VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

LOKÁLNÍ VYHLEDÁVÁNÍ PRO LINUX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MICHAL PRÍVOZNÍK
AUTHOR

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

LOKÁLNÍ VYHLEDÁVÁNÍ PRO LINUX

DESKTOP SEARCH FOR LINUX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL PRÍVOZNÍK
AUTHOR

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

BRNO 2010

Abstrakt

Tato práce se zabývá indexací, různými typy indexovacích struktur, jejich výhodami a nevýhodami. Poskytuje základ pro vytvoření vyhledávače s podporou morfologie, různých formátů souborů. Nastínila základní myšlenky, jejichž rozvinutí je cílem diplomové práce.

Abstract

This work deals with indexing, different types of indexing structures, their advantages and disadvantages. It provides the basis for a search engine with support of morphology or different file formats. It gives insight to the basic ideas, which answer is aim of the master's thesis.

Klíčová slova

Algoritmy, datové struktury, index, vyhledávaní, Linux

Keywords

Alogrithm, data structures, index, search, Linux

Citace

Michal Prívozník: Lokální vyhledávání pro Linux, diplomová práce, Brno, FIT VUT v Brně, 2010

Lokální vyhledávání pro Linux

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána doc. RNDr. Pavla Smrže, Ph.D.

.....
Michal Prívozník
19. května 2010

Poděkování

Rád by som poďakoval svojmu vedúcemu doc. RNDr. Pavlovi Smržovi, Ph.D., za ústretový prístup a hodnotné pripomienky.

© Michal Prívozník, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**
- 1.1 Information Retrieval systémy . . . 4
- 1.2 Úvod . . . 5
- 1.3 Index . . . 5
  - 1.3.1 Suffix tree . . . 6
  - 1.3.2 Inverted index . . . 7
  - 1.3.3 Document term matrix . . . 8
- 1.4 Kompresia indexu . . . 10
- 1.5 Proces indexácie . . . 10
- 1.6 Štruktúra IR systému . . . 11
- 1.7 Modely IR systémov . . . 12
  - 1.7.1 Booleovský model . . . 12
  - 1.7.2 Pravdepodobnostné modely . . . 12
  - 1.7.3 Vector Space Model . . . 13

2 Information Retrieval systémy a ich komponenty **15**
- 2.1 Booleovský model . . . 15
- 2.2 Slovníky a pozičné súbory . . . 20
  - 2.2.1 Tokenizácia . . . 21
  - 2.2.2 Stop slová . . . 22
  - 2.2.3 Normalizácia . . . 22
  - 2.2.4 Stemming a Lematizácia . . . 22
  - 2.2.5 Korekcia pravopisu . . . 25
- 2.3 Vytváranie indexu . . . 26
  - 2.3.1 BSBI . . . 26
  - 2.3.2 SPIMI . . . 27
  - 2.3.3 Dynamické indexy . . . 28
- 2.4 Kompresia indexu . . . 28
  - 2.4.1 Heapsov zákon . . . 28
  - 2.4.2 Zipfsov zákon . . . 29
  - 2.4.3 Kompresia slovníka . . . 29
  - 2.4.4 Kompresia postings zoznamu . . . 30
- 2.5 Ohodnocovanie výsledkov dotazu . . . 31
- 2.6 Vector Space model . . . 32
  - 2.6.1 Implementácia . . . 34
- 2.7 Latentná sémantická analýza . . . 34

3 Prehľad a porovnanie existujúcich nástrojov **37**
3.1 Google Desktop Search - GDS . . . 37
3.2 X1 Professional Client - XPC . . . 38
3.3 Microsoft Desktop Search - MDS . . . 39
3.4 Beagle . . . 40
3.5 Porovnanie vyhľadávačov . . . 41
3.5.1 Úvod do metrík . . . 41
3.5.2 Metodika . . . 44
3.6 Výsledky . . . 45

4 Návrh **49**
4.1 Indexer a klient . . . 49
4.2 Komunikácia . . . 50
4.3 Indexer . . . 50
4.3.1 Index . . . 50
4.3.2 Reader . . . 52
4.3.3 Konfigurácia . . . 53
4.3.4 Stemmer . . . 54
4.3.5 Ostatné . . . 54
4.4 Klient . . . 55

5 Implementácia **57**
5.1 Indexovanie v CLucene . . . 57
5.2 Komunikácia . . . 58
5.3 Indexer . . . 58
5.4 Klient . . . 61
5.5 Porovnanie s ostatnými aplikáciami . . . 62

6 Záver **66**

A Manuál **70**

B Konfiguračný súbor **72**

Predslov

V dnešnom svete plnom informácií je dôležité nájsť tú správnu rýchlo a spoľahlivo, pretože len tak možno konať efektívne. Ak aj máme veľkú zbierku kníh, ale nemáme spôsob, ako nájsť hľadanú informáciu, nie je pre nás táto prínosom. Najvodnejším príkladom je internet. Nájdeme tam veľmi veľa informácií o každom aspekte ľudského života: kutlúra, veda, šport, zábava, atď. Lenže čo ak nevieme kde sa hľadaná informácia nachádza? Použijeme vyhľadávač, ktorý nás nasmeruje tým správnym smerom. Na základe kľúčových slov, ktoré jednoducho a čo najpresnejšie špecifikujú cieľ nášho hľadania, vyhľadávač zobrazí výsledok. Podobne je to aj s lokálnym vyhľadávaním, kedy neprehľadávame internet, ale lokálne dáta, teda dokumenty v rôznych formátoch uložené na našom pevnom disku. Samozrejme, s podporou operačného systému nemusíme hľadať len na lokálnych diskoch, ale aj na zdieľaných. V tomto prípade je to už však úlohou operačného systému, ktorý zaručuje, že prístup k vzdialeným diskom sa nelíši od prístupu k lokálnym, teda sú plne transparentné a možno k nim pristupovať bez zmeny programu, aplikácie.

Túto diplomovú prácu som si vybral pre svoj pozitívny vzťah k operačnému systému Linux a verím, že jej výsledky budú prínosom.

Prvá kapitola predstavuje úvod do problematiky získavania informácií z dokumentov. Zaoberá sa štruktúrou a modelmi Information Retrieval systémov a tvorí základ pre druhú kapitolu, v ktorej sa vybraným témam venujem podrobne. Téma vyhľadávania je veľmi obsiahla a ani jej bližšia špecializácia na lokálne vyhľadávanie ju nezmenšuje natoľko, aby som sa mohol venovať celej problematike podrobne. V tejto kapitole predstavím jednotilvé indexovacie štruktúry, ich výhody a nevýhody.

V druhej kapitole sa potom venujem výstavbe invertovaného indexu, rozoberiem postupne všetky komponenty, ktoré tento proces ovplyvňujú. Dôraz je kladený na čo najmenšie množstvo spotrebovaných prostriedkov, keďže tieto sú veľmi kritické na poli lokálneho vyhľadávania. Každá preberaná problematika je znázornená stručne a jasne a podľa možnosti za použitia obrázku, či tabuľky so vzorovými dátami.

Tretia kapitola predstavuje jednotlivé vyhľadávače zo zadania, zaoberá sa problematikou porovnávania aplikácií tohto druhu, predstavuje základné metriky používané na konferenciách TREC a vyhodnocuje aplikácie na ich základe.

V štvrtej kapitole sa budeme zaoberať výstavbou vlastného výhľadávača, pričom budeme klásť dôraz na výsledky porovnávaní z tretej kapitoly. Takže sa budeme vedieť vyhnúť zisteným nedostatkom a odstrániť ich už pri návrhu. V tejto kapitole sa budeme tiež zaoberať stemmerom pre slovenský jazyk.

V piatej kapitole zhrnieme problémy, ktoré sa vyskytli pri implementovaní návrhu zo štvrtej kapitoly a uvedieme ich riešenie. Následne porovnáme vytvorený program s aplikáciami z tretej kapitoly a načrtneme možné zlepšenia.

V závere zhodnotíme prácu, zhodnotíme jej prínos.

Kapitola 1

Úvod

V tejto kapitole sa oboznámime so základnými pojmami, aby sme získali predstavu o IR systémoch, ktorú budeme ďalej rozširovať, predstavu o ich štruktúre, problémoch pri návrhu a konštruovaní systému. Vymedzíme funkcie, ktoré budeme od systému vyžadovať. Táto kapitola je prípravou pre druhú kapitolu, kde si predstavíme konkrétne algoritmy a riešenia.

1.1 Information Retrieval systémy

Information Retrieval (IR) systémy a ich výskum sa zameriava na vyvíjanie algoritmov a modelov pre získavanie dát z dokumentov. Sú súčasťou Natural Language Processing (NLP), čo je odbor na pomedzí informatiky, lingvistiky a umelej inteligencie, ktorého cieľom je spracovanie prirodzeného (ľudského) jazyka na počítači. Christopher Manning, jedna z najvýznamnejších postáv dnešných IR systémov, definuje IR ako získavanie materiálu (najčastejšie dokumentov) z neštruktúrovaného charakteru (zvyčajne text) tak, že vybrané entity spĺňajú potrebu informácií z veľkých zbierok (obyčajne uložených v počítači) [42]. V tejto definícii je obsiahnutý základ a odpovedá na otázky: Čo budeme robiť? Získavať, hľadať. Čo bude výsledkom? Dokumenty. Prečo to budeme robiť? Aby sme uspokojili informačnú potrebu. Nad čím budeme hľadať? Nad veľkou zbierkou neštruktúrovaných dát. Hoci táto definícia evokuje pocit, že vyhľadávanie je len doménou knihovníčok alebo úzkej skupiny ľudí, nie je tomu tak. Dnes ľudia bežne vyhľadávajú potrebné informácie na internete alebo v svojich e-mailoch. Pod neštruktúrovanými dátami z definície sú myslené dáta, ktoré nemajú jasnú a pre počítač ihneď pochopiteľnú štruktúru. Sú protikladom štruktúrovaných dát, ktorých krásnym príkladom môžu byť relačné databázy alebo dáta z burzy. V nich je každému číslu ihneď priradený jednoznačný význam. Príkladom neštruktúrovaných dát tak môže byť román alebo hovorené slovo. Ak si totiž predstavíme bohatosť prirodzeného jazyka, častokrát ani človek nedokáže vetu pevne uchopiť (napr. ak obsahuje zamlčané podmety a nie je v kontexte). V skutočnosti však máloktoré dáta sú naozaj neštruktúrované. Spomínaný román je členený nielen graficky (kapitoly, nadpisy, odstavce) ale aj významovo. Aj vetným členom možno prisúdiť štruktúru (podmet, prísudok). Jedným z cieľov NLP a IR je práve hľadanie štruktúry v dátach, ich klasifikácia.

Druhou oblasťou pôsobnosti je *clustering* – zoskupovanie dokumentov na základe podobnosti obsahu. Je to podobné triedeniu kníh na poličke na základe témy. Obdobou je klasifikácia, ktorá prvku z danej množiny tém priradí množinu (aj prázdnu) dokumentov. Často je prvých pár dokumentov klasifikovaných ručne s nádejou, že zvyšné budeme vedieť klasifikovať automaticky.

IR systémy sa líšia aj v poli pôsobnosti. To totiž určuje celú infraštruktúru sytému a jeho nároky na výpočtový výkon. Rozoznávame: vyhľadávanie na internete – mili óny ba až miliardy dokumentov uložených na miliónoch počítačov. Tu sa zameriavame na získavanie dokumentov pre indexáciu, budovanie systémov pracujúcich efektívne vo veľkých merítkach a zohľadňovanie špecifických aspektov webu ako prehľadávanie hypertextových dokumentov bez ovplyvnenia ich tvorcami, manipulujúcimi ich obsah za účelom zvýšenia rankingu. Na druhom konci je lokálne vyhľadávanie či už v rámci e-mailového klienta alebo operačného systému. Dôležitými aspektmi v tomto prípade sú: spotrebované miesto na disku, ľahká spravovateľnosť (v ideálnom prípade nebude žiadna potrebná) a minimálne systémové požiadavky, takže aplikácia môže bežať na jednom počítači a užívateľ nepociťuje nepríjemne jej prítomnosť, nie je obmedzovaný. Niekde uprostred potom ležia korporátne systémy, ktoré sú sprostredkovávané určitému okruhu užívateľov a zameriavajú sa na určitý typ dokumentov. Príkladom môže byť patentový úrad s jedným, prípadne viac, centrálnymi počítačmi kde sú uložené patenty v elektronickej podobe.

1.2 Úvod

V tejto časti sa budeme zaoberať základnými pojmami, ktoré budeme potrebovať pri návrhu aplikácie.

