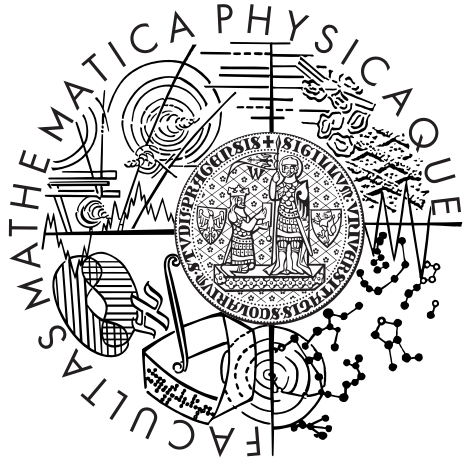
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Marek Švantner

Univerzální index textových dokumentů

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Martin Holub, Ph.D.

Studijní program: Informatika, Datové inženýrství

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

V Praze dne 20. prosince 2006 Marek Švantner

Obsah

1. Úvod **1**

2. Indexy v textových systémech **5**

2.1. Funkce a modely textových indexů 5

2.2. Invertované soubory 8

2.2.1. Struktura invertovaného souboru 8

2.2.2. Vyhodnocení dotazu 11

2.2.3. Omezení a rozšíření 13

2.3. Klasická implementace invertovaných souborů 14

2.3.1. Konstrukce invertovaného souboru 16

2.3.2. Efektivita, výhody a nevýhody invertovaných souborů 17

2.4. Optimalizace dotazů 18

2.4.1. Příklad 18

2.4.2. Optimalizační algoritmy 19

2.5. Zipfův zákon 20

3. Návrh dynamického univerzálního indexu **23**

3.1. Zobecnění funkce indexu 23

3.2. Požadavky na dotazování 24

3.3. Dynamický invertovaný soubor 25

3.3.1. Návrh struktury univerzálního indexu 25

3.3.2. Základní algoritmy nad univerzálním indexem 27

3.3.3. Získání seznamu indexových záznamů 27

3.3.4. Vkládání 28

3.3.5. Mazání 31

3.3.6. Vylepšení a rozšíření univerzálního indexu 34

4. Efektivita dynamického invertovaného souboru **35**

4.1. Obecné předpoklady a označení 35
4.2. Časová analýza expanze bloku 36
4.2.1. Vliv distribuce dat 45
4.3. Časová analýza ostatních operací 45
4.3.1. Operace insert 45
4.3.2. Operace delete 45

5. Odolnost proti výpadkům 46
5.1. Úvod do transakcí 46
5.2. Idempotence operace zotavení z výpadku 49
5.3. Implementace transakčního logu 50
5.3.1. Fyzická struktura logu 50
5.3.2. Kontrola přetečení logu 51
5.3.3. Algoritmus zotavení 51
5.4. Vliv logu na dotazování 55
5.5. Vliv logu na efektivitu indexu 56

6. Komprese dynamického invertovaného souboru 57
6.1. Požadavky na kompresní metody 58
6.2. Eliasovy kódy 58
6.2.1. Pomocné kódy 59
6.3. Diferenční kódování 62
6.4. B–blokové kódování 62
6.4.1. Základní B–blokové kódování 62
6.4.2. Kombinované B–blokové kódování 63
6.5. RLE komprese znaménkových bitů 66
6.6. Komprese textových atributů a obecných dat 67
6.7. Kombinace kompresních metod 67

7. Implementace a testování 68
7.1. Celková architektura 68
7.2. Hlavní třídy a moduly systému 69
7.3. Moduly Index a Dictionary & Directory 71
7.4. Třída PersistentStorage 73
7.5. Testování 73
7.5.1. Překlad knihovny univerzálního indexu 73

7.5.2. Testovací rozhraní 74

8. Experimenty **75**

8.1. Data pro experimenty 75

8.1.1. Datová kolekce *LN* 76

8.1.2. Datová kolekce *CW* 76

8.2. Časová složitost operace *insert* 77

8.3. Vliv volby parametru κ na dynamický invertovaný soubor 79

8.4. Experimenty s kompresními metodami 81

8.4.1. Žádná kompresní metoda 81

8.4.2. Diferenční kódování a B-blokové kódování 82

8.4.3. Diferenční kódování a Eliasův kód ω' 82

8.4.4. Diferenční a kombinované kódování ω 83

8.4.5. Diferenční a kombinované kódování ω' 83

8.4.6. Závěr experimentů s kompresí 84

8.5. Vliv transakčního zpracování na efektivitu 84

9. Závěr **87**

A. Rozhraní v jazyce C **89**

A.1. Identifikátory indexů 89

A.2. Pomocné definice – datové typy 89

A.3. Konstanty 90

A.4. Inicializace a konfigurace 91

A.5. Připojení a odpojení indexu 92

A.6. Vytváření a rušení indexů 93

A.7. Datové struktury pro výměnu dat 94

A.8. Datová struktura dotazu 95

A.9. Dotazování 101

A.10. Vkládání nových záznamů 104

A.11. Aktualizace indexových záznamů 104

A.12. Mazání indexových záznamů 105

A.13. Pomocné funkce 106

A.14. Rozhraní v jazyce C++ 106

B. Chybové kódy **108**

B.1. Neznámá chyba 108
B.2. Chyby v modulu indexu 108
B.3. Chyby v modulu `PersistentStorage` 110
B.4. Chyby v modulu `BasicDataStructures` 110
B.5. Chyby v modulu `FileSystem` 110

C. Formáty souborů **112**
C.1. Formát slovníku (dictionary – *.dic) 112
C.2. Formát adresáře indexu (index directory – *.dir) 113
C.3. Formát souboru indexových záznamů (index records storage – *.idx) . 113

D. Použité softwarové vybavení **115**

E. Obsah přiloženého CD **117**

Literatura **118**

Rejstřík **119**

Název práce: Univerzální index textových dokumentů

Autor: Marek Švantner

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Martin Holub, Ph.D.

E-mail vedoucího: holub@ufal.mff.cuni.cz

Abstrakt: Diplomová práce se zabývá návrhem a implementací vysoce efektivního univerzálního indexu textových dokumentů. Univerzální znamená možnost jednak konfigurovat struktury indexových záznamů a metod zpracování dat (bez nutnosti rekompilace), jednak použít knihovnu indexu i pro jiné účely, například pro tvorbu tezauru, reprezentaci bibliografických vztahů nebo pro reprezentaci určité třídy funkcí v jiných oblastech než jsou dokumentografické systémy. Pro implementaci je navržen *dynamický invertovaný soubor*, který umožňuje efektivně provádět aktualizační operace bez nutnosti přebudování datové struktury. Specifickými oblastmi práce jsou i on–line komprese indexu a zajištění odolnosti datové struktury proti výpadkům pomocí transakčního zpracování.

Je odvozena konstantní amortizovaná složitost struktury, která je poté experimentálně ověřena. Další experimenty se týkají i výkonu kompresních metod a vlivu parametrů datové struktury na její výkon a zabraný prostor.

Diplomová práce obsahuje vlastní implementaci univerzálního indexu v C/C++ testovanou v prostředích Linux a Windows XP.

Klíčová slova: index invertovaný univerzální dokumentografický vyhledávání textový

Title: Universal Full-Text Index

Author: Marek Švantner

Department: Department of Software Engineering

Supervisor: RNDr. Martin Holub, Ph.D.

Supervisor's email address: holub@ufal.mff.cuni.cz

Abstract: This diploma thesis deals with the design and implementation of a highly efficient universal index of textual documents. Universal stands for an opportunity to configure structures of index records and methods of the index data processing (without recompiling an application). Furthermore, it means that the index library can be used even for other purposes, for example to implement a thesaurus, to represent bibliographic relationships or even for generic representation of a specific class of functions in other areas than documentographic systems. The index is implemented using the *dynamic inverted file* which can be efficiently updated without need of the data structure rebuilding. Specific issue is on–line index compression and failure recovery via the transactional log.

It is shown that the amortized complexity of the data structure is linear. This fact is afterwards experimentally verified. Other experiments address the compression methods and the impact of the data structure parameters on its efficiency.

The diploma thesis contains the implementation of the universal index in C/C++. It has been tested in the Linux and Windows XP environments.

Keywords: index inverted universal documentographic search textual

Kapitola 1

Úvod

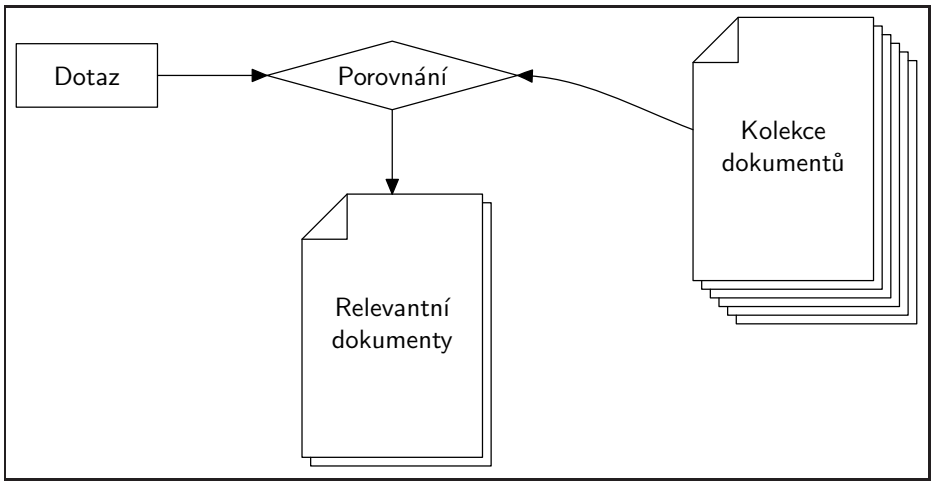
Dokumentografickým informačním systémem (DIS) rozumíme informační systém, jehož primárním určením je spravovat kolekci dokumentů psaných v přirozeném jazyce a umožnit uživateli efektivní vyhledávání a zpřístupnění žádaných textů. Původně tyto systémy vznikly automatizací postupů v oblasti knihovnictví, dnes mají i mnoho dalších aplikací (internetové vyhledávače atd.) – viz [8]. V praxi můžeme DIS rozdělit do několika hlavních modulů:

- *Dotazovací modul*: Umožňuje uživateli zadání požadavků na hledaný dokument.
- *Modul vyhodnocení dotazu*: Hledá množinu dokumentů vyhovujících dotazu (tzv. relevantních dokumentů).
- *Subsystém zpřístupnění dokumentu*: Poskytuje uživateli základní informace o každém z množiny relevantních dokumentů (název, autor, vydavatel, rok vydání, abstrakt, počet stran, množina klíčových slov atd.).
- *Subsystém dodání dokumentu*: Dodá vlastní text dokumentu, který si uživatel zvolí na základě informací subsystému zpřístupnění dokumentu.
- *Administrátorský modul*: Umožňuje správci aktualizovat textovou kolekci. Každé přidání či ubrání dokumentu často vyžaduje přebudování datových struktur používaných modulem vyhodnocení dotazu – v tomto textu však ukážeme takovou strukturu, která umožňuje aktualizační operace provádět efektivně za běhu.

Tato práce se bude zabývat pouze modulem vyhodnocení dotazu, implementace ostatních modulů se očekávají vždy od konkrétní aplikace, která bude tuto knihovnu využívat.

Základní obecnou představu o vyhodnocení dotazu poskytuje obrázek 1.1 (viz [5]). DIS načte ze vstupu dotaz a svým *modulem vyhodnocení dotazu* jej porovná se všemi

dokumenty, které spravuje v příslušné *kolekci dokumentů* . Výsledkem je určení, popř. dodání, dokumentů relevantních původnímu dotazu.[1]



Obrázek 1.1: „Naivní“ představa dokumentografického systému

Je však zřejmé, že je fyzicky neproveditelné porovnávat každý dotaz se všemi dokumenty v celé kolekci. Proto je třeba provést následující úpravu podle obrázku 1.2.

V této sofistikovanější variantě se dokumenty nejdříve předzpracují – z každého z nich se získají až řádově menší data, která budeme označovat jako *deskriptory dokumentů*. Deskriptor typicky ponese mnohem méně informace než původní dokument, ale tato nevýhoda bude vyvážena tím, že deskriptory půjde relativně snadno uspořádat do datové struktury, která umožní efektivní nalezení relevantních dokumentů. Této datové struktuře budeme říkat *index*. *Indexace* je potom proces přiřazení deskriptorů ke každému z dané množiny dokumentů.

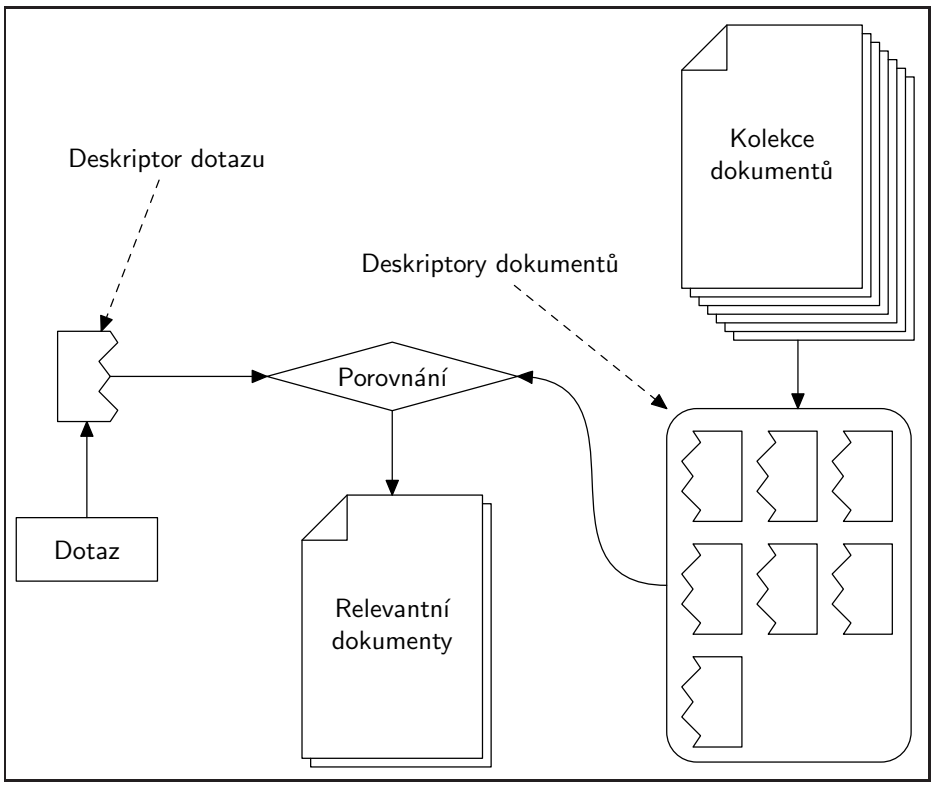
Podobné předzpracování je někdy prováděno i s dotazem. Tím je možno ještě více zefektivnit proces hledání relevantních dokumentů.

Poznámka: V literatuře lze narazit ještě na jinou terminologii – někteří autoři říkají deskriptoru dokumentu *index dokumentu* a datové struktuře spravující deskriptory *indexový soubor*. Tento text se však bude držet výhradně terminologie *deskriptor – index*.

Cíl práce

Cílem této práce je navrhnout a implementovat vysoce efektivní modul (C/C++ knihovnu) pro vyhledávání v kolekcích dokumentů. Tento modul musí být založen na speciální diskové datové struktuře – indexu – umožňujícím dynamickou konfigurovatelnost, tj. možnost volby struktury dat, která budou v této struktuře udržována

[1] Relevantními dokumenty se rozumí všechny dokumenty v textové kolekci, které splňují podmínky specifikované dotazem uživatele.



Obrázek 1.2: Realističtější představa dokumentografického systému

pro jednotlivá indexovaná slova. Tento index je navržen na základě principu znázorněného na obrázku 1.2, tj. na správě a vyhledávání v deskriptorech dokumentů.[2]

Díky zmíněné konfigurovatelnosti bude index využitelný nejen pro implementaci vyhledávání v kolekcích dokumentů, ale obecněji pro reprezentaci širší množiny funkcí mapujících identifikátory na záznamy složené z podporovaných datových typů. Například v oblasti dokumentografických systémů tedy může sloužit kromě mapování slov na dokumenty také pro reprezentaci různých vztahů mezi termy jako jsou tezaury, nadřazenost termů a další.

Datová struktura indexu dále musí být odolná proti výpadkům, což bude realizováno implementací transakčního zpracování aktualizačních operací. Přitom není nutné podporovat víceuživatelský přístup – každá aplikace využívající tento modul musí sama zaručit, aby nemohlo být paralelně spuštěno více aktualizačních operací ani aktualizační operace spolu s dotazem.

Posledním požadavkem je optimalizace velikosti indexu, která může radikálně zmenšit velikost jednotlivých částí datové struktury a tak podstatně zvýšit efektivitu jejich načítání při dotazování. Tato optimalizace bude realizována jednak zo-

[2]Ve skutečnosti není implementace principu vyhledávání v deskriptorech tak přímočará jako na obrázku 1.2, neboť deskriptory dokumentů v této implementaci „nedrží pohromadě“ – jsou rozptýleny po celé datové struktuře indexu. Během zpracování dotazu se potom hledají části deskriptorů dokumentů. Nalezení části deskriptoru v seznamu částí deskriptorů pro určité slovo potom implikuje výskyt tohoto slova v daném dokumentu. Celý tento princip bude podrobně popsán v následujících kapitolách.

hledněním empirických zákonitostí distribuce slov v dokumentech (zipfův zákon), které minimalizuje množství volného, nevyužitého prostoru v jednotlivých částech datové struktury. Další část redukce velikosti bude implementace specializovaného algoritmu komprese indexových dat.

Nedílnou součástí práce musí být i důkladné testování produktu. To bude realizováno jednak vývojovými testy jednotlivých modulů umožňujícími odhalit chyby již během vývoje a jednak jednoduchým nástrojem pro vytvoření konkrétní instance indexu a manipulaci s ní. Další množinou testů bude experimentální ověření časové složitosti vybraných operací nad indexem (která v rámci práce rovněž musí být teoreticky odvozena), srovnání efektivity operací s vypnutým a zapnutým transakčním zpracováním a srovnání výkonnosti různých implementovaných metod komprese. Tyto testy samy o sobě zároveň prověří funkčnost modulu jako celku.

Struktura práce

Druhá kapitola se zabývá úvodem do problematiky – popisuje význam indexů v dokumentografických systémech, stručně shrnuje některé jejich běžné typy. Zvláštním typem, na který je kladen důraz, je *invertovaný soubor*. Rovněž je uveden *Zipfův zákon* distribuce četností slov v dokumentech, který je spolu s invertovanými soubory základem pro tuto práci.

Invertovaný soubor ve své základní podobě neumožňuje efektivní aktualizační operace. Kapitola třetí prezentuje speciální datovou strukturu, tzv. *dynamický invertovaný soubor*, která tuto nevýhodu odstraňuje.

Na odvození teoretické efektivity navržené datové struktury se zaměřuje kapitola čtvrtá. Je ukázáno, že amortizovaná složitost operace vkládání nových záznamů je konstantní. Jiné operace kvůli rozsahu práce rozebrány nejsou.

Rozšířením základní funkcionality dynamického invertovaného souboru se zaobírají kapitoly pátá a šestá. Nejdříve je popsán způsob zajištění odolnosti proti výpadkům pomocí transakčního logu. Důraz je kladen na atomicitu operací, konkurentními akcemi více uživatelů se práce nezabývá. Poté jsou prezentovány metody komprese a kódování dat v indexu, které spojují některé běžně známé metody jako *diferenční* a *B-blokové kódování*, *Eliasovy kódy* a *RLE kompresi*.

Sedmá kapitola letmo představuje některé nejdůležitější implementační poznámky a poskytuje návod pro ověření funkcionality produktu pomocí velice jednoduchého nástroje příkazové řádky.

Kapitola osmá popisuje různé experimenty s výsledným produktem. Jedná se zejména o ověření časové složitosti operace insert, o vliv nastavení parametrů dynamického invertovaného souboru a o porovnání různých metod pro kompresi.

Celkové shrnutí celé práce je potom v závěrečné kapitole.

Kapitola 2

Indexy v textových systémech

Tato kapitola shrnuje různé obecné modely a mechanismy fungování textových indexů, zejména pak tzv. invertovaných souborů, které jsou základem pro implementaci popsanou v této práci.

2.1. Funkce a modely textových indexů

Jak již bylo popsáno v úvodu, textové indexy plní zejména funkci optimalizace zpracování dotazů nad kolekcemi textových dokumentů. V procesu dotazování tvoří jakési rozhraní mezi dokumenty v kolekci a dotazem, takže dotaz stačí vyhodnocovat efektivněji vzhledem k indexu, není třeba jej porovnávat s jednotlivými dokumenty.

Pro vyhodnocení dotazu nad kolekcí dokumentů je podstatné, jak jsou dotazy a dokumenty reprezentovány a jakou přesně příslušné reprezentace nesou informaci. Z tohoto hlediska rozlišujeme různé *modely* dokumentografických systémů. Uveďme pro představu alespoň několik nejzákladnějších:

- *Boolský model*: Dotazem je boolský výraz sestavený z jednotlivých termů (to mohou být např. slova obsažená v textu, jejich lemmata, kořeny, ale také slovní spojení, frazémy apod.) – např. *"obratlovec" OR "plaz" AND NOT "savec"* by mohl být dotaz na dokumenty obsahující slova obratlovec nebo plaz, ale neobsahující slovo savec. Dokument v boolském modelu je reprezentován množinou termů v něm obsažených. Systém považuje za relevantní všechny dokumenty, jejichž deskriptor odpovídá boolskému výrazu dotazu. Nevýhodou tohoto modelu je především to, že neumožňuje uspořádat množinu výstupních textů dle „míry relevance“ a že je poměrně nepřesný. To je vyváženo jednoduchostí a výpočetní efektivitou (při rozumné implementaci).
- *Vektorový model*: Dotaz je vyjádřen vektorem stejné dimenze, jako je počet „relevantních“ slov (tj. těch, která mohou být použita v dotazech – nejsou to např. předložky, příliš časté termy apod.) ve všech dokumentech. Každému slovu odpovídá jedna složka vektoru – to je číslo, které je tím větší, čím charakterističtější je podle uživatele pro relevantní dokumenty. Deskriptorem dokumentu je opět vektor stejné dimenze – tentokrát je jeho i–tá složka tím větší,

čím lépe i–té slovo charakterizuje obsah dokumentu. Dále je definována vektorová metrika, což je zobrazení z množiny dvojic těchto vektorů do přirozených čísel, které vyjadřuje, jak jsou si dva vektory podobné. Dotazování se v základní variantě děje tak, že se nalezne N vektorů dokumentů takových, aby byly co nejpodobnější vektoru dotazu. Výhodou tohoto modelu je větší přesnost než u boolského, za kterou se však platí vyšší složitostí výpočtu (nutnost vyhodnocovat metriky na relativně velmi dlouhých vektorech). Dalšími nevýhodami je například to, že vektorový model neobsahuje negaci (i když efektu negace lze dosáhnout rozšířením na záporné váhy), neobsahuje explicitně logický součet (OR), vůbec v něm nelze realizovat exkluzivní součet (XOR). Naopak snáze se implementuje *zpětná vazba* (uživatel interaktivně zpřesňuje dotaz na základě výsledků předchozího) a také tento model umožňuje přesnější zohlednění zastupitelnosti termů (synonymita atd.).

- *Rozšířený boolský model*: V tomto modelu jsou deskriptory dokumentů, stejně jako ve vektorovém modelu, vektory vah. Dotaz je však reprezentován boolským výrazem – od boolského modelu se liší tím, že může volitelně obsahovat váhy jednotlivých termů. Vzorce pro vyhodnocení dotazu zde nebudeme uvádět, zmíníme jen, že se opět jedná o metriky, tentokrát parametrizované parametrem p (kladné reálné číslo). Tento parametr určuje chování modelu – např. pro $p = 1$ dostáváme klasický vektorový model, pro rostoucí hodnotu p rozšířený boolský model stále více konverguje k jednoduchému boolskému.

- *Model „sistringů“*: Viz odstavec o lexikografických indexech dále na této straně.

V současnosti existuje několik různých druhů indexů, které nyní stručně shrneme a uvedeme některé jejich výhody a nevýhody:

- *Lexikografické indexy*: Tyto indexy jsou založeny na množině indexovaných termů, která je lexikograficky uspořádaná. Typickým představitelem této kategorie jsou *invertované soubory* (více viz kapitola 2.2. a další), které umožňují najít ke každému termu seznam jeho výskytů v textové kolekci a efektivně implementovat boolský model. Jinou variantou jsou *PAT stromy* (binární komprimované trie), pomocí kterých se však neindexují termy, ale tzv. *sistringy* (semi–infinite strings – podřetězce textu, které začínají vždy na určité pozici a pokračují až do konce textu nebo i textové kolekce). Vyhledávání v PAT stromu potom probíhá tak, že stejnou metodou jako u trie nalezneme uzel reprezentující hledaný term a v něm si přečteme pozici v textu, na které by tento term mohl být. Nemusí se tam však nacházet nutně, neboť PAT strom je komprimované trie. Výhodou PAT stromů oproti invertovaným souborům je zejména to, že umožňují efektivně řešit mnohem více typů dotazů (například vyhledávání podle regulárních výrazů, vyhledání nejfrekventovanějšího podřetězce atd.). V neposlední řadě dokument nemusí být členěn na termy, ale může se klidně jednat o obrovské binární slovo, ve kterém lze vyhledávat libovolné podřetězce. Nevýhodou je značný nárůst paměťové režie ve srovnání s invertovanými soubory[1] a také to, že primární text musí být k dispozici, abychom

[1]Velikost PAT stromu bude sice lineární vzhledem k délce indexovaného textu, ale vzhledem

mohli ověřit skutečné nalezení hledaného řetězce. Nevýhodu vysokých paměťových nároků řeší tzv. *PAT pole* (PAT arrays) – efektivní implementace PAT stromů, ovšem za cenu nárůstu časové složitosti. Na závěr poznamenejme, že v základní podobě PAT pole/stromy neimplementují žádný model DIS (boolský, ...), neboť nejsou založeny na termech. Pokud však množinu sistringů omezíme tak, že každý z nich musí začínat na pozici, kde začíná v textu nějaký term, lze pomocí nich boolský model úspěšně simulovat.

- *Clusterované (shlukované) indexy*: Clusterované indexy jsou známy především v souvislosti s implementací vektorového modelu DIS. V jeho popisu bylo uvedeno, že je založen na výpočtu vzdálenosti vektoru dotazů od vektorů dokumentů – je však zřejmé, že je nesmysl pro každý dotaz počítat vzdálenosti od všech dokumentů. Tuto situaci řeší clusterované indexy, které množinu vektorů dokumentů rozdělí do více podmnožin (*clusterů* – shluků) a každé podmnožině přiřadí vektor, který ji bude reprezentovat. Rozdělení do podmnožin je třeba dělat tak, aby vektory v každé z nich si byly co nejblíže (v dané metrice) a zároveň aby byly dost daleko od vektorů z podmnožin ostatních. Toto rozdělení lze dělat i hierarchicky. Vyhledávání potom probíhá tak, že se pro vektor dotazu najde vektor reprezentující shluk, který je nejbližší vektoru dotazu a dále se srovnávají jen vektory obsažené v tomto shluku. Pouze pokud ve shluku není dostatek vektorů, hledá se druhý nejbližší shluk atd.

- *Hašované indexy*: Příkladem indexů založených na hašování jsou např. *signaturové soubory*. Nejedná se vlastně o indexy v pravém slova smyslu, neboť umožňují pouze přibližné vyhledávání, a tudíž jsou vhodné především jako předstupeň jiné a přesnější metody. Fungují tak, že na základě nějaké malé a snadno dostupné informace o jednotlivých dokumentech umožňují snadno vyřadit mnoho dokumentů, které nemohou vyhovovat dotazu – přitom nijak nezaručují, že neodstraněné dokumenty dotazu vyhovují. Dotaz je představován konjunkcí termů (ostatní logické operace nejsou podporovány).

  Dokumenty jsou rozděleny do množin (ovšem, na rozdíl od clusterovaných indexů, bez ohledu na jejich obsah) a ke každé množině je přiřazena tzv. *signatura*. Ta určuje, které termy mohou být v této množině dokumentů obsaženy (například tak, že se na deskriptor vyhradí 1000 bitů a každému termu přiřadí bitový řetízek stejné délky; bitové řetízky se potom zkombinují pomocí bitové operace OR. Stejným způsobem se zformuje signatura odpovídající dotazu). Dotazování je založeno na operaci popsané výrazem SQ *and not* ST, kde SQ je signatura dotazu a ST signatura množiny dokumentů. Je-li hodnota tohoto výrazu nenulová, text nemůže vyhovovat dotazu. V opačném případě nelze na základě této metody rozhodnout a je třeba příslušný dokument (dokumenty) načíst a nebo implementovat druhou fázi pomocí nějaké přesnější techniky.

Index textových dokumentů realizovaný v této práci je založen na principu invertovaných souborů. Jak bylo uvedeno, pomocí tohoto typu indexů se přirozeným

k tomu, že obsahuje mnoho pointerů a dodatečných informací v každém uzlu, může jeho velikost růst více než u klasických lexikografických indexů.

způsobem realizuje boolský model. Vzhledem k požadavku na konfigurovatelnost struktury lze však strukturu indexu snadno rozšířit o atribut reprezentující relevanci slov v dokumentech a tím se v určitém smyslu přiblížit rozšířenému boolskému, potažmo vektorovému modelu. Kvůli specifické organizaci invertovaného souboru však nelze efektivně získat vektory dokumentů – efektivně lze získat pouze prvky těchto vektorů pro určitou malou množinu slov, například pro slova obsažená v dotazu. V tomto smyslu musejí být upraveny příslušné metriky. Vektorový model v pravém slova smyslu tedy z uvedených důvodů tímto způsobem implementovat nelze.

2.2. Invertované soubory

2.2.1. Struktura invertovaného souboru

Invertovaný[2] soubor se typicky skládá ze dvou základních částí. První z nich je *indexový soubor*, který vhodným způsobem uchovává seznamy *indexových záznamů* pro jednotlivé termy obsažené v kolekci dokumentů. Každý indexový záznam ve své minimální podobě obsahuje informaci o dokumentu, který obsahuje konkrétní výskyt daného termu. Pro daný term lze tedy triviálně procházením seznamu zjistit množinu všech dokumentů, které tento term obsahují.

Druhou částí je *slovník indexovaných termů*[3] – ten představuje jakýsi „rejstřík“ indexového souboru, tj. pro libovolný term umožňuje efektivně v indexovém souboru vyhledat příslušný seznam indexových záznamů.

Princip invertovaného souboru je znázorněn na obrázku 2.1. Slovník indexovaných termů je zjednodušeně představován levou částí obrázku, zatímco soubor indexových záznamů se nachází vpravo. Obrázek však nic neříká o fyzické organizaci dat v jednotlivých částech datové struktury.

Poznámka: Existuje mnoho metod konstrukce invertovaného souboru; příklad jedné z nich je uveden v části 2.3.1. Obecně se vyznačují vysokou časovou náročností a nutností kompletní reorganizace při jakékoli změně v indexovaných datech.

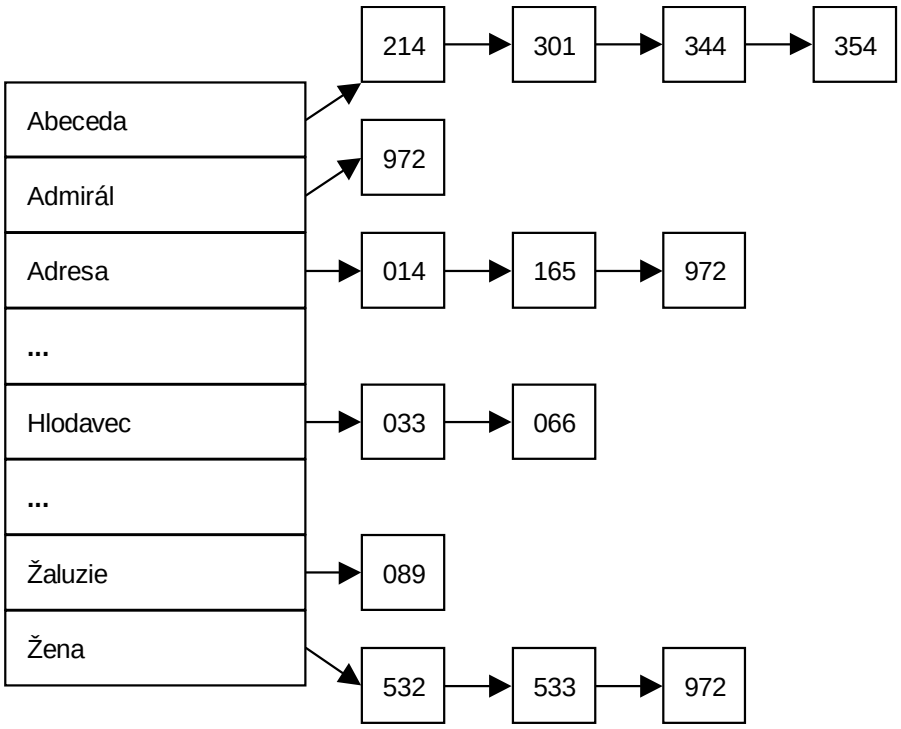
V dalších kapitolách bude navržena efektivní implementace invertovaného souboru umožňující inkrementální přidávání a odebírání indexových záznamů.

Nyní bude pojem invertovaného souboru a indexových záznamů zaveden formálně:

Definice 2.2.1: *Nechť n, n_1, n_2 jsou přirozená čísla taková, že $n = n_1 + n_2$, $n_1 > 0$. Potom definujeme* indexový záznam *jako n–tici, pro kterou platí:*

[2]Původ názvu „invertovaný soubor“ bude patrný z algoritmu v sekci 2.3.1.

[3]Ne vždy je slovník považován přímo za součást indexu – v dalších kapitolách uvidíme implementaci, která používá dvouúrovňový slovník, přičemž jen jedna úroveň je součástí knihovny indexu.



Obrázek 2.1: Základní struktura invertovaného souboru

- *Prvních n_1 složek nese informaci dostatečnou k jednoznačné identifikaci entity (dokumentu) v kolekci entit (dokumentů). Tyto složky nazveme* primárními složkami indexového záznamu.
- *Zbývajících n_2 složek nese libovolnou další informaci, která však není nutná k jednoznačné identifikaci. Tyto složky nazveme* sekundárními složkami indexového záznamu.

Přirozeným způsobem definujeme také primární, resp. sekundární část indexového záznamu jako n_1–tici primárních, resp. n_2–tici sekundárních složek.

V nejjednodušším případě se tedy indexový záznam skládá pouze z jediné primární složky obsahující identifikaci dokumentu, ve kterém se nachází daný term. Více primárních složek může být potřeba například pokud je nutné identifikovat i kapitoly nebo odstavce v rámci dokumentu a podobně. Naopak sekundární složky mohou sloužit k zajištění dodatečné funkcionality, jako jsou například proximitní dotazy[4] (lze realizovat přidáním informace o pořadovém číslu termu v dokumentu) nebo zohlednění relevance (přidání informace o váze termu).

Definice 2.2.2: *Pro indexový záznam z definujeme*

(i) $pri(z)$ jako indexový záznam obsahující pouze primární část z (zúžení na primární část);

[4]Proximitními dotazy se rozumí dotazy obsahující podmínky na vzájemnou polohu nebo vzdálenost slov v dokumentu.

(ii) $sec(z)$ *jako indexový záznam obsahující pouze sekundární část* z *(zúžení na sekundární část).*

Definice 2.2.3: Uspořádaný seznam indexových záznamů *je* m*–tice (*m *přirozené)* $\mathbf{S}$ *indexových záznamů uspořádaných vzestupně primárně podle první složky, sekundárně podle druhé a tak dále až do poslední primární složky záznamu. Sekundární část se pro toto uspořádaní ignoruje.*

Seznamy indexových záznamů jsou uchovávány v invertovaném souboru, odkud mohou být získány pomocí slovníku (viz dále). Během zpracování netriviálních dotazů budou různými způsoby slévány – tím budou dočasně vznikat seznamy, které v původní datové struktuře nebyly vůbec obsaženy. Právě kvůli slévání musí být seznamy indexových záznamů uspořádány podle svých primárních částí.

Je nutno zdůraznit, že dotazy mohou obsahovat také *negaci*. Proto je třeba umět pracovat také s negací seznamu indexových záznamů.

Bylo by velice nepraktické a neefektivní tuto negaci fyzicky konstruovat a to i za tzv. *předpokladu uzavřenosti světa*. Místo toho budeme rozlišovat mezi dvěma typy seznamů. První bude ukládán v invertovaném souboru a se druhým budeme pracovat při vyhodnocování dotazu – ten bude obsahovat navíc binární příznak negace. Místo fyzické konstrukce negace se potom jen nastaví tento příznak u příslušného seznamu.

Definice 2.2.4: Uspořádaný seznam indexových záznamů s negací *je uspořádaný seznam indexových záznamů podle definice 2.2.3 spolu s příznakem negace, který může nabývat hodnot z množiny* $\{true, false\}$.

Bez újmy na obecnosti lze nadále v teoretickém zápisu rozdíly mezi těmito dvěma typy seznamů zanedbat a předpokládat, že *oba* typy mají schopnost uchovávat příznak negace. V tomto textu tedy budeme pracovat jen s *uspořádanými seznamy indexových záznamů s negací*.

Značení operací pracujících s příznakem negace zavádí následující definice:

Definice 2.2.5: *Nechť* $\mathbf{S}$ *je seznam indexových záznamů. Potom*

(i) $\neg\mathbf{S}$ *značí negaci tohoto seznamu (tzn. seznam, který obsahuje stejné záznamy jako* $\mathbf{S}$*, ale má opačně nastaven příznak negace);*

(ii) $neg(\mathbf{S})$ *nabývá hodnoty true, jestliže* $\mathbf{S}$ *má nastaven příznak negace. V opačném případě nabývá hodnoty false.*

Definice 2.2.6: *Nechť* $\mathbf{S}$ *je seznam indexových záznamů. Potom definujeme následující operátory:*

(i) $set(\mathbf{S})$ *jako množinu indexových záznamů obsažených v* $\mathbf{S}$.

(ii) $pri(\mathbf{S})$ *jako množinu indexových záznamů obsažených v* $\mathbf{S}$ *zúžených na primární část.*

(iii) $sec(\mathbf{S})$ *jako množinu indexových záznamů obsažených v* $\mathbf{S}$ *zúžených na sekundární část.*

2.2.2. Vyhodnocení dotazu

V této části předpokládejme již zkonstruovaný invertovaný soubor pro nějakou kolekci dokumentů.

