarace funkcí z modulu *mz_utils.c*, které umožňují jejich použití v modulech *mz1.c* a *mz2.c*.

4.10 mz1.c a mz2.c

Tyto soubory obsahují funkce, které zajišťují předzpracování ukládaných dokumentů metodou 1 a 2 se statickým slovníkem. Názvy funkcí, jejich parametrů a význam parametrů jsou pro oba soubory shodné. Jednotlivé názvy funkcí se liší pouze prefixem mz1_ resp. mz2_ pro příslušný soubor. Popíšeme tedy funkce obou modulů najednou.

4.10.1 Funkce rozhraní

mz?_process()

```
int mz?_process(
  void *src_addr,
  size_t src_len,
  void *des_addr,
  size_t *des_len,
  int ci
)
```

Funkce provede předzpracování vstupu příslušnou metodou. Vstup je dán ukazatelem `src_addr` na začátek paměti, kde je uložen, a délkou vstupu `src_len`. Předzpracovaný vstup bude uložen do paměti, jejíž začátek je dán ukazatelem `des_addr` a jejíž délka je dána parametrem `des_len`. Po úspěšném předzpracování je hodnota `des_len` nastavena na skutečnou délku výstupu. Parametr `ci` udává, zda má metoda brát zřetel na velikost znaků kódovaných řetězců (`ci`=0), či nikoli (`ci`=1).

Pokud paměť určená pro výstup není dostatečně velká, vrací funkce `MZ_MEM`. Pokud nelze předzpracování provést z důvodu malého počtu volných znaků, vrací funkce `MZ_TOKENS`. Při výskytu jiné chyby vrací funkce `MZ_ERROR`. Při úspěšném pro vedení je návratový kód funkce `MZ_OK`.

mz?_reconstruct()

```
int mz?_reconstruct(
  void *src_addr,
  size_t src_len,
  void *des_addr,
  size_t *des_len,
  int ci
)
```

Funkce provede zpětnou rekonstrukci původního řetězce, který byl předzpracován příslušnou metodou. Zda byl původní řetězec předzpracován metodou nerozlišující velikost znaků kódovaných řetězců a zda má být podle toho také zpětně

rekonstruován, určuje parametr `ci`. Předzpracovaný řetězec je zadán ukazatelem `src_addr` na začátek paměti a délkou `src_len` této paměti. Rekonstruovaný řetězec bude uložen do paměti určené ukazatelem na její začátek `des_addr`, která má délku `des_len`. Hodnota `des_len` bude po provedení funkce nastavena na skutečnou délku rekonstruovaného řetězce.

Pokud by paměť na rekonstruovaný řetězec nestačila, vrací funkce `MZ_MEM`, nepodaří-li se rekonstruovat mapování znaků a slov ze slovníku, vrací `MZ_TOKENS`, v případě jiné chyby `MZ_ERROR`. Při úspěšném provedení vrací `MZ_OK`.

4.10.2 Pomocné funkce

mz*_build_tree()

```
int mz*_build_tree(
  int count,
  const char *exprs[],
  int ci
)
```

Tato funkce postaví strukturu TRIE pro `count` slov, která jsou zadána polem řetězců `exprs`. Parametr `ci` určuje, zda metoda bude brát zřetel na velikost znaků vkládaných řetězců (`ci`=0) či nikoli (`ci`=1). První úroveň TRIE je tvořena globálním polem `roots`.

V případě úspěchu vrací `MZ_OK`, při chybě `MZ_ERROR`.

mz*_insert_string()

```
int mz*_insert_string(
  const char *expr,
  int code,
  int ci
)
```

Funkce vloží do TRIE, jejíž první úroveň je tvořena globálním polem `roots` resp. `mz2_roots`, reprezentaci řetězce daného proměnnou `expr`. Poslední znak řetězce bude koncovým stavem a bude mu přiřazen kód `code`. Parametr `ci` určuje, zda metoda bude brát zřetel na velikost znaků kódovaných řetězců (`ci`=0) či nikoli (`ci`=1).

Pokud funkce proběhne úspěšně, vrací `MZ_OK`, v případě výskytu chyby vrací `MZ_ERROR`.

mz*_process_step()

```
int mz*_process_step(
  void *src_addr,
  size_t src_len,
  void *des_addr,
  size_t *des_len,
  int step,
  int ci
)
```

Vstupem této funkce je řetězec, který je zadán ukazatelem `src_addr` na začátek paměti, ve které je uložen. Jeho délka je dána parametrem `src_len`.