Predstavme si nasledovnú situáciu. Máme N objektov a potrebujeme nájsť jeden, ktorý má nejakú konkrétnu vlastnosť. Triviálnym riešením je prechádzať postupne všetky objekty a porovnávať, či sme našli ten hľadaný. V priemere tak úspešné hľadanie preskúma polovicu objektov, neúspešné všetky. Časová zložitosť je tak $O(N)$. Táto je však neúnosná, keď si predstavíme, že napr. databázy alebo dátové sklady obsahujú milióny objektov a vyhľadávanie je bežná operácia. Preto sa snažíme túto operáciu čo najviac zefektívniť tak, aby sme dosiahli lepšiu časovú zložitosť (sublineárnu) a vyhovujúcu pamäťovú náročnosť.

1.3 Index

Index je dátová štruktúra, ktorá zbiera, triedi a ukladá dáta za účelom rýchlejšieho a presného vyhľadávania informácií. Index design, teda návrh týchto štruktúr, spája v sebe matematiku, informatiku, fyziku, lingvistiku. Častokrát však použitie indexu spomalí niektoré iné operácie - v databázach je to napr. zapisovanie, kedy sa musí informácia uložiť nielen do samotnej tabuľky, ale aj do indexu, čo vyúsťuje do vyšších nárokov na úložný priestor. Na ilustrovanie potreby indexovania môže slúžiť aj fakt, že všetky databázy na trhu používajú index. Typicky na uloženie indexu je potrebného menej úložného priestoru než na uloženie celej tabuľky, keďže index sa nevytvára nad všetkými stĺpcami, ale len nad kľúčovými.

Keďže aj samotné vytváranie indexu je časovo náročná úloha, je potrebné ho prispôsobiť konkrétnym podmienkam, aby bolo čo najviac efektívne. V podstate sa z tohto hľadiska rozlišujú 2 spôsoby vytvárania indexu:

opakované indexovanie v pravidelných intervaloch Toto sa využíva najmä pri rozľahlých vyhľadávacích službách, ako napríklad internetové vyhľadávače, pretože umožňuje plánovanie vyťaženia výpočtového času a do istej miery aj priestorového.

agentové indexovanie Využíva sa najmä lokálne, na desktope, pretože objem dát k indexácií je omnoho menší a navyše operačné systémy umožňujú informovať procesy,

ktoré o to požiadajú, že ten či onen súbor sa zmenil. Vo výsledku je tak indexovanie na začiatku veľmi náročné a zaberie veľa výpočtového výkonu, no potom stačí, aby si indexovací agent zaregistroval odber informácií o zmene súborov (ide o služby linuxového jadra *inotify* a *dnotify*), čo mu umožňuje udržiavať aktuálny index.

Populárne vyhľadávače sa zameriavajú na indexovanie online dostupných, v prirodzenom jazyku písaných dokumentov a zatiaľ čo meta-vyhľadávacie služby využívajú indexy iných služieb.
Existujú aj vyhľadávače, služby, ktoré spolu s indexom ukladajú aj akýsi základ práve indexovaného dokumentu, napríklad pre prípad, že by sa dokument stal nedostupným.

Z hľadiska návrhu indexovacej štruktúry [28] rozoznávame niekoľko základných vlastností:

spájacie faktory určujú ako sú dáta pridávané do indexu a tým aj to, či môže viacero dalších indexovacích agentov pristupovať k indexu simultánne. Agent musí najskôr zistiť, či indexuje novú, alebo modifikuje starú položku.

techniky uloženia ako je index uložený. Určujú typ súboru, použitie komprimácie, prípadne rozdelenie do viacerých súborov, atď.

veľkosť indexu koľko úložného priestoru je zabraté indexom.

rýchlosť vyhľadávania ako rýchlo sme schopní v indexe nájsť požadovanú informáciu v porovnaní s tým, ako rýchlo môže byť informácia aktualizovaná, resp. odobratá.

spravovateľnosť či a ako je index spravovateľný.

tolerancia chýb dostupnosť služby, poruchy konzistencie indexu, redundancia.

Z hľadiska dátových štruktúr rozoznávame niekoľko typov indexov:

1. Suffix tree
2. Inverted index
3. Citation index
4. Ngram index
5. Document term matrix

1.3.1 Suffix tree

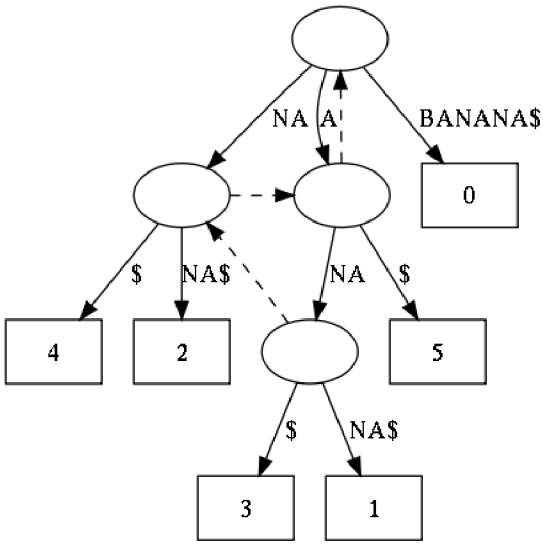
Je stromová dátová štruktúra, kde pre každý suffix reťazca S existuje práve jedna cesta od koreňa k listu. Je to špeciálny prípad *radix tree* pre suffixy S. Vytvorenie takého stromu má lineárnu časovú zložitosť, ale akonáhle je strom vytvorený, umožňuje rýchle vyhľadávanie, identifikáciu podreťazca, dokonca je odolné voči istému množstvu chýb (napr. preklepov). Toto zrýchlenie však niečo stojí - miesto pre suffix tree pre jeden reťazec vyžaduje viac miesta než uloženie samotného reťazca.

Suffix tree má nasledujúce vlastnosti:

1. cesta od koreňa k listu je zhodná so suffixom reťazca S
2. hrany sú neprázdne sledy znakov

3. všetky uzly majú najmenej 2 potomkov

Keďže takýto strom neexistuje pre všetky reťazce, vkladá sa doň jedinečný terminálny symbol, ktorý sa nevyskytuje v žiadnom reťazci obsiahnutom v strome, zvyčajne $.



Obrázek 1.1: Príklad pre slovo BANANA$. Šesť uzlov znázornených ako štvorec, prislúcha šiestim suffixom: A$,NA$,ANA$,NANA$,ANANA$ a BANANA$. Číslo v štvorci určuje poradie.

Na obrázku 1.1 vidíme príklad stromu pre reťazec BANANA$. Čiarkovane sú znázornené odkazy na uzol s nasledujúcou vlastnosťou: ak cesta od koreňa k niektorému uzlu určuje podreťazec $\alpha\beta$ a $|\alpha| = 1$, $|\beta| \geq 0$, tento uzol obsahuje odkaz na uzol určujúci β. V tomto prípade napr. uzol ANA odkazuje na uzol NA.

Táto indexovacia štruktúra má tú vlastnosť, že umožňuje vytváranie v lineárnom čase, vyhľadávanie regulárnych výrazov, hľadanie najdlhšieho spoločného podreťazca, atď. Nevýhodou však je pamäťová náročnosť: Strom pre reťazec S má $n = |S|$ listov. Keďže každý vnútorný uzol má minimálne 2 potomkov, môže byť týhto až $n - 1$ a teda celkovo $n + (n - 1) + 1 = 2n$ (n listov, $n - 1$ vnútorných uzlov a 1 koreň).

1.3.2 Inverted index

Táto indexovacia štruktúra uchováva zoznam slov a ku každému odkaz na pôvodný súbor. Rozlišujeme 2 druhy: prvý *inverted file* uchováva len odkaz ku každému slovu na príslušný súbor, kde sa vyskytuje. Druhý, *full inverted index* uchováva aj pozície slova v dokumente. Umožňuje tak hľadanie presných fráz, vyžaduje však viac úložného miesta a aj času na vytvorenie.

Majme vety: V_1=`'to je príklad'`, V_2=`'čo to je'` a V_3=`'dnes je nedeľa'`. Invertovaný index prvého typu bude potom vyzerať ako v tabuľke 1.3.2.

Ak budeme hľadať `'to' 'je'`, výsledkom je $\{1, 2\} \cap \{1, 2, 3\} = \{1, 2\}$. Tento index sa často nazýva aj *boolean index*, lebo dokáže len určiť, či sa daný výraz v dokumente nachádza alebo nie. Nedokáže určiť aj početnosť výskytu a teda ani relevantnosť výsledku, ktorý je síce správny – hľadané slová sa skutočne v tých dokumentoch nachádzajú, avšak

to	{1,2}
je	{1,2,3}
príklad	{1}
čo	{2}
dnes	{3}
nedeľa	{3}

Tabulka 1.1: Príklad inverted index pre vety V_1=`'to je príklad'`, V_2=`'čo to je'` a V_3=`'dnes je nedeľa'`.

môžu byť spomenuté len okrajovo a nemusia byť teda to, čo hľadáme. Touto problematikou sa budeme podrobne zaoberať v sekcií 1.7 o modeloch IR systémov.

Majme teraz vety: V_1=`'to je banán'`, V_2=`'je to to, čo to je'`, V_3=`'čo je to'`. Invertovaný index druhého typu bude už zložitejší:

to	{(1,1),(2,2),(2,3),(2,5),(3,3)}
je	{(1,2),(2,1),(2,6),(3,2)}
banán	{(1,3)}
čo	{(2,4),(3,1)}

Tabulka 1.2: Príklad plne invertovaného indexu pre vety V_1=`'to je banán'`, V_2=`'je to to, čo to je'`, V_3=`'čo je to'`.

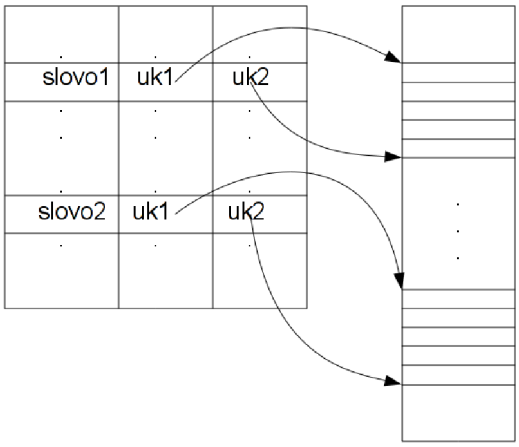
Každý odkaz je dvojica (a, b), kde a odkazuje na vetu, b na pozíciu v nej. Ak sa teraz pokúsime vyhľadať `'čo je to'`, dostaneme len vetu 3, hoci sa jednotlivé slová nachádzaju aj vo vete 2. [34][33] Tento typ invertovaného indexu umožňuje už aj vyhľadávať presné frázy, keďže si uchováva aj informáciu o slede slov. Ak by sme túto maticu chceli reprezentovať tak, že každý prvok zoznamu by bol v zvláštnom stĺpci, dostali by sme riedku maticu. Invertovaný index môže byť považovaný za hashovaciu tabuľku.

Tak ako v našom príklade aj v praxi obsahuje invertovaný index odkazy do tabuľky súborov, čo umožňuje lepšiu správu záznamov – ak napr. k súborom uchovávame aj metadáta, nie sú tieto uložene priamo v indexe, ale v zvlášť tabuľke. Ďalšou variáciou je *dynamický invertovaný index*. Ten sa skladá z 2 základných častí: slovník a pozičný súbor (postings file). Položka v slovníku má 3 zložky: slovo, počiatočný a koncový ukazateľ do postings file. V postings file je potom spojitý zoznam pozícií na ktorom sa dané slovo vyskytuje. Situáciu znázorňuje obrázok 1.2. Tento prístup umožňuje vyššiu kompresiu indexu. [24]

1.3.3 Document term matrix

je matica, ktorá popisuje početnosť výskytu slov v jednotlivých vetách, dokumentoch. V riadkoch je názov súboru, v stĺpcoch slová. Uvažujme nasledujúce vety: V_1=`'to je banán'`, V_2=`'je to to, čo to je'`, V_3=`'čo je to'`. Matica bude potom vyzerať nasledovne (tabuľka 1.3.3):

Nie vždy chceme uchovávať všetky slová z dokumentu, pretože by to viedlo k zbytočne veľkej matici. Ukladáme tak len tie slová, ktoré sú relevantné - napr. v rodine Indo–Európskych jazykov to sú podstatné mená, slovesá a prídavné mená, prípadne môžeme ukladať aj slovné spojenia. Toto nám potom umožňuje istým spôsobom definovať tému dokumentu.



Obrázek 1.2: Ukážka dynamického invertovaného indexu. Vľavo je slovník vpravo postings file.

	to	je	banán	čo
V_1	1	1	1	0
V_2	3	2	0	1
V_3	1	1	0	1

Tabulka 1.3: Príklad document term matrix pre vety V_1=`'to je banán'`, V_2=`'je to to, čo to je'`, V_3=`'čo je to'`.