Dotazem lze chápat logický výraz, kde jednotlivé atomy představují zaindexované nebo nezaindexované termy a přípustné operace jsou logický součin (*and*), logický součet (*or*) a negace (¬). Například pro dotaz ((mlži *or* plži) *and* ¬slimák) budou relevantní všechny dokumenty, které se zmiňují o mlžích nebo plžích, ale nemluví o slimácích.[5] To by se dalo zobecnit použitím lingvistických modulů jako např. lemmatizátoru a dalších. Tím se ale tato práce nebude zabývat.

Vyhodnocení dotazu probíhá podle následujícího schématu. Nejdříve se pomocí slovníku získá seznam indexových záznamů pro každý term obsažený v dotazu (nezaindexovaným termům je přiřazen prázdný seznam). Poté se rekurzivně dle struktury dotazu tyto seznamy slévají za použití příslušných operací. Možné operace nyní definujeme. Nejdříve však uveďme několik pomocných definic:

Definice 2.2.7: *Definujme unární operátor* $\langle\langle\ \rangle\rangle$ *takto:*

(i) Je–li $\mathbf{M}$ množina indexových záznamů, $\langle\langle\, \mathbf{M} \,\rangle\rangle$ *je* seznam indexových záznamů obsažených v množině $\mathbf{M}$ *(mj. splňující požadavek na uspořádání dle primární části).*

(ii) Je–li cond nějaký unární predikát m*–tého řádu, definujeme* $\langle\langle\ x \mid cond(x)\ \rangle\rangle$ *jako* seznam indexových záznamů vyhovujících predikátu *cond.*

Operátor $\langle\langle\ \rangle\rangle$ tedy uspořádá indexové záznamy z dané množiny na základě jejich primárních částí a z výsledku vyrobí seznam přidáním příznaku negace dle definice 2.2.4.

Definice 2.2.8: *Nechť* t_1, t_2 *jsou libovolné termy zaindexované v daném invertovaném souboru. Buďte* $\mathbf{S}_{t_1}$, *resp.* $\mathbf{S}_{t_2}$, *příslušné seznamy indexových záznamů. Potom definujeme:*

(i) Slití seznamů indexových záznamů v režimu logického součinu *jako operaci, jejímž výsledkem je*

$$\langle\langle\ x \mid (pri(x) \in pri(\mathbf{S}_{t_1}) \wedge pri(x) \in pri(\mathbf{S}_{t_2})) \wedge (x \in set(\mathbf{S}_{t_1}) \vee x \in set(\mathbf{S}_{t_2}))\ \rangle\rangle.$$

Tedy výsledný seznam obsahuje všechny indexové záznamy takové, že $\mathbf{S}_{t_1}$ *i* $\mathbf{S}_{t_2}$ *obsahují alespoň po jednom záznamu se stejnou primární částí, přičemž alespoň jedna z částí sekundárních musí odpovídat výslednému záznamu.*

(ii) Slití seznamů indexových záznamů v režimu logického součtu *jako operaci, jejímž výsledkem je* $\langle\langle\ set(\mathbf{S}_{t_1}) \cup set(\mathbf{S}_{t_2})\ \rangle\rangle$. *Tedy ve výsledném seznamu se budou vyskytovat všechny indexové záznamy takové, že v* $\mathbf{S}_{t_1}$ *nebo v* $\mathbf{S}_{t_2}$ *se vyskytuje alespoň jeden záznam se* všemi *složkami stejnými.*

[5] V základní verzi to přesně znamená, že tyto dokumenty obsahují (příp. neobsahují) daná slova ve tvaru uvedeném v dotazu.

Slití v režimu logického součinu označme symbolem $\sqcap$*, slití v režimu logického součtu potom symbolem* $\sqcup$*.*

Poznámka 2.2.9: Operace $\sqcap$ a $\sqcup$ nemusejí umět pracovat se všemi variantami negací ve svých operandech. Konkrétně například slévání v režimu logického součtu nemusí podporovat situaci, kdy jsou oba argumenty negovány. Algoritmus pro zpracování dotazu totiž tyto případy převede na slévání v režimu logického součinu. Pro logický součet stačí implementovat případy, kdy nejvýše jeden operand je negován – to lze jednoduše provést pomocí operace rozdílu seznamů indexových záznamů (povšimněme si ovšem, že se jedná pouze o pomocnou operaci, která se nemůže explicitně vyskytnout uvnitř dotazu).

Definice 2.2.10: *Nechť* t_1*,* t_2 *jsou libovolné termy zaindexované v daném invertovaném souboru. Buďte* $\mathbf{S}_{t_1}$*, resp.* $\mathbf{S}_{t_2}$*, příslušné seznamy indexových záznamů. Pak definujeme operaci* rozdíl seznamů indexových záznamů $\mathbf{S}_{t_2} - \mathbf{S}_{t_1}$ *jako*

$$\langle\!\langle\, x \mid x \in set(\mathbf{S}_{t_2}) \wedge pri(x) \notin pri(\mathbf{S}_{t_1}) \,\rangle\!\rangle.$$

Nyní následuje samotný algoritmus zpracování dotazu nad strukturou invertovaného souboru:

Vstup: (pod)dotaz Q; pro každý atom A obsažený v dotazu seznam odpovídajících indexových záznamů S_A

Výstup: množina indexových záznamů relevantních vzhledem k dotazu Q

```
1  if Q je atom then result ← S_Q;
2  else if Q je tvaru ¬Q_1 then result ← ¬S_{Q_1};
3  else
       /* Q je tvaru νQ_1 op νQ_2, kde op je logický součin nebo součet; ν nabývá
          hodnoty buď ¬ nebo ¬¬ (tj. ν může být buď negace nebo prázdná
          operace).                                                          */
4      result_1 ← rmerge( Q_1 );
5      result_2 ← rmerge( Q_2 );
6      if neg(result_1) is true then result_1 ← ¬result_1; Q_1 ← ¬Q_1;
7      if neg(result_2) is true then result_2 ← ¬result_2; Q_2 ← ¬Q_2;
8      if Q je tvaru Q_1 and Q_2 then result ← result_1 ⊓ result_2;
9      else if Q je tvaru ¬Q_1 and Q_2 then result ← result_2 − result_1;
10     else if Q je tvaru Q_1 and ¬Q_2 then result ← result_1 − result_2;
11     else if Q je tvaru ¬Q_1 and ¬Q_2 then result ← ¬(result_1 ⊔ result_2);
12     else if Q je tvaru Q_1 or Q_2 then result ← result_1 ⊔ result_2;
13     else if Q je tvaru ¬Q_1 or Q_2 then result ← rmerge( ¬(Q_1 and ¬Q_2 );
14     else if Q je tvaru Q_1 or ¬Q_2 then result ← rmerge( ¬(¬Q_1 and Q_2 );
15     else if Q je tvaru ¬Q_1 or ¬Q_2 then result ← rmerge( ¬(Q_1 and Q_2 );
16 end
17 return result;
```

Procedura `rmerge(`Q`)` vytvoří seznam indexových záznamů odpovídajících všem dokumentům relevantním dotazu Q.

Způsob získání seznamu S_Q pro daný term Q je popsán na straně 27.

2.2.3. Omezení a rozšíření

Je zřejmé, že přesnost, úplnost a efektivitu dotazování lze zásadně ovlivnit omezením na termy (slova), která se mohou vyskytnout ve slovníku. V praxi platí pozorování, že termy, které se v dokumentech vyskytují „příliš často“, téměř nezmění výsledek dotazování, a tudíž nejsou pro indexaci vhodné. Naopak použití termů, které se téměř nevyskytují, v dotazu mívá za následek prázdnou množinu dokumentů ve výsledku, a proto tyto termy také není vhodné do slovníku vkládat.

V jednoduchých systémech se setkáme s indexovými záznamy obsahujícími pouze primární část, tj. pouze odkaz na dokument obsahující příslušný term. V takových

případech lze vyhledávat pouze dokumenty obsahující/neobsahující zadané termy.

Zvýšení přesnosti vyhledávání lze provést mnoha způsoby, např. již dříve uvedené rozšíření o proximitní dotazy (tj. dotazy, které obsahují také podmínky na vzájemné umístění slov v dokumentech – např. v jednom odstavci atd.). Jsou dvě možnosti jak toho dosáhnout: buď v indexu budou jen odkazy na dokumenty a podmínky proximitního dotazu se fyzicky ověří přímo v textové kolekci (hlavní nevýhody jsou, že textová kolekce musí být přístupná a že se výrazně zvýší doba odezvy), nebo se do indexových záznamů přidají další informace – například pořadové číslo slova v dokumentu nebo n-tice obsahující číslo odstavce, číslo slova ve větě apod. Tyto informace potom budou součástí sekundární části indexového záznamu.

Definice 2.2.11: *Složky sekundární části indexového záznamu, které nesou informaci o umístění indexovaného termu uvnitř jednoho dokumentu (jednoznačně určeného primární částí záznamu), nazveme* souřadnicemi výskytu termu v dokumentu.

Zavedením více dimenzí souřadnic výskytu se sice zvýší velikost indexu, ale na druhou stranu bude umožněno přesnější dotazování.

Poznámka 2.2.12: Některé dokumentografické systémy mohou ukládat nikoli dokumenty jako celky, ale například jejich jednotlivé části zvlášť (části dokumentů se berou za samostatné dokumenty – např. kapitoly, odstavce, ...). Potom složka souřadnic výskytu odpovídající těmto částem bude patřit do primární části indexového záznamu, nikoli do sekundární.

Další možná rozšíření mohou být založena i na dodatečných informacích přímo ve slovníku (např. počet výskytů termu v kolekci atd.).

V praxi je vhodné zakázat dotazy, které nejsou tzv. *pozitivně omezující*, tj. neobsahují v disjunktivně konjunktivní formě alespoň jeden literál bez negace. Tím se předejde tomu, abychom jako výsledek zpracování dotazu dostali negovaný seznam indexových záznamů. Vyhodnocováním dotazů, které nejsou pozitivně omezující, by se výrazně snížil výkon systému jako celku. Index navrhovaný v této práci předpokládá na svém vstupu pouze pozitivně omezující dotazy, ale nikterak tuto vlastnost nekontroluje. To by měl provést uživatel knihovny indexu.

2.3. Klasická implementace invertovaných souborů

Nejdříve uveďme stručný přehled některých běžně známých datových struktur, pomocí kterých lze implementovat slovníkovou část:

- *Uspořádané pole (Sorted array)*: Uspořádané pole umožňuje vyhledávání v nejhorším případě v logaritmickém čase vzhledem k počtu uložených prvků, nicméně jeho nevýhodou je lineární složitost přidávání a odebírání prvků.

- *Prefixový B-strom*: Opět se jedná o obecně známou datovou strukturu. V listech tohoto stromu mohou být buď odkazy na seznamy indexových záznamů (tj. odkazy do souboru indexových záznamů), nebo mohou být jednotlivé indexové záznamy přímo součástí tohoto stromu. Nevýhody prvního přístupu budou popsány níže, nevýhodou druhého je, že při slévání seznamů budou jednotlivé stránky každého ze seznamů fyzicky rozptýleny na různých místech disku, takže diskové hlavy se budou muset často přesouvat mezi jednotlivými stopami. Tím výrazně klesne efektivita.

- *TRIE*: Jelikož TRIE není tak známá struktura jako předchozí dvě, připomeňme zde jednu z jejích možných definic:

  Definice 2.3.1: *Nechť Σ je n–ární abeceda ($n > 0$) a $\mathbf{S}$ množina slov ze Σ^*. Potom TRIE reprezentující $\mathbf{S}$[6] je strom splňující následující podmínky:*

  (i) Každý vnitřní vrchol má právě n odkazů na syny, přičemž libovolné množství z nich může být prázdných (null).

  (ii) Existuje vzájemně jednoznačné zobrazení π prvků Σ na množinu

  $$\mathbf{P} = \{1, \ldots, n\}.$$

  Prvky $\mathbf{P}$ značí pořadí odkazu na syny. Jednoduše řečeno, i–tému prvku abecedy odpovídá u každého uzlu $\pi(i)$–tý odkaz na syna atd.

  (iii) Každému vrcholu je přiřazeno slovo z $\mathbf{S}$ následujícím způsobem:

  - *Kořenu je přiřazeno prázdné slovo λ.*
  - *Je-li vrcholu v přiřazeno slovo α, potom jeho s–tému synovi je přiřazeno slovo $\alpha \cdot \pi^{-1}(s)$, kde $\cdot$ značí konkatenaci.*

  (iv) Pokud v je vnitřní vrchol a je mu přiřazeno slovo α, pak existuje $\sigma \in \mathbf{S}$ takové, že α je prefixem σ.

  (v) Pro každý vrchol v je definována funkce member následujícím způsobem:

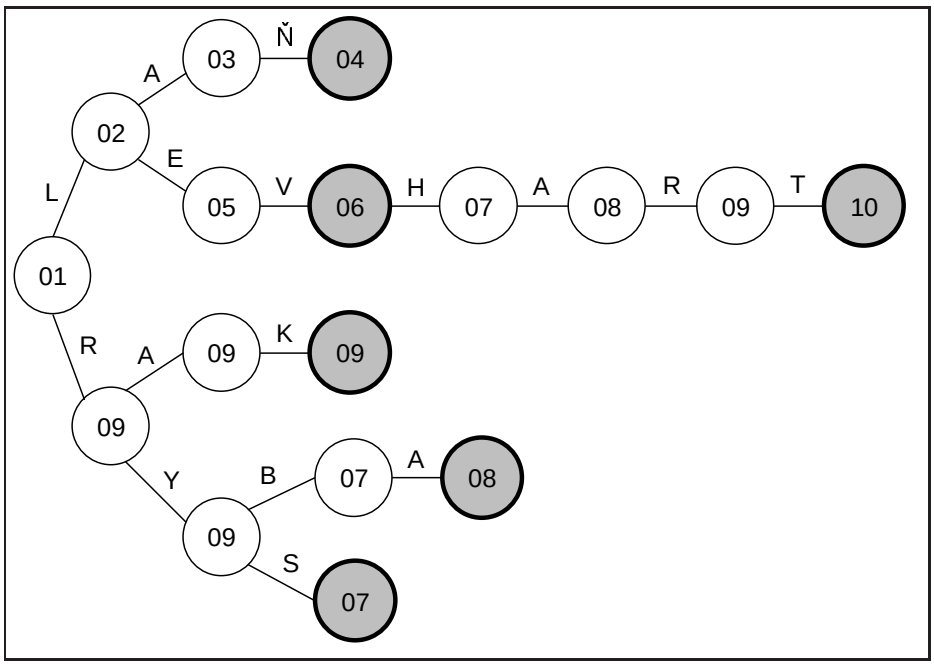
  $$member(v) = \begin{cases} true \ldots v\ reprezentuje\ slovo \in \mathbf{S} \\ false \ldots v\ reprezentuje\ slovo \notin \mathbf{S}. \end{cases}$$

  Více o TRIE se lze dozvědět například v [12]. Příklad TRIE je na obrázku 2.2.

  Slovník lze implementovat pomocí TRIE, z jehož uzlů s hodnotou *member* rovnou *true* vedou odkazy na příslušné seznamy indexových záznamů (do souboru indexových záznamů).

  Je-li slovník implementován pomocí jakékoli datové struktury takové, že indexové záznamy nejsou přímo její součástí, potom je soubor indexových záznamů typicky fyzicky uspořádaný tak, že seznam pro každý term zabírá souvislý úsek tohoto souboru, v němž jednotlivé seznamy indexových záznamů následují jeden za druhým. Je evidentní, že z takového souboru nelze efektivně mazat ani

[6] Tj. TRIE, ve kterém jsou uložena právě slova obsažená v $\mathbf{S}$.



Obrázek 2.2: Příklad *trie* reprezentujícího slova laň, lev, levhart, rak, ryba a rys. Tmavší barva uzlu naznačuje, že pro tento uzel nabývá funkce *member* hodnoty *true*.

do něj vkládat nové záznamy – aktualizační operace vyžadují tedy kompletní přebudování indexu. Výhodou ovšem je, že záznamy každého termu mohou být na disku pohromadě, takže diskové hlavy se nemusejí nadměrně pohybovat.

V dalších kapitolách bude ukázána dynamická implementace tohoto souboru, která rovněž udržuje záznamy jednoho každého termu na disku pohromadě, ale umožňuje navíc provádět aktualizační operace bez nutnosti přebudovat datovou strukturu.

2.3.1. Konstrukce invertovaného souboru

Existuje více algoritmů konstrukce invertovaného souboru pro danou množinu dokumentů a indexovaných termů. Protože však cílem této práce je prezentovat nový, dynamický algoritmus, bude zde uveden pouze základní princip klasické konstrukce, a to bez bližšího popisu.

Uvažujme invertovaný soubor implementovaný pomocí uspořádaného sekvenčního souboru. Základní postup jeho konstrukce je následující:

1. Projdi všechny dokumenty indexované kolekce dokumentů a zaznamenej všechny relevantní termy včetně jejich pozice v textu (je-li třeba – minimálně je však nutné ukládat jednoznačnou identifikaci dokumentu). Získaný seznam termů je uspořádán podle pozic termů v dokumentech.

2. *Inverze seznamu* – lexikograficky setřiď seznam podle termů. Je vhodné použít

stabilní třídící algoritmus,[7] neboť je nutné, aby výskyty jednoho termu byly uspořádány dle jejich pořadí v kolekci dokumentů. Z tohoto kroku je vidět původ názvu *invertovaný soubor*.

3. *Postprocessing* – spočítání vah a dalších informací, komprese indexu atd.

Je vidět, že konstrukce indexu, zejména pro větší textové kolekce, je časově velice náročná operace. Existuje i lineární algoritmus (popsaný např. v [2]) – jeho součástí není třídění, musí však 5-krát přečíst všechny dokumenty v kolekci. Ovšem i algoritmus, který by kolekci musel přečíst pouze jednou a přitom byl lineární, bude pro reálně rozsáhlé kolekce čas nezbytný pro konstrukci invertovaného souboru příliš velký na to, aby bylo možné index efektivně přebudovávat po každé aktualizaci, případně sérii aktualizací. Pro implementaci „dynamických“ DIS je tedy třeba navrhnout efektivnější strukturu.

2.3.2. Efektivita, výhody a nevýhody invertovaných souborů

Mnoho poznámek o efektivitě invertovaných souborů bylo uvedeno již v předchozích odstavcích. Nyní budou stručně shrnuty.

Na výpočetní složitost operací nad invertovanými soubory je třeba nahlížet z několika pohledů, a sice z pohledu konstrukce, dotazování a aktualizace. Konstrukce byla diskutována v předchozím odstavci a není třeba se k ní vracet. Poznamenejme jen, že asymptoticky nenajdeme jinou strukturu pro index, která by byla efektivnější pro jeho kompletní vybudování/přebudování (textovou kolekci je třeba přečíst vždy a lineární algoritmus pro invertované soubory existuje).

Dotazování nad invertovanými soubory může při vhodné implementaci být velmi efektivní. Záleží především na tom, zda indexové záznamy jednotlivých termů budou na disku udržovány dostatečně „pohromadě“, aby se předešlo nadměrným posunům diskových hlav (čas nutný pro načtení diskového bloku s posunem hlavy bývá řádově větší než čas načtení bloku při sekvenčním čtení).

Aktualizace invertovaných souborů představuje problém. Při použití klasických struktur lze volit mezi dvěma možnostmi: buď držet indexové záznamy jednoho termu pohromadě – potom aktualizace půjde dělat pouze přebudováním indexu; nebo se tohoto požadavku vzdát (použít např. prefixový B-strom, jehož klíče budou záznamy obsahující term spolu s jeho umístěním) – a tím výrazně snížit efektivitu dotazování. V dalších kapitolách bude prezentována datová struktura, která kombinuje oba tyto požadavky, tedy jednak udržuje záznamy jednotlivých termů pohromadě a jednak umožňuje efektivní provádění aktualizačních operací.

Co se týče slovníku, vzhledem k jeho obvyklé velikosti nebývá problém jej držet v hlavní paměti a ušetřit tak několik přístupů na disk při každém vyhledávání seznamu indexových záznamů pro nějaký term.

[7] Stabilní třídící algoritmus zachovává pořadí prvků, které považuje v rámci uspořádání za stejné.

Zbývá poznámka o paměťové náročnosti invertovaných souborů. Samozřejmě záleží na tom, jakou granularitu zvolíme pro indexové záznamy a jakou kompresi použijeme (pokud vůbec). Literatura uvádí, že velikost indexu implementovaného pomocí invertovaných souborů se v praxi pohybuje mezi 10 % – 300 % velikosti textové kolekce.

2.4. Optimalizace dotazů

Optimalizaci dotazů lze chápat jako dodatečné předzpracování dotazů, případně rozhodování o jejich konkrétním způsobu provedení, které vede ke zvýšení efektivity dotazování. Protože zkoumání ani implementace optimalizace dotazování není předmětem této práce, nabízí tato podkapitola pouze stručný přehled o dané problematice.

V kontextu invertovaných souborů[8] se dá hovořit o optimalizaci ve dvou hlavních oblastech:

- v elementárních operacích, tzn. optimalizace získání seznamu indexových záznamů pro daný term;
- v rámci komplexního dotazu, tzn. optimalizace pořadí prováděných operací, změna asociativity (uzávorkování) apod.

Elementární operace jsou v zásadě typicky optimalizovány již z principu struktury invertovaných souborů, neboť – jak již bylo popsáno – indexové záznamy každého konkrétního termu bývají fyzicky na disku uloženy pohromadě. Optimalizovat index z tohoto hlediska tedy znamená co nejefektivněji načíst příslušné diskové bloky. Protože sekvenční čtení z externí paměti příliš urychlit nelze (je závislé na fyzických parametrech daného zařízení), je nutné zajistit, aby příslušné záznamy byly uloženy na co nejmenším počtu bloků. Tyto metody se nazývají *komprese indexu* a zabývá se jimi kapitola 6.

Optimalizace komplexních dotazů může být při reálném použití zcela zásadní pro dosažení přijatelné odezvy systému. Jedná se o ekvivalentní úpravy stromu dotazu tak, aby se co nejvíce redukoval objem mezivýsledků. Přitom rozdíly v těchto objemech se mohou pohybovat i v mezích několika řádů.

2.4.1. Příklad

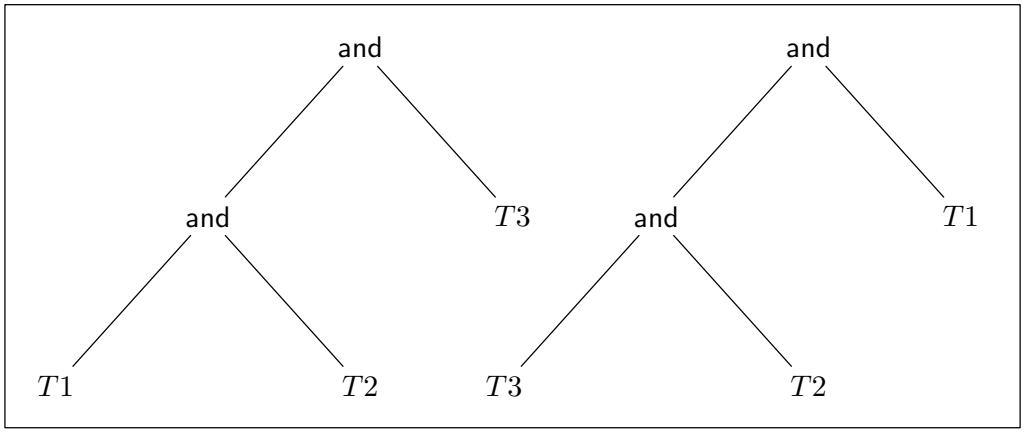
Následující příklad ilustruje, jaký vliv na efektivitu dotazu může mít pouhá dobrá nebo špatná volba pořadí zpracování jednotlivých logických operací v dotazu (tj. různé uzávorkování dotazu). Předpokládejme dotaz tvaru

$$T1 \; and \; T2 \; and \; T3,$$

[8] Od tohoto místa nepojednává tento text o jiných typech indexů než o invertovaných souborech, pokud nebude uvedeno jinak.

kde term $T1$ se vyskytuje v 50 000 dokumentech ($D1 - D50000$), $T2$ rovněž v 50 000 dokumentech ($D1001 - D51000$) a konečně $T3$ v deseti dokumentech $D2001 - D2010$. Dále předpokládejme, že do jednoho diskového bloku se vejde 500 indexových záznamů.

Pro jednoduchost uvažujme pouze dva možné tvary stromu dotazu, vzniklé ekvivalentními úpravami na základě komutativního a asociativního zákona, odpovídající výrazům ((*T*1 *and* *T*2) *and* *T*3), resp. ((*T*3 *and* *T*2) *and* *T*1). Oba tyto stromy jsou znázorněny na obrázku 2.3.



Obrázek 2.3: Příklady dvou různých stromů pro dotaz $T1$ *and* $T2$ *and* $T3$

Jak je patrné, v prvním případě (strom vlevo) jsou nejprve načteny záznamy pro termy $T1$ a $T2$, tj. celkem 100 000 indexových záznamů. Tyto jsou následně slity operací *merge* (viz algoritmus na straně 13), jejímž výsledkem je seznam L_1 velikosti 49 000 záznamů. Poté je načteno 10 záznamů termů $T3$, které musí být slity se seznamem L_1 do výsledného seznamu 10 záznamů. Celkem tedy z externí paměti muselo být načteno přibližně 200 diskových bloků a při slévání seznamů vzato do úvahy více než 100 000 indexových záznamů.

Oproti tomu v případě druhého stromu (vpravo) jsou nejprve načteny záznamy pro $T2$ a $T3$ (přitom vzhledem k omezenému výskytu $T3$ se lze vyhnout i načítání naprosté většiny diskových stránek pro $T2$ – například za pomoci dodatečných indexových struktur udávajících minimální a maximální čísla dokumentů v každém diskovém bloku). V ideálním případě se tedy jedná o načtení pouhých dvou diskových bloků. Ze stejných důvodů lze vystačit s načtením jediného bloku pro term $T1$. V porovnání s první variantou stačí tedy načíst celkem 3 diskové bloky z invertovaného souboru a při slévání vzít v úvahu řádově desítky indexových záznamů.

2.4.2. Optimalizační algoritmy

Jak již bylo uvedeno, způsob optimalizace naznačený v předchozím příkladu není předmětem této práce. Proto zde bude uvedeno pouze několik zcela elementárních implementačních poznámek.

Je okamžitě patrné, že tvar stromu vedoucí k co nejoptimálnějšímu dotazu je závislý na konkrétních datech. Čím detailnější informace o distribuci indexových záznamů bude při optimalizaci uvažována, tím lepší strategii vyhodnocení dotazu může optimalizátor sestavit. Zároveň s tím ovšem roste složitost vlastní optimalizace, a to teoreticky až na úroveň, kdy se tato stane kontraproduktivní. Evidentně je proto vždy nutné stanovit kompromis, kterým typicky může být nějaká heuristika. Prakticky by se mohlo jednat například o následující možnosti:

- brát v úvahu pouze velikosti seznamů indexových záznamů a slévat přednostně co nejkratší z nich (v případě, že se dané termy v dotazu vyskytují v negaci, tak naopak co nejdelší);
- uvažovat „nejmenší“ a „největší“ záznamy (z hlediska jejich uspořádání v seznamech indexových záznamů) v daných seznamech a slévat v takovém pořadí, aby výsledné seznamy byly co nejkratší;
- pro distribuci jednotlivých termů si pamatovat histogramy a na jejich základě slévat tak, aby výsledné seznamy byly co nejkratší.

Typickou metodou, jak nalézt v praxi co nejlepší variantu stromu, je *dynamické programování*, které spočívá v hledání optimálních dílčích řešení pro malé poddotazy, ze kterých jsou sestavována nejlepší řešení pro dotazy větší, až do nalezení řešení pro celý původní dotaz. Pro optimalizaci dotazů obsahujících větší počet termů mohou být místo dynamického programování efektivnější různé *randomizované algoritmy*, které spočívají v *náhodných procházkách* po grafu, jehož uzly jsou různé stromy ekvivalentní zadanému dotazu.

Více informací o zmíněných metodách optimalizace lze nalézt například v [4].

2.5. Zipfův zákon

Zipfův zákon (viz např. [2]) představuje model rozložení četností jednotlivých termů v dokumentu (a potažmo v celé textové kolekci).

Seřaďme všechny termy obsažené v daném dokumentu/kolekci sestupně dle jejich četností výskytu, označme t_j j-tý term v této výsledné posloupnosti a f_j jeho četnost. Zipfův zákon je potom dán vztahem

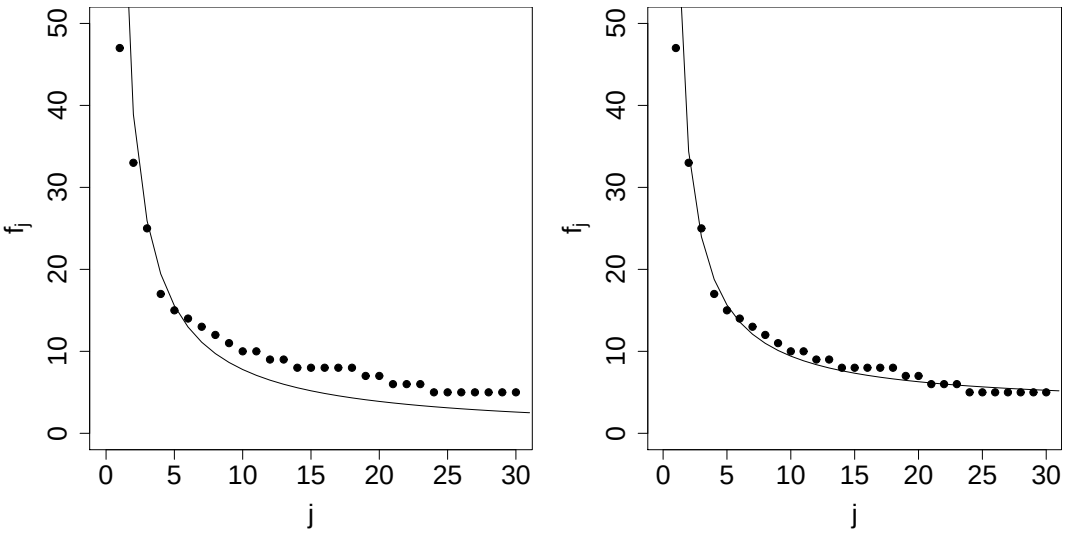
$$f_j \cdot j = const,$$

kde *const* je nějaká konstanta, která se pro různé dokumenty (kolekce) liší. Je-li však N počet všech výskytů všech slov v textu, pak se (viz [2]) například pro angličtinu hodnota *const* blíží $N/10$.

Označíme–li dále $p_j = f_j/N$, pak Zipfův zákon lze zapsat také následujícím způsobem:

$$p_j \cdot j = A,$$

kde hodnota A se pro angličtinu blíží $0,1$ (a nezávisí na konkrétním dokumentu).



Obrázek 2.4: Ilustrace Zipfova zákona

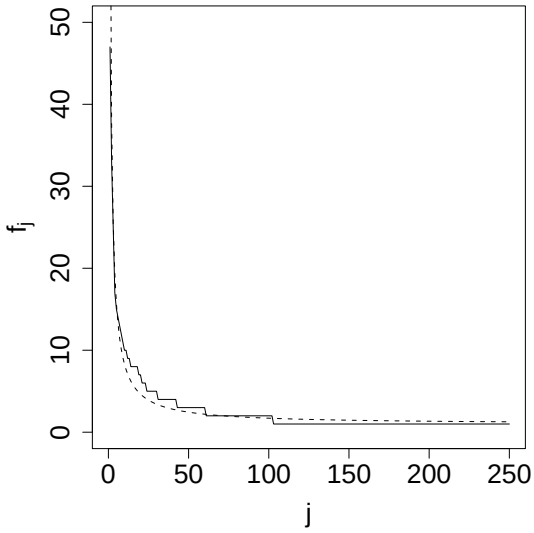
Graficky je Zipfův zákon ilustrován obrázkem 2.4. Jednotlivé tečky znázorňují četnosti třiceti nejčetnějších termů (uspořádáno podle klesajících četností) v úvodní kapitole tohoto textu.[9] Daty je navíc proložena část hyperboly znázorňující ideální rozložení četností podle Zipfova zákona. Hyperbola byla získána lineární regresí metodou nejmenších čtverců. V levém obrázku je prokládána křivka ve tvaru $const/j$ (tj. přesně podle znění Zipfova zákona), v obrázku pravém je pro zajímavost ukázáno, o co věrněji by skutečnost vystihoval Zipfův zákon modifikovaný na tvar

$$\frac{const_1}{j} + const_2,$$

kde $const_1$ a $const_2$ jsou konstanty optimalizované lineární regresí. Implementaci dynamického invertovaného souboru lze snadno modifikovat podle obou variant Zipfova zákona, stejně jako podle libovolné jiné vyčíslitelné reálné funkce.

Obrázek 2.5 znázorňuje to samé jako obrázek předchozí, ale je vygenerován pro veškeré významové termy z úvodní kapitoly, nikoli pouze pro prvních třicet. Plná čára znázorňuje původní data, přerušovaná je potom regresní křivka.

[9]Ve skutečnosti se jedná o prvních třicet nejčetnějších termů, které zbyly po ručním vyřazení „nevýznamových“ slov, jako jsou předložky, spojky apod. Dále byly sloučeny termy se stejným kořenem, lišící se například v případě podstatných jmen jen pádem. Stručně řečeno, zbylá slova jsou entity, které by se v praktickém DIS pravděpodobně skutečně indexovaly.



Obrázek 2.5: Ilustrace Zipfova zákona

Kapitola 3

Návrh dynamického univerzálního indexu

3.1. Zobecnění funkce indexu

Zatím bylo předpokládáno, že index je datová struktura, která danému termu přiřadí množinu dokumentů, ve kterých se tento term vyskytuje (nebo také boolskému výrazu složenému z termů přiřadí množinu dokumentů, které vyhovují danému výrazu – bez újmy na obecnosti však pro účely tohoto zobecnění lze uvažovat pouze boolský výraz složený z jediného atomu, rozšíření na obecný výraz je evidentní).

Obecnější pohled spočívá v tom, že indexované objekty nejsou nutně termy, ale tzv. *i–entity* (indexovatelné entity) – to jsou libovolné identifikátory, na jejichž množině existuje dobré uspořádání. Z pohledu indexu není podstatné, co tyto identifikátory identifikují; v případě DIS se typicky jedná o identifikátory termů. Prakticky tedy v případě „klasického“ použití indexu pojem termu a i–entity splývá.

Podobným způsobem pojem dokumentů zobecníme na *t–entity* (textové entity). I na množině t–entit musí existovat dobré uspořádání.

Na index (na nejnižší úrovni) lze potom nahlížet jako na funkci

$$idx : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{T}),$$

kde $\mathbf{I}$ je označuje množinu i–entit, $\mathbf{T}$ množinu všech t–entit a $\mathcal{P}(\mathbf{T})$ potenční množinu množiny $\mathbf{T}$.

Vzhledem k tomu, že i–entity i t–entity jsou pouze jakési (numerické) identifikátory, je zřejmé, že jejich role v indexu lze libovolně zaměňovat a získat tak pomocí jediné implementace indexu datovou strukturu mimo jiné například pro realizaci následujících vztahů (předpokládejme zde, že $\mathbf{I}$ je množina identifikátorů termů a $\mathbf{T}$ množina identifikátorů dokumentů):

- $idx_1 : \mathbf{T} \rightarrow \mathcal{P}(\mathbf{I})$ ... k libovolnému dokumentu přiřadí všechny termy, které se v něm vyskytují;

- $idx_2 : \mathbf{T} \rightarrow \mathcal{P}(\mathbf{T})$ ... umožňuje zachytit vztahy mezi dokumenty jako např. *citace*, *bibliografické párování* apod.;
- $idx_3 : \mathbf{I} \rightarrow \mathcal{P}(\mathbf{I})$ ... umožňuje zachytit vztahy mezi termy, např. *synonymitu*, *hierarchické uspořádání*[1] atd. a realizovat tak tzv. *tezaurus*.[2]

Toto zobecnění má dva důsledky:

Zaprvé, při použití indexu „klasickým způsobem“ dochází v podstatě k rozdělení slovníku na dvě části (samostatné slovníky). Jeden z nich, který umožňuje pro identifikátor i-entity získat údaje potřebné pro nalezení příslušného seznamu indexových záznamů, bude nadále považován za přímou součást indexu. Druhý slovník realizuje mapování konkrétního termu na její jednoznačný identifikátor. Tento slovník musí implementovat uživatel knihovny indexu, neboť ta nemůže předpokládat nic o tom, jaké entity budou indexovány. Tímto slovníkem se zde nebudeme zabývat, jeho implementace je výhradně v režii uživatele knihovny.

Druhý důsledek souvisí s použitím indexu pro jiný účel, např. pro implementaci funkcí idx_i, jak byly uvedeny výše. Ačkoli způsob implementace se nezmění, může se zásadně změnit např. účinnost komprese použité na tento index (ta může být založena na nějakém předpokladu o distribuci dat – to bude samozřejmě při použití indexu pro jiný účel jiné).

Vzhledem k předchozím zobecněním budeme navrhovaný index nazývat jako *univerzální*.

3.2. Požadavky na dotazování

Ještě před uvedením navrhované struktury univerzálního indexu je vhodné shrnout požadavky na dotazovací operace, které musí výsledný index splnit:

- Nalézt seznam všech t–entit vyhovujících boolskému výrazu složenému z i–entit. Za kontrolu, zda je výraz pozitivně omezující (viz výše), je zodpovědný uživatel knihovny.
- Navíc vybrat jen záznamy splňující nějakou podmínku na hodnoty zadaných položek v indexových záznamech. Přípustné podmínky a způsob jejich zadávání je zdokumentován v popisu rozhraní knihovny indexu (viz sekce A).
- Navíc vybrat n prvků s maximální, popř. minimální, hodnotou dané položky indexového záznamu (např. prvních 20).
- Navíc vybrané položky indexových záznamů dodat na výstup.

[1] Například zachytí, že term *zvíře* je hypertermem termu *kočka*. Dokumentografický systém potom může na dotaz požadující dokumenty o zvířatech vrátit i ty, ve kterých se vyskytuje slovo *kočka*, ale ne *zvíře*.

[2] Datová struktura, která zachycuje v DIS nějaký druh vztahu mezi indexovanými termy.

3.3. Dynamický invertovaný soubor

Tato část prezentuje datovou strukturu univerzálního indexu, která udržuje indexové záznamy jednotlivých termů pohromadě a zároveň umožňuje efektivně dynamicky provádět aktualizační operace. Tuto strukturu budeme nazývat také *dynamický invertovaný soubor*. Princip dynamického invertovaného souboru se poprvé objevil v [3, 6].