Pokud je `step`=1, funkce zjistí, která slova ze slovníku se vyskytují v zadaném řetězci, které znaky nebyly použity a zda jich bude dost na zakódování nalezených slov. Pokud jich není dost, vrací `MZ_TOKENS`. Pokud jich je dost, vytvoří mapování nalezených slov na volné znaky do globálního pole `map`.

Pokud je `step`=2, pak funkce ve vstupním řetězci nahradí nalezená slova přiřazenými znaky. Výsledek uloží do paměti dané ukazatelem `des_addr` na její začátek. Její délka je dána parametrem `des_len`. Pokud paměť pro výstup nestačí, vrací funkce `MZ_MEM`. Hodnota `des_len` je nastavena na skutečnou délku předzpracovaného vstupu.

Parametr `ci` určuje, zda metoda bude brát zřetel na velikost znaků kódovaných řetězců (`ci`=0) či nikoli (`ci`=1).

Při úspěšném průběhu vrací funkce `MZ_OK`, při jiné chybě než výše zmíněných vrací `MZ_ERROR`.

mz*_step_in_tree()

```
node* mz*_step_in_tree(
  node *p,
  unsigned char chr,
  int ci
)
```

Funkce vrací ukazatel na uzel v TRIE, do kterého se má pokračovat z uzlu `p` na základě znaku `chr`. Pokud není kam přejít, vrací funkce prázdný ukazatel NULL. Parametr `ci` určuje, zda má brát zřetel (`ci`=0) na velikost znaku `chr` a znaků v uzlech TRIE, či nikoli (`ci`=1).

4.11 store.c

Soubor obsahuje funkce, které zajišťují ukládání dokumentů. Některé z funkcí tvoří rozhraní pro použití knihovny html-repository.

4.11.1 Pomocné funkce

file_close()

```
int file_close(file_t *f)
```

Funkce zavírá soubor, který byl spojen se strukturou danou ukazatelem `f`. Pokud byl soubor otevřen pro zápis, zapisuje před uzavřením do souboru zbytek bufferu ze struktury dané ukazatelem `f`.

V případě úspěšného provedení vrací kód `STR_OK`, pokud se nepodařilo zapsat zbytek bufferu nebo se nepodařilo zavřít soubor, vrací kód `STR_ERROR`.

file_exists()

```
int file_exists(char *file)
```

Funkce zjišťuje, zda existuje soubor zadaný jménem `file`. Funkce se pokusí soubor zadaného jména otevřít pro čtení. Pokud toto systémové volání končí chybou, ověří se, zda chyba vrácená voláním odpovídá neexistenci souboru. Odpovídá-li vrácená chyba, vrací funkce hodnotu 0 (soubor neexistuje), pokud se vyskytla jiná chyba, vrací funkce kód `STR_ERROR`. Jestliže se podařilo soubor otevřít, pak existuje a funkce vrací hodnotu 1.

file_open()

```
int file_open(file_t *f)
```

Funkce otevře soubor spojený se strukturou danou ukazatelem `f`. Podle položky `rw` této struktury otevře soubor buď pouze pro čtení (`rw`=0), nebo pouze pro zápis (`rw`=1), přičemž soubor vytvoří, pokud ještě neexistuje (pouze při variantě pro zápis). Po otevření je položka `used` struktury dané ukazatelem `f` nastavena na hodnotu 1 (soubor byl použit).

Pokud funkce proběhla úspěšně, vrací kód `STR_OK`, jinak `STR_ERROR`.

file_readln()

```
ssize_t file_readln(
  char *row,
  size_t max,
  file_t *f
)
```

Funkce ze souboru spojeného se strukturou danou ukazatelem `f` načte jeden řádek a uloží do proměnné dané ukazatelem `row`. Nečte se přímo ze souboru, ale z bufferu struktury, do kterého se soubor načítá po blocích. Načtený řetězec obsahuje i znak konce řádku a je ukončen znakem `'\0'`. Maximální délka řádku je dána parametrem `max`.

Pokud od aktuální pozice v bufferu struktury do jeho konce není nalezen konec řádku, načte se do bufferu následující blok souboru.