Táto matica sa využíva v *Latentnej Sémantickej Analýze (LSA)*, čo je technika pri spracovaní prirodzeného jazyka, na analýzu vzťahov medzi súborom a výrazmi, ktoré obsahuje tak, že produkuje viacero množín konceptov.

Predstavme si nasledujúci príklad: Máme sadu textov a chceme určiť všetky tie, ktoré najviac korešpondujú s výrazom `'the game theory'`. Začneme tým, že eliminujeme všetky tie dokumenty, ktoré neobsahujú ani jedno zo slov `'the'`, `'game'` alebo `'theory'`. Avšak potom stále zostane veľa dokumentov. Ak by sme teraz chceli spočítať počet výskytov jednotlivých slov v dokumentoch, posunieme sa síce o krok bližšie, no relevantnosť takéhoto prístupu je otázna. Problém môže byť práve slovo `'the'`, ktoré sa v anglických textoch vyskytuje veľmi často a mohlo by tak dať vyššiu váhu menej relevantnému dokumentu. Tento nedostatok sa odstráni zavedením *inverznej frekvencie* (inverse document frequency), ktorá znižuje váhu podmienok, ktoré sa vyskytujú až príliš často a naopak zvyšuje váhu tých, ktoré sa vyskytujú zriedka.

Frekvencia výskytu jednotlivých slov sa navyše normalizuje (term frequency), aby sme nezvýhodňovali dlhšie dokumenty:

$$\mathtt{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{1.1}$$

kde $n_{i,j}$ označuje počet výskytov slova t_i v dokumente d_j.

Inverzná frekvencia je potom:

$$\texttt{idf}_i \;=\; \log \frac{|D|}{|\{d : t_i \in d\}|} \tag{1.2}$$

kde $|D|$ je počet všetkých dokumentov, $|\{d : t_i \in d\}|$ je počet dokumentov, kde sa vyskytuje slovo t_i. Ak sa slovo nenachádza v žiadnom dokumente, mohlo by to viesť k deleniu nulou, preto sa niekedy používa skôr: $1 + |\{d : t_i \in d\}|$. Potom

$$\texttt{(tf-idf)}_{i,j} \;=\; \texttt{tf}_{i,j} \;\times\; \texttt{idf}_i \tag{1.3}$$

Uvažujme dokument obsahujúci 100 slov, kde sa slovo `'game'` vyskytuje 5-krát. Term-frequency pre toto slovo je potom $\frac{5}{100} \;=\; 0.05$. Tento dokument je súčasťou ďalších 10 000 000 dokumentov, kde sa slovo `'game'` vyskytuje len v 1 000. Inverse document frequency je potom $\log(\frac{10\ 000\ 000}{1\ 000}) \;=\; 4$. Celkové skóre je teda $0.05 \;\times\; 4 \;=\; 0.2$.

1.4 Kompresia indexu

S veľkosťou a počtom indexovaných dát vzrastá aj objem indexu. Spravovanie a udržiavanie takéhoto indexu je časovo a priestorovo náročné. Veľa vyhľadávačov preto využíva kompresiu na jeho zmenšenie. Skúsme si predstaviť, k akým veľkostiam sa môžeme až dopracovať:

1. V roku 2002 bolo na internete okolo 1 400 000 000 verejných stránok [43].
2. V priemere má každá stránka 441 stránok (knižných) [43].
3. Každá knižná stránka má priemerne 250 slov [22].
4. Na uloženie jedného znaku, potrebujeme 1 bajt, pri niektorých kódovaniach 2 (napr. Unicode)
5. Slová majú v priemere 5 znakov.

V tomto príklade by sme potrebovali $1\ 400\ 000\ 000 * 441 * 250 * 5$ bajtov na uloženie indexu, čo je zhruba 700TB dát. V závislosti od zvolenej techniky kompresie, môže byť tento objem znížený na zlomok pôvodného. Za zmienku stoja aj výdavky spojené s napájaním dátového centra elektrickou energiou, pretože čím menej diskov musíme napájať, tým menej je spotrebovanej elektrickej energie.

1.5 Proces indexácie

sa skladá z viacerých častí:

parsovanie dokumentu pripravuje dokument pre indexáciu. Rozdelí vstupný text na jednotky, lexémy, nazývané *tokeny*. Preto sa niekedy tomuto procesu hovorí aj *tokenizácia*. Čas strávený dobrou tokenizáciou je neskôr vykompenzovaný kratším časom hľadania. Počítač nerozumie prirodzenému jazyku, jeho štruktúre a nevie tak identifikovať, kde končí jedno slovo a začína druhé. Pre počítač je veta len postupnosť znakov. Na rozdelenie sa tak požívajú špeciálne programy (ako napr. Lex), ktoré na základe definovaných pravidiel, rozdelia vstupný text na tokeny a prípadne ich aj identifikujú. Takýmito pravidlami môže byť napr. rozdelovanie na základe medzier,

interpunkčných znamienok a pod. Identifikácia tokenu ho potom zaradí do entít, ako napr. email, telefónne číslo, URL. V prípade anglického dokumentu je parsovanie relatívne jednoduché (vďaka skladbe anglických slovíčok príponami a predponami), no v iných jazykoch to predstavuje výzvu, napr. japončina alebo čínština, kde nie sú slová oddelované medzerami. Už v tejto časti je možné aspoň čiastočne vyselektovať slová, ktoré užívateľ bude hľadať, ktoré sa majú indexovať.

rozpoznávanie jazyka ak vyhľadávač podporuje viaceré jazyky (pri vyhľadávaní, nie z hľadiska užívateľského rozhrania), je tento krok zaradený medzi tokenizáciu a samotné vloženie do indexovacej štruktúry. Mnohé z nasledujúcich krokov totiž závisia od jazyka, v ktorom je dokument napísaný, napr. *stemming* – redukcia slova na jeho koreň odstránením predpon alebo prípon, stochastickou analýzou, slovníkom.[41]

identifikovanie formátu dokumentu často indexujeme nie len holý text, ale aj dokumenty v rôznych formátoch, napr. HTML, PDF, PS, XML, ODF, ... Keďže každý z týchto formátov je iný, je nutné k nim aj špecificky pristupovať. Ak by totiž indexovací agent ignoroval fakt, že súbor je v XML formáte, dostali by tak tokeny, ktoré nechceme indexovať – XML značky. Alebo v prípade ODF dokumentu, nepotrebujeme ukladať formátovanie a zvýraznenie niektorých častí textu. Tu treba spomenúť aj fakt, že veľa formátov je proprietárnych, zatiaľ čo špecifikácia ostatných je voľne dostupná. Otázkou je tiež podpora komprimovaných súborov (zip, rar, gzip, bzip2, ...). Tu je potrebné najskôr archív rozbaliť a až potom naindexovať jeho obsah.

rozpoznávanie hlavných častí je dôležité, pretože nie všetky dokumenty sú písané ako kniha. Napr. internetové noviny obsahujú okrem hlavného článku aj postranné reklamy, odkazy na iné články, a pod. Ak by sme k tomu pristupovali ako ku knihe, zaniesli by sme do indexu nepresnosť.

meta informácie veľa formátov súborov podporuje meta informácie – informácie o informáciách. Tak napr. v PDF dokumente môže byť uvedený autor ako meta informácia, hoci v samotnom texte nemusí byť uvedený vôbec. Pre moderný vyhľadávač je potrebné, aby tieto informácie vedel nájsť, identifikovať a získať.

Pri indexovaní na lokálnom počítači je nutné si najskôr ujasniť, čo všetko chceme do indexu zahrnúť, ktoré súbory a zložky. Prechádzanie tohto stromu sa nazýva *crawling*, proces, ktorý ho vykonáva *crawler*.

1.6 Štruktúra IR systému

Veľa dnešných IR systémoch je vystavaných na invertovanom indexe 1.3.2, resp. na jeho vylepšenej variante obsahujúcej aj informácie o pozícií v dokumente. Môžeme tak vyhľadávať presné frázy. Problémom však je, že naozaj len presné frázy. Ak by sme totiž hľadali `'IR system design'`, nenašli by sme dokumenty zaoberajúce sa `design of IR system`. Tento problém sa dá vyriešiť zavedením modulu na detekovanie fráz a vytvorením pomocného indexu fráz. Tieto sa však samotnom systéme nerozlišujú od slov a narába sa s nimi rovnako. Najjednoduchším postupom je vybrať najčastejšie bigramy (dvojice slov).

Angličtina, Čeština ale aj Slovenčina a iné jazyky obsahujú slová, nazývané funkčné, ktoré sa síce vyskytujú často, ale nesú minimálnu informačnú hodnotu, napr. spojky, predložky, osobné zámena. Ak by sme tieto slová indexovali, veľkosť indexu by vzhľadom na ich frekvenciu výskytu neúmerne vzrastala, avšak práve pre ich nízku informačnú hodnotu by

neboli citeľným prínosom. Príkladom takého slova je anglické *the*, alebo česká a slovenská spojka *a*. Tieto slová sa nazývajú *stop slová*, zoznam takýchto slov sa nazýva *stop list*. Použitie tejto techniky však pochopiteľne znemožňuje vyhľadávanie fráz obsahujúcich stop slovo.

Ďalšou používanou technikou je stemming. Je to proces, ktorého výsledkom je *stem*. Vznikne odstránením predpôn a prípon zadaného slova. Vyššou formou je lematizácia ktorá prevádza slová na ich základný tvar, lemmu. Lemma je základný tvar lexémy – množiny všetkých možných tvarov slova. Rozdiel spočíva vo vnímaní kontextu. Napr. stemmer (algoritmus vykonávajúci stemming) môže považovať slová `better` a `good` za rozličné, hoci prvé je lexémou druhého. Sila kontextu sa prejaví v nasledujúcich 2 vetách: `I am talking.`, `In his talking he outlined` ... V oboch sa vyskytuje slovo talking. V prvej ako sloveso, v druhej ako podstatné meno. Stemmer by obe zredukoval, pretože nie je schopný vnímať tento rozdiel. S týmto súvisí aj problém, keď sa dve rozdielne slová zredukujú na rovnaký stem: `gallery` a `gall`. Obe majú rovnaký stem `gall-`. Najpoužívanejšie stemming algoritmy sú *Porter*, *Lovins* a *Snowball*. Pri morfologicky bohatých jazykoch (napr. slovanské jazyky) je tvorba stemnera veľmi náročná. Pre predstavu uvážme slová *zámku-zámok*, *ruky-ruce* alebo *sníh-sněhu*

1.7 Modely IR systémov

Z hľadiska teórie, na ktorej je postavený IR systém, delíme tieto do 3 hlavných skupín:

teória množín Systémy postavené na tejto teórie získavajú výsledky užitím nástrojov tejto teórie, napr. prienik množín. Patrí sem: Standard Boolean model, Extended Boolean Model a Fuzzy Model.

algebraické modely reprezentujú dokumenty a dotazy ako vektory alebo matice. Pri vykonávaní dotazu potom vyberajú tie dokumenty, ktorých vektor je najviac podobný vektoru dotazu. Patrí sem: Vector Space Model, Extended Boolean Model, Latent Semantic Indexing

pravdepodobnostné modely sú založené na teórií pravdepodobnosti. Často pri tom využívajú Bayesov teorém.

1.7.1 Booleovský model

Modely tohto druhu sú jednoduché a ľahko implementovateľné. Vznikli ako prvé avšak s menšími obmenami sa používajú dodnes. Majú totiž tú vlastnosť, že majú ľahký dotazovací jazyk, ktorý sa skladá poväčšine z 3 či 4 operátorov (AND, OR, NOT, AND NOT) a umožňuje plnú kontrolu nad vykonávaním dotazu. Pracujú veľmi dobre v prípadoch keď vieme, čo presne (alebo hrubo) kolekcia obsahuje a čo hľadáme.[32]. Pre presnejšie zadávanie dotazu sú mnohé systémy doplnené o rôzne iné operátory (napr. blízkosť slov) či podporu regulárnych výrazov. Tento typ dotazovacieho jazyka nie je použitý len v Booleovských modeloch, ale aj iných. Základný princíp týchto modelov sme už mohli vidieť v pri invertovanom indexe (sekcia 1.3.2).