3.3.1. Návrh struktury univerzálního indexu

Navrhovaná efektivní implementace indexu má následující strukturu:

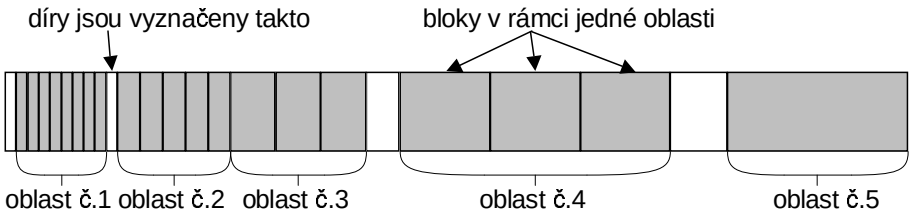
- *Slovník*. Realizuje mapování identifikátorů i–entit na údaje identifikující příslušné seznamy indexových záznamů. Konkrétní podoba těchto údajů bude zřejmá po popisu souboru indexových záznamů. Neříkáme nic o způsobu implementace slovníku; vzhledem k možnosti uchovávat jej v hlavní paměti efektivita slovníku není pro index příliš podstatná.
- *Soubor indexových záznamů*. V tomto textu budeme pro soubor indexových záznamů používat označení *dynamický invertovaný soubor*. Je podrobně popsán v následujícím textu.

Struktura dynamického invertovaného souboru je znázorněna na obrázku 3.1. Skládá se z *oblastí* (areas) a *děr* (gaps). Každá oblast je spojením předem neurčeného počtu bezprostředně navazujících *bloků* (blocks). Každý blok obsahuje seznam indexových záznamů pro jednu konkrétní i–entitu.

Oblasti jsou souvislé části souboru (neobsahují žádné díry), které obsahují jednak data a jednak určitý podíl nevyužitého prostoru. *Díry* jsou rovněž souvislé úseky souboru, které však neobsahují žádná užitečná data.

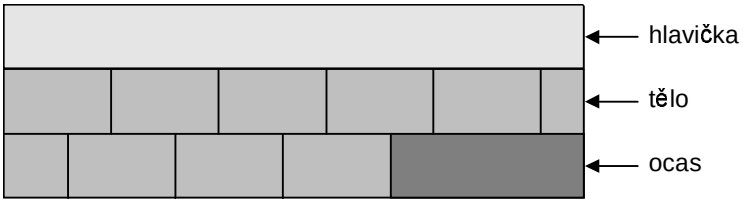
Soubor indexových záznamů po celou dobu své existence musí splňovat následující invarianty:

1. Mezi bloky v jedné oblasti nesmí být žádný nevyužitý prostor (díra).
2. Počet bloků v každé oblasti může být libovolné přirozené číslo včetně nuly.
3. Všechny bloky v téže oblasti musejí mít stejnou velikost.
4. Velikost každého bloku v i–té oblasti l_i je $l_i = l_{(i-1)} \cdot \kappa$, kde l_0 je minimální velikost bloku (tj. taková, aby se do něj vešla hlavička bloku – viz dále – a alespoň nějaká užitečná data) a κ je parametr indexu. Platí tedy $l_i = l_0 \cdot \kappa^i$. Defaultně použité hodnoty jsou $\kappa = 1/0,84$ a $l_0 \geq 20B$, pomocí konfigurace indexu je však lze měnit.
5. Mezi každými dvěma oblastmi, ale také na začátku souboru (před první oblastí), může být maximálně jedna díra (jedna nebo žádná). Na konci souboru (za poslední oblastí) díra být nesmí.



Obrázek 3.1: Struktura invertovaného souboru

Struktura jednotlivých bloků je znázorněna na obrázku 3.2. Skládají se z *hlavičky* (head), *těla* (body) a *ocasu* (tail). Hlavička obsahuje různé řídící informace, jako je např. délka těla v bitech a identifikátor i–entity příslušné k seznamu indexových záznamů obsaženému v tomto bloku. Dále může obsahovat parametry kompresního algoritmu pro tento blok a další pomocné údaje. V těle bloku jsou soustředěny vlastní indexové záznamy, které za sebou bezprostředně následují. Ocas je nevyužitý prostor, který vyplňuje zbytek velikosti bloku (neboť všechny velikosti bloků musí být hodnotou výrazu $l_0 \cdot \kappa^i$ pro nějaké i, jak bylo uvedeno výše).



Obrázek 3.2: Struktura bloku. Obdélníky uvnitř těla představují jednotlivé indexové záznamy.

Volba exponenciální funkce pro výpočet velikostí bloků reflektuje *Zipfův zákon*, který byl uveden v 2.5. Výběrem parametru κ lze tedy řídit přesnost, jak velikosti bloků odpovídají jednotlivým velikostem seznamů indexových záznamů. Menší hodnoty κ mají za důsledek

- menší velikosti ocasů (nevyužitých konců bloků), tj. i menší prostor zabraný dynamickým invertovaným souborem;
- častější potřebu spuštění algoritmu expanze bloků (přesun bloku do oblastí s vyšším pořadovým číslem, viz strana 30) – přitom se ale obecně během expanzí přesouvá menší množství dat.

Obecná myšlenka taková, že vzhledem k Zipfovu zákonu budou oblasti s nízkým pořadovým číslem obsahovat větší množství bloků s krátkými seznamy indexových záznamů, zatímco oblasti s vysokým číslem budou obsahovat málo bloku s dlouhými seznamy. Celkově potom velikosti oblastí budou řádově vyrovnané.

Zbývá dodat, jak vypadá struktura, na kterou slovník mapuje identifikátory i–entit. Jedná se o dvojici

$$\langle\, n \;;\; o \,\rangle,$$

kde n značí pořadové číslo oblasti, ve které se nachází blok se seznamem indexových záznamů pro danou i–entitu. o je potom offset začátku dat příslušného bloku v souboru indexových záznamů.

3.3.2. Základní algoritmy nad univerzálním indexem

Následující část popisuje algoritmy základních operací nad univerzálním indexem – dotazování, vkládání a mazání záznamů.

3.3.3. Získání seznamu indexových záznamů

Nejjednodušší operací je *získání seznamu indexových záznamů*. Jedná se o součást procedury dotazování. Způsob zpracování dotazu jako celku byl uveden výše (viz procedura rmerge na straně 13) – ten však neřešil právě způsob získání seznamu indexových záznamů pro danou i–entitu. Tento dílčí algoritmus je proto uveden na následujících řádcích:

Vstup: identifikátor i–entity, případně seznam různých omezení **C** na hodnoty položek indexových záznamů

Výstup: seznam indexových záznamů této i–entity, jejíž hodnoty odpovídají omezením **C**

1 Pomocí slovníku najdi offset bloku B příslušného k i–entitě na vstupu;

2 Načti do paměti obsah bloku B, přitom proveď všechny transformace potřebné k získání seznamu indexových záznamů (dekomprese atd.). Na jednotlivé indexové záznamy aplikuj všechna omezení ze seznamu **C**;

3 Zapiš výsledný seznam na výstup.

Algoritmus 2: Algoritmus Query

Na proceduru rmerge na straně 13 a na předchozí algoritmus se dá nahlížet jako na dva algoritmy na dvou různých úrovních. Přitom pouze získání seznamu indexových záznamů musí být implementováno přímo knihovnou indexu – zpracování dotazu by s využitím tohoto algoritmu mohl klidně provádět i uživatel knihovny.

Přesto univerzální index sám implementuje i samotné zpracování dotazu. Uživatel ovšem tuto funkcionalitu nemusí využívat; zpracování celých dotazů přímo indexem má výhody i nevýhody. Nesporná výhoda je, že index má, narozdíl od uživatele, možnost si vést interní statistiky o distribuci dat ve svých souborech a může na jejich základě poskytnout efektivnější optimalizaci dotazu.[3]

Nevýhodou je, že při zpracování dotazů indexem má uživatel menší volnost v přizpůsobení zpracování dotazů svým potřebám. Dále potom je nutné předzpracovat

[3]Optimalizace dotazů však není předmětem této práce.

dotaz do struktury přesně definované rozhraním knihovny (parsing dotazu univerzální index neprovádí – tím by uživateli zbytečně vnucoval nějakou předem danou syntax).

Ještě před popisem jednotlivých algoritmů uveďme několik pomocných definic:

Definice 3.3.1: *Definujme zobrazení*

$$blocksize \ : \ \mathbb{N} \rightarrow \mathbb{N},$$

které přirozenému číslu n přiřadí nejmenší velikost bloku, do kterého se vejdou data velikosti alespoň n. Číslo n vyjadřuje velikost dat s uvážením faktorů, které velikost bloku mohou změnit, jako je například komprese atd.

Za předpokladu, že se distribuce velikostí bloků v oblastech bude držet zipfova zákona (viz sekce 2.5.), bude funkce $blocksize(n)$ nabývat hodnoty nejmenšího přirozeného čísla, pro které platí

$$l_0 \cdot \kappa^i \geq n.$$

Pro rozšířený Zipfův zákon ve tvaru

$$f_j = \frac{const_1}{j} + const_2$$

bude pak výsledkem $blocksize(n)$ nejmenší přirozené číslo, pro které platí

$$l_0 \cdot \kappa^i + \lambda \geq n.$$

κ i λ jsou v obou případech parametry univerzálního indexu.

Definice 3.3.2: *Definujme zobrazení*

$$blockarea \ : \ \mathbb{N} \rightarrow \mathbb{N}$$

jako index oblasti, do které patří blok, kam se vejdou data velikosti dané v parametru zobrazení (jedná se zároveň o hodnotu exponentu i ve výrazu κ^i).

3.3.4. Vkládání

Následující dva algoritmy dohromady umožňují již dříve zmíněné inkrementální vkládání do indexu bez nutnosti jeho přebudování. Druhý z nich – algoritmus expanze bloku – je využíván algoritmem Insert jako ošetření speciálního případu, kdy pro danou i–entitu již v indexu existuje seznam indexových záznamů, ale po přidání nových záznamů se výsledek již nevejde do původního bloku. Všechny ostatní případy řeší přímo algoritmus Insert.

Poznámka 3.3.3: Algoritmy pro vkládání očekávají na svém vstupu mimo jiné i seznam identifikátorů t–entit (te_list), které obsahují výskyty příslušné i–entity (rovněž na vstupu). Bez újmy na obecnosti budeme o tomto seznamu mluvit někdy i přímo jako o seznamu indexových záznamů (který z něj lze snadno zkonstruovat).

Jako první je uveden algoritmus Insert:

Vstup: identifikátor i-entity (*iie*); seznam identifikátorů t–entit k zaindexování (*te_list*).

Výstup: prázdný

1 Pomocí slovníku se pokus najít pořadové číslo k bloku v rámci jeho příslušné oblasti a jeho offset *bofs* v souboru indexových záznamů. Pokud takový blok již existuje (tzn. že tato i–entita již má v indexu nějaké výskyty), pokračuj krokem 6. Jinak pokračuj krokem 2 (je indexován první výskyt této i–entity).

2 Vytvoř blok B o velikosti $blocksize(n)$, kde n je velikost seznamu *te_list* po aplikaci komprese a uvážení dalších faktorů, které mohou změnit velikost seznamu indexových záznamů po zapsání do bloku. Zapiš do B veškerá potřebná data (hlavičku, vkládaný seznam indexových záznamů, ...). Označme i hodnotu $blockarea(n)$.

3 Pokus se blok B vložit do díry těsně před oblastí s pořadovým číslem i (nejdříve ji vytvoř, pokud neexistuje). Pokud se to podařilo (díra byla dostatečně velká), algoritmus končí.

4 Pokus se blok B vložit do díry těsně za oblastí s pořadovým číslem i (podle skutečné velikosti bloku buď do oblasti i nebo $i + 1$. Pokud se to podařilo (díra byla dostatečně velká), algoritmus končí.

5 Vyjmi první blok B' následující neprázdné oblasti (její index může být větší než $i + 1$, jelikož některé oblasti mohou být prázdné), do vzniklé díry vlož blok B (do oblasti s indexem i), blok B' přejmenuj na B a vlož jej stejným algoritmem do oblasti $i + 1$ (tj. pokračuj znovu krokem 3, kde $i = i + 1$). Skonči.

6 Načti blok B na offsetu *bofs* souboru indexových záznamů. Zjisti, zda se seznam indexových záznamů již v bloku obsažených po slití se seznamem daným na vstupu vejde do tohoto bloku při zachování jeho velikosti. Pokud ano, zapiš B s novým (slitým) seznamem na jeho původní místo a skonči. Pokud ne, pokračuj krokem 2 algoritmu expanze bloku (viz dále) aplikovaným na blok B a na stejná vstupní data.

Algoritmus 3: Algoritmus Insert

Následuje algoritmus Expanze bloku:

Vstup: identifikátor i–entity (*iie*); seznam identifikátorů t–entit k zaindexování (*te_list*).

Výstup: prázdný

1 Předpoklady: blok B, do nějž mají být přidány indexové záznamy seznamu *te_list*, existuje, ale není dost velký, aby se do něj vešla výsledná data.

2 Pomocí slovníku najdi pořadové číslo k bloku v rámci jeho příslušné oblasti a jeho offset *bofs* v souboru indexových záznamů. Načti tento blok B a odstraň jej ze souboru indexových záznamů.

3 Pokud B nebyl blokem s nejvyšším indexem v rámci své oblasti (tzn. posledním), přesuň poslední blok této oblasti na původní místo bloku B.

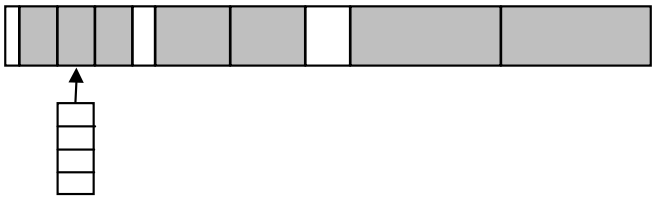
4 Vytvoř blok B' o velikosti $blocksize(n)$, kde n je velikost seznamu *te_list* slitého s původním seznamem v již existujícím bloku, to celé po aplikaci komprese a uvážení dalších faktorů, které mohou změnit velikost seznamu indexových záznamů po zapsání do bloku. Zapiš do B' seznam indexových záznamů vzniklý slitím seznamu z původního bloku B a vstupního seznamu.

5 Vlož do souboru blok B' stejným způsobem, jako kdyby dosud nebyl zaindexován (pomocí algoritmu Insert – krok 3).

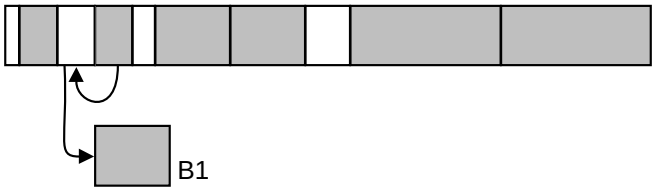
Algoritmus 4: Algoritmus Expanze bloku

Postup při vkládání do indexu je nyní ilustrován na jednom z mnoha možných scénářů. Předpokládejme, že v indexovém souboru jsou tři oblasti. První z nich přitom sestává ze tří bloků, zatímco zbylé dvě obsahují po dvou.

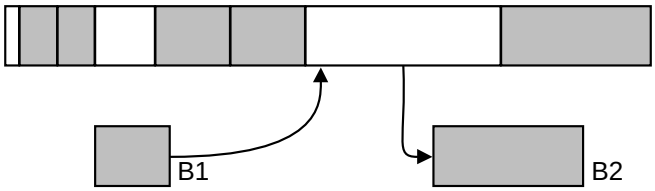
Do tohoto souboru jsou vkládány nové indexové záznamy pro term, který již je indexován a odpovídá mu druhý blok první oblasti. Navíc po slití nových záznamů se stávajícími se již výsledný seznam nevejde do bloku velikosti odpovídající této oblasti.



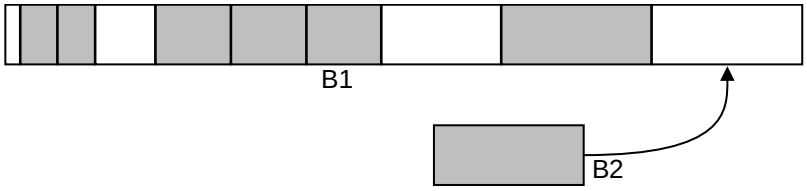
Proto je třeba tento blok (označme jej B_1) vyjmout a na jeho místo přesunout poslední blok první oblasti. Blok B_1 je tedy zvětšen a je do něj zapsán slitý seznam.



Mezi první a druhou oblastí vznikla díra, ale blok B se do ní nevejde. Nevejde se ani do díry za druhou oblastí. Proto je vyjmut první blok třetí oblasti (označme jej B_2) a do vzniklé díry na konec druhé oblastí vložen blok B_1.



Stejným postupem je nyní třeba vložit do souboru indexových záznamů právě vyjmutý blok B_2. Třetí oblast je však poslední a tedy je za ní dostatek místa, kam lze tento blok vložit a algoritmus tak ukončit.



3.3.5. Mazání

Budou uvedeny dva algoritmy mazání dat z indexu. První z nich odstraní všechny indexové záznamy spojené s danou i–entitou, což je užitečné například při zařazení nějakého termu na tzv. *stop-list* (seznam termů, které se neindexují).

Vstup: identifikátor i–entity (*iie*), která má být smazána

Výstup: prázdný

1 Pomocí slovníku se pokus k danému *iie* najít pořadové číslo i bloku B v rámci jeho příslušné oblasti a jeho offset *bofs* v souboru indexových záznamů. Pokud takový blok neexistuje, algoritmus končí.

2 Odstraň informace o dané i–entitě ze slovníku.

3 Pokud B nebyl poslední blok oblasti s indexem i, přesuň poslední blok na místo původního B.

4 Pokud za posledním blokem oblasti s indexem i vznikla díra velikosti alespoň l_j, kde j je pořadové číslo následující neprázdné oblasti (pro nejmenší možné j větší než i), přesuň do této díry poslední blok oblasti s indexem j. Jinak skonči algoritmus.

5 Polož $i = j$ a opakuj krok 4.

Algoritmus 5: Algoritmus Odstranění i–entity (tj. všech jejích výskytů)

Druhý algoritmus odstraňuje pouze specifikované indexové záznamy. Bez újmy na obecnosti lze předpokládat, že všechny mazané indexové záznamy náleží stejné i–entitě.

Vstup: identifikace mazaných indexových záznamů (např. offset bloku a pořadové číslo záznamů, případně místo toho přímo atributy záznamů)

Výstup: prázdný

1 Pomocí slovníku se pokus podle dané identifikace bloku najít pořadové číslo i bloku B v rámci jeho příslušné oblasti (její index označme i_a) a jeho offset *bofs* v souboru indexových záznamů. Pokud takový blok neexistuje, algoritmus končí.

2 Zjisti, zda po smazání všech požadovaných indexových záznamů ze seznamu v bloku B je velikost bloku potřebného pro uchování výsledného seznamu stále větší než $l_0 \cdot \kappa^{i-1}$, resp. $l_0 \cdot \kappa^{i-1} + \lambda$ (podle verze Zipfova zákona). Pokud ano, ulož B na původní místo (po úpravě seznamu indexových záznamů) a skonči. V opačném případě pokračuj krokem 3.

3 Najdi nejmenší j takové, že seznam indexových záznamů z kroku 2 (tj. po vymazání požadovaných) lze uskladnit v bloku velikosti $l_j = l_0 \cdot \kappa^{j-1}$, resp. $l_0 \cdot \kappa^{j-1} + \lambda$. Vytvoř blok B' velikosti l_j a zapiš do něj seznam indexových záznamů z bloku B po odebrání těch požadovaných.

4 Zneplatni blok B. Zneplatnění je totéž jako smazání, ale vzniklá díra je zaplněna pouze v případě, že je uprostřed příslušné oblasti – v tom případě je do ní přesunut poslední blok této oblasti. Díry před i za oblastí jsou zatím ponechány nezaplněné.

5 Blok B' vlož do souboru jako kdyby příslušná i–entita nebyla dosud zaindexována.

6 Pokud je oblast s indexem i_a neprázdná a zůstala před ní díra velikosti větší nebo rovné l_{i_a}, přesuň na začátek i_a–té oblasti její poslední blok.

7 Pokud je oblast s indexem i_a poslední (největší) oblastí, odstraň případné volné místo za touto oblastí a skonči.

8 Pokud za oblastí nebo po oblasti s indexem i_a zůstala díra velikosti větší nebo rovné l_{j_a}, kde j_a je index následující neprázdné oblasti (j_a je nejmenší možný index větší než i_a), přesuň poslední blok oblasti s indexem než j_a na začátek této oblasti. Pokud se jedná o poslední blok této oblasti, přesuň jej libovolně celým jeho obsahem do díry.

9 Polož $i_a = j_a$ pokračuj krokem 7.

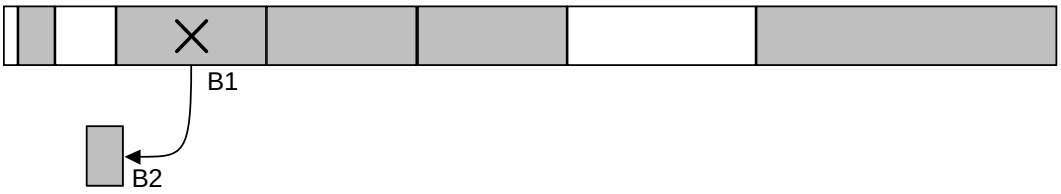
Algoritmus 6: Algoritmus Odstranění indexového záznamu

Podobně jako v případě vkládání nyní následuje ilustrace algoritmu mazání na

příkladě.

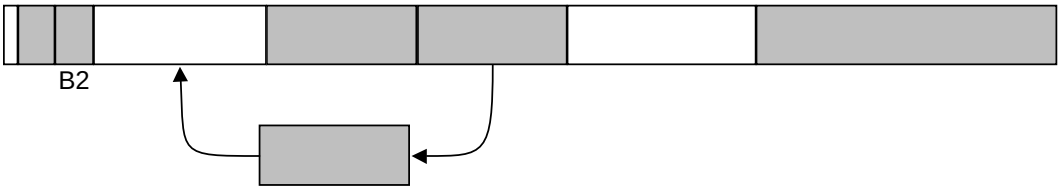
Předpokládejme, že neprázdné oblasti mají indexy 1, 3 a 4 (oblasti s indexy 0 a 2 jsou tedy prázdné). V první z nich je obsažen jeden blok, ve druhé tři a ve třetí opět jediný. Chceme smazat část indexových záznamů z prvního bloku oblasti s indexem 3 (na obrázku je vyznačen křížkem; označme jej B_1).

Po smazání příslušných indexových záznamů se potřebná velikost bloku, který pojme výsledek, zmenší na velikost odpovídající oblasti s indexem 1. Nejdříve je tedy vytvořen nový blok B_2 této velikosti a do něj je zapsán výsledný seznam indexových záznamů.

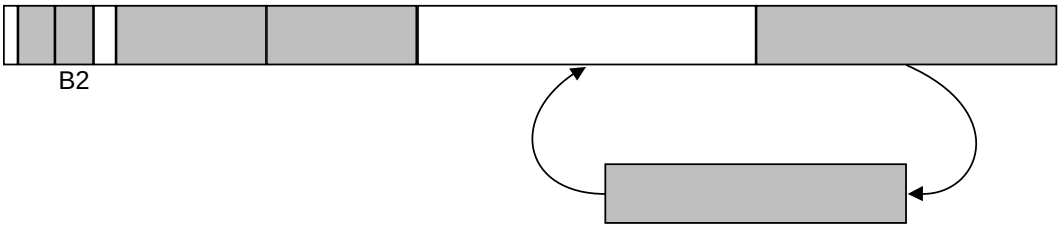


Dále je zneplatněn původní blok B_1. Po jeho odstranění vznikne díra na začátku druhé oblasti. Ta je zatím ponechána beze změny (to je rozdíl „zneplatnění“ a „smazání“ bloku). Blok B_2 je vložen do souboru, jako kdyby i–entita příslušná tomuto bloku nebyla dosud zaindexována. Toto vložení je v tomto případě triviální, neboť mezi první a druhou oblastí vznikla dostatečně veliká díra zneplatněním bloku B_2. Do ní se tento blok bez potíží vejde.

Je vidět, že před oblastí s indexem 3 (druhá v pořadí) vznikla příliš veliká mezera. Podle algoritmu Delete je do ní přesunut poslední blok téže oblasti.



Nyní se však vyskytuje příliš velká díra *za* toutéž oblastí. To lze napravit přesunutím posledního bloku (v tomto případě jediného) z oblasti s indexem 4 na začátek této oblasti. V tomto speciálním případě sice daný blok již na začátku oblasti byl, ale ve skutečnosti se přesune o jednu svou délku „dopředu“.



Protože právě byla zpracována poslední oblast v souboru, zbývá jen smazat zbytečné volné místo na konci souboru.

3.3.6. Vylepšení a rozšíření univerzálního indexu

V předchozí sekci prezentovaný návrh univerzálního indexu představil strukturu, která umožňuje efektivní dotazování (je zřejmé, že lze snadno implementovat tak, aby seznamy indexových záznamů byly uloženy v souvislých oblastech na disku) a zároveň aktualizaci. Z přehledu algoritmů je však vidět, že aktualizační operace jsou efektivní pouze *amortizovaně asymptoticky*. V nejhorším případě může aktualizační operace stále vyžadovat zpracování veškerých dat obsažených v indexu. To znamená, že uvedená struktura stále není vhodná pro zcela interaktivní použití (přestože zde je situace mnohem lepší než u klasických invertovaných souborů, které pro aktualizaci vyžadují přebudování indexu).

Dále současná implementace není odolná proti výpadkům systému, což by u „databázově orientovaných“ datových struktur dnes mělo být samozřejmostí.

Zlepšení doby odezvy v nejhorším případě lze relativně snadno dosáhnout například nějakou modifikací s použitím *oblastí přetečení*, případně tím, že se mezi oblastmi budeme záměrně snažit nechávat větší díry. Pro univerzální index však je použit jiný přístup vzniklý spojením řešení velké doby odezvy v nejhorším případě s požadavkem na odolnost proti výpadkům. Je použit tzv. *redo-log* (tj. log s odloženou realizací změn), který automaticky zároveň slouží jako přetoková oblast. Operace budou nejprve zapisovány do logu (což má další výhodu, že lze zapisovat sekvenčně, tzn. minimalizovat pohyby hlaviček disku a tudíž maximalizovat efektivitu) a poté, ve chvílích, kdy systém nebude příliš zatížen, realizovat změny uvedené v logu do skutečných struktur indexu. Obě možnosti vyžadují zavést zamykání bloků i ostatních struktur v indexu, čímž bude položen základ pro budoucí paralelně probíhající aktualizační operace (které však zatím nejsou cílem ani finální verze, neboť se předpokládá, že uživatel zajistí serializaci aktualizačních operací. Zajištění odolnosti proti výpadkům je rozebráno v kapitole 5.

Dalším rozšířením je implementace kompresních metod, které jednak šetří diskový prostor a jednak i nepřímo zlepšují efektivitu dotazování, neboť stačí načítat menší počet diskových stránek (viz kapitola 6).

Kapitola 4

Efektivita dynamického invertovaného souboru

4.1. Obecné předpoklady a označení

Podrobné teoretické zkoumání efektivity univerzálního indexu není cílem této práce. Přesto bude níže v této části, kromě evidentních fakt o složitosti elementárních dotazů, odvozen odhad amortizované složitosti operace `insert` nad univerzálním indexem. Operace `delete` je do značné míry analogická, takže lze předpokládat, že její složitost by se odvodila analogickým způsobem.

Úmluva 4.1.1: *Pro účely následujících tvrzení předpokládejme kvůli zjednodušení, že:*

1. *Indexové záznamy mají konstantní velikost r_s.*
   $\overline{r}_b$ potom označuje průměrný počet indexových záznamů, které se vejdou do jednoho diskového bloku.
2. *Bez újmy na obecnosti lze zanedbat velikost hlavičky bloku indexu.*
3. *V odhadech mohou být bez újmy na obecnosti uvažovány pouze operace nad souborem indexových záznamů.*[1]

Označení 4.1.2: *Nechť libovolný blok se nachází v oblasti s indexem i a obsahuje takový počet záznamů, že po odebrání jednoho z nich by musel být přesunut do oblasti s indexem $i-1$. Potom označení*

$$r_{\Delta[(i)\to(j)]}$$

[1] Adresář univerzálního indexu může být při reálném použití typicky uložen ve vnitřní paměti, a proto jeho podíl na celkové složitosti je zanedbatelný (jeho velikost je $2bp$ bytů, kde b je index největšího bloku a p velikost offsetu do souboru indexových záznamů; tedy pro $\kappa = 1/0,84$ a největší blok velikosti 1GB by se velikost adresáře pohybovala maximálně v řádu několika málo stovek kilobytů (roste logaritmicky v závislosti na maximální velikosti bloku uloženého v dynamickém invertovaném souboru). Slovník indexu lze implementovat tak, aby jeho složitost byla maximálně O(délka termu)).

pro přirozené $j > i$ značí nejmenší počet záznamů, po jejichž vložení již tento blok svou velikostí bude patřit do oblasti s indexem j.

Označení 4.1.3:

1. *l_0 značí počáteční velikost bloku (tj. velikost bloku náležícího k oblasti s nejmenším indexem).*
2. *κ značí koeficient růstu velikosti bloků, tj. konstantu takovou, že velikost bloku v oblasti s indexem i je $l_0\kappa^i$.*
3. *Konstanty d_{seek}, resp. d_{seq} označují maximální dobu nastavení diskové hlavy na stopu se začátkem požadovaných dat, resp. maximální dobu sekvenčního čtení diskové stránky v nejhorším případě. Případné posuny diskové hlavy na sousední stopu během sekvenčního čtení přitom budou zanedbány (resp. zohledněny v konstantě d_{seq}).*

Věta 4.1.4: *Nechť T je term, pro který je v univerzálním indexu uloženo s_T indexových záznamů. Potom složitost elemetárního požadavku na získání seznamu indexových záznamů měřená v počtu diskových bloků, které je nutno načíst, je v nejhorším případě rovna*

$$O(d_{seek} + \left\lceil \frac{s_T}{\overline{r}_b} \right\rceil \cdot d_{seq})$$

Důkaz: Zřejmý. □

Věta 4.1.5: *Složitost vkládání nového záznamu do univerzálního indexu má v nejhorším případě lineární časovou složitost vzhledem k délce souboru indexových záznamů, tj. $O(n_B)$, kde n_B je počet diskových bloků indexu.*

Důkaz: Zřejmý. Například pokud každá oblast obsahuje jediný blok indexu a zároveň mezi oblastmi nejsou žádné díry, může při vkládání do nejmenšího bloku dojít k přetečení, které bude mít za následek kaskádu operací *expanze bloku*. Během těch může dojít k načtení a opětovnému uložení až všech dat v souboru indexových záznamů. □

4.2. Časová analýza expanze bloku

Tato sekce se zaměřuje na analýzu chování operace expanze bloku. Jak již bylo uvedeno v předchozím textu, v nejhorším případě způsobí tato operace načtení a opětovné uložení veškerých dat obsažených v souboru indexových záznamů, tedy její složitost je lineární vzhledem k objemu zaindexovaných dat.

Zajímavější je ovšem analýza amortizované složitosti expanze bloku, tedy průměrná cena jedné operace expanze v posloupnosti těchto operací v nejhorším případě. V následujícím textu bude ukázáno, že tato cena je konstatní (a tedy nezávisí na množství dat v indexu).

Kvůli zjednodušení bude bez újmy na obecnosti následující text vycházet z těchto předpokladů:

- Nechť $j \in \mathbb{N}$ je index největší neprázdné oblasti obsažené v souboru indexových záznamů. Potom pro všechna $i \in \mathbb{N}, i < j$ platí, že oblast A_i je neprázdná.
- Omezíme se na posloupnost expanzí bloku iniciovaných přidáváním dalších bloků do oblasti s nejnižším indexem.
- Výchozí stav univerzálního indexu je takový, že mezi oblastmi A_i a $A_{(i+1)}$ pro každé i je díra velikosti alespoň $l_0\kappa^i$.
- Index je parametrizovaný parametrem $\kappa \in (1, 2)$.

Evidentně žádný z uvedených předpokladů nesníží asymptotickou složitost operace expanze bloku, takže horní odhad spočtený za těchto předpokladů bude tím spíše platit bez nich.

Definice 4.2.1: *Nechť A_i, resp. $A_{(i+1)}$, pro přirozené i jsou dvě po sobě následující oblasti v souboru indexových záznamů. Uvažujme operaci expanze bloku (viz algoritmus 4).*

Jestliže

- *poloha nebo velikost těchto oblastí není touto operací nijak ovlivněna, budeme říkat, že expanze bloku* nepřešla přes oblast A_i*;*
- *je ovlivněna pouze poloha nebo velikost oblasti A_i, budeme říkat, že expanze bloku se* zastavila mezi bloky A_i a $A_{(i+1)}$*, resp.* skončila za blokem A_i*;*
- *je ovlivněna poloha nebo velikost obou oblastí, budeme říkat, že expanze bloku* prošla přes oblast $A_{(i+1)}$.

Tvrzení 4.2.2: *Nechť $i, j, \delta \in \mathbb{N}$ jsou přirozená čísla taková, že $i = j - \delta$. Nechť A_i, resp. A_j, jsou dvě oblasti, které na sebe v souboru indexových záznamů bezprostředně navazují, tj. neexistuje žádné $k \in \mathbb{N}$ takové, že $i < k < j$ a A_k je neprázdná a dále velikost díry mezi nimi je nulová. Označme $c_e(\delta)$ počet expanzí bloku, které musejí projít přes tyto bloky, než mezi nimi nějaká expanze může skončit.*

Potom platí

$$c_e(\delta) = \left\lceil \frac{1}{\kappa^\delta - 1} \right\rceil .$$

Speciálně označme

$$c_e = c_e(1) = \left\lceil \frac{1}{\kappa - 1} \right\rceil .$$

Důkaz: Každá expanze bloku, která projde mezi oblastmi A_i a A_j $(i, j \in \mathbb{N};\ i < j)$, způsobí odebrání bloku ze začátku oblasti A_j (zvětšení díry mezi těmito oblastmi o $l_0\kappa^j$) a následně vložení bloku oblasti A_i (tedy zmenšení této díry o $l_0\kappa^i$). V součtu potom velikost díry při každé takové expanzi vzroste o $l_0\kappa^{(i+\delta)} - l_0\kappa^i$ pro $\delta = j - i$, tedy o

$$l_0\kappa^i(\kappa^\delta - 1).$$

Expanze může skončit mezi oblastmi A_i a A_j, jestliže se mezi těmito oblastmi vytvořila díra velikosti alespoň $l_0\kappa^i$. Platí přitom:

$$\begin{aligned} c_e(\delta)\, l_0\kappa^i(\kappa^\delta - 1) &\geq l_0\kappa^i \\ c_e(\delta) &\geq \frac{1}{\kappa^\delta - 1} \end{aligned}$$

Nejmenší přirozené číslo vyhovující této nerovnici je potom $\left\lceil \frac{1}{\kappa^\delta - 1} \right\rceil$. □

Konstanta c_e ve smyslu tohoto tvrzení bude používána i ve zbytku této kapitoly.

Podle 4.2.2 tedy alespoň každá c_e–tá expanze bloku se mezi těmito bloky zastaví. Ve skutečnosti zde může zastavit více expanzí, neboť při každém ukončení expanze mezi danými bloky se tam může kumulovat.

Definice 4.2.3: *Nechť c_e je konstanta ve smyslu definice 4.2.2 a i je přirozené číslo. Dále uvažujme univerzální index, kde h je index největší obsažené neprázdné oblasti. Potom funkce*

$$\varphi : \mathbb{N} \to \{0, 1, \ldots, c_e\}$$

se nazývá mapa velikosti děr v souboru indexových záznamů*, jestliže splňuje, že*

$$\varphi(n) = c_e$$

pro všechna $n \in \mathbb{N}$, $n > h$. Tato mapa reprezentující stav po provedení i–té expanze bloku se označuje symbolem φ_i.

Funkce $\varphi(i)$ nabývá hodnoty 0, jestliže velikost díry mezi oblastmi $A_{(i-1)}$ a A_i je menší než rozdíl velikosti bloků v těchto oblastech, hodnoty 1, jestliže velikost díry je alespoň tento rozdíl, ale méně než jeho dvojnásobek atd.

Předpokládejme, že hodnoty $\varphi_i(n)$ jsou definované pro všechna $n \in \mathbb{N}$. Nechť k je nejmenší přirozené číslo takové, že $\varphi_i(k) = c_e$. Potom hodnoty $\varphi_{(i+1)}$ definujme následujícím způsobem:

$$\varphi_{(i+1)}(n) = \begin{cases} \varphi_i(n) + 1, & x < k \\ 0, & x = k \\ \varphi_i(n), & x > k \end{cases}$$

Dále hodnoty φ_1 definujme jako $\varphi_i(n) = c_e$.

i	$\varphi_i(n)$**,** $n \in \{1,\ldots,7\}$						
1	2	2	2	2	2	2	2
2	0	2	2	2	2	2	2
3	1	0	2	2	2	2	2
4	2	1	0	2	2	2	2
5	0	1	0	2	2	2	2
6	1	2	1	0	2	2	2
7	2	0	1	0	2	2	2
8	0	0	1	0	2	2	2
9	1	1	2	1	0	2	2
10	2	2	0	1	0	2	2

Tabulka 4.1: Hodnoty φ_i pro $i \in \{1,\ldots,10\}$

Hodnoty funkcí φ_i a $\varphi_{(i+1)}$ modelují časově nejhorší průběh operace expanze bloku. Jestliže $\varphi_i(m) < c_e$, potom v díře mezi oblastmi A_i a $A_{(i+1)}$ není dostatek místa pro blok z oblasti A_i, a proto expanze přes tento blok projde dále. Přitom se příslušná díra zvětší o rozdíl velikosti bloku z $A_{(i+1)}$ (který přebírá úlohu expandovaného bloku) a bloku z A_i (který dosud byl expandovaným blokem a nyní je uložen ve zvětšené díře). Proto $\varphi_{(i+1)}(m) = \varphi_i(m) + 1$. Jestliže naopak $\varphi_i(m) = c_e$, může být právě expandovaný blok uložen do díry mezi oblastmi A_i a $A_{(i+1)}$ a expanze může být ukončena. Zde je třeba si povšimnout, že volné místo v díře může být o něco větší než velikost sem vkládaného bloku; přesto příslušná hodnota φ je definována jako 0. Toto zjednodušení modelu v těchto výpočtech nevadí, neboť znamená, že skutečný algoritmus expanze bloku může díky nakumulovanému volnému prostoru být ukončen po méně krocích, tedy že skutečný výsledek může být lepší než ten spočtený na základě tohoto modelu. Jinými slovy, pokud bychom upravili skutečný algoritmus expanze podle tohoto modelu tak, aby se zneplatnil volný prostor v díře, ve které byla ukončena expanze, tento upravený algoritmus by neměl lepší časovou složitost než ten původní.