V případě úspěchu vrací funkce počet přečtených znaků včetně znaku `'\0'`. Pokud došlo k chybě, vrací funkce kód `STR_ERROR`.

file_writeln()

```
ssize_t file_writeln(
  char *row,
  file_t *f
)
```

Funkce zapíše do souboru spojeného se strukturou danou ukazatelem `f` řádek daný řetězcem `row`. Nezapisuje se přímo do souboru, ale do bufferu struktury. Po naplnění bufferu se zapisuje celý najednou na konec souboru.

Vrací počet zapsaných znaků včetně znaku konce řádky, nastane-li chyba, vrací `STR_ERROR`.

parse_row()

```
int parse_row(
  char *row,
  const char *format,
  int argc,
  ...
)
```

Funkce rozdělí řetězec zadaný parametrem `row` ve formátu CSV podle oddělovačů (čárek) na `argc` hodnot. Hodnoty jsou dané formátovacím řetězcem `format` (`i` – hodnota typu `integer`, `l` – hodnota typu `long`, `s` – hodnota typu `char*`). Další parametry jsou ukazatelé na proměnné, do kterých mají být získané hodnoty uloženy.

Pokud řetězec zadaný parametrem `row` není ve formátu CSV, vrací funkce `STR_ERROR_CSV`, v případě výskytu jiné chyby vrací `STR_ERROR`. Při úspěšném provedení vrací `STR_OK`.

repair_consistency()

```
int repair_consistency(char *id)
```

Funkce překopíruje z adresáře pro dočasné soubory (daného položkou `init_dir_consist` globální struktury `args`) všechny soubory, které patří do archivu dokumentu s identifikátorem `id`, do příslušných adresářů. Příslušné adresáře se rozliší podle přípon.

Po překopírování všech souborů smaže soubor pro kontrolu konzistence. Tuto funkci mohou volat pouze procesy, které právě drží zámek (ať na čtení či zápis) na archiv příslušného dokumentu, jinak hrozí nekonzistence archivu.

Při úspěšném provedení vrací `STR_OK`, při chybě `STR_ERROR`.

set_buffers()

```
int set_buffers(size_t size)
```

Funkce přealokuje všechny předalokované globální buffery používané pro kompresi a dekompresi dokumentů pro soubor velikosti `size`.

Při úspěchu vrací `STR_OK`, při chybě `STR_ERROR`.

4.11.2 Funkce rozhraní

get_revision()

```
int get_revision(
  char *id,
  int rev_nr,
  char *fout,
  long *ts
)
```

Funkce do souboru zadaného cestou ve `fout` vrátí verzi číslo `rev_nr` dokumentu s identifikátorem `id`. Do proměnné dané ukazatelem `ts` nastaví časové razítko uložení požadované verze v sekundách od počátku unixové éry (1. 1. 1970).

Je-li zadáno číslo požadované verze -1, vrací funkce nejnovější verzi dokumentu. Pro čísla verzí ≥ 0 vrací verzi dokumentu daného čísla (nejstarší verze má číslo 0).

Na začátku zamyká příslušný archiv zámkem pro čtení. Pokud zjistí existenci souboru pro kontrolu konzistence tohoto archivu (archiv je nekonzistentní), volá nejprve funkci `repair_consistency()`.

Jestliže dokument se zadaným identifikátorem neexistuje, vrací tato funkce `STR_ERROR_NOT_FOUND`. Pokud požadovaná verze neexistuje, vrací funkce `STR_ERROR`, stejně jako v případě výskytu jiné chyby. Pokud funkce proběhne úspěšně, vrací `STR_OK`. Jestliže by některý použitý řádek ze souborů indexu nebyl ve správném formátu CSV, pak funkce vrátí `STR_ERROR_CSV`. Tato situace by ale za normálních okolností neměla nastat.

init()

```
int init(void)
```

Funkce inicializuje všechny globální proměnné (alokuje buffery používané při kompresi, globální struktury, proměnné pro názvy souborů, ...). Velikosti bufferů jsou dány globální strukturou `args`.

Při úspěšném provedení vrací `STR_OK`, při chybě `STR_ERROR`.

last_revision_nr()

```
int last_revision_nr(
  char *id,
  int *rev
)
```

Funkce do proměnné dané ukazatelem `rev` nastaví číslo nejnovější verze dokumentu s identifikátorem `id`.

Na začátku zamyká příslušný archiv zámkem pro čtení. Pokud zjistí existenci souboru pro kontrolu konzistence tohoto archivu (archiv je nekonzistentní), volá nejprve funkci `repair_consistency()`.