1.7.2 Pravdepodobnostné modely

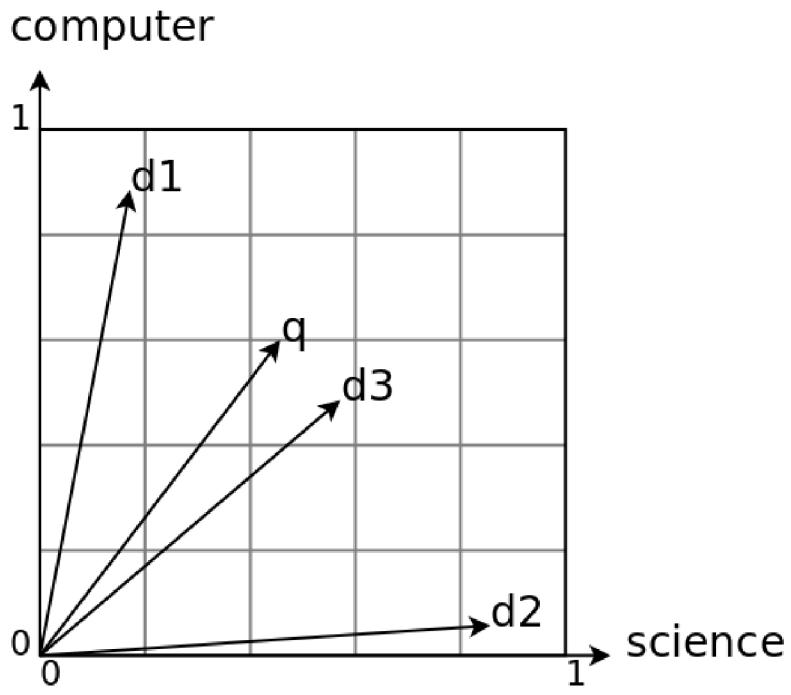
Moderné vyhľadávače umožňujú zoraďovať výsledky podľa miery relevancie. Výsledky vyhľadávania sú často prezentované na stránkach. Na prvej sú tie, ktoré systém považuje za

najrelevantnejšie – najviac zodpovedajúce dotazu a na každej ďalšej sú už menej relevantné dokumenty. Ako uvidíme neskôr (v sekcií 3.5.1) tento spôsob umožňuje užívateľovi kontrolovať mieru kompromisu medzi *precision* (presnosť) a *recall* (množstvo). Ak totiž na prvej stránke nie je hľadaný dokument, užívateľ môže prejsť na druhú stránku, čím sa zvíši recall za cenu zníženia precision.

Nosnou myšlienkou pravdepodobnostných modelov je, že zobrazujú dokumenty v zostupnom poradí vzhľadom na pravdepodobnosť optimálnosti relevancie. Pracujú tak, že pre každý dokument určia pravdepobnosť, že aktuálne spracovávaný dokument je to, čo hľadáme a zaradia ho do zoznamu. Tento zoznam je radený zostupne podľa pravdepodobnosti. Vyvstáva tu však viacero problémov. V prípade, že máme medzi dokumentmi 2 duplikáty d_1 a d_2, oba dostanú rovnaké ohodnotenie (ranking). Ale ak medzi výsledok vyhľadávania zaradíme d_2 zatiaľ čo d_1 už v zozname je, užívateľovi to neprinesie žiadnu novú informáciu. Iste, môžeme takéto duplicitné dokumenty z uvažovania vylúčiť, ale to porušuje princíp pravdepodobnostných modelov.

1.7.3 Vector Space Model

Je to jeden z najpoužívanejších modelov. Dokumenty a dotazy sú reprezentované ako vektory v mnohonásobnom priestore, kde každý rozmer zodpovedá jednému slovu obsiahnutom v dokumente. Za najrelevantnejšie dokumenty sú potom považované tie, ktoré sa čo najviac približujú vektoru dotazu. Namiesto porovnávania veľkosti vektorov sa však porovnáva ich vzájomný uhol.



Obrázek 1.3: Dvojrozmerný Vector space model s troma dokumentmi a jedným dotazom. Dva rozmery prislúchajú termom *science* a *computer*.

Na obrázku 1.3 vidíme Vector Space Model (VSM) pre slová *science* a *computer*. Vektor dotazu q má súradnice (0.45,0.6), $\vec{d_1} = (0.17; 0.88)$, $\vec{d_2} = (0.85; 0.05)$ a $\vec{d_3} = (0.57; 0.48)$. Tieto vektory sú odvodené z váhy jednotlivých termov. Term je jednotné označenie pre slovo alebo frázu. Ako sme už uviedli v sekcií 1.3.3, váhovanie termov len na základe početnosti nemusí dávať dobré výsledky, pretože niektoré slová sa vyskytujú príliš často a „zatienili“ by tak iné, podstatnejšie slová. V našom príklade bude dokument číslo 3 ohodnotený ako najrelevantnejší, pretože jeho vektor je najpodobnejší vektoru dotazu. V ostaných dvoch dokumentoch sa síce spomínajú obe hľadané slová, ale len okrajovo.

Podobnosť vektorov vo VSM určujeme na základe uhlu, ktorý zvierajú. Kosínus tohto uhla potom určuje ohodnotenie dokumentu. Formálne:

$$\cos(\vec{q}, \vec{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \sqrt{\sum_{i=1}^{n} d_i^2}} \tag{1.4}$$

kde $\vec{q}$ a $\vec{d}$ sú $n-$rozmerné vektory.

Kapitola 2

Information Retrieval systémy a ich komponenty

V tejto kapitole si predstavíme niektoré konkrétne riešenia problémov načrtnutých v predošlej kapitole. Budeme tak schopní navrhnúť a neskôr implementovať efektívny IR systém.

2.1 Booleovský model

sme už videli pri invertovanom indexe (v sekcií 1.3.2). Je to najjednoduchší model na ktorom sa dá postaviť IR systém. Dokumenty a dotazy sú reprezentované ako Booleovské výrazy, napr. *information AND retrieval NOT society*. Ak by sme tento dotaz chceli vyhľadávať sekvenčne, najskôr by sme prešli všetky dokumenty a vybrali tie, ktoré obsahujú *information*. Z nich by sme vybrali obsahujúce *retrieval* a nakoniec tie, kde sa slovo *society* nevyskytuje. Takýto postup isto vedie k správnemu výsledku ale nie je riešením. Stačí totiž väčšia kolekcia dokumentov a časová náročnosť bude neúnosná. Problémom je tiež *NOT society*. Toto nie je triviálne. Navyše tento prístup neumožňuje žiadne iné operácie napr. blízkosť slov [31].

Dobrým krokom smerom k riešeniu je použitie document term matice 1.3.3. Je to matica, kde riadky obsahujú termy (slová alebo výrazy), ktoré sa vyskytujú v kolekcií a stĺpce sú tvorené dokumentmi. Prvok matice je potom 0 alebo 1, áno alebo nie a vyjadruje, či sa term v dokumente nachádza alebo nie. Prvok samozrejme nemusíme reprezentovať len binárnou hodnotou ale aj prirodzeným číslom. Vtedy číslo udáva počet výskytov slova v dokumente. Vyhľadávanie potom môžeme vykonať takto: riadok s hľadaným termom reprezentujeme ako vektor a podľa príslušnej Booleovskej operácie z dotazu ju s ním a prípadne aj susedným vykonáme. Pre náš príklad by hľadanie mohlo vyzerať nasledovne: $\overrightarrow{information} = 101100$, $\overrightarrow{retrieval} = 011001$, $\overrightarrow{society} = 110010$ dáva $101100 \wedge 011001 \wedge 001101 = 001000$, teda tretí dokument zľava. Ak si však predstavíme, že máme milión dokumentov, každý nech má 1000 tokenov o priemernej veľkosti 6 bajtov, dostaneme veľkosť kolekcie zhruba 6GB. Nech je v kolekcií 500 000 rôznych termov. Matica by tak mala $500\ 000 * 1\ 000\ 000 = 500 * 10^9$ bitov, čo je zhruba 58GB, teda skoro 10-krát viac než kolekcia. Matica však bude zrejme veľmi riedka – bude obsahovať len veľmi málo 1. Ak teda zaznamenáme len tieto prípady, dostaneme invertovaný index. Vyhľadanie nášho príkladu bude potom vyzerať takto: zo slovníka vyberieme položky *information*, *retrieval* a *society*. Každá položka obsahuje odkaz na zoznam dokumentov v ktorom sa nachádza. Teraz teda musíme prejsť prvé dva zoznamy a vybrať z nich tie dokumenty, ktoré sú v oboch. Posledným krokom vylúčime tie

dokumenty, ktoré sú v zozname termu *society*. Ako však prehľadať dva zoznamy čo najefektívnejšie? Iste pomôže, ak budú zoradené podľa veľkosti. Potom sa totiž môžeme posúvať po oboch zoznamoch zľava doprava a porovnávať, tak ako je to vyjadrené nasledujúcim pseudo-kódom:

Algorithm 2.1 Prienik(p_1, p_2)

```
vysledok ← ∅
while p1 ≠NIL a p2 ≠ NIL do
  if docID(p1) = docID(p2) then
    vysledok → pridaj(p1)
    p1 ← (p1 → next)
    p2 ← (p2 → next)
  else if docID(p1) < docID(p2) then
    p1 ← (p1 → next)
  else
    p2 ← (p2 → next)
  end if
end while
return vysledok
```

Algoritmus začne na začiatku oboch zoznamov. V každom kroku porovnáva čísla dokumentov v zoznamoch. Ak sa zhodujú, našli sme dokument, ktorý je v oboch zoznamoch a môžeme sa posunúť na ďalšie prvky. Inak sa posúvame len v tom, ktorý má menšie číslo dokumentu. Tento algoritmus má lineárnu časovú zložitosť. V prípade, že dotaz obsahuje 3 a viac prvkov môže dôjsť k jeho optimalizácii. *Optimalizácia* v konotáciách IR je proces ktorého výsledkom je poradie v ktorom sa budú jednotlivé podčasti dotazu vykonávať tak, že množstvo vynaloženej práce je minimálne. V prípade Booleovského modelu to znamená určenie poradia spracovania poddotazov s ohľadom na možnú veľkosť výsledku. Predstavme si, že hľadáme *information AND retrieval AND system* a v kolekcií sa slová *information* a *retrieval* vyskytujú pomerne často ale slovo *society* len veľmi zriedka. Ak by sme dotaz spracovávali bez optimalizácie sprava doľava, prienik prvých dvoch termov by bol pravdepodobne dlhý, čo by znamenalo veľa krokov uvedeného algoritmu 2.1. Ak by sme však dotaz optimalizovali a vykonávali ho zľava doprava, tak prvý medzivýsledok – prienik *society* a *retrieval* by už bol menší a tak by bolo potreba menej krokov na vytvorenie prieniku s so zoznamom *information*. Túto myšlienku implementuje algoritmus 2.2.

Algorithm 2.2 Prienik($\{t_1, \ldots, t_n\}$)

```
termy ← ZoradVzostupnePodlaFrekvencie({t1, ..., tn})
vysledok ← postings(termy1)
i ← 2
while i < n a vysledok ≠ NIL do
  vysledok ← Prienik(vysledok, postings(termyn))
  n ← n + 1
end while
return vysledok
```

Predstavme si teraz zložitejší dotaz: *(Bratislava OR Praha) AND (veľkosť OR rozloha)*. Tu postupujeme podobne ako v predošlých príkladoch avšak s tým, že pri optimalizácií

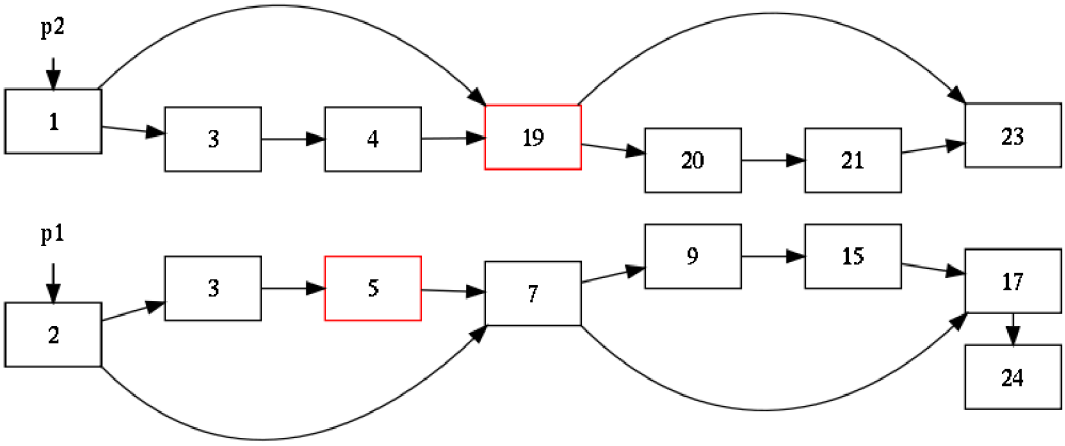
uvažujeme už súčet postings zoznamov pre *Bratislava* a *Praha*, resp. *veľkosť* a *rozloha*. Pri takomto všeobecnom dotaze musíme zlúčiť oba prislúchajúce postings zoznamy a počítať prienik až nad nimi.