Tabulka 4.1 pro ilustraci zobrazuje příklad definice funkcí φ_i pro několik i, pokud $c_e = 3$. Z tohoto příkladu je vidět, že hodnoty $\varphi_i(1)$ se pro jednotlivá i pravidelně střídají. Pro vyšší hodnoty parametru funkce již pravidelnost není příliš zřejmá. Pro další úvahy je však nutný nástroj, který umožní odhadnout kolik instancí algoritmu expanze skončí po kolika krocích.

K tomu je třeba následující značení:

Definice 4.2.4: *Pro $n \in \mathbb{N}$ definujeme funkci procentuální četnosti přenosů expanze $P : \mathbb{N} \to (0,1)$ vztahem*

$$P(n) = \lim_{j\to\infty} \frac{|\{i|(i \le j) \wedge (\forall m : m \in \mathbb{N} \wedge m \le n)(\varphi_i(m) < c_e)\}|}{j}.$$

Definice 4.2.5: *Pro $n \in \mathbb{N}$ definujeme funkci procentuální četnosti ukončení ex-*

panze $S : \mathbb{N} \to (0,1)$ *vztahem*

$$S(n) = \lim_{j\to\infty} \frac{|\{i|(i \leq j) \wedge (\forall m : m \in \mathbb{N} \wedge m < n)(\varphi_i(m) < c_e) \wedge (\varphi_i(n) = c_e)\}|}{j}.$$

Intuitivní interpretace definic 4.2.4 a 4.2.5 lze snadno vysvětlit pomocí tabulky 4.1. $P(n)$ označuje procento výskytu řádků i takových, pro které platí, že ve všech sloupcích $0, 1, \ldots, n$ je obsažena hodnota menší než c_e, tedy expanze bloku by *prošla* přes všechny příslušné oblasti. Naproti tomu $S(n)$ označuje procento výskytu řádků takových, pro které platí, že ve všech sloupcích $0, 1, \ldots, (n-1)$ je obsažena hodnota menší než c_e, zatímco ve sloupci c_e je hodnoda c_e. Tedy expanze bloku nad uspořádáním indexu reprezentovaným pomocí φ_i by se zastavila mezi oblastmi $A_{(n-1)}$ a A_n.

Lemma 4.2.6: *Pro funkce* S *a* P *a přirozené* $n > 1$ *platí následující soustava rovnic:*

$$S(n) = \frac{1}{c_e} \cdot P(n-1) \tag{4.1}$$

$$P(n) = P(n-1) - S(n) \tag{4.2}$$

Důkaz: Zřejmý dle definic 4.2.4, resp. 4.2.5, a definice hodnot φ. Neformálně:

Pro $n = 1$ zřejmě

$$S(1) = \frac{1}{c_e} \tag{4.3}$$

$$P(1) = 1 - \frac{1}{c_e} = \frac{c_e - 1}{c_e} \tag{4.4}$$

Pro $n = 2$ platí[2]

$$S(2) = \frac{1}{\frac{1}{P(1)} \cdot c_e} = \frac{1}{c_e} \cdot P(1)$$

Dále zřejmě platí $P(2) = 1 - S(2) - S(1)$, protože procentuální četnost přenosů přes druhý řád dostaneme odečtením procentuální četnosti expanzí, které skončily ve druhém nebo některém předcházejícím řádu, od jedničky.

Pro $n > 2$ analogicky dostáváme

$$S(n) = \frac{1}{\frac{1}{P(n-1)} \cdot c_e} = \frac{1}{c_e} \cdot P(n-1)$$

a

$$P(n) = 1 - \sum_{k=1}^{n} S(k).$$

[2]Výraz $1/P(1)$ vyjadřuje „každá kolikátá expanze má za následek přenos do 2. řádu“. Dle 4.2.2 aby se mohla nějaká pozice vynulovat, musí přes ni nejdříve projít c_e přenosů, což je důsledkem každé $(1/P(1)) \cdot c_e$–té operace expanze.

Protože z indukčního předpokladu víme, že

$$P(n-1) = 1 - \sum_{k=1}^{n-1} S(k),$$

dostáváme dosazením

$$P(n) = 1 - \sum_{k=1}^{n} S(k) = P(n-1) - S(n).$$

□

Lemma 4.2.7: *Pro přirozené n platí:*

$$S(n) = \left(1 - \frac{1}{c_e}\right)^{(n-1)} \cdot \frac{1}{c_e}. \tag{4.5}$$

Důkaz: Vyjádřením $S(n)$ z 4.1 a sečtením s 4.2 dostáváme:

$$\begin{aligned} S(n) &= \frac{1}{2}\left[\left(\frac{1+c_e}{c_e}\right)P(n-1) - P(n)\right] \\ &= \frac{1}{2}\left[(c_e+1)S(n) - c_e S(n+1))\right]. \end{aligned} \tag{4.6}$$

Pokračujeme dále vyjádřením $S(n+1)$ z 4.6 a rozepsáním výsledného rekurentního vztahu:

$$\begin{aligned} S(n+1) &= \left(1 - \frac{1}{c_e}\right) S(n) \\ S(n) &= \left(1 - \frac{1}{c_e}\right) S(n-1) \\ &= \left(1 - \frac{1}{c_e}\right)\left(\left(1 - \frac{1}{c_e}\right) S(n-2)\right) \\ &= \left(1 - \frac{1}{c_e}\right)\left(\left(1 - \frac{1}{c_e}\right)\left(\left(1 - \frac{1}{c_e}\right) S(n-3)\right)\right) \\ &\dots \\ &= \left(1 - \frac{1}{c_e}\right)^{(n-1)} S(1) \\ \text{(dle 4.3)} &= \left(1 - \frac{1}{c_e}\right)^{(n-1)} \cdot \frac{1}{c_e} \end{aligned}$$

□

Lemma 4.2.8: *Pro $c_e > 1$ představuje nekonečná posloupnost $(S(n))_{n=1}^{\infty}$ pravděpodobnostní rozložení, tj. platí:*

$$\sum_{n=1}^{\infty} S(n) = \sum_{n=1}^{\infty} \left(\left(1 - \frac{1}{c_e}\right)^{(n-1)} \cdot \frac{1}{c_e} \right) = 1$$

Důkaz:

$$\sum_{n=1}^{\infty} S(n) = \sum_{n=1}^{\infty} \left(1 - \frac{1}{c_e}\right)^{(n-1)} \cdot \frac{1}{c_e} = \frac{1}{c_e} \sum_{n=0}^{\infty} \left(1 - \frac{1}{c_e}\right)^{n} = \frac{1}{c_e} \cdot \frac{1}{1 - (1 - \frac{1}{c_e}))} = 1$$

□

Předpoklady lemmatu 4.2.8 jsou splněny díky předpokladu, že $\kappa \in (1, 2)$.

Definice 4.2.9: *Nechť $n, l_0, B \in \mathbb{N}$, $\kappa \in (1, 2)$. Potom funkce $\epsilon : \mathbb{N} \to \mathbb{N}$ je definována výrazem*

$$\epsilon(n) = \frac{l_0 \kappa^n}{B}.$$

Intuitivní interpretace předchozí definice je následující. l_0 a κ jsou dříve definované parametry indexu, B je velikost diskového bloku. Potom hodnota $\epsilon(n)$ označuje počet diskových operací, které jsou nutné pro dílčí operaci *ukončení expanze bloku* mezi oblastmi A_n a $A_{(n+1)}$, tj. pro uložení bloku oblasti A_n do příslušné díry.

V následujícím lemmatu bude dokázáno, že průměrný počet diskových operací provedených v rámci *ukončení expanze bloku* je asymptoticky konstantní.

Lemma 4.2.10: *Pro $\kappa \in (1, 2)$ platí, že $\sum_{l=1}^{\infty} (S(l) \cdot \epsilon(l)) = O(1)$.*

Důkaz:

$$\begin{aligned}
\sum_{l=1}^{\infty} (S(l) \cdot \epsilon(l)) &= \sum_{l=1}^{\infty} \left(S(l) \cdot \left\lceil \frac{l_0 \kappa^l}{B} \right\rceil \right) \\
&< \sum_{l=1}^{\infty} \left(S(l) \cdot \left(\frac{l_0 \kappa^l}{B} + 1 \right) \right) \\
&= \sum_{l=1}^{\infty} S(l) + \frac{l_0}{B} \sum_{l=1}^{\infty} S(l) \cdot \kappa^l \\
(4.2.8) \quad &= 1 + \frac{l_0}{B} \sum_{l=1}^{\infty} \left[\left(1 - \frac{1}{c_e}\right)^{(l-1)} \cdot \frac{1}{c_e} \cdot \kappa^l \right] \\
&= 1 + \frac{l_0 \kappa}{B\, c_e} \sum_{l=0}^{\infty} \left[\left(1 - \frac{1}{c_e}\right) \cdot \kappa \right]^l
\end{aligned}$$

$$= 1 + \frac{l_0\kappa}{B\, c_e} \cdot \frac{1}{1 - \left(1 - \frac{1}{c_e}\right)\kappa} \tag{4.7}$$

$$= 1 + \frac{l_0\kappa}{B\, \left\lceil \frac{1}{\kappa - 1} \right\rceil} \cdot \frac{1}{1 - \left(1 - \frac{1}{\left\lceil \frac{1}{\kappa - 1} \right\rceil}\right)\kappa}$$

$$\ldots$$

$$= 1 + \frac{l_0}{B} \cdot \frac{\kappa}{(1 - \kappa)\left\lceil \frac{1}{\kappa - 1} \right\rceil + \kappa} \tag{4.8}$$

Výraz 4.8 závisí pouze na statických parametrech indexu a na velikosti diskového bloku. Pro úplný důkaz je ještě nutno ukázat, že součet geometrické řady v kroku 4.7 je korektní. To je pro $\kappa \in (1, 2)$ splněno, protože

$$0 < \left(1 - \frac{1}{\left\lceil \frac{1}{\kappa - 1} \right\rceil}\right)\kappa < \left(1 - \frac{1}{\left(\frac{1}{\kappa - 1} + 1\right)}\right)\kappa = 1.$$

□

Nyní bude následovat konečné odvození amortizované složitosti operace expanze bloku. Důkaz bude proveden pomocí modifikované potenciálové metody. Pro datovou strukturu bude definována potenciálová funkce, která určí potenciál struktury po určitém počtu kroků. Každé operaci potom bude určena amortizovaná složitost a ta srovnána se skutečnou. Pokud je amortizovaná složitost větší, příslušný rozdíl je příspěvek této operace k potenciálu. Pokud je naopak menší, rozdíl musí být z potenciálu „uhrazen".

Označení 4.2.11: *Symbol D_i označuje stav datové struktury univerzálního indexu po provedení i–té operace (D_i je počáteční stav). V souladu se dříve zavedeným modelem lze D_i reprezentovat také posloupností $(\varphi(n))_{n=1}^{\infty}$.*

Označení 4.2.12: *Nechť $k \in \{0, 1, \ldots, c_e\}$ a $n \in \mathbb{N}$. Dále nechť h_n je největší přirozené číslo takové, že $\varphi_n(k) < c_e$. Potom definujme*

$$X_k^n = \{i | (i \in \mathbb{N}) \wedge (i \leq h_n) \wedge (\varphi_n(i) = k)\}.$$

Definice 4.2.13: *Nechť l_0 a κ jsou dříve definované parametry univerzálního indexu a B velikost diskového bloku. Potom pro datovou strukturu univerzálního indexu po provedení n–té expanze bloku definujeme* potenciálovou funkci *(potenciál)* Φ *následujícím způsobem:*

$$\Phi(D_n) = 2 \cdot \left[0 \cdot \sum_{l \in X^n_{(c_e - 0)}} \epsilon(l) + 1 \cdot \sum_{l \in X^n_{(c_e - 1)}} \epsilon(l) + 2 \cdot \sum_{l \in X^n_{(c_e - 2)}} \epsilon(l) + \ldots + c_e \cdot \sum_{l \in X^n_0} \epsilon(l)\right]$$

$$= 2 \cdot \sum_{k>1}^{\left\lceil \frac{1}{\kappa - 1} \right\rceil} \left(k \cdot \sum_{l \in X_k^n} \left\lceil \frac{l_0 \kappa^l}{B} \right\rceil\right)$$

Každé operaci bude níže určena amortizovaná cena (měřená v počtu diskových operací). Tato cena může být menší nebo větší než cena skutečná.

Potenciál (neformálně) funguje jako „účet“ datové struktury, ze kterého lze doplácet za operace, jejichž skutečná cena je vyšší než amortizovaná. Naopak pokud skutečná cena je nižší, příslušný rozdíl uložen „na účet“. Jestliže zůstatek v potenciálu je stále nezáporný, potom průměrná amortizovaná cena operací dává amortizovanou složitost operace expanze.

Definice 4.2.14: ***(Amortizovaná cena operace)***
Amortizovaná cena jedné operace expanze (ex), která skončí mezi oblastmi A_n a $A_{(n+1)}$, je

$$\tilde{c_n}(ex) = 2\epsilon(n) + 2c_e\epsilon(n) = 2(c_e + 1)\epsilon(n).$$

První sčítanec v definici 4.2.14 pokrývá skutečnou cenu dílčí operace *ukončení expanze*, tj. případné načtení posledního manipulovaného bloku z disku a jeho uložení do volné díry.

Druhý sčítanec představuje inkrement potenciálu pro tento krok, ze kterého budou pokryty skutečné ceny manipulace s bloky oblasti A_n při následujících expanzích procházejících mezi danými oblastmi.

Lemma 4.2.15: *Pokud počáteční stav datové struktury je takový, že za každou oblastí A_n je díra velikosti alespoň $l_0\kappa^n$, potom hodnota potenciálové funkce je během posloupnosti operací expanze stále nezáporná.*

Důkaz: Uvažujme díru nulové velikosti mezi neprázdnými oblastmi A_n a $A_{(n+1)}$. Každá procházející expanze ovlivní tuto díru dvěma způsoby:

1. Načtení prvního bloku oblasti A_n a jeho vyjmutí z této oblasti. Pokud daná operace expanze skončí mezi oblastmi $A_{(n+1)}$ a $A_{(n+2)}$, bude cena tohoto načtení pokryta přímo ze skutečné ceny této expanze.

2. Načtení (typicky prvního) bloku oblasti A_n a jeho uložení do zvětšené díry mezi oblastmi A_n a $A_{(n+1)}$. Skutečná cena této dílčí operace je $2\epsilon(n)$ a je pokryta z potenciálu z inkrementu, který vznikl, když se mezi oblastmi A_n a $A_{(n+1)}$ zastavila poslední expanze.

Protože počet expanzí procházejících mezi těmito oblastmi před tím, než se mezi nimi nějaká další zastaví, je maximálně c_e, bude v potenciál vždy obsahovat dostatečnou hodnotu pro pokrytí manipulací s bloky oblasti A_n během procházejících expanzí. Proto hodnota potenciálu je po celou dobu nezáporná. □

Věta 4.2.16: *Amortizovaná složitost T operace expanze bloku za dříve uvedených předpokladů je $O(1)$.*

Důkaz:

$$\begin{aligned} T &= \sum_{n=1}^{\infty} (S(n) \cdot 2(c_e + 1)\epsilon(n)) \\ &= 2(c_e + 1) \sum_{n=1}^{\infty} (S(n)\, \epsilon(n)) \\ (4.2.10) &= 2(c_e + 1) \cdot O(1) = O(1) \end{aligned}$$

□

4.2.1. Vliv distribuce dat

Předchozí analýza předpokládala přidávání záznamů vždy do oblasti s nejmenším indexem. V dokumentografických informačních systémech však distribuce dat odpovídá Zipfovu zákonu. To znamená, že oproti předpokladům předchozích tvrzení bude docházet častěji ke vkládání do oblastí s vyššími indexy a naopak. Intuitivně je patrné, že při vkládání do větších bloků dochází k expanzím bloku méně často a že tyto expanze mají i menší délku. Proto lze předpokládat, že časová složitost se tímto novým předpokladem distribuce dat nezhorší – konkrétním důkazem se však tato práce nebude zabývat. V jednotlivých experimentech v kapitole 8 lze sledovat vliv různých distribucí dat na efektivitu dynamického invertovaného souboru.

4.3. Časová analýza ostatních operací

4.3.1. Operace insert

Pokud během operace insert dojde k expanzi bloku, je zřejmě cena celého insertu stejná jako cena expanze, tedy amortizovaně konstantní. Insert do bloku v oblasti A_n bez expanze bloku má cenu $2\epsilon(n)$ (neboť dojde k načtení a opětovnému uložení celého bloku), tedy menší nebo stejnou ve srovnání s expanzí bloku, která se okamžitě zastaví. Proto i amortizovaná cena operace insert je konstantní.

4.3.2. Operace delete

Tato práce se nebude zabývat výpočtem amortizované složitosti této operace. Intuitivně je zřejmé, že je rovněž konstantní a že důkaz by se provedl analogicky jako v případě operace insert, resp. v případě expanze bloku.

Kapitola 5

Odolnost proti výpadkům

Odolnost proti výpadkům je implementována za použití principů transakčního zpracování dat známého z databázových systémů. Přitom je předpokládáno, že uživatel knihovny indexu zaručí serializaci aktualizačních operací i každé aktualizační operace s dotazem. Díky tomu není nutné implementovat zamykání dat v indexu. Bez tohoto předpokladu by složitost implementace narostla neúměrně rozsahu této práce.

5.1. Úvod do transakcí

Budeme požadovat, aby se jednotlivé uživatelské operace nad univerzálním indexem chovaly jako tzv. *transakce*.

Definice 5.1.1: Transakcí *nazveme libovolnou posloupnost operací, která splňuje tzv. vlastnosti* ACID*, tj. je*

- atomická *(atomic) – to znamená, že se provedou buď všechny její operace, nebo žádná;*
- konzistentní *(consistent) – převádí stav systému z jednoho konzistentního stavu do druhého, tj. není přípustné, aby transakce skončila (ať už úspěšně nebo neúspěšně) a zanechala za sebou nekonzistentní data;*
- nezávislá *(independent) – jednotlivé transakce se mezi sebou nemohou ovlivňovat, tzn. jedna transakce nevidí mezivýsledky ostatních, pokud tyto nebyly dokončeny před jejím započetím;*
- trvalá *(durable) – její výsledky jsou persistentě uloženy do externí paměti.*

Dále bude vysvětleno, jak implementačně zajistit, aby se volání každé aktualizační operace na rozhraní indexu chovalo jako transakce. Každé takové volání je ve skutečnosti posloupnost několika interních operací – například při indexaci nové

i–entity je třeba zapsat příslušné informace do slovníku, do souboru indexových záznamů a případně ještě upravit adresář indexu (kvůli změně hranic oblastí). Implementací transakčního zpracování lze zaručit, že například v případě havárie indexu se nezapíší data pouze do souboru indexových záznamů, ale už ne do adresáře apod.

Vlastnosti transakcí zde nebudou detailně rozebírány, neboť o nich v literatuře již bylo napsáno dost, viz např. [11] nebo [10].

Pro dosažení vlastností ACID pro operace nad indexem bude využito klasického prostředku známého z databází – tzv. *logu*. Konkrétně se zde jedná o *log s odloženou realizací změn* (redo–log), který je příznačný tím, že všechny aktualizace se nejdříve zapíší do logu a teprve později se propagují do datových struktur indexu.

Před vlastním popisem transakčního logu a jeho využitím uveďme pro přehlednost několik (neformálních) definic:

Definice 5.1.2: Časová známka *je jakékoli kladné reálné číslo, které nějakým způsobem identifikuje, relativně nebo absolutně, časový okamžik. Na množině časových známek je definováno* časové uspořádaní $\prec$. *Platí, že pokud* $t_1 \prec t_2$, *předchází časový okamžik odpovídající časové známce* t_1 *okamžiku odpovídajícímu známce* t_2.

Množinu časových známek označme $\mathbb{T}$.

Označení 5.1.3: *Označme symbolem* $-\infty$ *časovou známku reprezentující nejstarší reprezentovatelný časový okamžik. Tato speciální hodnota bude používána pro* neplatnou *časovou známku.*

Je jedno, jaká soustava časových známek bude zvolena, není ani nutná žádná vazba na fyzický (skutečný) čas.

Označení 5.1.4: *Označme množinu transakcí symbolem* $\mathfrak{T}$ *a množinu jejich* instancí *symbolem* $\mathfrak{T}_x$.

Transakcí se rozumí posloupnost operací ve smyslu definice 5.1.1, zatímco její instance je nějaké její konkrétní provedení nad konkrétními daty.

Definice 5.1.5: *Buď* T *transakce. Potom definujeme zobrazení*

(i) $beg(T) : \mathfrak{T}_x \to \mathbb{T}$, *které instanci transakce přiřadí časovou známku příslušnou pro okamžik, kdy systém tuto transakci začal zpracovávat.*

(ii) $end(T) : \mathfrak{T}_x \to \mathbb{T}$, *které instanci transakce přiřadí časovou známku příslušnou pro okamžik, kdy systém dokončil zpracování této transakce.*

Budeme používat klasickou sadu operací souvisejících s transakcemi:

- `begin(T)`: Zahájí zpracování transakce `T`.

- `commit(T)`: Způsobí ukončení transakce `T`. V průběhu volání musí všechna data být persistentně uložena, neboť jestliže se `commit` úspěšně vrátí, musí již být zaručeny vlastnosti ACID.

- `rollback(T)`: Způsobí neúspěšné ukončení transakce. Ani jedna z obsažených operací nebude ve výsledku provedena.

Nyní již lze popsat základní principy logu s odloženou realizací změn:

Je zřejmé, že jsou–li cílem vlastnosti ACID, není možné zapisovat data jen přímo do cílových datových struktur. Pokud by totiž během příslušné posloupnosti operací došlo k selhání systému, zůstala by část operací neprovedená, data by byla aktualizovaná jen částečně, a tím by byly porušeny vlastnosti A a C. V zásadě existují dvě možnosti, jak tuto situaci řešit. První z nich (undo–log) počítá s tím, že data jsou sice zapisována přímo do datových struktur, ale zároveň jsou do speciálního souboru – logu – ukládány informace, které v případě pádu systému umožní vrácení provedené části operací a tím obnovení konzistence dat.

Druhá možnost, již zmíněný redo–log, funguje obráceně. Do redo–logu jsou zapisovány instrukce, které odpovídají částem jednotlivých transakcí (jejich provedení převede cílové datové struktury do stejného stavu jako přímo provedení operací obsažených v příslušné transakci). Operace `commit` provede uložení příslušných částí logu na disk a ověří proveditelnost dané sekvence instrukcí. Od této chvíle (ale klidně i později) dojde k provedení těchto instrukcí a modifikaci cílových datových struktur. To ovšem znesnadňuje dotazování, protože kromě vyhledávání přímo v „primárních“ datových strukturách je nutné ještě ověřování, zda v logu nejsou nějaké informace o změnách relevantních dat.

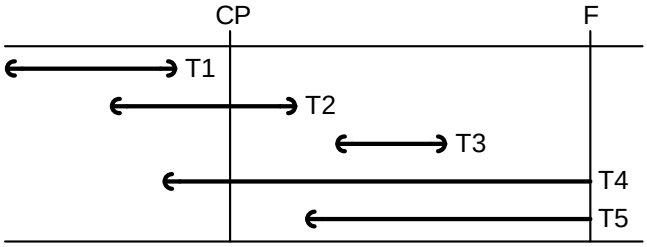
Důležitým prvkem je zapisování tzv. *checkpointů* (kontrolních bodů) do logu. Označme C časovou známku vložení checkpointu do logu. Potom musí být zaručeno, že každá transakce T, pro kterou platí, že $end(T) \prec C$, již je vykonána, tj. všechny jí odpovídající instrukce uložené v logu již byly provedeny.

V případě pádu systému je nutné spustit *proces zotavení z výpadku systému*. V tomto případě je třeba rozlišit transakce podle jejich časového vztahu k poslednímu zapsanému do logu. Diskuzi možných případů provádí obrázek 5.1. Znázorňuje dobu trvání jednotlivých transakcí – časová osa je orientována obvyklým způsobem zleva doprava. Obrázek rovněž obsahuje dvě vertikální osy. První z nich, označená CP, se vyskytuje v bodě, kdy byla do logu zapsána poslední časová známka. Druhá osa je označená F a znázorňuje okamžik pádu systému.

Transakce T_1 až T_3 jsou zakresleny jako horizontální úsečky, jejichž levý bod je určen časovou známkou operace `begin` a pravý podobně operací `commit`. Pro T_4 a T_5 dosud nebyl `commit` vyvolán, neboť vykonávání těchto transakcí bylo přerušeno pádem systému.

Během procesu zotavení je s jednotlivými typy transakcí zacházeno následujícím způsobem:

- T_1: $end(T_1) \prec CP$, proto všechny operace této transakce byly již úspěšně zapsány do cílových struktur a touto transakcí se není třeba zabývat.



Obrázek 5.1: Možné stavy transakcí vzhledem k poslednímu checkpointu (CP) a bodu selhání systému (F)

- T_2: $(beg(T_1) \prec CP) \wedge (end(T_1) \succ CP) \wedge (end(T_1) \prec F)$, proto se nelze spolehnout na to, že již všechny operace T_2 byly provedeny. Část operací pozdějších než CP se však do cílových struktur promítnout mohla. Všechny operace této transakce pozdější než poslední checkpoint musí být provedeny znovu. Proto bude nutné navrhnout záznamy do logu tak, aby jimi reprezentované operace byly *idempotentní*, tj. aby jejich vícenásobné provedení dávalo vždy stejný výsledek.

- T_3: $beg(T_3) \in (CP; F)$, $end(T_3) \in (CP; F)$, proto již mohla být provedena libovolná část operací této transakce. Nelze však zjistit jak velká část, a proto *všechny* operace této transakce musí být provedeny znovu (algoritmy musejí být navrženy tak, aby vícenásobné provedení jedné operace nemělo žádné negativní důsledky).

- T_4: $(beg(T_4) \prec CP) \wedge (end(T_4) = \infty)$, tedy pád systému přerušil tuto transakci před dokončením. Veškeré její operace je nutno „zapomenout“, neboť neexistuje způsob, jak ji bezpečně dokončit a přitom zaručit zachování vlastností ACID.

- T_5: $(beg(T_5) \succ CP) \wedge (end(T_5) = \infty)$, tedy situace je stejná jako v předchozím případě.

5.2. Idempotence operace zotavení z výpadku

Definice 5.2.1: *Řekneme, že operace α na množině $\mathbb{M}$ je* idempotentní, *jestliže platí $\alpha(x) = \alpha(\alpha(x))$ pro $\forall x \in \mathbb{M}$.*

V tomto smyslu budeme požadovat idempotenci operace zotavení z výpadku. Význam tohoto požadavku je zřejmý – během zotavení může dojít k dalšímu selhání systému. Požadavek idempotence je zde tedy ještě silnější než v předchozí definici – výsledek zotavení musí být korektní i za předpokladu, že předchozí zotavení proběhlo jen částečně. Proto operace jednotlivých transakcí v logu musejí být konstruovány tak, aby jejich vícenásobné provedení mělo stejný efekt jako provedení jediné.

5.3. Implementace transakčního logu

5.3.1. Fyzická struktura logu

Log se fyzicky skládá z bloků, přičemž je optimální, když tyto bloky splývají s bloky diskovými. Každý blok obsahuje časové razítko a určitou množinu záznamů. U jednotlivých bloků je důležité zajistit konzistenci – pokud se kvůli selhání systému nepodaří blok zapsat celý, musí index být schopen tuto skutečnost zjistit a zařídit se podle toho (ignorovat porušený blok). Konzistenci lze zajistit buď pomocí speciálního rozhraní pro tyto účely určeného, lépe se však jeví jednoduchý mechanismus zápisu stejného časového razítka na začátek i na konec bloku; blok se potom považuje za konzistentní, pokud se obě časová razítka rovnají. Další důležitou režijní informací v hlavičkách bloků je pozice (offset) naposled zapsaného bloku obsahujícího checkpoint. Význam tohoto údaje i časových razítek v blocích je patrný z algoritmů v sekci 5.3.3.

Uvnitř bloků transakčního logu mohou být obsaženy následující záznamy:[1]

1. `BEGIN`: Reprezentuje informaci o zahájení transakce identifikované číslem, které je součástí tohoto záznamu.

2. `COMMIT`: Záznam o potvrzení úspěšného dokončení transakce. Zapsáním tohoto záznamu se systém zavazuje, že bude schopen realizovat všechny operace obsažené v příslušné transakci. Z hlediska uživatele je již příslušná transakce zcela dokončena, z hlediska indexu ještě může „čekat“ na své provedení v cílových datových strukturách.

3. `ROLLBACK`: Záznam o zrušení transakce. Indikuje, že tato transakce se ukázala jako neproveditelná a byla zrušena systémem, nebo že ji zrušil přímo uživatel.

4. `FILE`: Záznamy tohoto typu slouží k přiřazení souboru/souborů k transakci. Každý z nich namapuje konkrétní soubor na nějaký identifikátor, který potom bude používán při I/O operacích.

5. `WRITE`: Tento záznam specifikuje zápis dat do specifikovaného souboru, který byl předtím svázán s transakcí pomocí záznamu `FILE`. Délka dat smí být nejvýše taková, aby tento záznam v logu nepřesáhl hranici diskové stránky, jinak je třeba data rozdělit do více záznamů.

6. `EXPAND`: Záznam `EXPAND` způsobí rozšíření příslušného souboru na cílovou velikost. Viz poznámky u `SHRINK`.

7. `SHRINK`: Opak záznamu `EXPAND` – znamená zkrácení souboru na cílovou velikost.

   Cílová velikost musí být dána absolutně kvůli tomu, aby operace zotavení z výpadku systému byla idempotentní. Zřejmě by se `EXPAND` a `SHRINK` daly

[1] Uvnitř znamená, že nesmí překračovat hranice jednotlivých bloků. V opačném případě by mohlo docházet k nekonzistencím.

sloučit do jediného záznamu, nicméně z historických důvodů a pro kontrolu zůstávají v indexu tyto dvě operace rozlišeny.

8. `CHECKPOINT`: Checkpoint je speciální záznam. Pokud je zapsán do logu, musí být zaručeno, že všechny transakce, jejichž záznam `COMMIT` byl zapsán před checkpointem, už byly fyzicky realizovány v cílových datových strukturách. Důležitou informací obsaženou v tomto typu záznamu je seznam transakcí aktivních v okamžiku zápisu checkpointu. Pro více informací viz sekci 5.1.

5.3.2. Kontrola přetečení logu

Přetečení logu je faktor, který by mohl porušit požadované vlastnosti ACID a je tedy nutno jej ošetřit.

Logu je předem určena jeho velikost; v rámci tohoto prostoru potom log funguje jako tzv. *cyklický buffer*. To znamená, že po vyčerpání prostoru logu se začne zapisovat opět od začátku souboru, kde jsou záznamy, které již pravděpodobně byly zpracovány a nejsou nadále potřeba. Pokud tyto záznamy dosud zpracovány nebyly, je třeba zastavit přijímání příchozích transakcí a místo toho zpracovat záznamy v logu a nějaký prostor uvolnit.

Přitom je třeba efektivně určit odkud kam v logu sahají aktuálně platné stránky. K tomu slouží atribut *časová známka zápisu stránky*, který je uveden v hlavičce každého diskového bloku zapsaného v logu.

Při procesu zotavení je třeba respektovat následující pravidla:

1. Narazíme–li během čtení platných diskových stránek logu na stránku se starší časovou známkou, než měla stránka předchozí, byla předchozí stránka poslední platnou stránkou logu.

2. Načteme–li poslední fyzickou stránku na konci souboru logu a tato je platná, následující stránka se nachází na začátku souboru logu.

Aby předchozí pravidla mohla fungovat, je třeba na začátku log inicializovat počátečními časovými známkami. Z toho důvodu je při vytvoření indexu nutné jeho log *naformátovat*. Formátování spočívá v tom, že se celý log zaplní diskovými stránkami, které mají definovánu pouze hlavičku. V té obsahují časovou známku reprezentující nejstarší reprezentovatelný čas a údaj o nula obsažených záznamech.

5.3.3. Algoritmus zotavení

Algoritmus zotavení z výpadku systému se skládá ze dvou dílčích algoritmů:

- nalezení nejnovějšího checkpointu;
- vlastní zotavení.

Nalezení nejnovějšího checkpointu

Vzhledem k tomu, že log funguje jako cyklický buffer a rozsah platných stránek se určuje pomocí časových známek bloků, nelze na začátku procesu zotavení z pádu systému jednoduše určit první (nejstarší) platný diskový blok logu.

Je snadno patrné, že posloupnost časových známek bloků seřazených od prvního (fyzicky) do posledního bloku souboru splňuje následující podmínky:

1. Buď je celá neklesající (podmínka A), nebo lze rozdělit na nejvýše dvě neklesající podposloupnosti (B).
2. Je–li splněno A, potom žádný blok nemá časovou známku s menší hodnotou než první blok. Rovněž žádný blok nemá časovou známku s větší hodnotou než blok fyzicky poslední.
3. Je–li splněno B, potom každý blok první podposloupnosti má vyšší časovou známku než libovolný blok podposloupnosti druhé.
4. Má–li časová známka fyzicky prvního bloku vyšší hodnotu než známka bloku fyzicky posledního, platí B. V opačném případě platí A.

Na základě těchto podmínek lze zformulovat algoritmus, který nalezne poslední checkpoint.

Předpokládejme, že soubor realizující log sestává z n diskových bloků, označme $\mathbf{B}[i]$ i–tý blok (číslováno od nuly). Označme $T(\mathbf{B}[i])$ časovou známku i–tého bloku. $\mathbf{B}[i].lastblock$ značí atribut hlavičky bloku logu, který obsahuje pozici naposled zapsaného checkpointu.

Vstup: soubor realizující log

Výstup: pozice diskového bloku obsahujícího poslední checkpoint; -1 v případě, že dosud žádný nebyl zapsán

1 **if** $T(\mathbf{B}[0]) = -\infty$ **then**
  /* *Do logu zatím nebyly zapsány žádné bloky* */
2 Ukonči algoritmus zotavení;
3 **end**
4 **if** $T(\mathbf{B}[n-1]) = -\infty$ **then**
  /* *Cyklický buffer dosud nebyl zcela zaplněn* */
5 *lastblock* ← pořadí posledního zapsaného bloku nalezeného metodou binárního půlení intervalů, přičemž hodnotu $-\infty$ považuj za ∞ (za největší reprezentovatelnou časovou známku);
6 **end**
7 **else if** $T(\mathbf{B}[0]) < T(\mathbf{B}[n-1])$ **then**
  /* *Cyklický buffer byl právě zaplněn* */
8 *lastblock* ← $(n-1)$;
9 **end**
10 **else if** $T(\mathbf{B}[0]) > T(\mathbf{B}[n-1])$ **then**
  /* *Časové známky bloků tvoří dvě rostoucí podposloupnosti* */
11 *lastblock* ← pořadí posledního zapsaného bloku nalezeného metodou binárního půlení intervalů, přičemž časové známky nižší než $T(\mathbf{B}[0])$ považuj za ∞ (za největší reprezentovatelnou časovou známku);
12 **end**
13 **if** $\mathbf{B}[lastblock].last\_checkpoint = 0$ **then**
  /* *Žádný checkpoint zatím nebyl zapsán* */
14 **return** *-1*;
15 **end**
16 return $(n + lastblock - \mathbf{B}[lastblock].last\_checkpoint)\ mod\ n$;

Algoritmus 7: Nalezení posledního checkpointu

Algoritmus zotavení z výpadku

Na nalezení posledního checkpointu bezprostředně navazuje algoritmus zotavení z výpadku systému:

Vstup: soubor realizující log; index posledního zapsaného checkpointu *last_checkpoint*

Výstup: pozice diskového bloku obsahujícího poslední checkpoint; 0 v případě, že dosud žádný nebyl zapsán

1 *commit_list* ← *empty_list*;

2 **if** *last_checkpoint* = −1 **then** *active_list* ← *empty_list*;

3 **else**

4 *active_list* ← seznam aktivních transakcí zaznamenaných uvnitř posledního checkpointu;

5 **end**

6 Procházej log od naposled zapsaného checkpointu směrem k novějším blokům (v případě dosažení konce souboru pokračuj od začátku). Pokud dosud žádný checkpoint nebyl zapsán, procházej místo toho od bloku s nejmenším časovým razítkem. Přitom:

- do *active_list*u přidávej všechny transakce, na jejichž `BEGIN` záznam narazíš;
- z *active_list*u odstraňuj všechny transakce, na jejichž `ROLLBACK` záznam narazíš;
- všechny transakce, na jejichž `COMMIT` záznam narazíš, přesuň z *active_list*u do *commit_list*u.

Přidej na konec logu `ROLLBACK` záznamy pro všechny transakce, které zůstaly v *active_list*u.

Najdi začátek nejstarší transakce z *commit_list*u (viz následující algoritmus).

Procházej log „doprava“ (směrem k později zapsaným blokům) od začátku nejstarší transakce v *commit_list*u a prováděj operace všech transakcí obsažených v *commit_list*u v cílových strukturách.

Zapiš na konec logu nový checkpoint.

Algoritmus 8: Zotavení z výpadku systému

Pro úplnost zbývá uvést algoritmus nalezení bloku obsahujícího záznam `BEGIN` dané transakce T. Bez újmy na obecnosti přitom předpokládejme, že $beg(T) < CP$, kde CP je časová známka zápisu nejnovějšího checkpointu.

Vstup: soubor realizující log; index posledního zapsaného checkpointu *last_checkpoint*; identifikátor transakce (TID), jejíž úvodní záznam má být nalezen

Výstup: pozice bloku obsahujícího záznam BEGIN příslušné transakce

Předpoklady: $beg(T) < CP$, kde CP je časová známka posledního zapsaného checkpointu

1 *chpblock* $\leftarrow$ **B**[*last_checkpoint*].*previous_checkpoint*;

2 **while** *(chpblock* $\neq -1$*)* ***and*** *(*$TID \in$ *seznam aktivních transakcí checkpointu v bloku s indexem chpblock)* **do**

3 *chpblock* $\leftarrow$ **B**[*chpblock*].*previous_checkpoint*;

4 **end**

5 **if** *chpblock* $= -1$ **then** *chpblock* $\leftarrow 0$

6 Projdi oblast souboru logu od bloku s indexem *chpblock* až do nejbližšího dalšího checkpointu nebo do konce souboru pokud žádný takový neexistuje. Vrať blok obsahující hledaný záznam `BEGIN`.

7 **return** *Úvodní záznam hledané transakce nalezený v předchozím bloku*

Algoritmus 9: Nalezení začátku dané transakce v logu

5.4. Vliv logu na dotazování

Je zřejmé, že princip redo–logu zpožďuje transparentně zápis dat do cílových struktur. Některé části dat mohou být stále pouze v logu, ačkoli příslušná transakce již byla ukončena a potvrzena uživateli.