Při úspěšném provedení vrací `STR_OK`, v případě chyby vrací `STR_ERROR`. Pokud by některý ze čtených řádků souborů indexu nebyl ve správném formátu CSV, vrací `STR_ERROR_CSV`. Za normálních okolností by ale k této situaci nemělo dojít.

parse_cmdline()

```
int parse_cmdline(
  int argc,
  char *argv[]
)
```

Funkce slouží ke zpracování parametrů, které jsou dány polem `argv`, které má `argc` prvků. Globální struktura `args` je naplněna buď implicitními hodnotami podle konstant definovaných v souboru *store.h*, nebo podle zpracovaných přepínačů, jestliže byly správně zadány. Funkce rozlišuje tyto přepínače:

`-a`*hodnota*
: Nastaví maximální velikost datového souboru na *hodnota*.
Ovlivňuje položku `init_max_archive_size`.
Omezení: *hodnota* > 0.

`-b`*hodnota*
: Nastaví velikost bufferu pro kopírování souborů na *hodnota*.
Ovlivňuje položku `init_sz_buff`.
Omezení: *hodnota* > 0.

`-c`*cesta*
: Určuje cestu k adresáři *consist*.
Ovlivňuje položku `init_dir_consist`.
Omezení: `init_sz_file_id`+`strlen(init_dir_consist))`<`init_sz_path`.

`-d`*cesta*
: Určuje cestu k adresáři *data*.
Ovlivňuje položku `init_dir_data`.
Omezení:
`init_sz_file_id`+`strlen(init_dir_data)`+`SUFFIX_LEN)`<`init_sz_path`.

`-e`*hodnota*
: Buffery budou předalokávány pro soubory do velikosti *hodnota*.
Ovlivňuje položku `init_sz_preallocated`.
Omezení: *hodnota* > 0.

`-f`*hodnota*
: Maximální počet verzí uložených za sebou jako rozdíl je *hodnota*.
Ovlivňuje položku `init_max_diff`.
Omezení: *hodnota* > 0.

`-i`
: Určuje, že se použijí i předzpracování bez rozlišení velikosti znaků řetězců ze slovníku.
Ovlivňuje položku `init_case_insensitive`.

`-k`*hodnota*
: Velikost bloku pro čtení souborů indexu je *hodnota*.
Ovlivňuje položku `init_sz_index_block`.
Omezení: *hodnota* > 0.

`-l`*hodnota*
: Určuje úroveň konstruovaného diffu.
Ovlivňuje položku `init_diff_level`.
Omezení: 1 (řádkový), 2 (slovní), 3 (znakový).

`-n`*hodnota*
: Nejmenší počet verzí v primárním datovém souboru je *hodnota*.
Ovlivňuje položku `init_min_revision`.
Omezení: *hodnota* ≥ 1.

`-o`*cesta*
: Určuje cestu k adresáři *locks*.
Ovlivňuje položku `init_dir_locks`.

Omezení: `init_sz_file_id+strlen(init_dir_locks))<init_sz_path`.

`-p`*hodnota*

Maximální délka cesty ke všem používaným souborům je *hodnota*.
Ovlivňuje položku `init_sz_path`.
Omezení: *hodnota* > 0.

`-r`*hodnota*

Velikost bufferu pro zpracování jedné řádky indexu je *hodnota*.
Ovlivňuje položku `init_sz_row`.
Omezení: *hodnota* > 0.

`-s`

Při použití se budou ukládat i stejné nejnovější verze.
Ovlivňuje položku `init_store_same_revision`.

`-t`*cesta*

Určuje cestu k adresáři *tmp*.
Ovlivňuje položku `init_dir_tmp`.
Omezení:
`init_sz_file_id+strlen(init_dir_tmp)+SUFFIX_LEN)<init_sz_path`.

`-x`*hodnota*

Nejvyšší počet verzí v primárním datovém souboru je *hodnota*.
Ovlivňuje položku `init_max_revision`.
Omezení: *hodnota* $>$ `init_min_revision`.

`-1`*cesta*

Určuje cestu k adresáři *index1*.
Ovlivňuje položku `init_dir_index1`.
Omezení:
`init_sz_file_id+strlen(init_dir_index1)+SUFFIX_LEN)`
`<init_sz_path`.

`-2`*cesta*

Určuje cestu k adresáři *index2*.
Ovlivňuje položku `init_dir_index2`.
Omezení:
`init_sz_file_id+strlen(init_dir_index2)+SUFFIX_LEN)`
`<init_sz_path`.