Algorithm 2.3 Zlucenie(p_1, p_2)

```
vysledok ← ∅
while p1 ≠NIL a p2 ≠ NIL do
  if docID(p1) = docID(p2) then
    vysledok ← Pridaj(vysledok, p1)
    p1 ← (p1 → next)
    p2 ← (p2 → next)
  else if docID(p1) < docID(p2) then
    vysledok ← Pridaj(vysledok, p1)
    p1 ← (p1 → next)
  else
    vysledok ← Pridaj(vysledok, p2)
    p2 ← (p2 → next)
  end if
end while
if p1 ≠NIL then
  while p1 ≠NIL do
    vysledok ← Pridaj(vysledok, p1)
    p1 ← (p1 → next)
  end while
else if p2 ≠NIL then
  while p2 ≠NIL do
    vysledok ← Pridaj(vysledok, p2)
    p2 ← (p2 → next)
  end while
end if
return vysledok
```

Uvedený algoritmus 2.3 pre zlúčenie 2 zoznamov má na vstupe 2 postings zoznamy z ktorých vytvorý jeden. Implementuje tak logickú funkciu *OR*. Samozrejme, dodržiava vzostupné radenie prvkov. Postupne prechádza oba zoznamy a pridáva aktuálny prvok z toho zoznamu, ktorý má pre aktuálny prvok menšie číslo dokumentu. Posun na ďalší prvok sa deje v tom zozname, z ktorého sme pridali prvok. Ak sú v oboch zoznamoch rovnaké prvky, pridáme len jeden, ale posunieme sa v oboch. A keďže vstupné zoznamy mohli mať vo všeobecnosti rôznu dĺžku, pridáme zostavajúce prvky z dlhšieho zoznamu do výsledku.

Jeden operátor sme však z hľadiska implementácie doposiaľ opomínali. Ide o unárny operátor *NOT*. Jeho aplikáciou získame všetky tie dokumenty, kde sa daný term nenachádza. Ak by sme ich však chceli reprezentovať ako množinový rozdiel $U \setminus postings(term)$ dostali by sme veľkú množinu s ktorou by sme museli pracovať. Omnoho efektívnejšie však bude, ak len upravíme príslušný algoritmus na výpočet prieniku. To však predpokladá dotaz v konjunktívnej normálnej forme. Ak by bol dotaz v disjunktívnej, museli by sme síce upraviť algoritmus 2.3 pre zlúčenie dvoch postings zoznamov, avšak výsledný zoznam by bol zase veľký. Tento problém sa však dá riešiť prevodom dotazu najskôr do konjunktívnej



Obrázek 2.1: Skip pointers. Červené prvky označujú aktívne prvky.

normálnej formy a až potom následným vyjadrovaním.

Uvedené algoritmy sa však dajú ešte zefektívniť tak, že budú mať sublineárnu časovú zložitosť. Postings zoznamy budú pre časté slová veľmi dlhé, takže efektívnosť výpočtu prieniku môže byť obmedzujúca. Ak by sa nám podarilo upraviť prvky v zozname tak, aby ukazovali nie len na následníka, ale aj o pár následníkov ďalej, mohli by sme oba zoznamy prehľadať rýchlejšie. Algoritmus 2.1 má totiž tú vlastnosť, že ak sme v zozname p_1 na dokumente s vyšším číslom dokumentu než v zozname p_2, posúvame sa len v zozname p_2 o toľko prvkov, než sa čísla dokumentov vyrovnajú (vtedy sme našli dokumenty v oboch zoznamoch), alebo sa stanú väčšími. Týchto posunutí však môže byť veľmi veľa a ak by sme sa vedeli posunúť o niekoľko prvkov dopredu, mohli by sme zoznam prehľadať skôr. Tento princíp sa nazýva *skip pointers*. Použiť ich na všetky prvky alebo len niektoré? O koľko prvkov dopredu by mali odkazovať? Odpovede na tieto otázky sú kompromisom medzi počtom preskočených elementov zoznamu a počtom prípadov, kedy môžeme skok vykonať. Pri kratšej vzdialenosti bude týchto záznamov viac, ale budú sa dať častejšie použiť. Kdežto pri dlhšej vzdialenosti bude záznamov menej, a budú sa dať použiť len zriedka. Príklad vidíme na obrázku 2.1.

Obrázok znázorňuje stav uprostred výpočtu. Červenou farbou sú znázornené aktívne prvky. Nasledujúcim krokom bude zrejme posunutie v zozname p_1 na prvok 7. Skip pointer tohto prvku však ukazuje na dokument s číslom 17. Je teda zrejmé, že môžeme prejsť až na tento prvok, pretože v zozname p_2 máme aktívny prvok 19 a je isté, že sa na „klasickej“ ceste (cez prvky 9 a 15) nebude vyskytovať žiadny prienik. Ak pokročíme vo výpočte ešte ďalej, zistíme, že aktívnymi prvkami sú 24 v p_1 a 19 v p_2 s tým, že skip pointer tohto prvku ukazuje na 23. Zase je zrejmé, že môžeme „doskočiť“ až na tento prvok. Z obrázka je zrejmý aj vyššie diskutovaný problém. Menšie vzdialenosti skokov vedú k ich častejšiemu použitiu, dlhšie sú menej používané, zato preskočia veľa prvkov. Algoritmus 2.4 pre výpočet prieniku v takto upravenom zozname je v tak len modifikáciou pôvodného algoritmu 2.1.

Prvky zoznamu sú rozšírené o ukazovateľ *skip*, ktorý buď ukazuje o niekoľko prvkov dopredu, alebo je nulový. Ak by sme ku každému prvku zoznamu pridali ďalší zoznam s pozíciami, na ktorých sa slovo v dokumente vyskytuje, mohli by sme vyhľadávať frázy a blízkosť slov.

Algorithm 2.4 Prienik(p_1, p_2)

```
vysledok ← ∅
while p1 ≠NIL a p2 ≠ NIL do
  if docID(p1) = docID(p2) then
    vysledok → pridaj(p1)
    p1 ← (p1 → next)
    p2 ← (p2 → next)
  else if docID(p1) < docID(p2) then
    if p1 → skip ≠ NIL a docID(p1 → skip) ≤ docID(p2) then
      while p1 → skip ≠ NIL a docID(p1 → skip) ≤ docID(p2) do
        p1 ← (p1 → skip)
      end while
    else
      p1 ← (p1 → next)
    end if
  else
    if p2 → skip ≠ NIL a docID(p1 → skip) ≥ docID(p2) then
      while p2 → skip ≠ NIL a docID(p1 → skip) ≥ docID(p2) do
        p2 ← (p2 → skip)
      end while
    else
      p2 ← (p2 → next)
    end if
  end if
end while
return vysledok
```

2.2 Slovníky a pozičné súbory

Ako sme už videli z príkladu invertovaného indexu (obrázok 1.2), slovník je zoznam slov vyskytujúcich sa v kolekcií a pozičný súbor obsahuje jednosmerne viazané zoznamy, ku každému slovu zo slovníka jeden. Tieto zoznamy obsahujú identifikáciu súborov v ktorých sa príslušné slovo vyskytuje. Ako sme videli v predošlej časti, je veľmi výhodne z hľadiska časovej náročnosti algoritmov, ak sú tieto zoznamy zoradené. V tejto časti si však predstavíme možné rozšírenia tejto entity za účelom presnejšieho hľadania.

Na začiatok si pripomenme, ako sa index vlastne vytvára. Najskôr vyberieme dokumenty, ktoré chceme indexovať. Potom ich otvoríme a získame z nich text (prípadne aj metadáta). Tento potom rozdelíme na sériu tokenov. Z nich vytvoríme invertovaný index, ktorý zlúčime s hlavným indexom. Už tu narážame na niekoľko zásadných problémov. Získavanie textu z dokumentov. Čo je to dokument? Súbor, e-mail, e-mailová príloha? Pre naše potreby postačí, ak si dokument definujeme ako sled znakov.

Každý dokument môže byť v inom kódovaní, niektoré sú zložené s menších dokumentov, ktoré môžu byť každý v inom jazyku (napr. zip súbor anglickej a francúzskej verzie nejakej knihy). No už pred samotným otvorením dokumentu musíme vedieť, v akom formáte je, pretože k rôznym typom sa pristupuje rôzne. Niektoré typy súborov dokonca ani nemusia obsahovať text priamo, ale obsahujú spôsob, ako text znovu reprodukovať. Od typu súboru sa odvíja voľba dekodéra. Dokonca aj pre niektoré čisto textové dokumenty je treba použiť dekodér. Pri HTML súbore chceme odstrániť tagy, nahradiť napr. < za < a podobne. Po otvorení dokumentu potom treba najskôr zistiť kódovanie a jazyk, v ktorom je napísaný. Od toho sa totiž odvíja tokenizácia – delenie textu na tokeny. V slovenčine alebo češtine sa slová oddeľujú medzerami alebo interpunkciou, existujú však jazyky (napr. sinotibetská jazyková rodina) ktorá takto slová nerozdeľuje, čiže treba zvoliť iný prístup. V praxi sa často zisťuje najskôr použité kódovanie, keďže už to môže dosť napovedať o jazyku dokumentu.

Až doteraz sme uvažovali, že dokument je to isté, čo súbor. Je však niekoľko prípadov, kedy je takéto priradenie nevyhovujúce. Mnohé e-mailové klienty ukladajú maily do jedného súboru. V tomto prípade by sme teda chceli, aby bol za dokument považovaný e-mail. Prípadne aj každý súbor vo vnútri zip súboru zaslaného ako príloha k e-mailu. Sú však aj prípady, kedy chceme, aby bol dokument „väčší“ ako jeden súbor. Ak je dokumentácia k programu rozdelená do niekoľko súborov, môžeme chcieť, aby boli tieto brané ako jeden dokument. Veľkosť dokumentu, dokumentová jednotka je tak určená granularitou indexu. Pre kolekciu kníh asi nebudeme chcieť považovať celú knihu za dokument. Môže sa totiž stať, že budeme hľadať nejaké dve slová, z ktorých prvé bude na začiatku knihy a druhé na konci. Lenže s najväčšou pravdepodobnosťou toto nie je to, čo sme hľadali. Zvolíme teda menšiu jednotku. Kapitolu, odstavec. Kde sa však zastaviť? Môžeme dokonca brať každú vetu ako zvlášť dokument. Ak by sme ju naozaj chápali ako dokument, potom by nám zasa unikali vzťahy medzi vetami.

V praxi veľmi často chceme, aby bol index multilinguálny a nezameriaval sa len na jeden typ jazyka. Teda algoritmy, ktoré navrhneme, musia byť buď viacjazyčné (napr. algoritmus na získanie koreňa slova), čo je však veľmi ťažko dosiahnuteľné a častokrát takéto algoritmy ak existujú tak pracujú len v rámci jednej jazykovej podskupiny, kde sú zoskupené veľmi blízke jazyky, napríklad česko-slovenská podskupina západoslovanských jazykov obsahuje češtinu a slovenčinu. Druhou možnosťou je navrhnuť pre každý jazyk, ktorý chceme do systému zahrnúť navrhnúť zvlášť algoritmus.

Z hľadiska implementácie je slovník asociatívne pole. Kľúčom je term, hodnotou potom odkaz na postings zoznam, prípadne iné užitočné informácie - napr. počet výskytov slova

a pod. Toto pole sa implementuje buď ako hashovacia tabuľka alebo samovyvažovací binárny vyhľadávací strom. Hashovacia tabuľka má tú výhodu, že vyhľadávanie v nej je rýchlejšie než v strome: na term aplikujeme hashovaciu funkciu, ktorá nám vráti index do tabuľky. K tomuto indexu potom stačí pripočítať adresu začiatku tabuľky a máme adresu prvku. Môže sa však stať, že hashovacia funkcia vráti pre dva rôzne termy rovnaký hash. V tom prípade je v tabuľke nie priamo prvok ale krátky zoznam (pri použití dobrej hash funkcie), ktorý však rýchlo prehľadáme. Toto pole je statické a nemení svoju veľkosť. Neumožňuje však vyhľadávanie na základe zhodných prefixov, či podobnosti, keďže častokrát 2 podobné reťazce majú odlišný hash. Stromy naproti tomu vyhľadávajú v logaritmickom čase (platí pre vyvážené stromy). Vyvažovanie stromov je však náročná operácia. Kompromisom sú *B-stromy*. Položky stromu, termy, sú organizované na základe prefixovej zhody. Teda po dosiahnutí niektorého uzla vieme, ktorou vetvou sa vydať. Tento prístup navyše umožňuje vyhľadávanie so zástupnými znakmi, *wildcards*. Ľahko totiž vyhľadáme tie podstromy, ktoré začínajú s rovnakým prefixom. Vyhľadávanie slov s rovnakým suffixom je však už skoro nemožné, ak nechceme postupne prechádzať všetky položky v strome. Riešenie spočíva v súčasnom budovaní stromu pre reverzie slov. Napríklad: $ins*$ vráti všetky termy z rozsahu $ins \leq \mathrm{t} < int$ a $*ins$ zasa $sni \leq \mathrm{t} < snj$.