To je třeba mít na paměti při dotazování. Data získaná například ze souboru indexových záznamů již totiž mohou být neplatná, přepsána nějakou transakcí.

Zároveň však v logu jsou i záznamy, které náleží dosud neukončeným transakcím. Ty na dotazování nesmějí mít žádný vliv, neboť jinak by bylo porušeno pravidlo I (nezávislost) z vlastností ACID.

Při dotazování tedy potřebujeme efektivní mechanismus, který umožní určit, které oblasti příslušného souboru byly změněny, ale příslušná data mohou zatím být jen v logu. V případě, že chceme číst ze změněné oblasti, budeme ještě potřebovat efektivně najít aktuální data.

Nejjednoduším a zároveň maximálně výkonným řešením je tzv. *explicitní cache* pro příslušné soubory. Transakce se provádějí formou zápisu instrukcí do logu a přitom zároveň se vytvářejí v paměti kopie změněných diskových stránek. Zároveň s tím existuje datová struktura realizující mapování pozic v souboru na stránky v této cache; v případě, že stránky měněny nebyly, obsahuje toto mapování neplatnou hodnotu.

Zbývá vyřešit, co se má stát, jestliže dojde prostor vyhrazený pro cache. Podle nastavení si to může vynutit buď propagaci změněných dat do cílových struktur nebo

tzv. *swapování* (odkládání na disk) bloků cache. Prostor pro swapování (tzv. *swap*) se přitom může nacházet na jiném fyzickém disku, čímž se zvýší celková efektivita indexu. Swapování v této implementaci není realizováno.

5.5. Vliv logu na efektivitu indexu

Přestože log všechny operace s cílovými datovými strukturami zřejmě výrazně komplikuje, může mít výrazně pozitivní vliv na efektivitu systému (hlavně na dobu odezvy v nejhorším případě). Zapisuje se do něj totiž sekvenčně a tudíž hlavičky disku se nemusejí zbytečně pohybovat, zejména pokud se pro log používá speciální vyhrazený fyzický disk. Skutečná práce – propagace změněných dat do cílových struktur – může být odložena na dobu, kdy je celková zátěž relativně malá.

I při provádění instrukcí obsažených v logu lze docílit úspory času. Představme si, že bychom měnili v rámci různých aktualizací postupně několik záznamů na téže diskové stránce. Při zápisu přímo na disk by se mohlo stát, že příslušnou stránku budeme muset zapsat několikrát – jednou pro každou z těchto operací. Nicméně při provádění logu lze tuto stránku bezpečně podržet v paměti co nejdéle a zapsat ji až po relizaci změn všech obsažených záznamů.

Tato situace je ještě markantnější při použití cache popsané v odstavci 5.4. Potom vlastně všechny instrukce již máme provedeny v kopiích stránek obsažených v cache a pro aktualizaci stačí pouze přepsat původní stránky těmito kopiemi.

Implementace v této sekci zmíněných optimalizačních technik ovšem již přesahuje rámec této práce.

Kapitola 6

Komprese dynamického invertovaného souboru

Při intenzivním nasazení a „naivní“ implementaci může velikost některých souborů vytvářených univerzálním indexem příliš růst (například při indexaci celých knihoven dokumentů atd.). Proto je vhodné se zabývat metodami efektivnějšího uložení dat – tzv. *kompresí*.

Jak bylo popsáno v dřívějších kapitolách, dynamický invertovaný soubor (který tvoří základ univerzálního indexu) je datová struktura, která se skládá ze tří typů datových souborů – a sice ze slovníku, adresáře a souboru indexových záznamů.

Pro adresář platí, že pokud i je index oblasti obsahující největší bloky, potom velikost adresáře je (až na hlavičku a další režijní informace zabírající konstantní prostor) $2\,p \cdot i$ bytů, kde p je velikost datového typu ukazatele na dané platformě. Za předpokladu $p = 4$ a při standardním nastavení parametrů uvedeném na straně 25 by například oblast s indexem 150 zahrnovala bloky o velikosti asi 4248 GB dat. Pokud by to zároveň byla oblast poslední, adresář by se vešel do prostoru o něco málo většího než 1 KB. Z tohoto příkladu je patrné, že při rozumně nastavených parametrech je velikost adresáře prakticky zanedbatelná.

To samé platí ve většině případů i pro slovník. Jeho velikost je $n \cdot m$ bytů, kde n je počet obsažených i–entit a m velikost jednoho záznamu. Ta se typicky pohybuje v řádu 10 až 20 bytů. Například při indexaci 10000 i–entit by velikost nekomprimovaného slovníku (při zanedbání pomocných vyhledávacích struktur) stále byla menší než 100 KB – 200 KB. Přitom očekávaný počet indexovaných významových termů pro kolekce psané v jediném jazyce se pohybuje v řádu jednotek tisíců.

V praxi se ovšem mohou vyskytnout i případy, kdy jsou indexovány dokumenty psané ve více různých jazycích, nebo je datová struktura používána pro nějaké jiné učely. Potom již může velikost slovníku být problematická. Ovšem vzhledem k tomu, že i např. do slovníku o velikosti pouhých 20 MB se vejde přes $1,3$ milionu záznamů, nebude se tato práce kompresí slovníku zabývat.

Problém velikosti se nejvíce dotýká souboru indexových záznamů. Těch může být velmi mnoho a navíc index umožňuje volitelně definovat jejich strukturu, včetně použití atributů typu „řetězec“ nebo „obecná data“. Metody pro kompresi tohoto

souboru jsou rozebrány dále.

Další soubory, jako jsou například transakční log, se nekomprimují. Log je soubor předem určené velikosti a pokud by se do něj již nevešlo požadované množství dat, lze jej částečně zpracovat a trochu místa v něm uvolnit (za předpokladu, že není příliš mnoho velice dlouhých nedokončených transakcí – potom by výsledkem byla chyba).

6.1. Požadavky na kompresní metody

Primárním cílem při implementaci indexu musí být především jeho časová efektivita. Z toho důvodu nelze používat kompresní metody, kde dekódování dat vyžaduje příliš mnoho strojového času. Časová složitost zakódování není na druhou stranu příliš kritická, neboť v typickém případě budou dotazovací operace o několik řádů převažovat nad aktualizačními.

Na druhou stranu čas potřebný na vykonání diskové operace (zde se jedná především o sekvenční čtení, neboť data každého bloku jsou uložena na disku sekvenčně) je stále řádově větší než čas operací v paměti. Bude–li navíc dosaženo dobrého kompresního poměru, dojde k výraznému snížení množství načítaných diskových bloků, a tím k vyvážení času spotřebovaného na dekódování.

Efektivita většiny kompresních metod závisí na charakteru, zejména na distribuci, vstupních dat. Vzhledem k univerzalitě indexu nelze předvídat, z jakých typů atributů se bude skládat schéma dynamického invertovaného souboru a jakému pravděpodobnostnímu rozdělení tyto atributy budou odpovídat. Nejpředvídatelnější jsou přitom atributy z primární části indexových záznamů. Vzhledem k tomu, že se vždy jedná o neklesající (pod)posloupnosti, lze na ně úspěšně aplikovat přinejmenším tzv. *diferenční kódování* – viz 6.3.

Z důvodů popsaných v předchozím odstavci není zřejmě vhodné implementovat žádné extrémně úsporné specializované kompresní metody. Místo toho implementujeme několik poměrně univerzálních, které dávají přes svou jednoduchost poměrně slušné výsledky. Ne pro každý typ dat jsou všechny tyto metody vhodné, ovšem uživatel indexu má možnost si pro *každý atribut* ze schématu zvolit libovolnou z nich na základě své znalosti příslušných dat.

V budoucnu by bylo vhodným rozšířením implementovat možnost přidávání dodatečných, uživatelsky definovaných modulů, které by byly volány pomocí *callbacků* přes definované rozhraní a pomocí kterých by uživatel mohl implementovat libovolnou vlastní kompresní metodu.

6.2. Eliasovy kódy

Eliasovy kódy (viz např. [10]) ve své podstatě nejsou kompresní metodou, ale způsobem kódování přirozených čísel. Jedná se o poměrně známou metodu, jejíž znalost je ovšem podstatná pro pochopení dalšího textu, a proto bude v této části stručně

popsána. Eliasovy kódy splňují následující vlastnosti:

- Kódové slovo libovolného přirozeného čísla n není omezeno žádným předem stanoveným počtem bitů; zároveň však neobsahuje žádné nevýznamové bity.
- Eliasovy kódy jsou *prefixové*, tj. kódová slova sama v sobě obsahují informaci o tom, kde končí. Nejsou tedy vyžadovány žádné oddělovače.
- Eliasovy kódy rostou logaritmicky v závislosti na velikosti vstupního čísla n.

6.2.1. Pomocné kódy

Nejdříve je nutné definovat některé pomocné kódy.

Definice 6.2.1: *Nechť n je nenulové přirozené číslo (v případě β je přípustná i nula). Potom definujeme následující kódy:*

(i) *$\alpha(n)$ je unární kódování. Je definováno následujícím způsobem:*

$$\alpha(n) = \begin{cases} 1 & \ldots \text{ pro } n = 1 \\ 0 \cdot \alpha(n-1) & \ldots \text{ pro } n > 1. \end{cases}$$

(ii) *$\beta(n)$ je standardní binární kódování. Definujeme jej takto:*

$$\beta(n) = \begin{cases} 0 & \ldots \text{ pro } n = 0 \\ 1 & \ldots \text{ pro } n = 1 \\ \beta(n \; div \; 2) \cdot \beta(n \; mod \; 2) & \ldots \text{ pro } n > 1, \end{cases}$$

kde div značí celočíselné dělení a mod zbytek po něm.

(iii) *$\beta'(n)$ je binární kódování bez prvního bitu. Zde se vychází z faktu, že pro $n \neq 0$ je první bit $\beta(n)$ vždy jednička, a tudíž nenese žádnou užitečnou informaci. Proto jej v takovém případě lze vynechat.*

Uveďme příklady definovaných kódů například pro $n = 9$:

$$\begin{aligned} \alpha(9) &= 000000001 \\ \beta(9) &= 1001 \\ \beta'(9) &= 001 \end{aligned}$$

Definice 6.2.2: *Nechť n je nenulové přirozené číslo a λ prázdný řetězec. Potom definujeme následující Eliasovy kódy:*

(i) *$\gamma(n)$ definujeme[1] pomocí kódových slov $\alpha(|\beta(n)|)$ a $\beta'(n)$ tak, že bity těchto slov proložíme. Přitom začínáme prvním bitem (delšího) kódu $\alpha(|\beta(n)|)$.*

(ii) *$\gamma'(n)$ definujeme předpisem $\alpha(|\beta(n)|) \cdot \beta'(n)$.*

[1]Tento kód je uveden pouze pro úplnost; v indexu ve skutečnosti není implementován, neboť jej lze nahradit ekvivalentním kódem $\gamma'(n)$.

(iii) $\delta(n)$ *definujeme předpisem*[2] $\gamma'(|\beta(n)|) \cdot \beta'(n)$.

(iv) $\omega(n)$ *je definován předpisem*

$$\omega(n) = \begin{cases} 0 & \dots \text{ pro } n = 1 \\ \overline{\omega}(n) \cdot \beta(n) \cdot 0 & \dots \text{ v ostatních případech,} \end{cases}$$

kde

$$\overline{\omega}(n) = \begin{cases} \lambda & \dots \text{ pro } 0 < n < 4 \\ \overline{\omega}(\beta(|\beta(n)| - 1)) \cdot (\beta(|\beta(n)| - 1)) & \dots \text{ ostatní případy.} \end{cases}$$

(v) $\omega'(n)$ *je definováno podobně jako* $\omega(n)$*, ale první „skupina bitů“ má velikost* 3 *místo* 2*. V prvním bodě definice* $\overline{\omega}$ *tedy stačí nahradit* $0 < n < 4$ *podmínkou* $0 < n < 8$*.*

Uveďme příklady právě definovaných kódů:

$\gamma(9) = \underline{0}0\underline{0}0\underline{1}1$ (podtržené bity odpovídají $\alpha(|\beta(n)|)$, nepodtržené potom $\beta'(n)$).
$\gamma'(9) = \underline{0001}001$
$\delta(9) = \underline{001}00\overline{001}$ (skupiny bitů z definice jsou rozlišeny podtržením/nadtržením).
$\omega(9) = \underline{11}1001\overline{0}$
$\omega'(9) = \underline{011}1001\overline{0}$

Je zřejmé, že kódová slova Eliasových kódů mají až dvojnásobnou délku oproti binárnímu zápisu; přesto však je jejich použití výhodné. Posloupnost čísel zapsaná binárně totiž neobsahuje žádné informace o tom, kde bity jednotlivých čísel začínají a končí. Nejjednodušší řešení je vyhradit pro každé číslo pevný počet bitů – častá délka na současných architekturách bývá 32 bitů. Ve většině případů je ovšem pravděpodobné, že většina bitů z těchto čtyř bytů zůstane zbytečně nevyužitá.

Eliasovy kódy pro danou hodnotu sice většinou zaberou více bitů než binární zápis, nicméně jsou *prefixové* – tzn. že každé kódové slovo samo nese informaci o svém začátku a konci. Z toho důvodu bity Eliasových kódů jednotlivých čísel mohou za sebou bezprostředně následovat. Není tedy třeba vyhrazovat předem nějaký prostor a využití bitů je stoprocentní. V součtu tak celkový prostor potřebný k zápisu Eliasových kódů posloupnosti čísel je typicky menší, než pokud by se použilo klasické binární kódování s předem vyhrazeným počtem bitů pro každé číslo.

Nyní ukážeme horní meze délky Eliasových kódů v závislosti na velikosti kódovaného vstupu n bitové délky $|n|$. Tento odhad je v dynamickém invertovaném souboru nutný pro předběžné určení délky zakódovaných seznamů indexových záznamů.

Délka pomocného kódu $\alpha(n)$ je zřejmě $|n|$, kódu $\beta(n)$ $\lceil \log_2(|n|) \rceil$, a konečně pro $\beta'(n)$ je to $\lceil \log_2(|n| - 1) \rceil$.

Z konstrukce Eliasových kódů dále evidentně plyne, že počet bitů kódu γ a γ' je $2\lceil \log_2(|n|) \rceil - 1$. Vyjádření délky $\delta(n)$ je už složitější a je rovna výrazu

$$2\lceil \log_2(\lceil \log_2(n) \rceil) \rceil + \lceil \log_2(n) \rceil - 2.$$

[2] V literatuře bývá často tento kód zaváděn pomocí γ místo γ', tento text se však drží změněné definice, neboť kód γ není implementován.

Všechny předchozí kódy jsou natolik jednoduché, že správnost uvedených velikostí není třeba dokazovat a může být vyjádřena přesně. Jiná situace nastává v případě ω a ω'. Omezme se nyní na první z nich. Z jeho definice je vidět, že má následující strukturu:

$$\omega(n) = b_n \cdot b_{n-1} \ldots b_2 \cdot b_1 \cdot n \cdot 0,$$

kde b_n jsou bloky bitů. Přitom platí, že

- b_1 kóduje velikost binárního zápisu n zmenšenou o jedničku (tedy $|\beta(n)| - 1$).
- b_{k+1} kóduje velikost binárního zápisu b_k zmenšenou o jedničku.
- Velikost poslední skupiny b_n (pro n největší možné) je buď přesně 2 bity, nebo je kódovaná hodnota 1 (a její kódové slovo 0).

Tvrzení 6.2.3: *Z výše uvedeného plyne, že velikost výsledného kódového slova pro vstup větší než* 1 *je*

$$\begin{aligned} |\omega(n)| = \; & 1 + \lceil \log_2(n) \rceil \\ & + \lceil \log_2(\lceil \log_2(n) \rceil - 1) \rceil \\ & + \lceil \log_2(\lceil \log_2(\lceil \log_2(n) \rceil - 1) \rceil) \rceil \\ & + \ldots \\ & + 2 \end{aligned}$$

Věta 6.2.4: *Buď n kladné přirozené číslo. Potom $|\omega(n)| \leq 2 \log_2(n)$.*

Důkaz: Zápis vstupního čísla spolu s ukončovací nulou má velikost $\lceil \log_2(n) \rceil + 1$. Stačí tedy dokázat, že

$$\lceil \log_2(n) \rceil > \text{součet následujících sčítanců až do 2.}$$

Jelikož ovšem $\lceil x \rceil - 1 < x$, plyne toto tvrzení přímo z následujícího lemmatu. □

Lemma 6.2.5: *Buď n přirozené číslo větší než* 4. *Potom platí, že*

$$(*) = \log_2(n) + \log_2(\log_2(n)) + \log_2(\log_2(\log_2(n))) + \ldots + \log_2(\log_2(\ldots)) < n,$$

kde poslední sčítanec je z intervalu $< 1, 2 >$.

Důkaz: Je snadno vidět, že pro každé $n \geq 4$ platí $\log_2(n) < \frac{n}{2}$. Proto je

$$(*) < \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + \frac{n}{2^k} < n \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{n}{2} \sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{n}{2} 2 = n$$

□

Je patrné, že asymptoticky lze hodnotu nejdelšího Eliasova kódu čísla n omezit výrazem $2\lceil \log_2 n \rceil$. Z toho je třeba vycházet při předběžném odhadu velikosti indexových bloků kódovaných Eliasovými kódy.

6.3. Diferenční kódování

V předchozí sekci bylo navrženo kódování přirozených čísel, které obecně menší čísla kóduje kratšími bitovými řetězci. V případě atributů z primární části indexových záznamů lze využít jejich vlastnosti, že se jedná o neklesající (pod)posloupnosti čísel. Stačí místo každé hodnoty kódovat rozdíl od předchozí, v případě prvního čísla potom od nuly.

Toto je zcela triviální metoda, která ovšem mívá více než dobré výsledky. V případě velmi frekventovaných i–entit je jejím výsledkem posloupnost velkého množství malých diferencí (a tedy krátkých bitových řetězců), zatímco v případě i–entit řídkých máme málo diferencí větších (ovšem stále ještě to jsou menší čísla než v původní posloupnosti).

6.4. B–blokové kódování

B–blokové kódování je jiný způsob, jak prefixově zakódovat přirozené číslo (včetně nuly). Jeho myšlenka spočívá v tom, že místo jedné hodnoty se kódují dvě. První je původní číslo dělené nějakou vhodně určenou konstantou B, druhé potom zbytek po tomto dělení. Lze předpokládat, že tyto zbytky jsou s poměrně vysokou pravděpodobností v intervalu $\{0, \ldots, B\}$ rozptýleny rovnoměrněji než původní čísla v intervalu $\{0, \ldots, 2^{\log_2 max}\}$, kde max je největší číslo z příslušné posloupnosti. Proto všechny bity „zbytkové“ části kódových slov budou relativně dobře využity. První část každého slova je kódována prefixovým kódem, neboť může mít proměnnou délku.

Dále budou popsány celkem dvě varianty B–blokového kódování; první z nich se uvádí v literatuře (např. [8]), druhá je rozšířením pro potřeby této práce.

6.4.1. Základní B–blokové kódování

Definice 6.4.1: *Buď b vhodné přirozené číslo ve tvaru 2^k, kde $k \in \mathbb{N}$. Potom* základní B–blokové kódování *je definováno následujícím vztahem:*

$$B_z(n) = \alpha((n-1)\ div\ b + 1) \cdot \beta^{<\log_2 b>}((n-1)\ mod\ b)$$

kde $\beta^{<\log_2 b>}$ je binární kódování, které kódová slova doplňuje zleva nulovými bity na velikost $\log_2 b$.

Například pro $b = 4$ a $n = 45$ dostáváme kódové slovo 00000000000100, pro $b = 8$ máme 000001100 a pro $b = 32$ 101101.

Z uvedeného příkladu je vidět, že efektivita tohoto přístupu kriticky závisí na dobré volbě hodnoty b. Neúměrně malé b implikuje příliš dlouhou unární část kódu; příliš velké má naopak za následek redundanci binární části.

Problém optimálního parametru b pro určitou kódovanou posloupnost hodnot řeší následující věty (jejich důkazy jsou uvedeny v [8]).

Věta 6.4.2: *Buď b vhodný parametr B–blokového kódování, p počet prvků kódované posloupnosti přirozených čísel (včetně nuly) a N horní mez součtu všech jejích prvků. Potom celkový počet bitů potřebných k zakódování této posloupnosti je nejvýše*

$$p(1 + \log_2 b) + \frac{N - p}{b}.$$

Věta 6.4.3: *Buď b vhodný parametr B–blokového kódování, p počet prvků kódované posloupnosti přirozených čísel včetně nuly a N horní mez součtu všech jejích prvků. Potom optimální volba b, která minimalizuje horní mez z věty 6.4.2, je určena vztahem*

$$b_{opt} = \begin{cases} 2^{\lceil \log_2 \frac{N-p}{p} \rceil} & \dots \text{ pro } 1 \leq p \leq N/2 \\ 1 & \dots \text{ pro } p > N/2. \end{cases}$$

6.4.2. Kombinované B–blokové kódování

Při zkoumání klasického B–blokového kódování se objevuje otázka, zda by použití (obecně) velice neefektivního kódu α v první části kódových slov nešlo nahradit efektivnějším kódováním. V kontextu prvních částí této kapitoly se nabízí Eliasovy kódy. Ty jsou navíc prefixové, a proto odpadne nutnost zapisovat oddělovač (bit 0) mezi levou a pravou polovinu každého kódového slova. Výslednou metodu označme jako *kombinované B–blokové kódování* (dále jen kombinované kódování).

Při použití kombinovaného B–blokového kódování je třeba řešit zejména následující dvě otázky:

- Který z dostupných Eliasových kódů zvolit?
- Hodnota b určená metodou z části 6.4.1. je optimální za předpokladu použití α kódu. Jaké je optimální b pro jiný typ Eliasova kódu?

Definice 6.4.4: *Buď Π libovolný prefixový kód a b přirozené číslo, které je mocninou 2. Potom* B–blokové kódování kombinované s prefixovým kódem Π *je zobrazení B_c, které libovolnému přirozenému číslu n přiřazuje kódové slovo podle předpisu*

$$B_c(n) = \Pi((n-1)\ div\ b + 1) \cdot \beta^{<\log_2 b>}((n-1)\ mod\ b).$$

Poznámka 6.4.5: Pro $\Pi = \alpha$ a b spočtené dle návodu v 6.4.1. je výsledkem klasické B–blokové kódování.

Implementace univerzálního indexu umožňuje nastavit Π jako libovolný z Eliasových kódů γ', δ, ω nebo ω'. Knihovna sama tuto volbu neřeší hlavně z důvodů redukce počtu parametrů při rozhodování o optimální podobě kombinovaného kódování pro daný blok. Praktické testy ukazují, že většinou jsou v levé části kódových

slov malá čísla. Proto není vhodné používat kódy δ a ω', které jsou právě pro některé malé hodnoty o jeden bit delší. Jako nejefektivnější volba v obecném případě se ukazuje kód ω.

Důležitějším rozhodnutím je modifikace parametru b tak, aby kombinované kódování přinášelo znatelnou úsporu oproti klasickému. Ve druhém případě totiž jeho volba musí být taková, aby součet unárních částí kódových slov příliš nenarostl. Při použití efektivnějšího kódu než unárního tato část kódových slov tolik nenarůstá a zbytková část by tedy při zachování původního stanovení b mohla být zbytečně velká. Z uvedeného důvodu se budeme snažit tuto část zmenšovat (budeme hovořit o *redukci* parametru b).

Jako problém se ukazuje značná nepravidelnost rozdílů mezi délkou unárního kódování a délkou kódu ω, zejména pro menší hodnoty. Pro některé z nich je dokonce kód α efektivnější než ω (viz tabulka 6.1). Důsledkem mohou v extrémním případě (např. určité posloupnosti s menším rozptylem a „nevhodnou“ střední hodnotou) být ztráty až několik bitů na jeden prvek kódované posloupnosti.

n	$\alpha(n)$	$\omega(n)$	$\lvert\alpha(n)\rvert$	$\lvert\omega(n)\rvert$	$\lvert\alpha(n)-\omega(n)\rvert$
1	1	1	1	1	0
2	01	100	2	3	-1
3	001	110	3	3	0
4	0001	101000	4	6	-2
5	00001	101010	5	6	-1
6	000001	101100	6	6	0
7	0000001	101110	7	6	1
8	00000001	1110000	8	7	1
9	000000001	1110010	9	7	2
10	0000000001	1110100	10	7	3
11	00000000001	1110110	11	7	4
12	000000000001	1111000	12	7	5
13	0000000000001	1111010	13	7	6
14	00000000000001	1111100	14	7	7
15	000000000000001	1111110	15	7	8
16	0000000000000001	10100100000	16	11	5

Tabulka 6.1: Prvních patnáct kódů α a ω a jejich délky

Zvoleným řešení uvedeného problému je adaptivní volba mezi základním a kombinovaným B–blokovým kódováním, přičemž ve druhém případě se navíc hledá optimální redukce parametru b.

Bohužel se nepodařilo najít žádnou přímočarou metodu pro stanovení optimálních parametrů kombinovaného B–blokového kódování, zejména kvůli zmíněným nepravidelnostem. Proto je nutné použít nepříliš elegantní řešení, které sestává z následujících kroků:

1. *První průchod blokem.* Během této fáze se provede prvotní analýza bloku tak,

jak byla popsána výše (stanovení optimálního parametru b pro klasické kódování, ...).

2. *Heuristické určení redukce parametru b pro případ kombinovaného kódování.* Viz dále.

3. *Určení optimálního kódování.* Tím je kódování s nejlepším výsledkem mezi všemi kódováními vyzkoušenými v předchozím kroku a klasickým b–blokovým kódováním.

Heuristické určení optimální redukce b je časově nejsložitější fází. Probíhá tak, že se opakovaně prochází blok a počítá se délka jeho zakódovaného obrazu s použitím různých redukcí parametru b (vycházejících z parametru určeného pro klasické kódování). Z výsledných délek a z délky čistého klasického zakódování je potom zvolena optimální metoda.

Nebylo by ovšem vhodné zkoušet blok kódovat pro všechny možné redukce od 0 do b.

Uvažujme posloupnost $\mathcal{B}$ délek zakódování nějakého bloku pro všechny přípustné redukce b. Experimenty na reálných kolekcích ukazují[3], že pro velkou většinu bloků lze tato posloupnost rozložit na dvě posloupnosti, z nichž první je nerostoucí a druhá neklesající. Optimální redukce je potom posledním prvkem první z posloupností (případně prvním prvkem druhé).

Uvedené pozorování ovšem neplatí obecně u všech bloků. Jako jednoduchý příklad lze uvést blok sestávající z hodnot 3 303 a 17 959; pro něj je posloupnost $\mathcal{B}$ rovna 32, 30, 31, 32, 37, 36, 36, 39, 40, 40, 40, 41, 41, 41. Tuto posloupnost na podposloupnosti s danými omezeními nelze rozdělit. Pořád zde ovšem platí, že optimální délka je posledním prvkem nerostoucí podposloupnosti začínající prvním prvkem té původní. Obecně však tato vlastnost nemusí být zaručena.

Zde se proto uplatní zmíněný heuristický přístup; cyklus ve fázi heuristického určení redukce b se opakuje jen do té doby, dokud spočtené délky zakódovaného obrazu daného bloku *nerostou*. Empirické testování vypovídá, že v naprosté většině případů nalezneme optimum. Navíc platí, že i pokud toto optimum nenalezneme, výsledek nemůže být horší než původní (klasické) kódování, neboť v takovém případě by byla pro daný blok adaptivně zvolena právě klasická metoda.

Průměrná velikost redukce se v testovaných kolekcích pohybuje kolem $0,6$ bitu;[4] tato hodnota zvětšená o 1 je přitom zřejmě i průměrný počet průchodů bloky ve fázi heuristického určení redukce. Tyto průchody jsou však prováděny nad daty v hlavní paměti, takže jejich čas můžeme pokládat za zanedbatelný v porovnání s časem zápisu bloků na disk.

Kompresní zisk kombinovaného kódování oproti klasickému se přitom v testech pohyboval přibližně od $1,5$ do $10\,\%$.

Uveďme nyní konkrétní příklad. Pro jednoduchost předpokládejme vstupní posloupnost, ve které je osmdesátkrát hodnota 40 a dvakrát 6 000. Věty 6.4.3 dává op-

[3]Podrobněji o testech viz kapitola 8.

[4]Maximální redukce se pohybovala kolem 14 bitů.

timální hodnotu b jako $2^{\lceil \log_2 (15\,200-82)/82 \rceil} = 256$. Množina levých částí kódových slov B–blokového kódování tedy bude obsahovat osmdesátkrát hodnotu 1 a dvakrát 24. Na zakódování pravých částí v obou případech potřebujeme celkem $8 \cdot 82 = 656$ bitů. Levé části při kódování α zaberou prostor 128 bitů, zatímco při kódování ω pouze 102 bitů[5]. Rozdíl je tedy 26 bitů. To znamená kompresní zisk oproti klasickému B–blokovému kódování necelá 3,5 %. V praxi tento zisk může být i výrazně vyšší. Intuitivně lze říci, že bude tím vyšší, čím více bude ve vstupní posloupnosti hodnot několikanásobně vyšších než je aritmetický průměr celé této posloupnosti. Tedy čím více nebo čím vzdálenějších shluků se bude v textové kolekci vyskytovat.

Data použitá ve výše uvedeném příkladu nemusí být nijak výrazně vzdálená od praxe; odpovídají situaci, kdy se v kolekci nacházejí shluky dokumentů s podobným tématem (v tomto případě konkrétně 3 shluky), které byly do DIS vkládány po sobě. Ty obsahují více stejných („specializovaných“) termů. Naopak mezi těmito shluky je větší prostor dokumentů pojednávajících o něčem jiném. Stejná situace je i v případě izolovaných rozsáhlejších specializovaných dokumentů.

Na závěr této části poznamenejme, že adaptivní kombinované kódování vyžaduje oproti základní klasické verzi více strojového času pouze ve fázi komprese (konkrétně její první části – analýzy). Složitost dekomprese je v obou případech stejná.

6.5. RLE komprese znaménkových bitů

Další možností úspory prostoru je vhodné kódování znaménkových bitů. V případě obecných atributů celočíselného typu nelze předvídat rozložení jejich hodnot. Pro posloupnosti, kde uživatel předpokládá střídání delších sekvencí kladných a záporných čísel, může být výhodná právě RLE komprese znaménkových bitů. Jinak se délka všech indexových záznamů prodlouží o jeden znaménkový bit.

Stejný problém se týká i atributů v primární části indexových záznamů. Tam se sice vyskytují jen kladné hodnoty, nicméně posloupnost čísel v atributu, podle kterého není tříděno primárně, se může skládat z několika neklesajících podposloupností. Přitom celková řada neklesající být nemusí. Při použití diferenčního kódování se na hranicích mezi těmito podposloupnostmi ukládají do výsledku záporné hodnoty. Proto má smysl použít i zde RLE kompresi znaménkových bitů.

Tento jev lze ilustrovat na jednoduchém příkladě:

Předpokládejme index, jehož schéma se skládá v primární části ze dvou atributů (sekundární část neuvažujeme). Konkrétně soubor indexových záznamů obsahuje následující dvojice:

$$< 1; 5 >, < 2; 3 >, < 2; 5 >, < 3; 2 > .$$

Primární atribut po diferenčním zakódování bude tvořit posloupnost nezáporných čísel $1, 1, 0, 1$. Diferenčně zakódovaná posloupnost pro sekundární atribut ovšem již bude obsahovat i záporná čísla: $5, -2, 2, -3$. Proto je nutné ukládat tyto atributy i se znaménkem.

[5] $\omega(24) = 10100110000$

Pro kompresi posloupnosti znaménkových bitů je použito klasické *RLE kódování* (run length encoding). To obecně spočívá v tom, že místo každého běhu[6] se pouze zapíše jeho délka a hodnota jednoho prvku, ze kterých běh sestává.

V tomto konkrétním případě je RLE kódování implementováno tak, že první atribut, jehož znaménkový bit je součástí daného běhu, je kódován *i s tímto bitem*. Bezprostředně potom následuje údaj o délce běhu, který je vždy kódován Eliasovým kódem ω bez ohledu na způsob komprese ostatních dat. Další atributy se stejným znaménkem již jsou kódovány bez znaménkového bitu, který je při načítání odvozen ze znaménka prvního atributu v běhu.

6.6. Komprese textových atributů a obecných dat

Pro atributy obsahující text nebo obecná data jsou zřejmě výše uvedené kódovací metody nevhodné. Navíc se jedná o data, jejichž sémantika je v obecném univerzálním indexu prakticky nepředvídatelná. Vhodné řešení by mohlo být implementovat dostatečně obecnou metodu, jako například statické či dynamické huffmanovo kódování, případně nějakou variantu slovníkové metody LZW.[7] V současné době ovšem index kompresi vektorových atributů nepodporuje.

6.7. Kombinace kompresních metod

Uvedené kompresní metody lze v univerzálním indexu i kombinovat.[8] Ne každá metoda je však povolená pro každý typ atributu.

Příslušná pravidla a omezení lze shrnout do následujících bodů:

- V současné verzi není podporována žádná komprese atributů typu reálných čísel, řetězců a obecných dat.
- Diferenční kódování lze použít v kombinaci s libovolnými jinými metodami.
- Současné použití Eliasova kódování a B–blokového kódování vede na *kombinované B–blokové kódování* (viz 6.4.2.).

[6] Posloupnost čísel stejné hodnoty – zde stejných znaménkových bitů – se nazývá *běh*.

[7] Na tuto metodu vlastní patent firma Unisys, která za její využití donedávna vyžadovala licenční poplatky. Během první poloviny roku 2004 však tento patent vypršel a metoda LZW lze nyní používat volně a zdarma.

[8] Tím je míněno, že pro různé atributy v souboru indexových záznamů lze nakonfigurovat různé typy komprese.

Kapitola 7

Implementace a testování

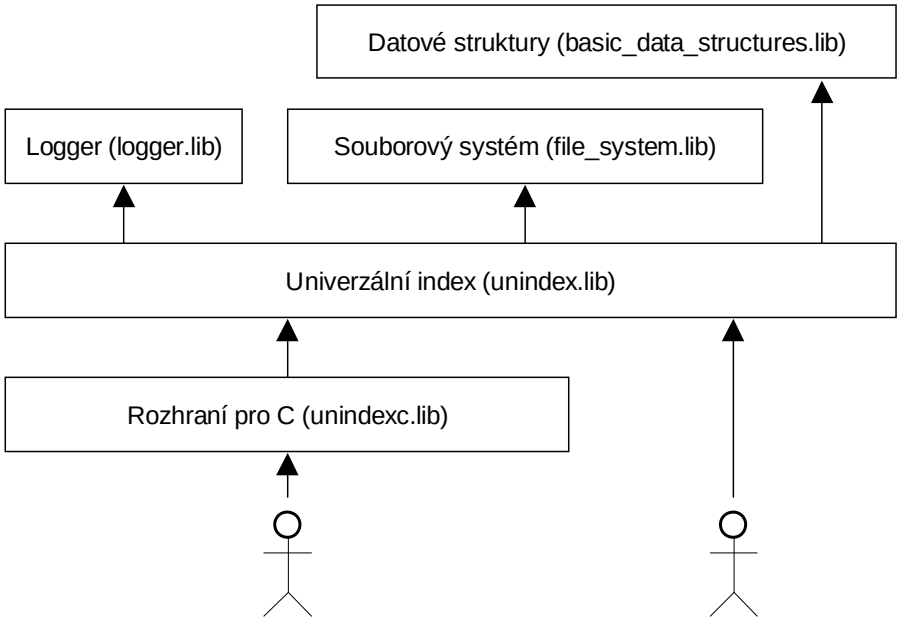
Cílem této kapitoly je především prezentovat některé implementační záležitosti, které nebyly uvedeny v předchozí „implementačně nezávislé“ kapitole. Přitom není cílem, především z důvodů rozsahu, se zde zabývat architekturou implementace příliš do hloubky. Pro bližší informace je na přiloženém CD dostupná vygenerovaná vývojová dokumentace a dále potom popis C-interface v příloze A. V závěru této kapitoly se nachází stručný popis jednoduchého rozhraní, pomocí kterého lze ověřit funkčnost univerzálního indexu.

7.1. Celková architektura

Implementace univerzálního indexu je přenositelná mezi platformami Windows32 a Linux. Pro snadný překlad na obou platformách je do projektu začleněn i speciální opensource nástroj *Jam* (více v části 7.5.).

Celý modul univerzálního indexu není tvořen pouze jedinou knihovnou v jazyce C++, ale skládá se z několika dílčích (sub)knihoven, jak je znázorněno ve schématu 7.1. Tyto knihovny mají následující význam:

- *Rozhraní pro jazyk C* (`unindexc.lib`) – umožňuje používání indexu i v programech psaných v „čistém“ C, ačkoli vlastní index je implementován v C++.
- *Univerzální index* (`unindex.lib`) – obsahuje jádro implementace indexu.
- *Logger* (`logger.lib`) – Jednoduchý systém pro logování zpráv. Zprávy lze rozlišit podle různých kritérií a podle nich je filtrovat a zapisovat do různých souborů, na konzoli atd. Tím se řeší tzv. „problém ladících hlášek“ (situace, kdy jsou do zdrojových kódů přidávány různé výpisy obalené direktivami jako např. `#ifdef _DEBUG_` apod.) a zároveň se umožňuje logování důležitých informací při normálním běhu indexu. V současné implementaci je však využití loggeru spíše okrajové.
- *Souborový systém* (`file_system.lib`) – Knihovna pro práci s adresářovou strukturou nezávislá na platformě (podporuje Windows/Unix).