Za přepínači mohou ještě následovat dva argumenty, které mohou mít hodnoty s významem:

mz1 při hledání nejlepší komprese bude vyzkoušeno i předzpracování vstupu metodou 1 se statickým slovníkem, položka `init_try_mz1` globální struktury `args` bude nastavena na hodnotu 1

mz2 při hledání nejlepší komprese bude vyzkoušeno i předzpracování vstupu metodou 2 se statickým slovníkem, položka `init_try_mz2` globální struktury `args` bude nastavena na hodnotu 1

Pokud funkce proběhne úspěšně, vrací `STR_OK`, pokud nejsou splněna omezení na přepínače týkající se cest k adresářům nebo pokud jsou špatně argumenty, vrací `STR_ERROR_PARAMS`.

store_new_revision()

```
int store_new_revision(
  char *id,
  char *path,
  int *rev,
  long *time_stamp
)
```

Funkce uloží novou verzi dokumentu s identifikátorem `id`, která je uložena v souboru daném cestou k němu parametrem `path`. Do proměnné dané ukazatelem `rev` nastaví číslo ukládané verze, do proměnné dané ukazatelem `time_stamp` uloží časové razítko reprezentované počtem sekund od počátku unixové éry (1. 1. 1970) uložení této verze dokumentu.

Na začátku zamyká příslušný archiv zámkem pro zápis. Pokud zjistí existenci souboru pro kontrolu konzistence tohoto archivu (archiv je nekonzistentní), volá nejprve funkci `repair_consistency()`.

V případě úspěšného provedení vrací `STR_OK`, v případě chyby vrací funkce `STR_ERROR`. Dále může vracet `STR_ERROR_CSV`, pokud by některý přečtený řádek ze souborů indexu nebyl ve správném formátu CSV. Tato situaci by ale za normálních okolností neměla nastat.

4.12 store.h

Tento hlavičkový soubor obsahuje definice datových typů používaných modulem *store.c*, dále konstanty, návratové kódy a makra používaná tímto modulem a v neposlední řadě také deklarace funkcí, které tvoří rozhraní modulu *store.c*.

4.12.1 Definice datových typů

struct rev_t

```
typedef struct rev_t{
  size_t orig;
  size_t compr;
  int method;
  struct rev_t *next;
  struct rev_t *prev;
  int index_nr;
  long ts;
}
```

Tato struktura se používá při vytváření obousměrného spojového seznamu s informacemi o jednotlivých verzích dokumentu. Význam jednotlivých položek:

orig velikost původní nekomprimované verze dokumentu

compr velikost uložené verze dokumentu v datovém souboru

method číslo metody (komprese) dané verze

next ukazatel na následující uzel spojového seznamu

prev ukazatel na předcházející uzel spojového seznamu

index_nr číslo souboru indexu druhé úrovně, kterým je indexován datový soubor s touto verzí dokumentu

ts časové razítko uložení této verze dokumentu

struct row_lev1

```
typedef struct row_lev1{
  int from;
  int to;
  int index_nr;
  size_t size_df;
  int diff_count;
}
```

Struktura se používá pro uchování informací jednoho řádku souboru indexu první úrovně. Její položky odpovídají významu položek každého řádku souboru.

struct row_lev2

```
typedef struct row_lev2{
  size_t compr;
  size_t orig;
  int method;
  long ts;
}
```

Struktura se používá pro uchování informací jednoho řádku souboru indexu druhé úrovně. Její položky odpovídají významu položek každého řádku souboru.

struct file_t

```
typedef struct file_t{
  int fd;
  char *buf;
  char *name;
  size_t pos;
  size_t sz;
  int rw;
  int used;
}
```

Struktura se používá pro práci se soubory indexu obou úrovní. Zápisy a čtení jednotlivých řádků jsou z důvodu časové efektivity bufferovány a do souboru se zapisují po celých blocích. Význam jednotlivých položek:

fd deskriptor přiřazený otevřenému souboru

buf ukazatel na buffer pro čtení resp. zápis

name jméno souboru, se kterým se prostřednictvím této struktury pracuje

pos aktuální pozice v bufferu

sz velikost bufferu

rw příznak, zda se ze souboru pouze čte (`rw`=0), nebo do něj pouze zapisuje (`rw`=1)

used příznak, zda byla tato struktura pro soubor použita

4.12.2 Definice konstant a maker

V tomto hlavičkovém souboru jsou definovány konstanty pro návratové kódy volání funkcí modulu *store.c*. Jedná se o tyto kódy:

STR_OK vše proběhlo v pořádku

STR_OK_SAME_REVISION ukládaná verze dokumentu byla shodná s naposledy uloženou, stejné verze se mají ukládat, verze byla úspěšně uložena

STR_ERROR blíže nespecifikovaná chyba

STR_ERROR_CSV chyba formátu řádku v souboru indexu

STR_ERROR_SAME_REVISION ukládaná verze dokumentu byla shodná s naposledy uloženou, stejné verze se nemají ukládat, proto chyba

STR_ERROR_NOT_FOUND požadovaný dokument nebyl nalezen

STR_ERROR_PARAMS chybně zadané parametry

Dále jsou zde definovány konstanty:

SUFFIX_LEN délka přípon v názvech používaných souborů

DEFAULT_? jsou konstanty definující implicitní hodnoty parametrů knihovny. Parametry jsou uloženy v globální struktuře `args`. Znak `?` v textu zastupuje název příslušné položky zapsaný velkými písmeny

Definovaná makra slouží pro určení minimální velikosti bufferů použitých pro jednotlivé kompresní metody na základě velikosti souboru:

gz_out_buf_size(x) minimální velikost bufferu pro výstup komprese souboru o velikosti x metodou gzip

bz_out_buf_size(x) minimální velikost bufferu pro výstup komprese souboru o velikosti x metodou bzip2

mz_out_buf_size(x) minimální velikost bufferu pro výstup předzpracování souboru o velikosti x metodou 1 nebo 2 se statickým slovníkem

mz_com_out_buf_size(x) minimální velikost bufferu pro výstup komprese předzpracovaného souboru o velikosti x metodou gzip nebo bzip2

Kapitola 5

Závěr

V souladu se zadáním byla navržena a implementována knihovna pro ukládání verzovaných html dokumentů.

Při návrhu bylo přihlédnuto k tomu, že očekávaná frekvence přístupů k nejnovějším verzím dokumentu je výrazně vyšší než k verzím starším. Nejnovější verze je proto dostupná přímo, při přístupu ke starším verzím může dojít k jistému zpomalení.

Ukládání verzí dokumentů využívá víceúrovňového diffu [14] a následné komprese. Tím bylo dosaženo snížení nároků na prostor za cenu zpomalení přístupu ke starším verzím dokumentů.

V rámci práce byly navrženy, implementovány a testovány nové kompresní metody založené na předzpracování html dokumentů před kompresí stávající metodou (některé z nich byly přijaty k publikaci v [13]). Ukázalo se, že ve většině případů se podařilo dosáhnout mírného zlepšení účinnosti komprese.

Implementovaná knihovna tak využívá jak stávajících, tak zejména nově navržených kompresních metod – při ukládání se testují jednotlivé metody a ukládá se nejlépe zkomprimovaná varianta dokumentu. Testy se provádí v paměti, a tak dochází jen k minimálnímu zpomalení ukládání. Výrazněji se na době ukládání podílí tvorba rozdílových souborů, které mohou být vytvářeny v závislosti na nastavení při ukládání druhé a dalších verzí dokumentu.

Při testování práce s archivem byly pro porovnání vyzkoušeny i archivy vytvořené archivátorem tar a zkomprimované metodou gzip a bzip2. Z pohledu časových nároků se ukázalo jako nepřijatelné celý archiv dekomprimovat a znovu komprimovat při každém přidání nové verze.

Pro účely testování byla knihovna zapouzdřena do jednoduché služby a byly vyvinuty testovací nástroje na zasílání požadavků této službě, které jsou také na přiloženém CD.

Pro praktické nasazení ve vyhledávači je zapotřebí dořešit ještě několik problémů: jak ukládat současně více verzí jednoho dokumentu (dosáhnout proudového zpracování příchozích verzí), jak nastavit optimální poměr mezi účinností komprese a nároky na zdroje, jak implementovat úložiště (využití klasického souborového systému je možná příliš náročné), atd. Víceúrovňový diff, který byl v této práci využit, je ještě v prototypové verzi – bude třeba jej doladit a zoptimalizovat. Pak by byl využitelný nejen ve vyhledávači, ale obecně (například v CVS či obdobných systémech).