2.2.1 Tokenizácia

Skôr, než sa začneme tokenami a tokenizáciou zaoberať, musíme si definovať kľúčové pojmy. Pod slovom rozumieme vymedzenú sled znakov, tak ako idú v texte. Term je množina slov, ktoré spája určitá vlastnosť: pád, význam, hláskovanie. Token je konkrétna reprezentácia slova alebo termu v texte. Prevod textu na tokeny, *tokenizáca*, však vôbec nie je triviálny problém. Nie len pre to, čo sme načrtli už vyššie, že v nie každom jazyku sa slová oddeľujú medzerou alebo iným znakom, ale aj preto, že aj tu existujú vínimky, kedy práveže chceme dve slová oddelené medzerou interpretovať ako jeden token. Ide o časté dvojslovné spojenia, mená s interpunkciou, či obchodné značky: New York, Karlovy Vary, O'Reilly, C#, Red Hat ai. Niekedy zasa chceme rozdeliť jedno slovo na viacero menších, z ktorých je zložené. Toto je veľmi častý jav v jazykoch bohatých na zložené slová ako napríklad nemčina.

Najtriviálnejší prístup k tokenizácií je oddeľovať slová na základe medzery prípadne interpunkcie. Ale ako vidíme, na príkladoch, tento prístup nedáva dobré výsledky, pretože rozdelí meno O'Reilly na 'O' a 'Reilly'. Ak by potom užívateľ chcel toto meno vyhľadať, tak by neuspel. Jedinou možnosťou by bolo aplikovať tokenizáciu aj na jeho vstup. Vo všeobecnosti však môžeme z jedného slova dostať niekoľko tokenov, ktoré musíme v striktne Booleovských systémoch, resp. v systémoch používajúcich striktný Booleovský dotazovací jazyk pospájať použitím logického operátora AND.

Ďalším z problémov, ktoré pri návrhu tokenizátora treba vyriešiť, je delenie slov. V mnohých dokumentoch sa delia slová, či už na konci riadku, alebo v prípade ustálených slovných spojení aj inde, pomlčkou. Avšak v prvom prípade chceme danú pomlčku odstrániť, pretože nie je súčasťou slova, ale len naznačuje jeho delenie. V druhom prípade je pomlčka použitá ako oddeľovač slov, a nie je isté, či je vhodné ju odstrániť. Ak by sme ju v slove Československo odstránili, zostali by nám 2 tokeny a ak by následne užívateľ hľadal dokumenty obsahujúce Československo, čo je jeden token, nenastala by zhoda a dokument by tak nebol vybraný.

2.2.2 Stop slová

Niekedy je výhodné najčastejšie slová z jazyka vôbec nezahŕňať do slovníka. Vzhľadom na ich vysokú frekvenciu by aj príslušné postings zoznamy boli veľmi dlhé, čo by v konečnom dôsledku znamenalo veľa zabranej pamäte a dlhší čas trvania algoritmov na výpočet prieniku postings zoznamov. Navyše, prínos týchto slov k informačnej hodnote je minimálny. Tieto slová sa nazývajú *stop slová*. Ich výber prebieha nasledovne: Zoradíme slová podľa frekvencie výskytu v nejakej kolekcií (čím väčšia a všeobecnejšie zameraná, tým lepšie) a potom ručne vyberáme tie slová, počnúc od najfrekventovanejších, ktoré nie sú sémanticky významné. Takto získame zoznam stop slov, *stoplist*. Príklad stoplistu pre angličtinu môžeme vidieť na tabuľke 2.1.

a	an	and	are	as	at	be	by	for	from
has	he	in	is	it	its	of	on	that	the
to	was	were	will	with					

Tabulka 2.1: Príklad stop slov pre angličtinu [42].

Použitie stoplistu síce značne znižuje veľkosť pozičného súboru, ale znemožňuje hľadanie fráz obsahujúcich čo i len jedno stop slovo. Ak budeme hľadať pieseň *„Let it Be“* tak po vypustení stop slov z tabuľky 2.1 nám zostane len *„Let“*, čo bude zodpovedať omnoho viac dokumentom. Postupom času, ako klesala cena pevných diskov, operačných pamätí a celkovo cena za uložený bajt, zmenšoval sa aj ich počet. Z pôvodných skoro až 200 slov, na 7–12 a dnes sa už vôbec nepoužíva. Práve preto, že situácia je dnes opačná, veľké postings zoznamy už nie sú problém a dnešné moderné procesory ich dokážu prejsť v zlomku sekundy. Zato vyhľadávanie fráz je štandard, ktorý sa snažia poskytnúť všetky vyhľadávače.

2.2.3 Normalizácia

Uvedené problémy smerujú k procesu, nazývanému *normalizácia*. Jej zámerom je určiť ekvivalenčné triedy pre jednotlivé tokeny takže sa odstránia (v ideálnom prípade) problémy pomlčiek, veľkých a malých písmen, množného čísla, skratiek bez bodiek (USA vs. U.S.A.) a i. Zo zadaného tokenu tak získame term, ktorý vložíme do indexu.

Alternatívnym prístupom je *asymetrická expanzia*, ktorá pre zadané slovo vytvorí množinu všetkých možných realizácií slov. Napr. pre slovo window dostaneme triedu {window, windows}, pre windows {windows, Windows, window}. Vyhodnocovanie užívateľom zadaného dotazu potom prebieha tak, že zo získanej množiny postupne hľadáme slová v slovníku, k nim prislúchajúce postings zoznamy, ktoré spojíme do jedného.

Pri normalizácií už môžeme uvažovať aj synonymá. Jeden zo spôsobov je, že k danému termu získame množinu synoným, a ku každému prvku množiny potom vytvoríme príslušný záznam v postings zozname. V jednom zozname tak nie sú obsiahnuté len dokumenty obsahujúce daný term, ale aj dokumenty, ktoré obsahujú niektorý zo synoným.

2.2.4 Stemming a Lematizácia

V predošlej kapitole sme spomenuli tieto dva termíny. Stemming je proces odstraňovania prípon a predpôn s nádejou, že dostaneme koreň slova. Nie je však zaručené, že koreň získame a častokrát výsledný stem nemá so skutočným koreňom slova nič spoločné. Presnejšia definícia by bola: Stem je unikátny token, nie nevyhnutne zmysluplný v ľudskom jazyku, ktorý môže poslúžiť ako jedinečné označenie skupiny vyskloňovaných foriem slov namiesto

lemy. Takto to funguje celkom dobre v analytických jazykoch, ktoré nemajú skloňovanie, napr. angličtina. V jazykoch flektívnych, ktoré vyjadrujú gramatické funkcie pomocou ohýbania slov, t.j. pridávaním predpon a prípon fungujú horšie. Najťažšie na tento typ spracovania sú jazyky introflektívne, kde sa pri skloňovaní mení koreň slova. Príkladom takéhoto jazyka je hebrejčina. Lematizácia je presnejšia a prevádza slová na ich základný tvar. Je však omnoho komplexnejšia a náročnejšia na implementáciu. Musí totiž brať do úvahy nie len samotný token, ale častokrát aj kontext v ktorom sa vyskytuje. Existujú totiž slová, ktoré keď vyskloňujeme, tak jeden z tvarov je totožný s tvarom iného slova. V angličtine je to napríklad *saw*. Bez kontextu nevieme určiť, či ide o minulý čas slovesa *see*, alebo o podstatné meno označujúce motorovú pílu. V slovenčine môže byť príkladom slovo *nos*, kde nevieme, či ide o časť ľudského tela, alebo rozkazovací spôsob slovesa *nosiť*.

Stemmer býva implementovaný jedným z nasledujúcich prístupov alebo ich kombináciou [30]:

slovníkový algoritmus Stemmer používa rozsiahly slovník, v ktorom vyhľadáva rozvinutý tvar slova a k nemu prislúchajúci základný tvar. Táto metóda je jedna z najpresnejších, avšak je časovo a pamäťovo veľmi náročná (slovenský jazyk obsahuje vyše 240 000 slov [20]). Problémom je aj takýto slovník vytvoriť a prípadne aj aktualizovať o nové slová a ich príslušné formy. Oproti zvyšným metódam má však tú výhodu, že dokáže správne odhaliť a spracovať výnimky existujúce pri niektorých slovách.

porovnávací algoritmus Je obdobou predošlého algoritmu. Namiesto celých slov uchováva len ich základ, ktorý je spoločný pre všetky tvary slova. Napríklad, pre slová „ruka“, „ruky“, „rukami“ sa zdá byť základom „ruk“, čo však ani nie slovo. Lenže tento sa môže vyskytovať aj v iných slovách. Preto musí byť zabezpečené sériou pravidiel, aby sa vybralo zo slovníka to správne slovo. Výhodou je podstatne menší slovník a teda aj čas alebo pamäť potrebná na vyhľadávanie.

odstraňovanie prípon Táto metóda vychádza z gramatiky daného jazyka. Postupne odstraňuje prípony a prípadne ich nahrádza základnou príponou, napr. v slove „kolesami“ je príponou „ami“, ktorú možno nahradiť za príponu „o“.

odstraňovanie predpôn a prípon Vychádza z predošlého prístupu a aplikuje pravidlá nie len na suffixy ale aj prefixy. Je teda mocnejší, napr. dokáže zo slova „nepísali“ odstrániť nie len príponu „ali“ a nahradiť ju „ť“, ale aj predponu „ne“, takže dostaneme „písať“. Pravidlá ktoré uplatňujeme však musíme aplikovať s veľkou obozretnosťou a najskôr musíme zistiť, ktorá časť slova je prefix a ktorá suffix.

stochastické algoritmy Sú príkladom strojového učenia. Najskôr sa algoritmus naučí pravidlá na množine testovacích dát (rozvinuté tvary a k nim prislúchajúce základné tvary), čoho výsledkom je pravdepodobnostný model – séria pravidiel. V normálnej prevádzke sa potom vyberá to pravidlo, ktoré má pre dané slovo najväčšiu pravdepodobnosť. Nevýhodou je značná závislosť na trénovacej množine.

Najznámejším algoritmom pre stemming a anglický jazyk je *Porter stemmer* [44]. Algoritmus pozostáva zo 71 pravidiel, rozdelených do 5 krokov aplikovaných sekvenčne. V každom kroku sa aplikuje najdlhšie možné pravidlo – to s najdlhším zhodným suffixom. Pravidlá sú v tvare `(podmienka) S1 -> S2`, ktoré hovorí, ak je splnená podmienka, tak suffix reťazca S1 prepíš na S2. Podmienkou môže byť napr:

- *S – stem končí na S

- *v* – stem obsahuje samohlásku (vowel)
- *d – stem končí na dve rovnake spoluhlásky (napr. -TT, -SS)
- *o – stem končí na spoluhlásku, samohlásku a spoluhlásku, ktorá sa nerovná W, X alebo Y (napr. -WIL)

Aby sa predišlo aplikovaniu skracujúcich pravidiel na už aj tak krátke slová, zavádza sa miera slov *m*. Táto udáva počet slabík slova. Napr. pravidlo `'(m > 1) EMENT ->'` sa aplikuje na slovo REPLACEMENT a skráti ho na REPLAC, ale na slovo CEMENT sa toto pravidlo kvôli podmienke aplikovať nedá. Podmienky sa navyše dajú spájať logickými spojkami.

Príklad pravidiel Porterovho stemmera môžeme vidieť v tabuľke 2.2.

(m>0) EED	→ EE	feed → feed
		agreed → agree
(*v*) ED	→	plastered → plaster
		bled → bled
(*v*) ING	→	motoring → motor
		sing → sing

Tabulka 2.2: Príklad pravidiel z Porterovho stemmera. Vľavo sú pravidlá s podmienkami, vpravo príklady aplikovania, resp. neaplikovania pravidla v dôsledku nesplnenia podmienky.

Slovo GENERALIZATIONS je v prvom kroku zredukované na GENERALIZATION, v druhom na GENERALIZE v treťom na GENERAL, vo štvrtom na GENER. Piaty krok sa síce aplikuje tiež, ale keďže slovo nespĺňa podmienku žiadneho pravidla, ani jedno sa neuplatní.

Tento stemmer však nepracuje dobre pre slovenský alebo český jazyk. Oba patria medzi jazyky, ktoré majú bohatú morfológiu. Keď si predstavíme slovo „rada“, ktorého koreň je „rad“ a pri zanedbaní diakritiky zahŕňa tieto slova: rada – podstatné meno, orgán; rád – podstatné meno, vyznamenanie; rád – sloveso; rad – podstatné meno, zoradenie; rada – podstatné meno, ponaučenie. Zaujímavou je z tohto pohľadu práca Lea Galamboša [26], ktorý vypracoval stemmer vhodný pre slovanské jazyky na základe ktorého bol vyvinutý stemmer pre poľský jazyk [23].