Obrázek 7.1: Základní knihovny využité v modulu univerzálního indexu

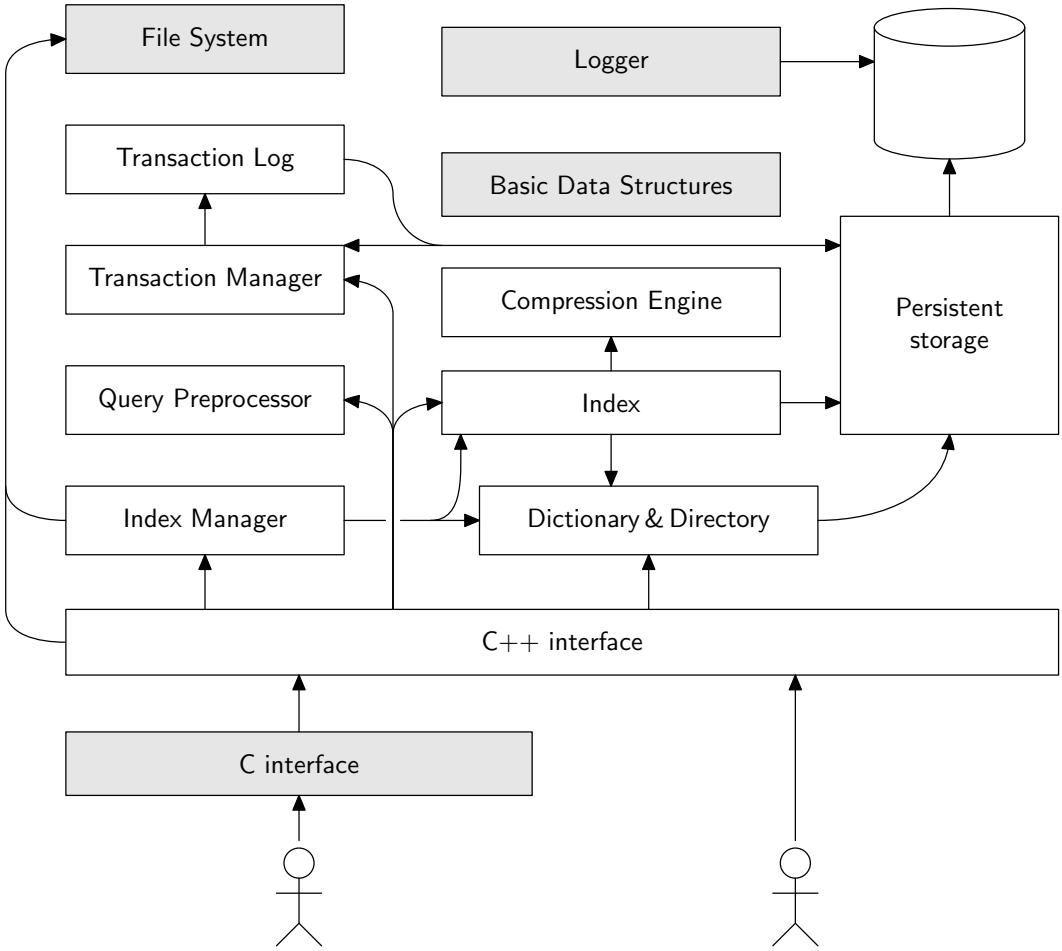
- *Datové struktury* (`basic_data_structures.lib`) – Obsahuje implementaci takových datových struktur, které jsou dostatečně obecné na to, aby se dalo předpokládat, že budou použity i mimo tento projekt. Jejich izolace do zvláštní knihovny usnadňuje reusabilitu kódu. V současné verzi implementace již je pro některé obecné datové struktury použita standardní knihovna STL. V počátcích této práce ještě ovšem některé běžně používané překladače mohly mít s touto knihovnou problémy.
- *Test* (`test.lib`) – Pomocná knihovna (není na obrázku), která obsahuje vývojové testy ostatních modulů (na základě metodiky testování všech oddělených částí systému zároveň při jejich vývoji). Základní ani výkonnostní testy celé knihovny indexu jako celku v knihovně test.lib obsaženy nejsou. Více o testování a experimentování bude uvedeno v kapitole 8.

7.2. Hlavní třídy a moduly systému

Obrázek 7.2 znázorňuje hlavní moduly indexu a jejich vzájemné závislosti (šipka znázorňuje, že zdrojový modul ke své činnosti potřebuje ten cílový). Všechny moduly, kromě těch se šedým podkladem, jsou součástí hlavní knihovny `unindex.lib`. Ty šedé potom zřejmě jednoznačně odpovídají výše uvedeným knihovnám.

Význam jednotlivých modulů:

- *C interface* – bylo popsáno výše.
- *C++ interface* – množina exportovaných prostředků pro manipulaci s indexem z prostředí C++. Fyzicky se nejedná o oddělený modul – C++ rozhraní se



Obrázek 7.2: Základní moduly knihoven univerzálního indexu

skládá z rozhraní několika jiných tříd, konkrétně `IndexManager` a `Index`. To jsou zároveň jediné třídy, které uživatel univerzálního indexu smí používat k manipulaci s ním.

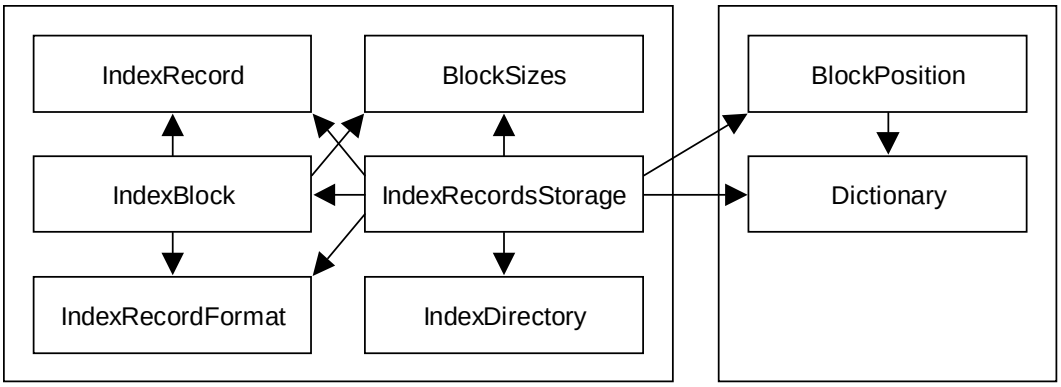
- *Index Manager* – spravuje seznam aktivních indexů (tato knihovna bude umožňovat současně provozovat více indexů například pro několik „paralelně“ existujících textových kolekcí) a k nim příslušných informací.
- *Index* – většina funkcionality vlastního indexu v podobě, jak byl popsán v předchozí kapitole.
- *Dictionary* – slovník mapující identifikátory i–entit na pozice bloků v souboru indexových záznamů.
- *Kompresní engine* – implementuje množinu kompresních algoritmů použitelných pro kompresi bloků (v souboru indexových záznamů).
- *Transaction Manager* – řídí a spravuje transakce.

- *Transaction Log* – datové struktury potřebné pro implementaci transakčního zpracování.
- *Query Preprocessor* – je zodpovědný za korektní manipulaci se strukturami popisujícími dotaz (zkopírování pro potřeby indexu, dealokace, ...).
- *Logger* – byl popsán výše.
- *FileSystem* – obsahuje například funkce pro vytváření, mazání a procházení adresářů, vše nezávisle na platformě (podporuje Windows a Unix/Linux).
- *BasicDataStructures* – bylo popsáno výše.
- *PersistentStorage* – třída, který exportuje rozhraní pro ukládání dat na disk. Více viz 7.4.

Detailnější informace k některým z těchto modulů jsou uvedeny v následujících sekcích.

7.3. Moduly Index a Dictionary & Directory

Tyto moduly se skládají především ze tříd znázorněných na obrázku 7.3.



Obrázek 7.3: Diagram tříd modulů Index a Dictionary & Directory.

Index poskytuje rozhraní ke konkrétní instanci indexu.

Slovník (dictionary) slouží k nalezení pozice bloku, ve kterém je seznam indexových záznamů daného termu, spolu s indexem oblasti. Na začátku života indexu je slovník načten z externího souboru a na konci do něj opět uložen. Jelikož během provozu jsou jeho data neustále v paměti, není příliš podstatná organizace (ani efektivita) tohoto souboru.

Pokud jde o datové struktury, bude slovník (jeho reprezentace v hlavní paměti) používat hašování s řetězci.

IndexBlock implementuje blok, jak byl popsán v předchozí části, tj. instance této třídy spravuje seznam indexových záznamů spolu s obecnými údaji o těchto datech.

IndexRecord zapouzdřuje indexový záznam. Jeho formát je dán jednou instancí třídy `IndexRecordFormat`.

IndexRecordFormat definuje formát indexových záznamů (ten je jednotný pro jeden konkrétní index; různé indexy spravované jedinou instancí knihovny zároveň však mohou mít formáty odlišné). Formát je dán posloupností identifikátorů, které postupně deklarují typy jednotlivých položek. Kromě toho obsahuje informace například o druhu použité komprese a jejích parametrech apod.

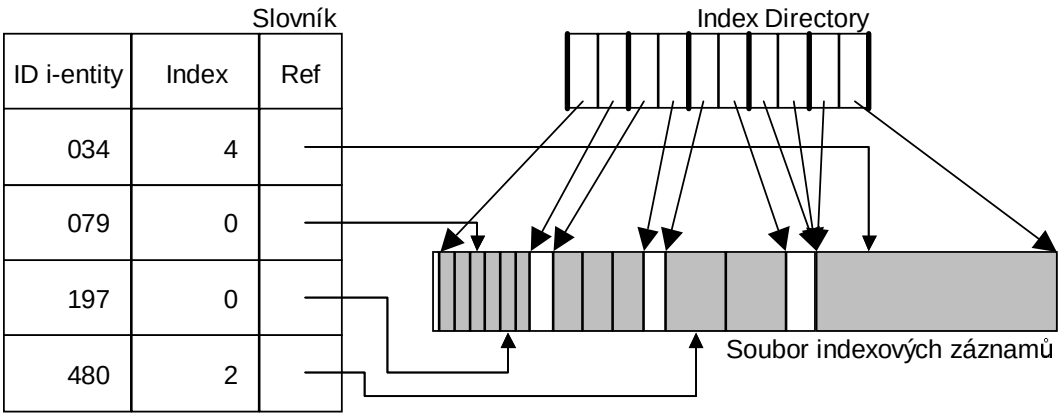
IndexRecordStorage představuje zapouzdřující třídu pro soubor bloků indexových záznamů.

BlockPosition reprezentuje údaje o pozici bloku v souboru indexových záznamů. Obsahuje především následující atributy:

- offset bloku v souboru indexových záznamů;
- index oblasti, ve které se daný blok nachází (počítáno od nuly).

Jakákoli úprava indexu, která si vyžádá manipulaci s bloky v souboru indexových záznamů (přesunutí, posunutí atd.) může mít za následek zneplatnění této struktury. Aktuální pozice jsou vraceny z metod tříd `IndexDirectory` nebo `Dictionary`.

IndexDirectory je pomocná struktura („adresář indexu“), která umožňuje efektivně zjistit začátky a konce jednotlivých oblastí v souboru indexových záznamů (oblast i najdeme v adresáři indexu vždy na stejné pozici). Je tedy nutný k orientaci v tomto souboru. Pokud je oblast j prázdná, ale existuje $k > j$, že k je neprázdná oblast ve stejném souboru, je oblast s indexem j v `IndexDirectory` rovněž uvedena – její počátek je však shodný s jejím koncem (a s koncem oblasti s indexem $j - 1$ nebo začátkem oblasti předchozí).



Obrázek 7.4: Struktura souborů spravovaných moduly Index a Dictionary. Položka `index` znamená index oblasti, ve které se nachází blok odkazovaný přes `ref`. Fyzicky je `ref` offset do souboru indexových záznamů.

BlockSizes je třída obsahující dynamické pole, které pro každý index i obsahuje velikost bloku i–té oblasti. Tato třída tedy eliminuje nutnost opakovaně vypočítávat tyto velikosti (což vyžaduje mocniny v reálné aritmetice a další relativně časově náročné operace). Při inicializaci se spočítá uživatelem určený počet hodnot;pokud během života knihovny přestane tento počet stačit, dopočítají se automaticky další.

Na obrázku 7.4 je schématicky znázorněn vztah jednotlivých datových struktur v těchto modulech.

Údržba souboru `IndexDirectory` v konzistentním stavu během aktualizace je jednoduchá, pokud jsou známy indexy oblastí, ve kterých se provádí nějaké změny. Není to ani nikterak omezující, protože v hlavičkách bloků jsou uvedeny identifikátory i–entit. Stačí tedy danou entitu najít ve slovníku, ve kterém je uveden i index příslušné oblasti. Ten už jednoznačně určuje místo v adresáři, které je třeba opravit. To je důvod, proč je ve slovníku jak ofset v souboru tak i index oblasti.

7.4. Třída PersistentStorage

Jak bylo uvedeno výše, jedná se o třídu, která exportuje rozhraní pro ukládání dat na externí paměťové zařízení. Pomocí této třídy je realizována většina I/O operací univerzálního indexu (výjimkou jsou některé okrajové operace, jako třeba výstup loggeru apod.).

Smyslem `PersistentStorage` je jednak to, že implementuje platformě nezávislé diskové I/O operace (podporován je Linux a Windows32), ale hlavně transparentní[1] *zajištění transakčního zpracování.*[2]

`PersistentStorage` si ve spolupráci se třídou `TransactionManager` sám zjišťuje, zda se má operace zpracovat transakčně, případně získá aktivní transakci (ev. vytvoří novou) a automaticky použije transakční struktury, jako je například transakční log.

7.5. Testování

Tato část obsahuje odkazy na instrukce potřebné ke zprovoznění jednoduchého testovacího rozhraní, pomocí kterého lze ověřit funkčnost univerzálního indexu jako celku.

7.5.1. Překlad knihovny univerzálního indexu

Knihovna indexu je implementována tak, aby byla přenositelná jak na platformu Windows, tak na Unix/Linux. K dosažení lepší přenositelnosti je využit speciální

[1] Transparentní zde znamená, že implementace ostatních tříd nemusí vůbec zohledňovat, zda je právě index v módu transakčního zpracování dat nebo ne.

[2] Více o transakčním zpracování bylo uvedeno v kapitole 5.

nástroj pro platformě nezávislý překlad – tzv. Jam. Ten je volně šiřitelný a je umístěn i na přiloženém CD v adresáři `/tools`.

Hlavní konfigurační soubor je `Jamrules`, který obsahuje obecná nastavení a pravidla pro překlad. Objekty pro přeložení se specifikují v souborech `Jamfile`, které jsou umístěny na různých místech zdrojového adresářového stromu, přičemž se na sebe vzájemně odkazují. Hlavní je soubor ve stejném adresáři jako `Jamrules`, kde se specifikuje cíl překladu (práce obsahuje různé cíle, jako vývojové testy nebo rozhraní pro experimenty).

Další informace k překladu produktu lze nalézt na CD v adresáři `/implementation/doxydocs`, případně přímo v dokumentaci programu Jam.

7.5.2. Testovací rozhraní

Univerzální index lze testovat pomocí jednoduchého nástroje příkazové řádky, který se spouští pomocí příkazu `test_general`. Jeho vstupem je speciální XML soubor obsahující popis operace, kterou tento program vykoná, včetně vstupních a výstupních souborů. Cesta ke konfiguračnímu XML je v programu explicitně uvedená v souladu se strukturou CD; lze však explicitně změnit pomocí jediného parametru příkazové řádky.

Jedná se pouze o testovací nástroj, u kterého není kladen žádný důraz na robustnost. V důsledku toho i pro načítání XML je použit triviální parser `TinyXML`, který nevaliduje soubor oproti žádnému schématu ani DTD a nehlásí příliš srozumitelně případné chyby. Uživatel proto musí sám předem zajistit, aby XML odpovídalo tomu, co předpokládá tento testovací nástroj. V adresáři `/implementation/dist/xml` jsou již nějaké ukázkové xml soubory, které byly otestovány a měly by být správné.

Dokumentaci struktury těchto XML dat i vlastního nástroje pro provedení experimentů lze najít buď v uvedeném adresáři v souboru `test_config_example.xml`, kde jsou jednotlivé elementy dobře okomentovány, nebo ve vygenerované dokumentaci v adresáři `/implementation/doxydocs`.

Kapitola 8

Experimenty

V této kapitole jsou popsány experimenty, které byly s univerzálním indexem provedeny. Tyto experimenty se zaměřují především na

- experimentální zjištění časové složitosti operace *insert*;
- vliv volby parametru κ na parametry indexu;
- vliv různých kompresních metod na výsledný prostor obsazený souborem indexových záznamů a na vliv těchto metod na časovou efektivitu operací nad indexem;
- vliv použití transakčního zpracování na časovou efektivitu operací.

Poznámka 8.0.1: Konkrétní reporty zde uvedených experimentů lze nalézt na přiloženém CD v adresáři `/reports`. Tam se nacházejí také informace o dalších experimentech, které v této kapitole z důvodu rozsahu nebyly zmíněny.

8.1. Data pro experimenty

Experimenty popsané v této kapitole pracují s následujícími kolekcemi testovacích dat:

1. *Náhodně vygenerovaná data, která respektují Zipfův zákon* (viz 2.5.) To znamená, že při seřazení i–entit sestupně dle četností jejich výskytů klesá jejich četnost exponenciálně. Uvedený soubor dat je v tabulkách s výsledky testů označen zkratkou *RNDZ*. Z důvodu příliš velkého nárůstu četností výskytů bylo vygenerováno pouze cca $1,2$ miliónu záznamů.
2. *Náhodně vygenerovaná data z rovnoměrného rozdělení.* To znamená, že četnost výskytů všech i–entit je přibližně rovnoměrná. Označení pro tato data z rovnoměrného rozdělení je *RAND*.

3. *Sada dat z textů z Lidových novin* (zkratka *LN*). Tato data jsou podrobněji popsána v 8.1.1.

4. *Sada textů z časopisu Computer World* (zkratka *CW*). Více informací je uvedeno v 8.1.2.

Ve většině experimentů jsou uvedená data z praktických důvodů ještě předzpracována – seskupena po blocích 100 záznamů stejné i-entity, přičemž tyto bloky jsou náhodně rozptýleny ve vstupním souboru.

8.1.1. Datová kolekce *LN*

Kolekce *LN* pochází z [7] a obsahuje články (texty), které poskytla redakce deníku Lidové noviny. Příslušné texty pokrývají poměrně široké spektrum témat.

Tato data jsou převzata již v předzpracované podobě, tj. v této práci již máme k dispozici data ve formátu záznamů

< identifikátor i–entity; identifikátor t–entity; četnost termu v dokumentu > .

Přitom z původní kolekce byly vyřazeny všechny dokumenty, které obsahovaly více než 4 % nerozpoznaných slov (opatření kvůli vyřazení irelevantních textů, jako např. texty v cizím jazyce či tabulky s výsledky sportovních utkání). Dále všechny termy ve vstupní kolekci byly převedeny na lemmata,[1] tzn. že jednotlivé identifikátory i–entit již odpovídají lemmatům, nikoli původním slovům.

Přiřazení konkrétních identifikátorů lemmatům, resp. dokumentům není pro účely experimentů podstatné a není součástí této práce. Zrovna tak nebudeme pracovat ani s atributem *četnost termu v dokumentu* – ten bude použit pouze pro posouzení, nakolik data odpovídají Zipfovu zákonu.

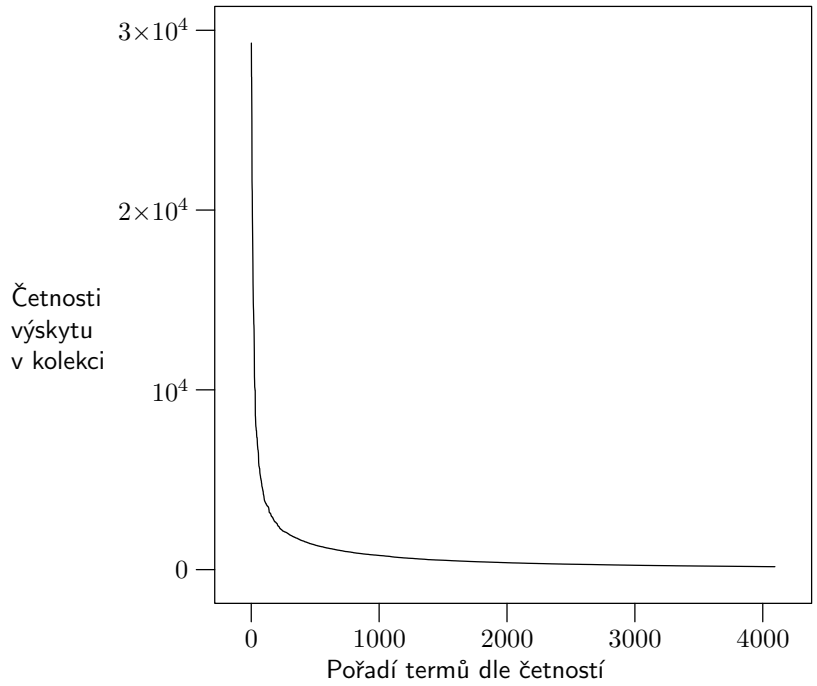
Distribuce dat v kolekci *LN* je znázorněna na obrázku 8.1. Toto zobrazení zanedbává atribut popisující četnost výskytu termů v dokumentech (tj. pokud se konkrétní term v nějakém dokumentu vyskytuje víckrát, je zde započítán pouze jediný výskyt). Na grafu v tomto měřítku by však stávající křivka s křivkou vykreslenou se zohledněním četnosti výskytů v konkrétních dokumentech opticky splývaly.

Rozdíl mezi oběma typy dat je lépe vidět na obrázku 8.2, kde je na osách použita logaritmická stupnice. Je patrné, že při započítání zmíněných četností data lépe odpovídají Zipfovu zákonu; nicméně z hlediska základního vyhledávání dokumentů není tato informace relevantní (neboť se uchovává v sekundární části indexových záznamů).

8.1.2. Datová kolekce *CW*

Tato kolekce je, podobně jako *LN*, převzata z [7], původně se však jedná o dokumenty poskytnuté redakcí časopisu *Computer World*. V tomto případě jde o odbornější texty zaměřené na počítačovou tématiku.

[1] Základní morfologický tvar slova, tj. u substantiv 1. pád singuláru, u sloves infinitiv apod.



Obrázek 8.1: Distribuce dat v souboru LN

O předzpracování těchto dat platí totéž co o kolekci *LN*, pouze hranice podílu neznámých slov pro vyřazení dokumentu byla posunuta na 10 % (z důvodu vyššího výskytu cizích a odborných výrazů). Distribuce dat je na obrázku 8.3.

Obecná data o použitých experimentálních kolekcích shrnuje následující tabulka:

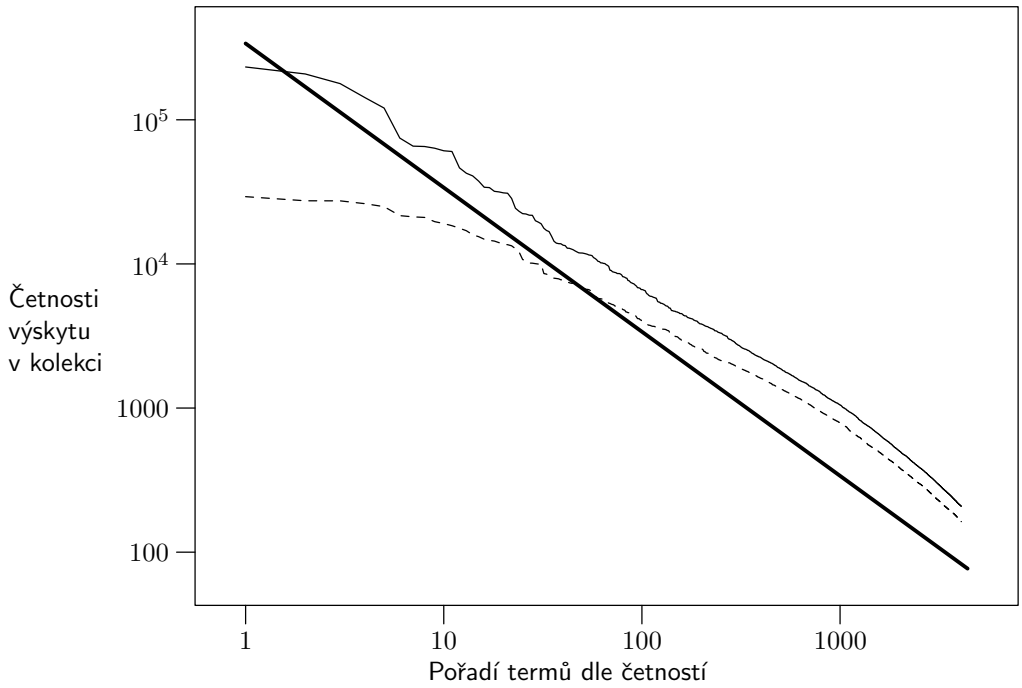
	LN	*CW*
Název souboru[2]	export-ld-CID42-LN-32377doc	export-ld-features-165
Počet dokumentů	32 377	9 815
Počet různých i–entit	161 130	171 688
Celkový počet i–entit	7 311 087	8 024 681
Počet ind. záznamů[3]	4 504 938	3 547 437
	RNDZ	*RAND*
Název souboru	random-zipf	random-norm
Počet dokumentů	10 000	10 000
Počet různých i–entit	100 000	100 000
Celkový počet i–entit	1 300 000	1 166 750

8.2. Časová složitost operace *insert*

Poznámka: Tento experiment, stejně jako všechny další v této kapitole, byl proveden na stroji s procesorem AMD Athlon64 3200+ s 2GB RAM a pevným diskem

[3]Soubory jsou na přiloženém CD v adresáři `data`.

[3]Bez zohlednění atributu *četnost termu*, tj. v podstatě počet řádek v datovém souboru.



Obrázek 8.2: Distribuce dat v souboru *LN*. Plná čára znázorňuje distribuci výskytů termů v celé kolekci (zohledňuje i četnosti výskytů v jednotlivých dokumentech). Tučně je potom znázorněna „ideální“ distribuce dle Zipfova zákona, přičemž konstanta je spočtena pomocí regresní analýzy metodou nejmenších čtverců. Čárkovanou čárou je vyznačena distribuce dat při zanedbání četností v jednotlivých dokumentech (tj. uvažuje se maximálně jeden výskyt dané i–entity v konkrétním dokumentu).

WD2500KS (7200 RPM). Použitými operačními systémy byly Mandriva Linux a Windows XP Home SP2 (verze 2002).

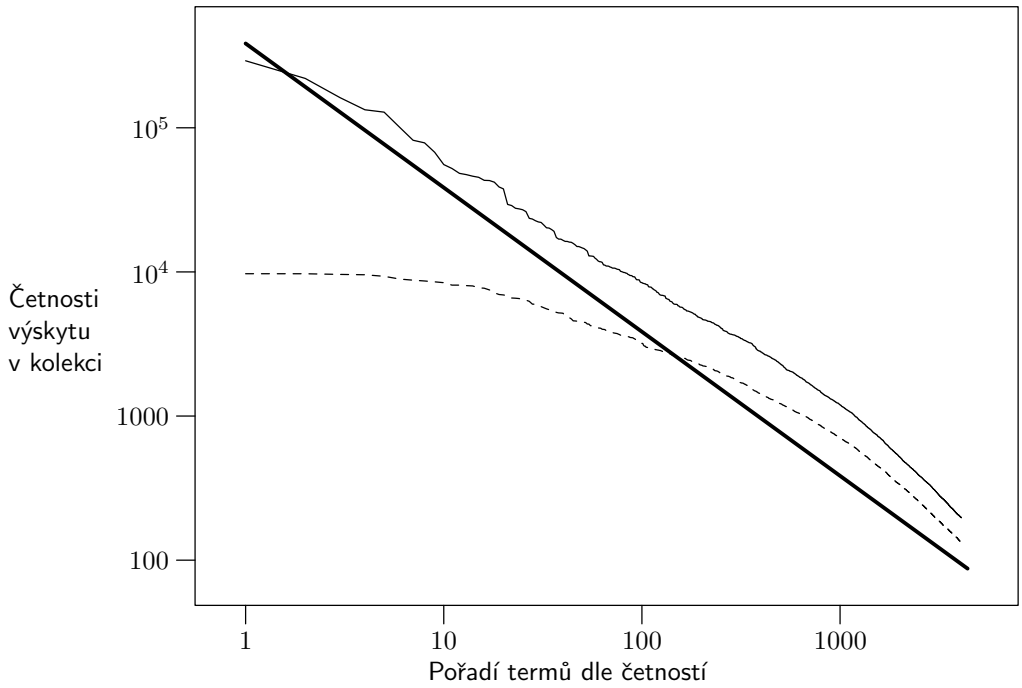
V kapitole 4 bylo teoreticky odvozeno, že amortizovaná složitost operace *insert* je konstantní. Následující experiment tento výsledek ověřuje na testovacích datech.

Způsob jeho provedení je takový, že do prázdného univerzálního indexu je postupně vkládáno počátečních n záznamů z konkrétní kolekce testovacích dat, přičemž n postupně nabývá hodnot 300 000, 450 000 atd. Měřený čas v sekundách je rozdíl času ukončení a zahájení testu.

Hodnoty naměřené v operačním systému Linux jsou uvedeny v následující tabulce:

	Velikost vstupních dat (v tisících záznamů)										
	300	450	600	750	900	1050	1200	1350	1500	1650	1800
LN	29	45	63	80	98	114	130	149	165	184	197
CW	26	39	53	67	81	93	106	119	133	148	169
RNDZ	30	45	60	76	89	103					
RAND	29	43	58	71	85	99					

Časy pro Windows jsou potom shrnuty zde:



Obrázek 8.3: Distribuce dat v souboru *CW*. Pro typy čar platí stejné konvence jako byly popsány u obrázku 8.2.

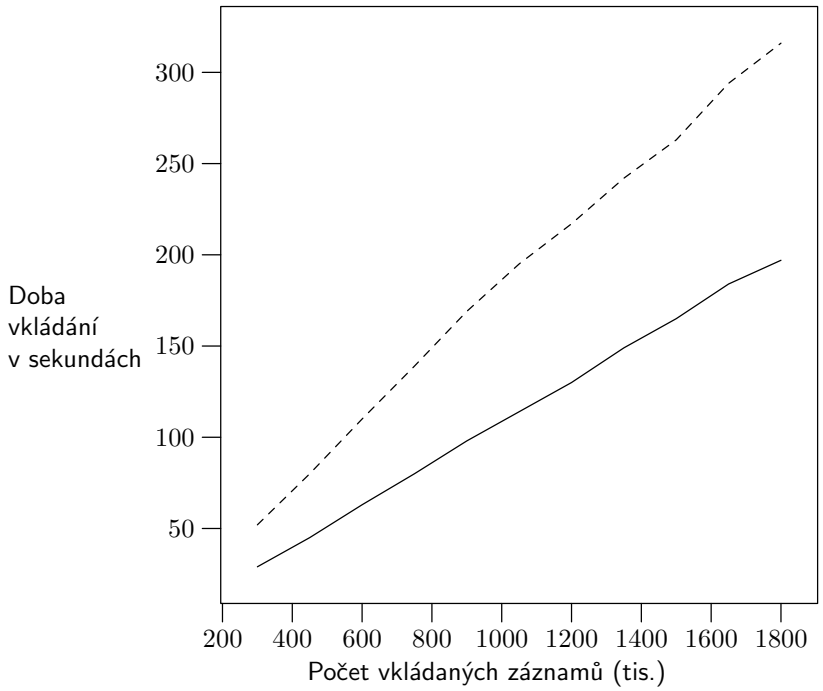
	Velikost vstupních dat (v tisících záznamů)										
	300	450	600	750	900	1050	1200	1350	1500	1650	1800
LN	52	80	110	139	169	195	217	242	263	294	316
CW	38	54	71	89	107	119	140	158	170	180	199
RNDZ	36	52	82	98	113	128					
RAND	36	53	67	80	101	104					

Výsledná data pro kolekci *LN* jsou graficky znázorněna na diagramu 8.4. Vizuálně je ihned patrná lineární závislost doby vkládání na počtu vkládaných záznamů. Z toho plyne, že průměrná doba na vložení jednoho záznamu v rámci posloupnosti operací (amortizovaná složitost) je konstantní. Pro ostatní kolekce není diagram uveden – jak se ovšem lze snadno přesvědčit, typ závislosti je ten samý.

Zajímavý je pomalejší růst doby vkládání, pokud experiment běží v operačním systému Linux (na diagramu plná čára). Tento rozdíl by šel vysvětlit např. lepší správou přístupu k disku, lepším cachováním a nebo větší prioritou defaultně přidělovanou procesům. Skutečnou příčinu tohoto jevu v této práci nebudeme zkoumat.

8.3. Vliv volby parametru κ na dynamický invertovaný soubor

Pro experimentální zkoumání vlivu hodnoty parametru κ na výsledné parametry indexu byla z důvodu rozsahu zvolena pouze jedna z testovacích kolekcí, a to kolekce



Obrázek 8.4: Závislost doby vkládání do indexu na počtu vstupních záznamů. Plnou čárou je zobrazen vztah pro Linux, čárkovaně potom pro Windows.

LN seskupená do skupin po 100 záznamech. Výsledky tohoto experimentu budou uvedeny pouze pro platformu Windows.

Sledovanými veličinami jsou zejména počet expanzí, jejich průměrná délka[4], průměrný počet uložených bytů a součet velikostí oblastí, resp. děr v souboru indexových záznamů.

Z níže uvedené tabulky je patrné, že s rostoucí hodnotou κ parametru roste velikost souboru indexových záznamů a zároveň velmi výrazně klesá doba běhu experimentu. Je to proto, že pro větší hodnoty κ rychleji rostou velikosti bloků v jednotlivých oblastech. Proto méně přesně „kopírují“ skutečné délky seznamů indexových záznamů a v ocasech bloků zůstává více nevyužitého místa. Protože se však do bloků vejde více dat, není tak často nutná expanze bloku a také je tato expanze kratší. Zřejmě tedy vhodnou volbou hodnoty κ lze radikálně ovlivnit efektivitu dynamického invertovaného souboru. Poněkud překvapivým zjištěním v tomto experimentu je fakt, že v literatuře doporučovaná hodnota $\kappa = 1/0,84$ zřejmě není pro optimální chování datové struktury příliš vhodná (právě z důvodu, že zbytečně přesně kopíruje skutečné rozložení dat a nenechává v ocasech bloků příliš mnoho rezervního prostoru; z toho důvodu je častěji nutná expanze bloku).

[4]Délka expanze bloku je rovna počtu různých bloků, které v rámci této expanze musely být v dynamickém invertovaném souboru přesunuty, vyjma právě vkládaného bloku. Pokud nějaký blok expanduje, ale po zvětšení se vejde na své původní místo, jedná se o expanzi délky 0.

	κ		
	1, 19	1, 55	1, 90
doba běhu	273 s	197 s	159 s
počet expanzí	18 765	14 058	11 137
průměrná délka expanze	4, 39	2, 30	1, 83
průměrně bytů v expanzi	4 483, 82	3 166, 2	3 245, 53
velikost oblastí	7, 88 MB	8, 69 MB	9, 57 MB
velikost děr	21, 97 KB	20, 99 KB	17, 35 KB

8.4. Experimenty s kompresními metodami

Cílem těchto experimentů je hlavně porovnat efektivitu jednotlivých kompresních metod a jejich kombinací, jak byly popsány v kapitole 6. Dále potom by tyto experimenty měly potvrdit (nebo vyvrátit) hypotézu, že použití komprese může výrazně zmenšit čas zpracování operací nad indexem (omezíme se zde na operaci *insert*) z důvodu redukce množství zapisovaných diskových bloků.

Vstupem těchto experimentů je postupně každá z výše popsaných kolekcí dat, která je vkládána do prázdného univerzálního indexu. Po úplném vložení celé kolekce je vždy vyhodnocena celková velikost souboru indexových záznamů, součet velikostí oblastí (tj. celková velikost souboru bez děr), celkový čas potřebný k vložení všech záznamů, počet expanzí bloků a množství dat zapsaných celkem do souboru indexových záznamů nebo z něj načtených. Dále je v každé tabulce uveden kompresní poměr oproti variantě „bez komprese“, přičemž jako data pro výpočet se bere velikost těl bloků.

Z důvodu rozsahu je experiment omezen pouze na některé varianty komprese. Výsledky všech zvolených variant jsou shrnuty v následujících sekcích.

8.4.1. Žádná kompresní metoda

Tento test slouží pro získání referenčních hodnot pro spočtení výkonnosti jednotlivých kompresních metod. Není použita žádná kompresní metoda. Výsledky jsou shrnuty v následující tabulce a nevyžadují žádný komentář.

	LN	*CW*	*RNDZ*	*RAND*
velikost oblastí	19,43 MB	14,98 MB	5,07 MB	5,63 MB
velikost děr	39,00 KB	35,66 KB	1,54 KB	0,94 KB
velikost těl bloků	17,18 MB	13,53 MB	4,45 MB	4,96 MB
čas vkládání	752 s	214 s	93 s	89 s
expanze bloků	27 196	8 257	8 407	8 412
průměrná délka expanze	5,67	5,35	1,91	1,51
průměrně dat v expanzi	9 228,73 B	18 977 B	1 309,98 B	1 096,13 B
zapsaných dat celkem	257,16 MB	163,15 MB	15,14 MB	13,94 MB
kompresní poměr	100 %	100 %	100 %	100 %

8.4.2. Diferenční kódování a B-blokové kódování

Diferenční kódování samo o sobě nepřináší žádnou úsporu – proto je třeba jej testovat ve spojení s nějakou metodou pro kódování čísel s proměnnou délkou kódového slova. V následující tabulce je výsledek spojení diferenčního kódování s B-blokovým (viz 6.3. a 6.4.).

	LN	*CW*	*RNDZ*	*RAND*
velikost oblastí	7,67 MB	5,16 MB	2,14 MB	2,30 MB
velikost děr	14,67 KB	11,26 KB	0,22 KB	0,25 KB
velikost těl bloků	6,42 MB	4,54 MB	1,77 MB	1,92 MB
čas vkládání	716 s	202 s	74 s	92 s
expanze bloků	27 256	8 208	8 504	8 439
průměrná délka expanze	5,45	5,10	1,11	1,56
průměrně dat v expanzi	3 273,89 B	5 780,03 B	294,37 B	469,33 B
zapsaných dat celkem	92,13 MB	163,15 MB	4,35 MB	5,89 MB
kompresní poměr	37, 37 %	33, 56 %	39, 78 %	38, 71 %

Je vidět, že použitím této relativně jednoduché komprese bylo dosaženo kompresních poměrů pod 40 %, a to shodně jak pro kolekce odpovídající Zipfovu zákonu tak i pro kolekci s rovnoměrným rozložením. Pro tři z kolekcí se i zmenšil čas potřebný na vkládání, ovšem nikoli výrazně.

8.4.3. Diferenční kódování a Eliasův kód ω'

Následují výsledky kombinace diferenčního kódování s Eliasovým ω' kódem (viz 6.3. a definice 6.2.2).