Literatura

[1] Burrows M., Wheeler D. J.: A Block Sorting Loseless Data Compression Algorithm, SRC RR 124, Digital Equipment Corporation, Palo Alto, CA, U.S.A., 1994, also available as http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

[2] Card R., Ts'o T., Tweedie S.: Design and Implementation of the second extended filesystem, *Proceedings of the First Dutch International Symposium on Linux*, 1994, also available as http://web.mit.edu/tytso/www/linux/ext2intro.html

[3] Gailly J. L., Adler M.: zlib version 1.2.3, http://www.zlib.net

[4] Gailly J. L., Adler M.: The gzip home page, http://www.gzip.org

[5] Galamboš L., et al.: EGOTHOR (2006), http://www.egothor.org

[6] Google: Web authoring statistics (2006) http://code.google.com/webstats/index.html

[7] Huffman D. A.: A method for the construction of minimum redundancy codes, *Proc. Inst. Radio Eng.*, Vol. 40, 1098–1101, 1952.

[8] Hunt J. W., McIlroy M. D.: An Algorithm for Differential File Comparison, Comput. Sci. Tech. Rep. 41, Bell Telephon Labs, Murray Hill, NJ, USA, Aug. 1976.

[9] Knuth D. E.: Dynamic Huffman Coding, *J. of Algorithms*, Vol. 6, 1985, 163–180.

[10] Kopecký M.: Dokumentografické informační systémy, přednáška a materiály k přednášce, MFF UK, Praha, 2006.

[11] Lánský J., Galamboš L., Chernik K.: Komprese webového úložiště, *Proceedings of ITAT 2006 Information Technologies – Applications and Theory*, PF UPJŠ, Košice, 2006, 107–112.

[12] Liefke H., Suciu D.: XMill: An efficient compressor for XML data, In Chen W., Naughton J. F., Bernstein P. A., eds.: *SIGMOD Conference*, ACM (2000) 153–164.

[13] Matouš V., Žemlička M.: Zvýšení účinnosti komprese HTML souborů vhodným předzpracováním, *Proceedings of ITAT 2007 Information Technologies – Applications and Theory*, PF UPJŠ, Košice, 2007.

[14] Paščenko P.: Verzovaná komprese textových dokumentů, Bakalářská práce, MFF UK, Praha, 2007.

[15] Pokorný J., Žemlička M.: *Základy implementace souborů a databází*, 2. upravené vydání, Karolinum, Praha, 2004.

[16] Seward J.: libbzip2 version 1.0.4, http://www.bzip.org

[17] Seward J.: The bzip2 and libbzip2 official home page, http://sources.redhat.com/bzip2/

[18] Skibiński P.: Reversible data transforms that improve effectiveness of universal lossless data compression, PhD Thesis, University of Wrocław, Wrocław, Poland, 2006.

[19] Storer J. A., Szymanski T. G.: Data compression via textural substitution, *J. ACM*, Vol. 29 No. 4, 1982, 928-951.

[20] World Wide Web Consorcium: HyperText Markup Language (HTML), http://www.w3.org/MarkUp

[21] Ziv J., Lempel A.: A universal algorithm for sequential data compression, *IEEE Transac. on Information Theory*, Vol. IT-23. No. 3, 1977, 337–343.

[22] Ziv J., Lempel A.: Compression of individual sequences via variable-rate coding, *IEEE Transac. on Information Theory*, Vol. IT-24. No. 5, 1978, 530–536.

Dodatek A

Obsah přiloženého CD

Přiložené CD obsahuje vedle této práce v elektronické podobě a zdrojových kódů vytvořené knihovny také všechny soubory, které sloužily jako testovací data. Dále obsahuje výsledky těchto testů včetně PHP skriptů, které byly použity pro zpracování výsledků. Adresáře mají tuto strukturu:

Adresář html-repository

Adresář obsahuje zdrojové kódy vytvořené knihovny včetně zdrojových kódů pro přeložení potřebné knihovny zlib, libbzip2 a DiffLib (každé ve svém vlastním adresáři). Kromě souborů, které jsou nutnou součástí knihovny obsahuje i zdrojové kódy ukázkové aplikace, která byla použita při testování knihovny.

Adresář data

Adresář obsahuje testovací data použitá jak pro testování předzpracování, tak pro testování ukládání verzovaných dokumentů pomocí knihovny.

Adresář výsledky

Adresář obsahuje naměřené výsledky uložené v textových souborech. Obsahuje rovněž PHP skripty použité pro zpracování výsledků.