Na Univerzite Pavla Jozefa Šafárika v Košiciach vznikol lematizátor pre slovenský jazyk [35]. Pokúsili sa odvodiť podobné pravidlá ako v Porterovom algoritme, avšak ukázalo sa, že je to úloha zložitá a v mnohých prípadoch až nemožná. Táto skutočnosť je daná tým, čo sme spomínali vyššie – slovenčina patrí medzi flexívne jazyky. Preto použili iný prístup, ktorý odvodili z pozorovania, že ohýbanie slova závisí primárne od jeho ukončenia. Algoritmus predpokladá existenciu slovníka so základným a rozvinutým tvarom slova. Vstupom je podstatné meno v ľubovolnom tvare, výstupom zoznam možných základných tvarov zdaného slova. Algoritmus pracuje v troch fázach. V prvej hľadá v zozname vyskloňovaných tvarov zodpovedajúcu predlohu, t.j. slov s rovnakou, najdlhšou možnou koncovkou. V druhom kroku vytvorí možný základný tvar slova na základe „podvojnej výmeny“. V poslednej fáze overí, že vytvorený základný tvar sa nachádza v slovníku. Celý algoritmus si ukážeme na príklade slova „ponúk“. Predpokladajme, že v slovníku už existuje slovo „ruka“ vrátane jeho možných tvarov.

1. slovo: X = „ponúk“

2. jedna z nájdených predlôh: Y = „rúk“
3. spoločný koniec: K = „úk“
4. začiatok predlohy: Y’ = „r“ (Y = Y’ + K)
5. začiatok slova: X’ = „pon“ (X = X’ + K)
6. základný tvar predlohy: Y = „ruka“
7. koniec základného tvaru predlohy: K’ = „uka“ (Y = Y’ + K’)
8. základný tvar slova: X = X’ + K’ = „ponuka“
9. overenie existencie slova X v zozname základných tvarov

Algoritmus bol implementovaný a odskúšaný. Testy ukázali až 90% úspešnosť. Treba si však uvedomiť, že vzhľadom na nejednoznačnosť jazyka demonštrovanú vyššie, nie je možné dosiahnuť 100% úspešnosť.

Treba však povedať, že zatiaľ čo pri neiktorých druhoch dotazov je stemming výhodou (napr. keď budeme hľadať *ekonomický* nájdeme aj *ekonomická*), pri iných je nevýhodou (ak by sme hľadali presnú frázu, avšak stemming ju naruší).

2.2.5 Korekcia pravopisu

Rozlišujeme dve úrovne korekcie: dokumenty a užívateľské dotazy. Nejde tu však o interaktívnu korekciu, tak ako sme možno zvyknutý z kancelárskych textových editorov. Rovnako existujú dva prístupy: izolovaná korekcia každého slova zvlášť alebo s ohľadom na kontext. Korekciu pravopisu v dokumentoch využívame hlavne pri prevode zoskenovaného dokumentu na text.

Izolovaná korekcia predpokladá existenciu slovníka správnych tvarov slov a metriky, ktorá nám povie, ako veľmi sa dve slová líšia. Výsledkom korekcie je potom to slovo, s najnižšou vzdialenosťou. Ako slovník môžeme použiť zoznam naindexovaných termov, čo však so sebou nesie riziko, že ak už v indexe máme slovo s pravopisnou chybou, táto sa bude ďalej propagovať aj na gramaticky správne slová, alebo môžeme použiť štandardné slovníky (napr. Krátky slovník slovenského jazyka), prípadne špecializované z oblasti budúceho pôsobenia IR systému. Používanými vzdialenosťami sú:

edit distance určuje počet operácií vyžadovaných na pretransformovanie jedného reťazca na druhý

Levenshteinova vzdialenosť je počet operácií potrebných na transformáciu jedného reťazca na druhý, pričom povolené operácie sú vkladanie, mazanie a nahradenie jedného znaku [27].

Damerau–Levenshteinova vzdialenosť vychádza z predošlej a pridáva štvrtú operáciu: výmenu dvoch znakov.

Alternatívnym prístupom je používanie *n-gramov*, t.j. série znakov dĺžky n. Ak $n = 2$ hovoríme o bigramoch, $n = 3$ o trigramoch. Napr. slovo „automobil“ obsahuje tieto bigramy: au, ut, to, om, mo, ob, bi, il; trigramy: aut, uto, tom, omo, mob, obi, bil. Paralelne s indexom termov, vytvárame aj index n-gramov. Každý n-gram obsahuje zoznam slov, v ktorých sa vyskytuje. Slovo, v ktorom chceme opraviť gramatiku rozdelíme na n-gramy. K získaným nájdeme príslušné zoznamy a za správne slovo budeme považovať to, ktoré sa vyskytuje vo všetkých.

2.3 Vytváranie indexu

V zásade existujú dva hlavné spôsoby vytvárania indexu. Blokové so zlučovaním a radením (Block Sort-Based Indexing)[38] a jednoprechodové blokové indexovanie bez radenia (Single Pass In-Memory Indexing)[42].

Je zrejmé, že s rastúcou veľkosťou zbierkou, kolekciou dát, rastie aj veľkosť indexu. Nie je ničím výnimočným, keď je v organizáciach vyhradený jeden server len na vytváranie a obsluhu indexu. Tieto počítače majú často rádovo gigabajty operačnej pamäte, stovky GB až desiatky TB diskového priestoru a výkonné procesory. Alternatívou je prístup spoločnosti Google, ktorá sa práve naopak, snaží použiť viac slabých počítačov (častokrát ide o klasické osobné počítače) a rozložiť tak záťaž. Tento prístup je navyše viac odolný voči výpadkom.

Aby sme mohli navrhnúť efektívne indexovacie algoritmy musíme brať do úvahy hardvér a jeho obmedzenia. Prístup k informácií do operačnej pamäte trvá typicky niekoľko desiatok až stoviek hodinových cyklov procesora, získanie informácie z disku trvá rádovo desiatky miliónov [36]. Iste, moderné operačné systémy dokážu rôznymi technikami tieto časy znížiť (napr. cachovaním), no aj tak musíme tento fakt brať do úvahy. Zatiaľ je totiž stále zoradenie poľa uloženého v operačnej pamäti rýchlejšie než zoradenie toho istého poľa uloženého na disku. Toto je dané mimo iného aj tým, že čítanie alebo zapisovanie súvislej oblasti na disku je rýchlejšie než čítanie alebo zapisovanie menších častí, hoc by vo výsledku išlo o rovnaké miesta na disku. Istú dobu totiž trvá samotné „naštartovanie“ prenosu dát na disk: musia sa vystaviť hlavičky, počkať na dotočenie disku, atď. Preto sa budeme snažiť, uchovávať čo najviac dát v pamäti RAM a zapisovať na disk čo najviac súvislé dáta.

2.3.1 BSBI

Predstavuje základný prístup k problematike. K dokumentu (to, že dokument nemusí byť to isté čo súbor sme vysvetlili už tu 2.2) vytvoríme slovník a príslušné postings zoznamy. Slovník potom zoradíme podľa abecedy. Tento dočasný index je teraz potrebné zlúčiť s výsledným indexom. Vyhľadáme teda zhodné termy v oboch slovníkoch, zlúčíme príslušné postings zoznamy a aktualizujeme atribúty daných termov (napr. počet výskytov termu).

Problémom tohto prístupu je, že pri veľkých kolekciách sa index nemusí vmestiť do RAM pamäte (nie dočasný, ale výsledný). Zlučovanie dvoch indexov priamo na disku rozhodne nie je efektívnym riešením, z dôvodov popísaných vyššie. Riešením však je, rozdeliť index na menšie časti – bloky – ktoré sa už do operačnej pamäte vôjdú. Tak môžeme zlučovať indexy po častiach. Na konci zasa vzniknuté bloky zlúčime.

V tomto bode je dôležité si uvedomiť, ako sú postings záznamy reprezentované na disku. V pamäti je to jednoduchý zoznam, kde každý element (až na posledný) obsahuje ukazovateľ s adresou nasledovníka. Lenže v súbore je tento prístup veľmi nevýhodný. Iste, aj v tomto prípade môžu elementy obsahovať ukazovateľ na nasledujúci prvok. Otázkou však je, čo by bolo adresou, aký význam jej prisúdiť. Ak by ňou bola pozícia od začiatku súboru, pridaním prvku, by sa posunuli offsety zvyšných prvkov, ktoré by bolo treba aktualizovať. Keby sme aj pridávali prvky na koniec súboru, takže by sme museli aktualizovať len jeden prvok – priameho predchodcu – problém by bol s následným čítaním zoznamu, kedy by sme museli „skákať“ po celom súbore.

Omnoho efektívnejšia sa tak javí cesta, keď ukladáme elementy bez nejakej priamej náväznosti, ktorú potom v prípade potreby kedykoľvek dokážeme vytvoriť. Aby sme tak mohli učiniť, potrebujeme v zásade 2 informácie: term a dokument. Vzhľadom na snahu mať prvky postings zoznamov čo najmenšie, pretože ich je obrovské množstvo, nahrádzame

identifikátory odkazmi *termID* a *docID*.

Algoritmus pre tento typ vytvárania indexu je 2.5.

Algorithm 2.5 BSBI

$i \leftarrow 0$
while (neboli naindexované všetky dokumenty) **do**
  $i \leftarrow i + 1$
  $blok \leftarrow ZiskajDalsiBlok()$
  $BSBI(blok)$
  $VytvorDocasnySuborNaDisku(block, f_i)$
end while
$ZlucBloky(f_1, \ldots, f_i)$

Funkcia $ZiskajDalsiBlok()$ získa z indexu uloženého na disku ďalší blok. Funkcia $BSBI(blok)$ potom nad ním a aktuálne spracovávaným dokumentom vykoná všetky kroky spomínané vyššie. Takto upravený blok potom uložíme na disk ($VytvorDocasnySuborNaDisku(block, f_i)$), ktoré v poslednom kroku zlúčime.

2.3.2 SPIMI

Predchádzajúci algoritmus má skvelú škálovateľnosť, veľkosť bloku môžeme voliť. Potrebuje však štruktúru pre mapovanie termID na term. Pri niektorých kolekciách sa však ani táto štruktúra nemusí vmestiť do operačnej pamäte (najmä, ak sa počet termov pohybuje v rádoch stoviek miliónov, čo však pri multilinguálnych IR systémoch nie je nič výnimočné). Tento problém rieši SPIMI. Vychádza z 2 myšlienok. Prvá, každý blok bude mať svoj vlastný slovník. Druhá, nezoraďujeme postings zoznamy v súbore. Vlastne tak vytvárame viacero menších indexov, ktoré môžeme neskôr zlúčiť. Kompletný algoritmus potom vyzerá nasledovne 2.6

Algorithm 2.6 SPIMI

$subor \leftarrow NovySubor()$
$slovnik \leftarrow NovySubor()$
while je voľná pamäť **do**
  $token \leftarrow dalsiToken()$
  if $term(token) \in slovnik$ **then**
    $postings_zoznam \leftarrow Postings(slovnik, term(token))$
  else
    $postings_zoznam \leftarrow PridajDoSlovnika(slovnik, term(token))$
  end if
  $PridajDoPostingsZoznamu(postings_zoznam, docID(token))$
end while
$ZoradSlovnik(slovnik)$
$ZapisNaDisk(slovnik, subor)$

Tento prístup umožňuje lepšiu kompresiu indexu.

2.3.3 Dynamické indexy

Až doteraz sme neuvažovali dynamickosť obsahu kolekcií. Dokumenty sú často menené, pridávané, odoberané. Najjednoduchsím prístupom je: udržiavať jeden hlavný index, nové dokumenty pôjdu do malého indexu v RAM pamäti. Vyhľadávame v oboch a zlučujeme výsledky. V pravidelných intervaloch potom pridávame malý index do hlavného. Mazanie dokumentov potom reprezentujeme bitovým vektorom a filtrujeme vrátené dokumenty týmto vektorom. Nevýhodou je, že pri zlučovaní dokumentu, ktoré musí byť časté, aby sme zbytočne nezaberali miesto v operačnej pamäti, je výkon systému slabý. Ak by sme každý postings zoznam uchovávali v zvlášť súbore, zlučovanie dvoch zoznamov by nebolo náročné - išlo by len o pridávanie na koniec súboru. Lenže tak by sme dostali veľmi veľa malých súborov, čím by dochádzalo k fragmentácií, čo je neefektívne. Tento problém elegantne rieši algoritmus logaritmického zlúčenia. Spravujeme sériu indexov, následník je dvakrát tak veľký ako predchodca. Najmenší index, udržiavame v pamäti a ak prekročí istú veľkosť, zapíšeme ho na disk ako I_0, prípadne zlúčime s I_0. Ak je tento tiež príliš veľký, nadbytočné prvky presunieme do I_1 atď.