	LN	*CW*	*RNDZ*	*RAND*
velikost oblastí	7,39 MB	4,72 MB	1,73 MB	3,10 MB
velikost děr	5,95 KB	10,12 KB	0,42 KB	0,38 KB
velikost těl bloků	6,16 MB	4,14 MB	1,40 MB	2,66 MB
čas vkládání	700 s	192 s	78 s	74 s
expanze bloků	27 217	8 264	8 487	8 545
průměrná délka expanze	5,37	5,04	1,31	0,99
průměrně dat v expanzi	2 983,74 B	5 192,57 B	284,76 B	377,77 B
zapsaných dat celkem	84,23 MB	45,25 MB	3,90 MB	5,93 MB
kompresní poměr	35, 86 %	30, 60 %	31, 46 %	53, 63 %

Diferenční kódování kombinované s kódem ω' dává překvapivě lepší výsledky než kombinace s B-blokovým kódem. Kompresní poměry pro data s rozložením Zipfova zákona se blíží k hranici 30 % – ovšem pro rovnoměrně rozložená data je výsledný poměr výrazně horší, přes 53 %. To lze vysvětlit tím, že Eliasovy kódy kódují efektivněji

[4]Soubor indexových záznamů, zde speciálně dynamický invertovaný soubor.

menší čísla – přitom malých diferencí je v datech díky Zipfovu zákonu výrazně více než velkých (méně efektivně kódovaných). B-blokové kódování bere naproti tomu v úvahu globálně celou konkrétní sadu dat, proto dává slušné výsledky i pro kolekci *RAND*.

Pro dokumentografické systémy se tedy zdá lepší dát přednost Eliasovu kódu ω' před B-blokovým kódováním.

8.4.4. Diferenční a kombinované kódování ω

Výsledky diferenčního kódování ve spojení s kombinovaným ω kódem jsou v následující tabulce. Pro více informací o kombinovaném kódování viz 6.4.2.

	LN	*CW*	*RNDZ*	*RAND*
velikost oblastí	6,41 MB	4,21 MB	1,70 MB	2,31 MB
velikost děr	13,74 KB	5,93 KB	0,18 KB	0,26 KB
velikost těl bloků	5,27 MB	3,67 MB	1,37 MB	1,93 MB
čas vkládání	746 s	239 s	103 s	91 s
expanze bloků	27 167	8 293	8 462	8 545
průměrná délka expanze	5,45	5,14	1,41	1,39
průměrně dat v expanzi	2 733,13 B	4 902,28 B	303,36 B	416,26 B
zapsaných dat celkem	76,70 MB	42,63 MB	4,01 MB	5,50 MB
kompresní poměr	30, 68 %	27, 12 %	30, 79 %	38, 91 %

Diferenční a kombinované kódování ω dává zatím nejlepší výsledky – nejenže kompresní poměry se u dat se Zipfovým rozložením pohybují těsně kolem 30 % a v některých případech i pod touto hranicí, ale poměrně dobře zkomprimovaná jsou i data s rovnoměrným rozdělením.

Na druhou stranu, čas potřebný pro vložení všech záznamů je ze všech dosud uvedených metod nejhorší. To je způsobeno výpočetně náročným hledáním optimálního posunutí v metodě kombinovaného kódování.

8.4.5. Diferenční a kombinované kódování ω'

Výsledky diferenčního kódování ve spojení s kombinovaným ω' kódem jsou v následující tabulce:

	LN	*CW*	*RNDZ*	*RAND*
velikost oblastí	6,58 MB	4,36 MB	1,71 MB	2,38 MB
velikost děr	9,85 KB	9,16 KB	0,08 KB	0,27 KB
velikost těl bloků	5,42 MB	3,81 MB	1,39 MB	1,99 MB
čas vkládání	760 s	243 s	108 s	99 s
expanze bloků	27 204	8 246	8 455	8 450
průměrná délka expanze	5,46	5,13	1,36	1,40
průměrně dat v expanzi	2 804,66 B	4 984,08 B	294,06 B	425,67 B
zapsaných dat celkem	78,80 MB	43,19 MB	3,95 MB	5,61 MB
kompresní poměr	31, 55 %	28, 16 %	31, 24 %	40, 12 %

Výsledky tohoto testu jsou ve všech ohledech horší než výsledky diferenčního a kombinovaného kódování ω', čímž se potvrzuje hypotéza uvedená z dřívějších kapitol o větší perspektivě kódu ω v kombinovaném kódování.

8.4.6. Závěr experimentů s kompresí

Z uvedených dat plyne, že použitím komprese lze výrazně snížit velikost potřebnou pro uložení dynamického invertovaného souboru. Prakticky dosahované kompresní poměry se v experimentech pohybovaly i pod hranicí 30 %. Volbou vhodné kompresní metody lze přitom volit mezi větší úsporou místa a rychlejším zpracováním.

Nejlepší výsledky přináší kombinované kódování, které se skládá z diferenčního, B-blokového a Eliasova kódování. Na druhou stranu důsledkem této metody je až o 25 % větší doba zpracování. Pokud je tato doba kritická, je lepší volit kombinaci diferenčního a B-blokového kódování. Pro data distribuovaná dle Zipfova zákona dává lepší výsledky varianta s Eliasovým kódem ω', zatímco pro obecná data je výhodnější verze s ω.

8.5. Vliv transakčního zpracování na efektivitu

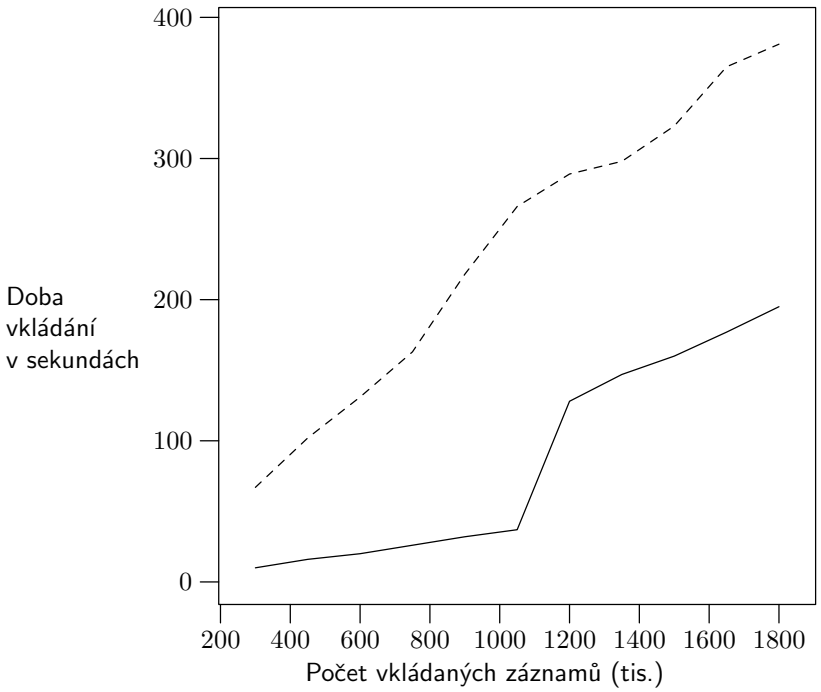
Experimenty s transakcemi mají za cíl zjistit vliv transakčního zpracování na čas potřebný pro provedení operací. Omezí se přitom, stejně jako v ostatních případech, především na *insert*.

Existuje předpoklad, že efektivitu transakčního zpracování ovlivňuje především použití speciálního API pro okamžitý zápis dat na disk bez použití bufferů operačního systému nebo disku (v okamžiku, kdy dojde k návratu z funkce pro zápis, lze předpokládat, že data jsou již permanentně zapsána na disku). Toto API je, stejně jako implementace bufferů, rozdílné pro Windows a Linux, a proto následující experimenty jsou provedeny v prostředí obou těchto systémů.

Pro snadné srovnání je použita stejná metodika jako při experimentálním ověřování časové složitosti v části 8.2., tedy pro každou kolekci jsou vždy do prázdného indexu vkládána data představující nejdříve 300 000 prvních záznamů z této kolekce, potom opět do prázdného indexu jsou vkládána data velikosti 450 000 atd. Výsledné časy jsou potom porovnány s hodnotami z části 8.2.

Následující tabulka uvádí naměřené hodnoty, přičemž se omezuje pouze na data z kolekce *LN*. První řádek obsahuje hodnoty pro Linux a druhý řádek pro Windows. V případě Windows se jedná o průměr hodnot naměřených při trojnásobném opakování experimentu (je to z toho důvodu, že v tomto případě se výsledné hodnoty při opakování nezanedbatelně lišily a chovaly se poměrně nepravidelně).

	Velikost vstupních dat (v tisících záznamů)										
	300	450	600	750	900	1050	1200	1350	1500	1650	1800
LN-Lin	10	16	20	26	32	37	128	147	160	177	195
LN-Win	67	102	131	163	218	266	289	298	323	365	381



Obrázek 8.5: Závislost doby vkládání do indexu na počtu vstupních záznamů – se zapnutými transakcemi. Plnou čárou je zobrazen vztah pro Linux, čárkovaně potom pro Windows.

Z uvedené tabulky a z grafu 8.5 je ihned patrné, že v případě zapnutých transakcí se datová struktura nechová zdaleka tak pravidelně jako při transakcích vypnutých. Vysvětlení by mohlo spočívat v tom, že časy operací synchronního zápisu na disk se mohou chovat značně nedeterministicky v závislosti na momentální aktivitě procesů na pozadí atd.

Ve Windows se přírůstek času vkládání oproti variantě bez jakýchkoli kompresí pohybuje až do přibližně 40 %.

V Linuxu nastávají dva podivné jevy – jednak fakt, že ve většině případů je výsledný čas lepší než u experimentu bez transakcí a jednak náhlý schod po vložení cca milionu záznamů.

Cesta k větší efektivitě (alespoň u systému Windows) by pravděpodobně šla směrem k co největší redukci množství dat zapisovaných do logu. Největší nevýhoda této implementace je nutnost řešit pomocí logu i např. algoritmus expanze bloků. Do transakčního logu se tak může zapisovat i velké množství dat, která nebyla přímo součástí vstupní operace.[5] Lepší řešení by bylo zajistit přesuny dat v rámci souboru indexových záznamů bez účasti logu – protože však transakční operace musejí být idempotentní (viz 5.2.), bylo by třeba pro tyto účely vyvinout poměrně sofistikovaný mechanismus, což již je nad rámec této práce.

Pozitivní zjištění, které z předchozího experimentu evidentně plyne je, že použitím transakčního zpracování se nezvyšuje časová složitost operace *insert*.

[5]Jedná se o daň za transparentnost transakčního zpracování, které realizuje modul `PersistentStorage` bez spolupráce s ostatními částmi univerzálního indexu.

Kapitola 9

Závěr

Výsledkem této diplomové práce je především návrh a implementace univerzálního indexu postaveného na principu dynamického invertovaného souboru. Obecná myšlenka je převzata z [3] a rozšířena o další funkcionalitu. Dynamický invertovaný soubor přináší oproti klasickým technikám možnost efektivně aktualizovat index, aniž by bylo nutno jej přebudovávat. Přitom seznam indexových záznamů pro každý term je uchováván pohromadě, čímž se zachovávají předpoklady pro efektivní dotazování.

Rozšíření se týkají zejména možnosti konfigurovat strukturu indexového záznamu, podpory komprese dat a zajištění atomického zpracování operací (odolnosti proti výpadkům).

Dalším podstatným výsledkem je teoretická analýza operace vkládání do datové struktury dynamického invertovaného souboru. Bylo ukázáno, že amortizovaná složitost této operace je konstantní. Mazáním se práce nezabývá. Vzhledem k tomu, že se jedná o operaci dosti analogickou k vkládání, se dá předpokládat stejná složitost. Podrobná analýza by mohla být zajímavým tématem jiné práce.

V experimentální části se podařilo ověřit, že vkládání do indexu se amortizovaně chová lineárně. Dále byla úspěšně ověřena hypotéza o klesajících četnostech expanzí bloků v závislosti na rostoucí hodnotě parametru indexu κ. Chování datové struktury lze vhodným nastavením tohoto parametrem výrazně ovlivnit. Byly také srovnány navržené metody komprese dat v indexu. Diferenční kódování se díky své efektivitě a jednoduchosti projevilo jako zcela nezbytná součást indexu; nejlepších kompresních poměrů se potom podařilo dosáhnout s metodou *kombinovaného kódování*, které spojuje diferenční a B-blokové kódování spolu s Eliasovým kódem ω'.

Experimenty s transakčním zpracováním ukázaly značně nepravidelné chování časů pro zpracování, přičemž jejich přírůstky oproti experimentům bez transakcí se pohybovaly až do cca 40 % (ve Windows bylo dokonce nutné kvůli nepravidelnostem spočítat průměry z více nezávislých testů). V Linuxu byly časy s transakčním zpracováním kupodivu i kratší než časy v experimentech, kde byla tato funkcionalita vypnutá. Na druhou stranu v tomto systému se alespoň chovaly předvídatelněji.

Dá se říci, že diplomová práce splnila všechny své cíle. Neplatí však, že by výsledný produkt byl připraven např. pro komerční použití. To by vyžadovalo implementaci další funkcionality, zejména optimalizátoru dotazů, a zefektivnění transakč-

ního zpracování pomocí redukce množství dat v transakčním logu a případně pomocí nějakého částečného manuálního bufferování. Pravděpodobně by si reálná aplikace vyžadovala i řízení paralelního přístupu k datové struktuře. To všechno může být tématem případné navazující práce.

Příloha A

Rozhraní v jazyce C

Rozhraní v jazyce C bylo navrhováno již před zadáním této práce a je součástí požadavků kladených na index.

Konvence: *Všechny funkce s návratovou hodnotou typu* `int`*, není–li uvedeno jinak, vrací hodnotu* `0` *při svém uspěšném dokončení. Jinak vrací chybový kód (jejich kompletní přehled je uveden v příloze B). V dalším textu už význam těchto návratových hodnot nebude zmiňován.*

A.1. Identifikátory indexů

K identifikaci konkrétního indexu mnoho funkcí používá typ `idx_index_id`. Tyto identifikátory jsou celá čísla, která přiděluje uživatel knihovny. Indexy, které jsou uloženy v tzv. *hlavním adresáři* (viz dále), mají všechny soubory pojmenovány tak, aby identifikátor byl z těchto jmen jednoznačně odvoditelný. Proto i při opětovném zavedení knihovny lze rovnou používat identifikátory těchto indexů.

U indexů, jejichž všechny soubory nejsou v *hlavním adresáři*, specifikuje jména souborů uživatel. Proto z nich obecně identifikátory indexů nejsou odvoditelné a uživatel je musí specifikovat v rámci tzv. *registrace* (viz dále).

A.2. Pomocné definice – datové typy

V první řadě rozhraní specifikuje následující makra pro různé datové typy:

```
typedef unsigned long           idx_size_type;
typedef int                     idx_index_type;
typedef int                     idx_attribute_id;
typedef int                     idx_index_id;
typedef int                     i_entity_id;
```

Typ `idx_size_type` je určen pro proměnné obsahující různé velikosti a offsety. Naproti tomu pro indexy (ve smyslu *index do pole*) je definován `idx_index_type`. Na určení identifikátorů indexových záznamů se používá `idx_attribute_id` a pro identifikátory indexů `idx_index_id`. Konečně pro identifikátory i–entit slouží `i_entity_id`.

A.3. Konstanty

Rozhraní dále definuje nejrůznější konstanty. V první řadě to jsou identifikátory pro jednotlivé podporované datové typy (*Mates data type*). Ty jsou využívány například pro zápis požadavků na index.

```
#define MDT_MASTER              101
#define MDT_INTEGER             102
#define MDT_FLOAT               103
#define MDT_STRING              104
#define MDT_DATA                105
```

Položky typu `MDT_MASTER` dohromady tvoří *primární část indexových záznamů* (viz 2.2.1). Jedná se tedy v podstatě o typ klíčových atributů, v praxi je to celé číslo. `MDT_INTEGER` označuje typ „celé číslo“, `MDT_FLOAT` pseudoreálné číslo v pohyblivé řádové čárce, `MDT_STRING` řetězec znaků zakončený nulou a konečně `MDT_DATA` pole libovolných dat. Všechny tyto typy jsou v implementaci mapovány přirozeným způsobem na nativní typy jazyka C. Je třeba si ovšem uvědomit, že některé z těchto nativních typů nejsou nezávislé na platformě – to platí zejména o typu `FLOAT`, ale i o `INTEGER` (viz například rozdíl mezi tzv. *big endian* a *little endian*[1] systémy). V konfiguraci si lze nastavit, zda se do souborů mají data ukládat přímo tak, jak jsou implementována v nativních typech C, nebo zda se má provádět jejich konverze do tvaru, který bude použitelný i při přesunutí dat na jinou platformu. To má ovšem nevýhodu ve snížení efektivity (ovšem nijak výrazném – počet diskových I/O operací se totiž nezmění). Tato funkcionalita ovšem nebyla testována, takže se doporučuje ponechat výchozí nastavení.

Dále C rozhraní definuje zástupná makra pro každý z uvedených datových typů:

```
#define INDEX_MASTER            int
#define INDEX_INTEGER           int
#define INDEX_FLOAT             float
#define INDEX_STRING            char *
#define INDEX_DATA              void *
```

K tomu ještě existují „přetypovávací“ makra:

[1]Rozdíl mezi systémy s big, resp. little endianem spočívá v pořadí ukládání jednotlivých bytů vícebytových hodnot v paměti. Pokud se na disk ukládá přesná kopie paměti, nelze binární soubor přímo přenést na architektutu s odlišným endian typem.

```
#define as_master(X)            (*((int*)(X)))
#define as_integer(X)           (*((int*)(X)))
#define as_float(X)             (*((floar*)(X)))
#define as_string(X)            ((char*)(X))
#define as_data(X)              ((void*)(X))
```

Další třída konstant jsou konstanty pro relace nad výrazy s položkami. Pomocí takových výrazů lze omezit množinu indexových záznamů, které mají být uvažovány v dotazu:

```
#define MREL_EQUAL              201
#define MREL_NEQUAL             202
#define MREL_LESSEQUAL          203
#define MREL_GREQUAL            204
#define MREL_LESS               205
#define MREL_GREATER            206
```

Pro samotné výrazy nad položkami jsou přípustné následující aritmetické operace:

```
#define MAOP_MULTIPLY           301     /* * */
#define MAOP_DIVIDE             302     /* / */
#define MAOP_PLUS               303     /* + */
#define MAOP_MINUS              304     /* - */
#define MAOP_MOD                305     /* % */
```

Logické operace mohou být taktéž použity ve výrazu nad položkami; zároveň však jimi lze spojovat i více predikátů vytvořených pomocí operací třídy `MREL`.

```
#define MLOP_AND                401
#define MLOP_OR                 402
#define MLOP_NOT                403
```

A.4. Inicializace a konfigurace

Pro konfiguraci indexu slouží speciálně pro tento účel navržené struktury. Struktura `t_global_settings` obsahuje především globální konfigurační údaje pro celou knihovnu.

```
typedef struct
  {
  char *        main_directory;
  int           disk_page_size;
  } t_global_settings;
```

Význam `main_directory` byl popsán výše – udává hlavní adresář, kam si knihovna indexu ukládá svá data. `disk_page_size` by mělo být nastaveno na velikost diskové stránky na dané platformě (slouží k optimalizaci diskových I/O operací).

Struktura `t_index_info`, jak již bylo uvedeno, umožňuje informovat knihovnu o indexech, které nejsou uloženy v adresáři `main_directory` uvedeném v předchozí struktuře. Její jednotlivé položky specifikují umístění jednotlivých souborů, ze kterých sestává konkrétní index.

```
typedef struct _t_index_descriptor
  {
  char *        data_file_dir;
  char *        log_data_dir;
  char *        aux_files_dir;
  idx_index_id  iid;
  int           dictionary_size;
  } t_index_info;
```

Položkami této struktury jsou cesty postupně k adresáři se souborem indexových záznamů, k adresáři s logem (viz kapitola 5) a k adresáři s pomocnými daty. Dále `t_index_info` obsahuje identifikátor indexu `iid` a nakonec `dictionary_size` udává velikost slovníku – může mít různý význam podle aktuální implementace slovníku. V případě hašování to většinou bude velikost hašovací tabulky.

Následující funkce rovněž souvisí s konfigurací indexu:

```
void idx_init( t_global_settings * settings_data )
void idx_done()
void idx_register( t_index_info * info )
void idx_unregister( int index_id )
```

`idx_init` slouží k inicializaci knihovny pomocí první z předchozích dvou struktur, zatímco `idx_done` spouští proces deinicializace – tj. dealokaci dynamických proměnných, uložení neuložených bufferů atd.

Funkci `idx_register` lze použít pro zavedení indexů, jejichž všechny soubory nejsou v adresáři `main_directory`. Je třeba ji explicitně volat při každém zavedení knihovny, ta si data o těchto indexech nikam persistentně neukládá. Dodatečné indexy lze registrovat i opakovaně, avšak jejich skutečná existence příslušných souborů není v tuto chvíli kontrolována. Protějškem `idx_register` je funkce `idx_unregister`, která odregistruje dané indexy. Od okamžiku odregistrace ztrácí knihovna veškeré informace o daném indexu.

A.5. Připojení a odpojení indexu

Od chvíle registrace konkrétního indexu, ať už explicitně funkcí `idx_register` nebo implicitně díky jeho existenci v adresáři `main_directory`, má knihovna informace

o umístění všech souborů tohoto indexu. Avšak k tomu, aby s ním bylo možné pracovat (klást dotazy, vkládat a mazat, ...), je nutné jej nejdříve *připojit*. Připojení provádí operace *mount* a způsobí zavedení informací o indexu do seznamu aktivních indexů a dalších interních struktur knihovny. Opakem je *odpojení* (operace *unmount*). Připojení a odpojení se provádí následujícími funkcemi:

```
int idx_mount( int index_id )
int idx_unmount( int index_id )
```

A.6. Vytváření a rušení indexů

Zatím jsme předpokládali, že indexy, se kterými pracujeme, již existují. Ve skutečnosti je třeba je nejdříve vytvořit. To lze provést následujícími dvěma funkcemi, které příslušný index ihned po vytvoření i automaticky připojí:

```
int idx_create( idx_index_id id, t_index_format * format );
int idx_create_ex( t_index_info info, t_index_format * format );
```

`idx_create` vytváří index v hlavním adresáři, přičemž uživatel specifikuje jeho identifikátor. Druhá funkce `idx_create_ex` oproti tomu umožňuje vytvoření indexu i v jiných adresářích. Indexy v jiných než hlavním adresáři je třeba dodatečně zaregistrovat.

Formát indexových záznamů (jednotný pro celý jeden konkrétní index) je dán strukturou, na kterou odkazuje parametr `t_index_format`:

```
typedef struct
  {
  t_record_attribute * ir_attributes;
  int                  ir_attributes_count;
  } t_index_format;
```

`ir_attributes_count` udává počet atributů v indexových záznamech, zatímco `ir_attributes` odkazuje na pole struktur typu `t_record_attribute`, které popisují jednotlivé atributy. Tato dílčí struktura je uvedena dále:

```
typedef struct
  {
  int    type;
  int    id;
  int    size;

  int    difference_encoding;
  int    b_block_encoding;
  int    rle_signum_encoding;
  t_elias_code_type elias_code_type;
  } t_record_attribute;
```

Tato struktura se používá kromě vytváření indexů i při dotazování – v tom případě se ovšem berou v úvahu pouze její první tři atributy.

`type` udává datový typ položky – může nabývat hodnoty některé z konstant s prefixem `MDT_` popsaných na straně 90. `id` je jednoznačný identifikátor položky a `size` její velikost v bitech. Další atributy se týkají nastavení kódování a komprese indexu. `difference_encoding`, `b_block_encoding` a `rle_signum encoding` jsou příznaky, které určují, zda se bude při ukládání dat aplikovat příslušná metoda komprese (všechny jsou popsány v kapitole 6). `elias_code_type` specifikuje typ použitého Eliasova kódování pomocí konstanty příslušného výčtového typu.

Rušení indexů je jednodušší a je k němu určena funkce `idx_drop`. Zrušení indexu je definitivní operace, kterou nelze nijak vrátit zpět.

```
int idx_drop( idx_index_id id );
```

A.7. Datové struktury pro výměnu dat

Datové struktury pro výměnu dat mezi indexem a jeho uživatelem slouží například pro předání výsledku dotazu uživateli nebo pro specifikaci nových indexových záznamů, které mají být do indexu vloženy. Jedná se v podstatě o seznam indexových záznamů (viz jeho definice na straně 10 – s výjimkou, že seznam v této struktuře nemusí být ve všech případech uspořádaný).

```
typedef struct
  {
  idx_size_type      index_record_count;
  idx_size_type      index_records_capacity;
  t_index_record *   index_records;
  bool               negate;
  t_record_format *  format;
  } t_record_list;
```

`index_record_count` udává počet indexových záznamů – položek v poli `index_records`.

`index_records_capacity` udává kapacitu pole `index_records`.

`index_records` odkazuje na pole struktur typu `t_index_record`, které tvoří tento seznam indexových záznamů. Tvar této struktury je popsán níže.

`negate` je příznak negace. Má–li hodnotu `TRUE`, reprezentuje tento seznam doplněk záznamů, které v něm jsou ve skutečnosti obsaženy. Tento příznak je využíván pouze v průběhu vyhodnocování dotazu, ve všech ostatních případech by měl mít hodnotu `FALSE`. Je–li uživateli jako výsledek dotazu vrácena tato struktura s nastaveným příznakem `TRUE`, může to znamenat, že kladený dotaz nebyl pozitivně definitní.

`format` popisuje strukturu indexových záznamů v daném seznamu. Při dotazování určuje, které položky indexových záznamů mají být dodány na výstup; při změně (operace *update*) specifikuje atributy, které mají být změněny; při vkládání může být použito pro kontrolu.

Zbývá uvést strukturu `t_index_record`, která nese data jednoho konkrétního indexového záznamu:

```
typedef struct
  {
  t_any_type * attributes;
  } t_index_record;
```

Typ `t_any_type` je union, který může nést jakýkoli typ indexového záznamu. Je definován následujícím způsobem:

```
typedef struct
  {
  t_data_type_identifier          type;

  union
    {
    INDEX_MASTER                  t_master;
    INDEX_INTEGER                 t_integer;
    INDEX_FLOAT                   t_float;
    INDEX_STRING                  t_string;
    INDEX_DATA                    t_data;
    };
  } t_any_type;
```

A.8. Datová struktura dotazu

Tato sekce popisuje datovou strukturu, která musí být vytvořena uživatelem jako reprezentace kladeného dotazu. Uživatel je posléze také zodpovědný za její dealokaci.

Dotaz jako celek je reprezentován strukturou typu `t_query`, která zároveň tvoří kořen stromu dotazu. Následuje až sedm různých vrstev uzlů, které mají v rámci dotazu rozdílnou sémantiku. Kromě třetí z nich vždy dvě sousední vrstvy tvoří jeden podstrom dotazu s nějakým konkrétním přesně definovaným významem. Přitom vrstva s nižším číslem tvoří vnitřní uzly, zatímco vyšší vrstva listy.

První a druhá vrstva uzlů je typu `t_list_expr_node`. Ty odpovídají logickému výrazu nad i–entitami. Operace v této vrstvě budou typicky AND, OR a NOT (viz například v dotazu „manažer OR účetní AND NOT sekretářka“).

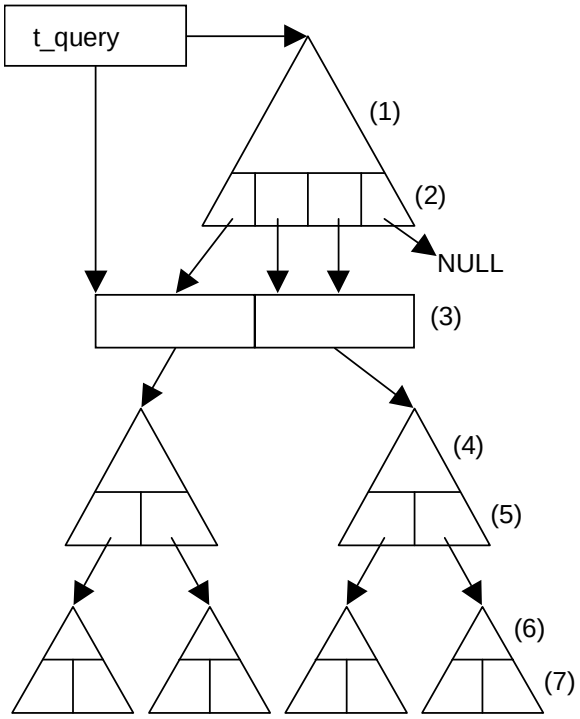
Pro každou i–entitu lze navíc klást různá omezení daná predikáty nad jeho atributy, která mohou zredukovat počet indexových záznamů v jednotlivých seznamech během vyhodnocení dotazu. Tyto výrazy jsou popsány zvláštními podstromy, které

tvoří dohromady čtvrtou až sedmou vrstvu. Za své kořeny mají vždy struktury typu `t_rel_expr_node`, ze kterých sestává i celá čtvrtá vrstva. Pátá vrstva je složena ze stejných typů tvořících listy. Příslušné uzly reprezentují logický výraz nad aritmetickými podvýrazy.

Poslední dvě vrstvy jsou tvořeny typem `t_arit_expr_node` a reprezentují aritmetický výraz nad atributy (např. $x + 3 * y$).

Mezi druhou a čtvrtou vrstvou se nachází ještě oddělující vrstva uzlů, která má hloubku přesně 1 a je tvořena strukturami typu `t_attribute_expr`. Narozdíl od jiných vrstev tato je celá tvořena *jediným* polem příslušných struktur. Listy podstromů
`t_list_expr_node` se odkazují vždy do této mezivrstvy, ale nikdy přímo na `t_rel_expr_node`. Tam se dá odkazovat až z `t_attribute_expr`.

Důvodem tohoto opatření je, že často budou podstromy pod `t_rel_expr_node` totožné a potom by bylo zbytečně neefektivní konstruovat množství shodných stromů. Místo toho se sestrojí strom jediný a odkáže se na něj z více prvků pole této mezivrstvy.



Obrázek A.1: Schéma struktury invertovaného souboru univerzálního indexu. Číslice v závorkách odpovídají číslu vrstvy.

Nyní budou zformulována pravidla, která musí zadavatel dotazu (uživatel knihovny) splnit. Některá již byla uvedena, některá budou nová. Zároveň s těmito pravidly bude uvedena i přesná podoba jednotlivých zmíněných typů struktur:

Obecná pravidla:

1. Za alokaci i dealokaci celého stromu dotazu je zodpovědný uživatel knihovny, nikoli samotná knihovna.

   Dotaz lze bezpečně dealokovat, jakmile dojde k návratu z funkce, ve které byl předán knihovně – ta si pro sebe dělá jeho kopii, především z důvodu vyhnutí se nutnosti modifikovat původní data.[2]

Pravidla pro `t_query`:

2. Struktura typu `t_query` se v celém stromě musí vyskytovat právě jednou, a to jako jeho kořen.
3. Proměnná `root_node` se musí odkazovat na existující strukturu typu `t_list_expr_node`.
4. Proměnná `attribute_exprs` se musí odkazovat na existující pole struktur typu `t_attribute_expr`.
5. Proměnná `attribute_expr_count` musí obsahovat počet prvků pole z předchozího bodu.

Následuje deklarace typu `t_query`:

```
typedef struct
  {
  t_list_expr_node *  root_node;
  t_attribute_expr *  attribute_exprs;
  int                 attribute_expr_count;
  } t_query;
```

Pravidla pro `t_list_expr_node`:

6. Má–li proměnná `arity` hodnotu `0`, představuje příslušná instance struktury list první vrstvy stromu dotazu a v unionu se používá proměnná `leaf` typu `t_list_expr_node_leaf`. V opačném případě se jedná o vnitřní vrchol vrstvy a v unionu platí proměnná `node` typu `t_list_expr_node_node`.
7. Pokud je v unionu platná proměnná `leaf`, potom
   (a) `leaf.ieid` musí obsahovat identifikátor i–entity, jejíž seznam indexových záznamů reprezentuje v rámci stromu dotazu tento uzel.

[2]Například proto, že uživatel může programovat jiném jazyce, než jaký je jazyk knihovny. V tomto případě se jedná o C/C++. Alokátory těchto dvou jazyků ale nejsou kompatibilní – struktura alokovaná pomocí `malloc` nelze dealokovat pomocí `delete` a to samé platí obráceně pro `new` a `free`. Proto by bylo velmi nebezpečné povolit do původní struktury začleňovat položky alokované odlišným alokátorem.

(b) `leaf.attribute_expr` obsahuje odkaz do pole struktur typu `t_attribute_expr`. Příslušný prvek tohoto pole je kořenem podstromu reprezentujícího omezení daná výrazem nad atributy indexových záznamů. Pokud žádná omezení nejsou, může tato proměnná nabývat i hodnoty `NULL`.

(c) `leaf.negate` nabývá buď hodnoty `0` (`FALSE`) nebo `1` (`TRUE`). Pokud je `1`, reprezentuje příslušný uzel ve skutečnosti doplněk (negaci) seznamu indexových záznamů než které by reprezentoval bez negace.

8. Pokud je v unionu platná proměnná `node`, potom

(a) `node.op` musí nabývat hodnoty jedné z konstant s prefixem `MLOP_`, jak byly uvedeny v sekci A.3.

(b) `node.nodes` musí odkazovat na pole struktur typu `t_list_expr_node` o tolika prvcích, jaká je hodnota `arity`. Všechny tyto prvky budou představovat operandy logické operace v `node.op` a jejich počet by proto vzhledem k této operaci měl dávat smysl. Pro negaci je přípustná jedině arita rovná jedné, zatímco pro logický součin a součet je přípustný libovolný počet argumentů větší než dva.

Zde je deklarace příslušné struktury. Její implementace se ve skutečnosti skládá ze tří struktur „spojených“ dohromady přes konstrukt `union`. Nejdříve uvedeme dvě dílčí struktury:

```
typedef struct
  {
  struct _t_list_expr_node *      nodes;
  t_logical_operation_identifier  op;
  } t_list_expr_node_node;

typedef struct
  {
  i_entity_id                     ieid;
  t_attribute_expr *              attribute_expr;
  int                             negate;
  } t_list_expr_node_leaf;
```

A nakonec zbývá struktura, která tvoří pro všechny tři pojítko:

```
typedef struct _t_list_expr_node
  {
  int                             arity;

  union
    {
    t_list_expr_node_node         node;
    t_list_expr_node_leaf         leaf;
    };
  } t_list_expr_node;
```

Pravidla pro `t_attribute_expr`:

9. Struktura typu `t_attribute_expr` se v celém stromě smí vyskytovat nejvýše v jediném poli odkazovaném mimo jiné z kořene (`t_query`) proměnnou `attribute_exprs`.

10. `reference_count` má stejnou hodnotu, kolik existuje pointerů odkazujících se na příslušnou strukturu.

11. `root_node` musí odkazovat na existující strukturu `t_rel_expr_node`, která tvoří kořen podstromu výrazu nad atributy indexového záznamu.

```
typedef struct
  {
  short                         reference_count;
  t_rel_expr_node *             root_node;
  } t_attribute_expr;
```

Pravidla pro `t_rel_expr_node`:

12. `requested_type` obsahuje identifikátor typu, na který má být přetypována hodnota, kterou tento uzel reprezentuje. Nemá–li dojít k přetypování, musí být v této proměnné nastavena hodnota `MDT_INVALID`.

13. Má–li proměnná `arity` hodnotu `0`, představuje příslušná instance struktury list třetí vrstvy stromu dotazu a v unionu se používá proměnná `leaf` typu `t_rel_expr_node_leaf`. V opačném případě se jedná o vnitřní vrchol vrstvy a v unionu platí proměnná `node` typu `t_rel_expr_node_node`.

14. Pokud je v unionu platná proměnná `leaf`, potom

    (a) `leaf.op` musí nabývat hodnoty jedné z konstant s prefixem `MREL_`, jak byly uvedeny v sekci A.3.

    (b) `leaf.left_expr` musí odkazovat na existující instanci struktury typu `t_arit_expr_node`. Ta představuje kořen výrazu, který bude vystupovat jako levý operand operátoru v `leaf.op`.

    (c) `leaf.right_expr` musí splňovat stejná pravidla jako `leaf.left_expr` – pouze odkazovaný uzel bude kořenem *pravého* operandu.

15. Pokud je v unionu platná proměnná `node`, potom

    (a) `node.op` musí nabývat hodnoty jedné z konstant s prefixem `MLOP_` nebo `MREL_`, jak byly uvedeny v sekci A.3.

    (b) `node.nodes` musí odkazovat na pole struktur typu `t_rel_expr_node` o tolika prvcích, jaká je hodnota `arity`. Všechny tyto prvky budou představovat operandy logické operace v `node.op` a jejich počet by proto vzhledem k této operaci měl dávat smysl. Pro negaci je přípustná jedině arita rovná jedné, zatímco pro logický součin a součet je přípustný libovolný počet argumentů větší než dva.

Výpis `t_rel_expr_node`:

```
typedef struct
  {
  struct _t_rel_expr_node *       nodes;
  int                             op;
  } t_rel_expr_node_node;

typedef struct
  {
  t_arithmetic_expr_node *        left_expr;
  t_arithmetic_expr_node *        right_expr;
  int                             op;
  } t_rel_expr_node_leaf;

typedef struct _t_rel_expr_node
  {
  t_data_type_identifier          requested_type;
  int                             arity;

  union
    {
    t_rel_expr_node_node          node;
    t_rel_expr_node_leaf          leaf;
    };
  } t_rel_expr_node;
```

Pravidla pro `t_arithmetic_expr_node`:

16. `requested_type` obsahuje identifikátor typu, na který má být přetypována hodnota, kterou tento uzel reprezentuje. Nemá–li dojít k přetypování, musí být v této proměnné nastavena hodnota `MDT_INVALID`.

17. Má–li proměnná `arity` hodnotu `0`, představuje příslušná instance struktury list poslední (čtvrté) vrstvy stromu dotazu a v unionu se používá proměnná `leaf` typu `t_arithmetic_expr_node_leaf`. V opačném případě se jedná o vnitřní vrchol vrstvy a v unionu platí proměnná `node` typu `t_arithmetic_expr_node_node`.

18. Pokud je v unionu platná proměnná `leaf`, potom

    (a) Pokud `leaf.attr_value` obsahuje hodnotu `MDT_INVALID`, je platná `leaf.attr_id`. Pokud `leaf.attr_value` obsahuje jinou hodnotu, je platná tato proměnná a naopak `leaf.attr_id` není definována.

    (b) Pokud je platná proměnná `leaf.attr_id.` obsahuje identifikátor atributu indexového záznamu, jehož hodnotu má tento uzel reprezentovat.

    (c) Pokud je platná proměnná `leaf.attr_value.` určuje přímo svým obsahem hodnotu, kterou má tento uzel reprezentovat.

19. Pokud je v unionu platná proměnná `node`, potom

    (a) Proměnná `node.op` musí nabývat hodnoty jedné z konstant s prefixem `MAOP_`, jak byly definovány v sekci A.3.

    (b) Proměnná `node.nodes` musí odkazovat na pole instancí struktur typu `t_arithmetic_expr_node`, které reprezentují hodnoty, jež mají být operandy operace v `node.op`.

```
typedef struct
  {
  struct _t_arithmetic_expr_node * nodes;
  int                               op;
  } t_arithmetic_expr_node_node;

typedef struct
  {
  idx_attribute_id                attr_id;
  t_any_type                      attr_value;
  } t_arithmetic_expr_node_leaf;

typedef struct _t_arithmetic_expr_node
  {
  t_data_type_identifier          requested_type;
  int                             arity;

  union
    {
    t_arithmetic_expr_node_node   node;
    t_arithmetic_expr_node_leaf   leaf;
    };
  } t_arithmetic_expr_node;
```

Na závěr této části uvedeme na obrázku A.2 ještě příklad, jak by mohl vypadat diagram instancí pro nějaký dotaz. Vzhledem k jeho rozsahu je diagram jen částečný; navíc ignoruje fakt, že k některým proměnným je přistupováno přes uniony.

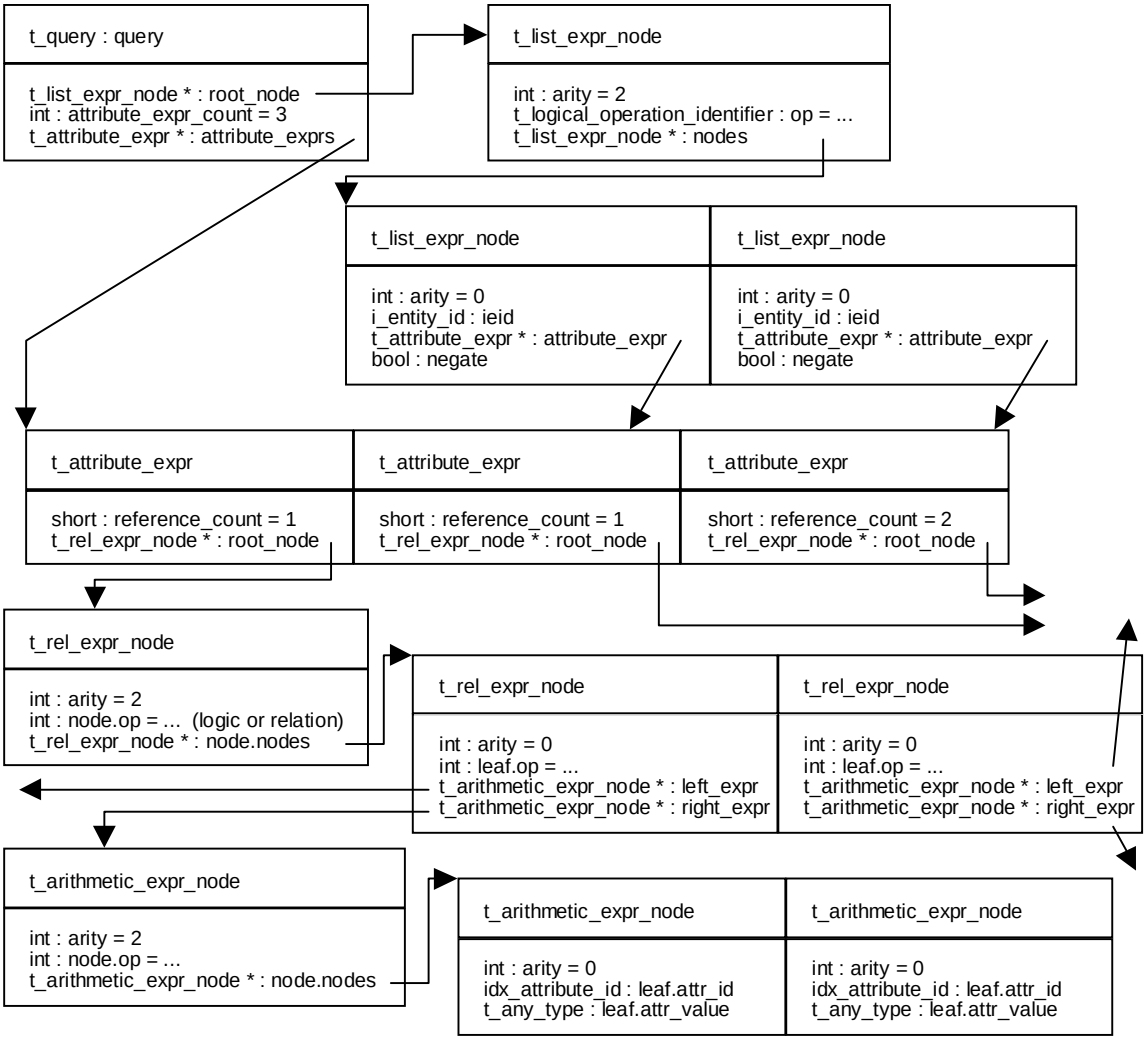
A.9. Dotazování

Kladení dotazů indexu je implementováno pomocí mechanismu využívajícího myšlenky tzv. *kurzorů* běžné například v relačních databázích. Kurzor je struktura, která si udržuje informace o tom, jaká část dat výsledku dotazu již byla předána uživateli.

V této implementaci je kurzor reprezentován typem `t_query_cursor`.

Při dotazování se rozlišují následující fáze:

1. *Vytvoření reprezentace dotazu.* Příslušné struktury byly detailně popsány v sekci A.8.



Obrázek A.2: Diagram instancí znázorňující strukturu dotazu na index

2. *Zadání dotazu knihovně indexu (otevření dotazu)*
3. *(Postupné přebírání výsledku dotazu od knihovny.*
4. *Uzavření dotazu.*

Ve fázi *zadání dotazu knihovně indexu* index dotaz převezme, a vyhodnotí jej. Během této fáze by se také mohly provádět určité druhy optimalizací, což ale není předmětem této práce.

Zadání dotazu se provede funkcí `idx_open_select`. Jejími parametry je identifikátor konkrétního indexu, se kterým chce uživatel pracovat, a ukazatel na dříve popsaný strom reprezentující dotaz. Funkce vrátí kurzor, který bude dále sloužit jako identifikace této instance dotazu při přebírání výsledných dat uživatelem. Přesná hlavička této funkce vypadá následujícím způsobem:

```
t_query_cursor idx_open_select
  (
  idx_index_id                  idx_id,
  t_query *                     query
  );
```

Po zadání dotazu si uživatel musí explicitně požádat o jeho výsledek. To udělá funkcí `idx_fetch_next`, která zadá jako parametr opět identifikátor příslušného indexu, kurzor vrácený při zadání dotazu, maximální počet indexových záznamů, který mu smí být vrácen a konečně pole, do kterého index tyto záznamy uloží. Je garantováno, že toto pole nebude indexem modifikováno až do dalšího volání `idx_fetch_next` nebo `idx_close_select`.

Počet indexových záznamů, které index ve skutečnosti do daného pole uložil, lze získat z kurzoru jako `query_cursor->records_number`.

Po zpracování vrácených výsledků může uživatel volat `idx_fetch_next` znovu. Index mu vrátí dalších `n` záznamů počínaje prvním záznamem, který nebyl vrácen při minulém volání.

Funkce pro výběr dat z indexu vypadá takto:

```
int idx_fetch_next
  (
  idx_index_id                  idx_id,
  t_query_cursor *              query_cursor,
  idx_size_type                 n,
  t_index_record **             result
  );
```

Poznámka A.9.1: Pole `result` není samo o sobě nijak ukončené. To znamená, že vrátí–li například index při třetím volání `idx_fetch_next` 10 záznamů, přičemž při druhém jich vrátil 20, bude výsledkové pole ve skutečnosti obsahovat nejméně těchto 20 záznamů, přičemž druhých 10 se bude opakovat z minulého volání. Proto je důležité dávat pozor na hodnotu `query_cursor->records_number` (viz výše).

Jestliže již uživatel získal celý výsledek nebo již další data nepotřebuje, je důležité dotaz zavřít. Zavření umožní knihovně odalokovat všechny speciální struktury, které kvůli dotazu musel dočasně spravovat. Zavření se provádí funkcí `idx_close_select`, jejíž parametry mají obdobný význam jako u funkcí předchozích:

```
int idx_close_select
  (
  idx_index_id                  idx_id,
  t_query_cursor *              query_cursor
  );
```

A.10. Vkládání nových záznamů

Vkládání nových dat do indexu se provádí pomocí funkcí `idx_insert_record` a `idx_insert_records`. Jejich parametry jsou identifikátor příslušného indexu, identifikátor i–entity, které se týkají nově vkládané záznamy a vlastní indexový záznam v případě funkce první (která umožňuje vložení jen jednoho prvku) nebo seznam záznamů v případě funkce druhé.

```
int idx_insert_record
  (
  idx_index_id                  idx_id,
  i_entity_id                   ieid,
  t_index_record *              record
  );

int idx_insert_records
  (
  idx_index_id                  idx_id,
  i_entity_id                   ieid,
  t_record_list *               records
  );
```

A.11. Aktualizace indexových záznamů

Funkce `idx_update_record` umožňuje změnit atributy jednoho nebo více indexových záznamů v daném indexu patřících k nějaké konkrétní i–entitě (tj. záznamy v rámci jednoho bloku).

Záznamy, které mají být změněny, jsou v příslušném seznamu vyhledávány pomocí tzv. *vyhledávacího vzoru*. Vyhledávací vzor je indexový záznam, na který odkazuje parametr `record` pole `attrs_master` (jeho velikost je v `attrs_master_count`). Tento indexový záznam určuje, které atributy vyhledávacího vzoru jsou *relevantní*. Změněny budou všechny záznamy, pro které se všechny relevantní atributy shodují s relevantními atributy vyhledávacího vzoru.

Naopak pole `attrs_to_update` (jeho velikost udává `attrs_to_update_count`) specifikuje, které atributy indexových záznamů jsou určeny ke změně. Odpovídající atributy v proměnné `record` potom udávají příslušné nové hodnoty.

Důležité omezení je, že relevantními atributy mohou být pouze atributy *primární* (`INDEX_MASTER`), zatímco měněnými pouze *sekundární* (jiné než `INDEX_MASTER`). Viz také 2.2.1.

```
int idx_update_record
  (
  idx_index_id            idx_id,
  i_entity_id             ieid,
  t_index_record *        record,
  t_record_attribute *    attrs_master,
  int                     attrs_master_count,
  t_record_attribute *    attrs_to_update,
  int                     attrs_to_update_count
  );
```

Funkce `idx_update_records` je obdobou předchozí, ale umožňuje zadat víc vyhledávacích vzorů najednou.

```
int idx_update_records
  (
  idx_index_id            idx_id,
  i_entity_id             ieid,
  t_index_record *        records,
  int                     records_count,
  t_record_attribute *    attrs_master,
  int                     attrs_master_count,
  t_record_attribute *    attrs_to_update,
  int                     attrs_to_update_count
  );
```

A.12. Mazání indexových záznamů

Existuje několik funkcí pro mazání dat z indexu.

První z nich, `idx_delete_all_records`, smaže všechny indexové záznamy pro danou i–entitu, bez jakýchkoli dalších omezení.

```
int idx_delete_all_records
  (
  idx_index_id                idx_id,
  i_entity_id                 ieid
  );
```

`idx_delete_record` umožňuje smazat jeden nebo více indexových záznamů spojených s danou i–entitou. Ty jsou specifikovány v proměnné `record` pomocí jednoho *vyhledávacího vzoru*. Další pole `attrs`, jehož délka je dána v proměnné `attrs_count`, specifikuje, podle kterých atributů záznamu v `record` se mají vyhledávat záznamy ke smazání. Ostatní atributy se ignorují.

```
int idx_delete_record
  (
  idx_index_id                  idx_id,
  i_entity_id                   ieid,
  t_index_record *              record,
  t_record_attribute *          attrs,
  int                           attrs_count
  );
```

Funkce `idx_delete_records` je obdobou předchozí, ale umožňuje zadat víc vyhledávacích vzorů najednou.

```
int idx_delete_records
  (
  idx_index_id                  idx_id,
  i_entity_id                   ieid,
  t_index_record *              records,
  int                           records_count,
  t_record_attribute *          attrs,
  int                           attrs_count
  );
```

A.13. Pomocné funkce

Rozhraní knihovny navíc poskytuje ještě následující pomocné funkce:

```
int idx_flush( int index_id );
int idx_flush_all( idx_index_id idx_id );
int idx_print( idx_index_id idx_id, FILE * output, int verbatim );
```

`idx_flush` způsobí zapsání všech dat daného indexu kromě slovníku na disk, pokud tam ještě nejsou. Pokud chce uživatel zapsat data včetně slovníku, musí použít `idx_flush_all`.

`idx_print` slouží k ladicím účelům. Textově vytiskne obsah indexu do souboru daného parametrem `output`. Pokud je `verbatim FALSE`, vytiskne pouze informace o jednotlivých oblastech a blocích; v opačném případě vypíše i indexové záznamy. K dispozici jsou i ladicí funkce `idx_print_area` a `idx_print_block` určené k výpisu pouze jedné konkrétní oblasti, resp. bloku. Ty zde nebudeme uvádět.

A.14. Rozhraní v jazyce C++

Rozhraní v jazyce C++ v zásadě odpovídá předchozímu. Je realizováno rozhraními tříd `Index` (operace na konkrétním namountovaném indexu, jako např. insert, delete,

atp.) a `IndexManager` (operace nad celou knihovnou, jako např. inicializace; dále potom namountování a odmountování indexu). Z důvodu rozsahu zde toto rozhraní nebude uvedeno; lze jej snadno nalézt v příslušných hlavičkových souborech.

Příloha B

Chybové kódy

Chybové kódy slouží pro indikaci způsobu dokončení operace z C–rozhraní. Příslušné funkce, není–li uvedeno jinak, vrací 0 při úspěšném dokončení; jinak vrací kladný chybový kód.

Kódy jsou uspořádány hierarchicky. Technicky toto uspořádání má význam například ve spojitosti s loggerem, kdy si lze vytvořit filtr, díky kterému se budou logovat například jen hlášky spjaté s nějakým daným modulem apod. V tomto přehledu je toto uspořádání naznačeno příslušným odsazením.

B.1. Neznámá chyba

`#define IDX_ERROR_UNKNOWN`
Blíže nespecifikovaná obecná chyba

B.2. Chyby v modulu indexu

`#define IDX_INDEX_ERROR`
Blíže nespecifikovaná chyba v modulu indexu.

`#define IDX_INPUT_OUTPUT_ERROR`
Vstupní/výstupní chyba v modulu indexu.

`#define IDX_CANNOT_SAVE_INDEX_DIRECTORY`
Nelze uložit adresář indexu.

`#define IDX_CANNOT_LOAD_INDEX_DIRECTORY`
Nelze načíst adresář indexu.

`#define IDX_INDEX_DIRECTORY_WRONG_FORMAT`
Špatný formát nebo nekonzistence v adresáři indexu.

`#define IDX_CANNOT_SAVE_INDEX_DICTIONARY`
Nelze uložit slovník na disk.

`#define IDX_CANNOT_LOAD_INDEX_DICTIONARY`

Nelze načíst slovník.

`#define IDX_INDEX_DICTIONARY_WRONG_FORMAT`
Špatný formát nebo nekonzistence ve slovníku.

`#define IDX_CANNOT_SAVE_INDEX_RECORDS`
Nelze uložit soubor indexových záznamů.

`#define IDX_CANNOT_LOAD_INDEX_RECORDS`
Nelze načíst soubor indexových záznamů.

`#define IDX_INDEX_RECORDS_WRONG_FORMAT`
Špatný formát nebo nekonzistence v souboru indexových záznamů.

`#define IDX_INDEX_OUT_OF_RANGE`
Nějaký atribut/index/proměnná/... se dostal mimo povolený rozsah hodnot.

`#define IDX_ATTRIBUTE_INDEX_OUT_OF_RANGE`
Atribut s daným indexem v daném indexovém záznamu neexistuje.

`#define IDX_NO_ATTRIBUTE_WITH_SUCH_ID`
Atribut s daným identifikátorem v daném indexovém záznamu neexistuje.

`#define IDX_WRONG_FORMAT_OF_BLOCK`
Neplatný formát bloku v indexu

`#define IDX_BUFFER_TOO_SMALL`
Buffer je příliš malý.

`#define IDX_INDEX_RECORDS_STORAGE_ERROR`
Nespecifikovaná chyba v souboru indexových záznamů

`#define IDX_INVALID_AREA_INDEX`
Neplatný index oblasti v souboru indexových záznamů.

`#define IDX_INDEX_MANAGER_ERROR`
Nespecifikovaná chyba v manažeru indexů.

`#define IDX_INDEX_NOT_MOUNTED`
Pokus pracovat s indexem, který není připojen.

`#define IDX_INDEX_MOUNTED`
Index je připojen, přestože příslušná operace očekává, že bude odpojen.

`#define IDX_UNKNOWN_INDEX_ID`
Neznámý identifikátor indexu (např. pokud nebyl zaregistrován nebo neexistuje).

`#define IDX_INDEX_QUERY_ERROR`
Nespecifikovaná chyba během zpracování dotazu.

`#define IDX_NO_MORE_RECORDS_IN_RESULT`
Ve výsledku už nejsou další záznamy. Touto chybou index indikuje například to, že kurzor již je na konci dostupných záznamů.

`#define IDX_QUERY_NOT_OPENED`
Dotaz není „otevřen“. Například při použití kurzoru před „otevřením“ dotazu.

`#define IDX_INVALID_ATTRIBUTE_TYPE`
Nedefinovaný typ atributu indexového záznamu.

B.3. Chyby v modulu `PersistentStorage`

`#define PERSISTENT_STORAGE_ERROR`
Nespecifikovaná chyba v modulu `PersistentStorage`.

`#define FILE_OPERATION_ERROR`
Nespecifikovaná chyba při práci se souborem.

`#define FSEEK_UNSUCCESSFUL`
Nelze provést operaci `FSEEK` *(přesun na danou pozici v souboru).*

`#define FWRITE_UNSUCCESSFUL`
Nelze zapsat do souboru.

`#define FREAD_UNSUCCESSFUL`
Nelze číst ze souboru.

`#define FOPEN_UNSUCCESSFUL`
Soubor nelze otevřít.

`#define FCLOSE_UNSUCCESSFUL`
Soubor nelze zavřít.

`#define FFLUSH_UNSUCCESSFUL`
Nelze provést operaci `FLUSH` *(zapsání dat z cache na disk).*

`#define FTELL_UNSUCCESSFUL`
Nelze zjistit aktuální pozici v souboru.

B.4. Chyby v modulu `BasicDataStructures`

`#define BASIC_DATA_STRUCTURES_ERROR`
Nespecifikovaná chyba v modulu `BasicDataStructures`.

`#define OUT_OF_RANGE`
Atribut/index/...je mimo povolený rozsah.

`#define NOT_SUFFICIENT_CAPACITY`
Datová struktura nemá dostatečnou kapacitu pro dokončení této operace.

`#define DATA_STRUCTURE_EMPTY`
Datová struktura neobsahuje žádná data.

B.5. Chyby v modulu `FileSystem`

`#define FILE_SYSTEM_ERROR`
Nespecifikovaná chyba v modulu `FileSystem`.

`#define UNRECOGNIZED_IMAGE_FILE`
Nenalezený image souboru pro virtuální souborový systém.

`#define INTEGRITY_ERROR`
Chyba integrity souborového systému.

`#define INVALID_FS_ITERATOR`
Neplatný iterátor.

`#define INVALID_OS_FS_ITERATOR`
Neplatný iterátor v nativním souborovém systému.

`#define INVALID_VOLUME_ITERATOR`
Neplatný iterátor ve virtuálním souborovém systému.

`#define CANNOT_CHANGE_DIRECTORY`
Nelze změnit adresář (například cílový adresář neexistuje, . . .).

`#define CANNOT_OPEN_FILE`
Soubor nelze otevřít.

`#define NOT_A_DIRECTORY`
Specifikovaná entita není adresář.

`#define NOT_A_FILE`
Specifikovaná entita není soubor.

`#define PATH_TOO_LONG`
Adresářová cesta je příliš dlouhá.

`#define ERROR_ENUMERATING_DIRECTORY`
Chyba při procházení adresáře.

`#define CANNOT_GET_FILE_INFORMATION`
Nelze získat informace o souboru.

`#define DIRECTORY_NOT_FOUND`
Adresář nenalezen.

`#define FILE_NOT_FOUND`
Soubor nenalezen.

`#define CANNOT_CREATE_DIRECTORY`
Nelze vytvořit adresář.

`#define CANNOT_DELETE_DIRECTORY`
Nelze zrušit adresář.

`#define DIRECTORY_NOT_EMPTY`
Adresář není prázdný (například při jeho mazání).

`#define UNRECOGNIZED_VOLUME_IMAGE`
Nebyl rozpoznán souboru image *svazku.*

`#define VOLUME_IMAGE_NOT_FOUND`
Nebyl nalezen soubor image *svazku.*

Příloha C

Formáty souborů

Tato příloha obsahuje přehled struktury souborů, které jsou z hlediska knihovny univerzálního indexu nejdůležitější. Jedná se zejména o soubory, pomocí kterých se implementuje invertovaný soubor.

Každý z těchto souborů na svém začátku obsahuje tzv. „magic number“. To je identifikátor, který slouží pro ověření typu souboru a okrajově k indikaci jeho případného poškození nebo záměny. Pro každý typ souboru je předdefinována nějaká konkrétní hodnota; pokud při načítání jeho neodpovídá, načítání selže a je ohlášena chyba.

Za magic number následuje „verze“. Toto číslo slouží jako identifikace verze knihovny, která daný soubor vytvořila. Při načítání musí hodnota verze uvedená v souboru buď souhlasit se skutečnou verzí knihovny, nebo ji musí knihovna alespoň považovat za kompatibilní.

Za magic number následuje „version“. Toto číslo slouží jako identifikace verze knihovny, která daný soubor vytvořila. Při načítání musí version uvedené v souboru souhlasit se skutečnou verzí knihovny, jinak načítání opět selže. Všechny vícebytové atributy v souborech jsou uloženy ve formátu „little endian“, tj. nejméně významný byte je na disku uložen jako první.

C.1. Formát slovníku (dictionary – *.dic)

1. Hlavička:
   - 8B ... Magic number
   - 4B ... Version
   - 16B ... Reserved
   - 4B ... Počet záznamů v těle
2. Tělo: V těle se nachází 0 nebo více záznamů typu:
   - 4B ... Identifikátor i–entity
   - 4B ... Index oblasti, ve které se nachází blok příslušný tomuto termu[1]
   - 4B ... Offset příslušného bloku v souboru indexových záznamů[2]

C.2. Formát adresáře indexu (index directory – *.dir)

1. Hlavička:
   - 8B ... Magic number
   - 4B ... Version
   - 16B ... Reserved
   - 4B ... Počet zapsaných dvojic offsetů, které následují v těle

2. Tělo: V těle se nachází 0 nebo více záznamů typu (označme i pořadí daného záznamu v rámci souboru):
   - 4B ... Offset počátku i–tého bloku v souboru indexových záznamů
   - 4B ... Offset konce i–tého bloku v souboru indexových záznamů

C.3. Formát souboru indexových záznamů (index records storage – *.idx)

1. Hlavička:
   - 8B ... Magic number
   - 4B ... Version
   - 20B ... Reserved
   - 2B ... Počet atributů indexových záznamů tohoto indexu
   - xB ... Reprezentace indexových záznamů

   Následuje formát *reprezentace indexových záznamů*. Tato oblast dat obsahuje pro každý atribut ze schématu indexu (dáno formátem) 16 bytů, ve kterých je specifikován typ daného atributu, způsob komprese a případně další informace. Pro daný atribut mají tyto informace následující strukturu:
   - 4B ... Item type
   - 4B ... Item id
   - 4B ... Item length
   - 4B ... Storage flags

   Položka *storage flags* specifikuje způsob komprese příslušného seznamu atributů. Jedná se o bitové pole s následující strukturou (b značí bity):

[1] Nebo −1, pokud žádný takový blok neexistuje.
[2] Pokud takový blok neexistuje, není hodnota definována.

1b ... Použít diferenční kódování[3]
1b ... Použít B–blokové kódování[4]
1b ... Použít RLE kódování znaménkových bitů[5]
1b ... Nevyužito
4b ... Indikuje použití Eliasova kódování, případně specifikuje jeho typ[6]
8b ... Nevyužito
8b ... Dvojkový logaritmus parametru b v B–blokovém kódování[7]
8b ... Nevyužito

Kompresní metody lze kombinovat, ovšem ne všechny kombinace jsou platné. Více informací je uvedeno v sekci 6.7. na straně 67.

2. Tělo: obsahuje 0 nebo více záznamů typu:

xB ... Díra[8]
yB ... Oblast[9]

Každý blok v rámci oblasti má následující strukturu:

4B ... Identifikátor i–entity
4B ... Délka seznamu indexových záznamů v bitech
4B ... Počet indexových záznamů v tomto bloku
4B ... Reservováno
xB ... Tělo bloku[10].

Tělo bloku sestává ze seznamu indexových záznamů zakódovaných podle nastavení uvedených v hlavičce. Existuje jediný případ, kdy se nejedná o pouhý seznam indexových záznamů – je jím použití RLE komprese znaménkových bitů. Ta je popsána v sekci 6.5. na straně 66, včetně změněného formátu těla bloku.

[7]Viz 6.3. na straně 62.
[7]Viz 6.4. na straně 62.
[7]Viz 6.5. na straně 66.
[7]Viz 6.2. na straně 58.
[7]Viz 6.4. na straně 62.
[8]Její velikost se může pohybovat od 0 do velikosti bloku v následující neprázdné oblasti.
[9]Skládá se z 1 nebo více bloků velikosti $l_i = l_0 \cdot \kappa^i$, kde i je pořadové číslo oblasti vrámci souboru (včetně prázdných oblastí, které zde nejsou explicitně – nezabírají žádný prostor) (viz dále).
[10]Jeho velikost byla udána druhou položkou hlavičky „délka seznamu souřadnic v bitech“.

Příloha D

Použité softwarové vybavení

Při zpracování této diplomové práce byly použity následující softwarové aplikace/moduly:

- *ActivePerl, verze 5.6.1* (`http://www.activestate.com/Products/ActivePerl`) je implementace jazyka Perl, který byl použit pro předzpracování textových vstupů (například testovací data poskytnutá vedoucím práce nebo vytvoření statistiky četností termů v úvodní kapitole) a pro napsání systému generujícího *makefiles* nezávisle na platformě.
- *csindex* (`http://www.filewatcher.com/?q=csindex%255B-_.%255D*&t=g&p=2`) je česko-slovenská implementace programu *makeindex* pro tvorbu indexu pro dokumenty sázené v TeXu.
- *Dia, verze 0.94-pre1* (`http://www.gnome.org/projects/dia`) je vektorový editor diagramů. Byl použit pro nakreslení některých diagramů v tomto textu, které nejsou vytvořeny systémem R.
- *MetaGraph-5* (`http://w3.mecanica.upm.es/metapost/metagraf.php`) je vektorový editor, který byl použit pro vytvoření části obrázků. Některé obrázky byly i vytvořeny ručně přímo v jazyce MetaPost.
- *GCC, verze 2.9* (součást linuxových distribucí) spolu s ostatními standardními linuxovými vývojovými nástroji bylo použito pro překlad v prostředí Linuxu.
- *AFPL GhostScript 8.3* (`http://www.cs.wisc.edu/~ghost/doc/AFPL/get853.htm`) je interpret jazyka PostScript.
- *GSview, verze 4.3* (`http://www.cs.wisc.edu/~ghost`) je prohlížeč postscriptových souborů postavený nad AFPL Ghostscriptem. Byl používán jednak pro určení žádoucích dimenzí tzv. *bounding boxu* souborů ve formátu EPS, jednak pro prohlížení poscriptové verze diplomové práce a její export do formátu PDF.
- *KDevelop, verze 3.0.4* (`www.kdevelop.org`) bylo použito pro psaní části zdrojových textů a pro ladění v prostředí Linuxu.

- *Microsoft Platform SDK, build 3790*
  (`http://www.microsoft.com/msdownload/platformsdk/sdkupdate`)
  je vývojové prostředí pro Windows obsahující nástroje jako překladač C++ nebo linker. Bylo použito pro překlad v prostředí Windows.

- *R, verze 1.9.0* (`http://www.r-project.org`) je statistický systém, který byl použit zejména pro vyhodnocení a prezentaci výsledků testování. Tímto systémem byly vytvořeny také např. obrázky 2.4 a 2.5.

- *TeXLive, verze TeXu 3.14* (`http://www.tug.org/texlive`) je distribuce typografického systému TeX, která byla použita pro vysázení tohoto textu. Kromě TeXu byly z této distribuce používány zejména programy *WinDVI* pro prohlížení souborů DVI, *dvips* pro převod z formátu DVI na PS a utility pro tvorbu indexu.

- *WMF2EPS, verze 1.32* (`www.wmf2eps.de.vu`) je konvertor souborů z formátu WMF do EPS. Jedná se o jediný nalezený program, který byl schopen tuto konverzi uspokojivě provést. Soubory WMF byly používány pro export z editoru Dia, neboť při exportu přímo do EPS nešlo správně zachovat české fonty. Ani WMF2EPS ovšem nebyl schopen správně nastavit u EPS souborů tzv. *bounding box*. Program byl použit v rámci své zkušební lhůty.

- *XEmacs, verze 21.5* (`www.xemacs.org`) je textový editor, který byl použit pro napsání většiny textů této práce včetně zdrojových kódů.

Příloha E

Obsah přiloženého CD

Většina klíčových adresářů na CD obsahuje soubor `readme.txt`, který informuje o jejich obsahu i o obsahu příslušných podadresářů. V této příloze jsou proto pro základní orientaci shrnuty pouze nejdůležitější adresáře:

`/data` – Původní i předzpracovaná data použitá k experimentům.

`/implementation/dist` – Přeložené knihovny i všechny spustitelné soubory jak pro Linux tak i pro Windows.

`/implementation/dist/xml` – Příklady ověřených XML souborů pro konfiguraci experimentů.

`/implementation/doxydocs` – Vygenerovaná vývojová dokumentace tříd obsahující i instrukce pro otestování produktu.

`/implementation/src/tests` – Zdrojové kódy vývojových testů a experimentů.

`/implementation` – Implementace univerzálního indexu.

`/implementation/doc` – Vývojová dokumentace.

`/reports` – Reporty z experimentů.

`/theory` – Zdrojové texty diplomové práce v LaTeXu.

`/theory/dist` – Výsledný text diplomové práce v různých formátech.

`/tools` – Některé pomocné nástroje použité při vývoji nebo testování.

Literatura

[1] Baeza-Yates R., Ribeiro-Neto B. (1999): Modern Information Retrieval, Addison-Wesley, New York.

[2] Frakes W. B., Baeza-Yates R. (1992): Information Retrieval, Prentice Hall, New Jersey.

[3] Holub M., Míka P.: MATES - An Experimental Linguistic Database System (2001), *Proceedings of the IRCS Workshop on Linguistic Databases, str. 134–140*, University of Pennsylvania, Philadelphia.

[4] Ioannidis, Yannis E. (1996): Query Optimization, University of Wisconsin, Madison.

[5] Kopecký M. (2002): Dokumentografické informační systémy, *Elektronické skriptum k předmětu DBI010*, viz `http://www.ms.mff.cuni.cz/~kopecky/vyuka/dis`, MFF UK Praha.

[6] Míka P. & kol. (2000): Dokumentace systému MATES (studentský softwarový projekt), *kapitola III.9*, MFF UK Praha.

[7] Pávek J. (2005): Testovací kolekce českých dokumentů a dotazů, *diplomová práce vedená M. Holubem*, MFF UK Praha.

[8] Pokorný J., Snášel V., Húsek D. (1998): Dokumentografické informační systémy, Karolinum, Praha.

[9] Pokorný J. (2002): Dotazovací jazyky, Karolinum, Praha.

[10] Pokorný J. (1997): Základy implementace souborů a databází, Karolinum, Praha.

[11] Procházka M. (2000): Transakce, *Elektronické skriptum k předmětu DBI016*, viz `http://nenya.ms.mff.cuni.cz/~ceres/dnl/TXy_0102/Prochazka-000529.pdf`.

[12] Poznámky z přednášky Datové struktury (MFF UK, kód TIN066).

[13] Poznámky z přednášky Dokumentografické informační systémy (MFF UK, kód DBI010).

[14] Poznámky z přednášky Dotazovací jazyky (MFF UK, kód DBI001 a DBI006).

[15] Poznámky z přednášky Transakce (MFF UK, kód DBI016).

Rejstřík

ACID, 46
aktivace indexu, 92
aktualizace, 104
algoritmus
    expanze bloku, 30
    insert, 29
    nalezení posledního checkpointu, 53
    nalezení začátku transakce v logu, 55
    odstranění i–entity, 31
    odstranění indexového záznamu, 32
    query, 27
    zotavení z výpadku, 54

B-strom
    prefixový, *viz* prefixový B-strom
blockclass, 28
BlockPosition, 72
blocksize, 28
BlockSizes, 73
blok, 25
boolský model, *viz* model boolský

C++ rozhraní, 106
checkpoint, 48
cluster, 7
clusterovaný index, 7
cyklický buffer, 51

datové typy, 89
    makra, 90
        přetypovávací, 91
deskriptor
    dokumentu, 2
dictionary, *viz* slovník, *viz* slovník
dotaz
    pozitivně omezující, 14
    proximitní, 9, 14
dotazování, 101
drop, *viz* rušení indexů
dynamické programování, *viz* programování dynamické
díra, 25

entita
    indexovatelná, *viz* i–entita
    textová, *viz* t–entita

funkce
    `idx_close_select`, 103
    `idx_create_ex`, 93
    `idx_create`, 93
    `idx_delete_all_records`, 105
    `idx_delete_records`, 106
    `idx_delete_record`, 106
    `idx_done`, 92
    `idx_drop`, 94
    `idx_fetch_next`, 103
    `idx_flush_all`, 106
    `idx_flush`, 106
    `idx_init`, 92
    `idx_insert_record`, 104
    `idx_mount`, 93
    `idx_print`, 106
    `idx_register`, 92
    `idx_unmount`, 93
    `idx_unregister`, 92
    `idx_update_records`, 105
    `idx_update_record`, 105

hašovaný index, *viz* index hašovaný
hlavička, 26

i–entita, 23
identifikátor indexu, 89
index, 2, 5, 71
    clusterovaný, 7
    hašovaný, 7
    lexikografický, 6
    shlukovaný, 7
indexace, 2
IndexBlock, 72
IndexDirectory, 72, 113
indexovatelná entita, *viz* i–entita
indexový soubor, *viz* soubor indexový
indexový záznam, 8

IndexRecord, 72
IndexRecordFormat, 72
IndexRecordsStorage, 113
IndexRecordStorage, 72
Indices, 5
invertovaný soubor
    dynamický, 25
inverze seznamu, 16

knihovna
    basic_data_structures.lib, 69
    C interface, 68
    file_system.lib, 68
    logger, 68
    logger.lib, *viz* knihovna logger
    test.lib, 69
    unindex.lib, 68
    unindexc.lib, *viz* knihovna C interface
    univerzální index, 68
kolekce
    dokumentů, 2
komprese, *57*
konstanty
    `MAOP_`, 91
    `MDT_`, 90
    `MLOP_`, 91
    `MREL_`, 91
kontrolní bod, *viz* checkpoint
kurzor, 101
kód
    prefixový, 59
kódování
    B–blokové, 62
      kombinované, 63
      základní, 62
    diferenční, 62
    Eliasovo, 58
    RLE znaménkových bitů, 66
    vektorových atributů, 67

lexikografický index, *viz* index lexikografický
little endian, 112
log, 47
    formátování, 51
    redo-log, 47
    s odloženou realizací změn, 47
log s odloženou realizací změn, 34

mazání, 105
metrika
    vektorová, 6
model
    boolský, 5
    rozšířený boolský, 6
    vektorový, 5
modul
    administrátorský, 1
    BasicDataStructures, 71
    C interface, 69
    C++ interface, 69
    dictionary, *viz* slovník, 71
    directory, 71
    dodání dokumentu, 1
    dotazovací, 1
    FileSystem, 71
    index, 70, 71
    IndexManager, 70
    kompresní engine, 70
    logger, 71
    PersistentStorage, 71
    QueryProcessor, 71
    transakční log, 71
    TransakčníManažer, 70
    vyhodnocení dotazu, 1
    zpřístupnění dokumentu, 1
mount, 93

oblast, 25
oblast přetečení, 34
ocas, 26
odpojení indexu, *viz* unmount
operace
    dotazování, *viz* operace query
    insert, 28
    query, 27
    vkládání, *viz* operace insert
operátor
    pri, 10
    sec, 10
    set, 10
optimalizace dotazů, 18

PAT pole, 7
PAT strom, 6
pole
    PAT, *viz* PAT pole
prefixový B-strom, 15
programování
    dynamické, 20

proximitní dotaz, *viz* dotaz proximitní, *viz* dotaz proximitní
připojení indexu, *viz* mount

query optimizer, *18*

redo-log, 34
rozšířený boolský model, *viz* model rozšířený boolský
rušení indexů, 94

seznam
  indexových záznamů, 10
  indexových záznamů s negací, 10
shlukovaný index, 7
signatura, 7
signaturový soubor, *viz* soubor signaturový
sistring, 6
slovník, 25, 71
  indexovanýchch termů, 8
složka indexového záznamu
  primární, *9*
  sekundární, 9
soubor
  indexový, 2, 8
  invertovaný, 8
  signaturový, 7
soubor indexových záznamů, 25
souřadnice výskytu termu v dokumentu, 14
stabilní třídící algoritmus, 17
stop-list, 31
strom
  PAT, *viz* PAT strom
subsystém
  dodání dokumentu, *viz* modul dodání dokumentu
  zpřístupnění dokumentu, *viz* modul zpřístupnění dokumentu
swapování, 56

t–entita, 23
teoretická efektivita, *35*
textová entita, *viz* t–entita
tezaurus, 24
transakce, 46
transakční log, 71
TRIE, 15
typ
  `t_any_type`, 95
  `t_arithmetic_expr_node`, 101
  `t_attribute_expr`, 99
  `t_global_settings`, 91
  `t_index_descriptor`, 92
  `t_index_format`, 93
  `t_index_record`, 95
  `t_list_expr_node`, 98
  `t_query`, 97, 103
  `t_record_attribute`, 93
  `t_record_list`, 94
  `t_rel_expr_node`, 100
tělo, 26
třída
  persistent storage, 73
třídící algoritmus
  stabilní, *viz* stabilní třídící algoritmus

unmount, 93
uspořádané pole, 14
uspořádání časové, 47

vazba
  zpětná, 6
vektorová metrika, *viz* metrika vektorová
vektorový model, *viz* model vektorový
vkládání, 104
vytvoření indexu, 93
výměna dat, 94

Zipfův zákon, 20, 26
zpětná vazba, *viz* vazba zpětná
zákon Zipfův, *viz* Zipfův zákon
záznam
  indexový, 8
záznam indexový, *viz* indexový záznam

časová známka
  neplatná, 47