2.4 Kompresia indexu

Táto problematika sa dotýka nie len IR systémov, ale vo všeobecnosti všetkých systémov využívajúcich potenciálne veľké indexy, ako napr. databázové systémy. Nosnou myšlienkou je fakt, že čítanie komprimovaných dát z disku a ich následná dekomprimácia je rýchlejšia než čítanie nekomprimovaných dát priamo. Aby toto tvrdenie platilo, musíme vybrať také komprimačné, resp. dekomprimačné algoritmy, ktoré sú rýchle. Dôvodov, pre ktoré použiť kompresiu v IR systémoch je však viac. Ak uvážime, že slovník je najpristupovanejšia dátová štruktúra celého systému (v podstate pri každom vyčíslovaní dotazu k nej pristupujeme), je veľmi výhodné ju mať v rýchlej operačnej pamäti. Lenže častokrát je slovník veľký, aby sa do nej vošiel. A tak je kompresia logickou voľbou. Druhou motiváciou môže byť veľkosť zabratého miesta na disku. Súbory s postings zoznamami môžu byť extrémne veľké, takže aj tu by sme ušetrili miesto. Veľkosť indexu je navyše kritická na desktopoch, kde sa musíme snažiť, aby bol index čo najmenší a užívateľ tak nepocítil nepríjemne prítomnosť IR systému. Hoci sme si to doteraz ani neuvedomovali, už nejakú kompresiu sme robili. Stemming, stop slová, transformovanie na jednu veľkosť písmen, vynechávanie čísel – toto všetko sú príklady pre stratovú kompresiu. Pri nej vynechávame isté informácie. Ďalej si však predvedieme bezstratovú komprimáciu.

2.4.1 Heapsov zákon

Aby sme vedeli zvoliť vhodný komprimačný algoritmus, musíme vedieť aký veľký slovník alebo postings súbor je alebo môže byť. Exituje nejaká horná hranica? Pre príklad, česká abeceda má 42 písmen, slovenská 46. A to sú len písmená jednej veľkosti. Pre jednoduchosť teraz zanedbajme fakt, že niekoľko málo písmen sa nevyskytuje na začiatku slov. V čestine je tak počet všetkých možných slov dĺžky 10: $(2*42)^{10} \approx 10^{19}$ v slovenčine $(2*46)^{10} \approx 10^{19}$. V angličtine, resp. nemčine, kde sú dlhšie slová je tento horný odhad ešte rádovo vyšší a to aj napriek faktu, že majú menšiu abecedu. Slovník teda bude rásť spolu s rastúcou kolekciou. Ako veľmi? Odpoveď ponúka práve Heapsov zákon. Je to empirický zákon. Nekladie si teda veľké požiadavky na presnosť. Je formula:

$$V(n) \;=\; K\, n^{\beta} \tag{2.1}$$

kde $V(n)$ je veľkosť slovníka pre kolekciu o počte n tokenov. K a β sú empricky určené konštanty. Typicky: $10 \leq K \leq 100$ a $\beta \approx 0.5$.

2.4.2 Zipfsov zákon

Pri navrhovaní vhodného algoritmu pre kompresiu je vhodné tiež vedieť rozloženie termov medzi dokumentmi. Umožní nám to charakterizovať požiadavky na algoritmus. Zipfsov zákon hovorí, že frekvencia nejakého termu je nepriamo úmerná k jeho poradiu vo frekvenčnej tabuľke termov. Teda najčastejšie sa vyskytujúci term t_1 sa bude vyskytovať dvakrát častejšie než druhý najčastejší term t_2. Formálne:

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} (1/n^s)} \tag{2.2}$$

kde N je počet prvkov, k je ich ohodnotenie a s je exponent charakterizujúci rozloženie. Často sa však udáva v zjednodušenej verzií:

$$f \propto \frac{1}{r} \tag{2.3}$$

Zipfsov zákon študoval aj Mandelbrot, ktorý ho spresnil a odvodil všeobecnejší vzťah medzi poradím a frekvenciou:

$$f = P(r + \rho)^{-B} \tag{2.4}$$

kde P,B a ρ sú parametre textu merajúce bohatosť slovnej zásoby a pre špeciálny prípad $B = 1$ a $\rho = 0$ ide o verziu Zipfsovho zákona.

2.4.3 Kompresia slovníka

Slovník je oproti postings súboru veľmi malý a keďže je často pristupovanou dátovou štruktúrou, chcem ho uchovať v pamäti. No pre dôvody uvedené vyššie ho chceme komprimovať. Položkou v slovníku je trojica (term, frekvencia výskytu, odkaz na postings zoznam). Ako veľká je? Predpokladajme 32 bitovú architektúru. Odkaz bude 4 bajtový. Rovnako na uloženie počtu výskytov bude stačiť 4 bajtové číslo (bezznamienkové). Koľko miesta však treba na uloženie termu? Ak by sme použili pevnú dĺžku, pri krátkych termoch by sme plytvali miestom, a dlhšie by sa nám nezmestili. Elegantným riešením je mať slovník ako jeden dlhý reťazec a v trojici nahradiť položku term odkazom na term. Jeden prvok tak odkazuje na začiatok termu v reťazci. Term potom končí tam, kde začína druhý term – odkaz na začiatok ďalšieho termu je v nasledujúcom prvku (trojici) slovníka. Ako veľký však bude tento odkaz? Nemusí byť nutne štvorbajtový - nejde predsa o odkaz do pamäti, ale len o offset od začiatku reťazca. Je veľkosť tak závisí od veľkosti reťazca, t.j. maximálneho počtu termov, ktoré chceme v slovníku uchovať. Napríklad pre počet 400 000 slov s priemernou dĺžkou 8 znakov je to $\log_2(400\ 000 * 8) \approx 22$ bitov, teda 3 bajty. No nemať v trojici term a ani žiadnu jeho časť by síce zmenšilo slovník, ale zvýšilo čas potrebný na nájdenie konkrétneho termu. Chýba totiž systém na prevod term → položka slovníka. Preto rozšírime trojicu o jeden prvok: prvých n znakov termu (často volíme n rovné priemernému počtu znakov slov v jazyku). Ono, ako sme videli vyššie - slovník býva implementovaný ako asociatívne pole alebo strom, a bez toho, aby sme vedeli k termu prisúdiť prvok slovníka, by sa tieto štruktúry len veľmi ťažko udržiavali (napr. vyvažovanie stromu).

Túto metódu možno ešte upraviť tak, že odkaz do reťazca budú obsahovať len prvé prvky z rovnakej „triedy“, teda skupiny termov, ktoré majú rovnaký začiatok. Reťazec

bude potom nutné upraviť tak, aby obsahoval aj dĺžky termov. Teda pred samotný term vložiť aj jeho dĺžku. Ak má táto trieda k prvkov, ušetríme $k-1$ odkazov, ale do reťazca vložíme k bajtov nesúcich informáciu o dĺžke.

Rovnako možno komprimovať aj samotný reťazec termov. Predpokladáme, že dva po sebe idúce termy majú rovnaký začiatok (slovník je usporiadaný). Obsahuje tak redundantné informácie, ktoré možno vypustiť. Uchováme začiatok, ale potom si už len pamätáme meniace sa konce. Situáciu znázorňuje obrázok 2.2.

8automata**8**automate**9**automatic**10**automation

$\Downarrow$

8automat⋆a**1**e**2**ic**3**ion

Obrázek 2.2: Ukážka kompresie slovníka

2.4.4 Kompresia postings zoznamu

Až doteraz sme uvažovali položky v postings zozname nesúce číslo dokumentu docID za štvorbajtové. Toto však môže byť mrhanie miestom. Ak máme rádovo milión dokumentov, nepotrebujeme plných 32 bitov, ale môže stačiť 24. Naopak, pri veľkých kolekciách zasa potrebujeme viac bitov.

Všimnime si, že postings zoznamy sú vlastne vzostupné rady čísel. Tento rad sa dá reprezentovať ako prvý prvok a diferencie. Napr. 1653, 1688, 1720, 1721, 1763 → 1653, 35, 32, 32, 1, 42. Na uchovanie týchto rozdielov už potrebujeme menej bitov. V tomto príklade 6 oproti 11 pôvodným. Navyše, tieto rozdiely tiež môžeme kódovať premenlivým počtom bitov. Príkladom môžú byť nasledujúce algoritmy 2.7,2.8 a 2.9.

Algorithm 2.7 Zakóduj($\{n_1, \ldots, n_k\}$)

```
bytestream ← ⟨⟩
for i ← 1 to k do
  bytes ← ZakodujCislo(n_i)
  bytestream ← PridajNaKoniec(bytestream, bytes)
  i ← i + 1
  return bytestream
end for
```

Algorithm 2.8 ZakodujCislo(n)

```
bytes ← ⟨⟩
while true do
  PridajNaZaciatok(bytes, n mod 128)
  if n < 128 then
    break
  end if
  n ← n div 128
end while
bytes[Dlzka(bytes)] ∧ = 128
return bytes
```

Algorithm 2.9 Dekóduj(*bytestream*)

```
cisla ← ⟨⟩
n ← 0
for i ← 1 to Dlzka(bytestream) do
  if bytestream[i] < 128 then
    n ← 128 * n + bytestream[i]
  else
    n ← 128 * n + (bytestream[i] − 128)
    PridajNaKoniec(cisla, n)
    n ← 0
  end if
end for
```

Algoritmy využívajú prvý bit na poznačenie informácie, či k číslu patrí aj nasledujúci bajt alebo nie. Čiže zo zadaného postings zoznamu so relatívnymi prírastkami, algoritmus 2.7 dekóduje číslo po čísle užívaním 2.8. Ten vezme číslo, a prekódováva ho na 7 bitov s tým, že do prvého bitu poznamenáva informáciu o tom, či je číslo rozprostrené aj do tohto bajtu, alebo je tento posledný, resp. prvý. Algoritmus 2.9 potom pracuje presne inverzne. Možno však použiť akékoľvek iné kódovanie.

2.5 Ohodnocovanie výsledkov dotazu

Až doposiaľ sme uvažovali dotazy ako booleovské a dokumenty im buď vyhovovali alebo nie. Tento prístup je dobrý pre expertov, ktorí približne poznajú dokumenty v kolekcií, alebo pre aplikácie, ktoré dokážu veľmi rýchlo spracovať tisícky dokumentov. Avšak pre väčšinu užívateľov je tento model nevyhovujúci. Bežný užívateľ nemusí vedieť písať dotazy v Booleovej logike, ale keby aj vedel, nemusí byť ochotný, pretože to predsa len predstavuje námahu. Booleovské modely sa totiž vyznačujú okrem iného tým, že na dotaz vrátia veľa dokumentov alebo veľmi málo. Lepším prístupom teda je k zadanému dotazu ohodnotiť každý dokument číslom, vyjadrujúcim jeho relevanciu. Dokumenty potom zoradíme zostupne, teda prvých, povedzme, 10 dokumentov je najrelevantnejších a užívateľ tak nájde hľadanú informáciu skôr. Základnou premisou tohto prístupu je, že vieme ohodnocovať dokumenty vzhľadom na zadaný dotaz, a vieme ich ohodntiť rýchlo.

Čo môže byť hľadanou ohodnocovacou funkciou? V časti o Document term matici (1.3.3), sme videli, že nestačí počítať obyčajný počet výskytov slov. Tento prístup je použiteľný pre jednoslovné dotazy. Čím viac sa hľadané slovo v dokumente nachádza, tým vyššie bude mať dokument výsledné skóre. Ale už pri dvojslovných dotazoch je táto metóda nevyhovujúca, pretože sa v dotaze môže vyskytovať často používané slovo (napr. v angličtine „the“, v slovenčine alebo češtine „a“ alebo „sa“), ktoré môže zvýhodniť nesprávné dokumenty a potlačiť váhu zriedkavého slova, ktoré má však väčšiu informačnú hodnotu.

Frekvencia termu t v dokumente d sa označuje $\mathrm{tf}_{t,d}$ a vyjadruje, koľkokrát sa term v dokumente nachádza. Teraz však neuvažujeme poradie slov v dokumente. Zrejme, samotná hodnota $\mathrm{tf}_{t,d}$ nie je to, čo chceme. Päťnásobne väčšia $\mathrm{tf}_{t,d}$ nemusí znamenať aj paťnásobne väčšie ohodnotenie dokumentu. Relevancie nerastie priamo úmerne s frekvenciou termu.

Môžeme použiť logaritmické váhovanie. Váha termu t v dokumente d je:

$$w_{t,d} = \begin{cases} 1 + \log \mathrm{tf}_{t,d} & \text{ak } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{inak} \end{cases} \tag{2.5}$$

Skóre dvojice dotaz-dokument (q, d) je potom:

$$\text{skóre}(q, d) \;=\; \sum_{t \in q \cap d} w_{t,d} \tag{2.6}$$

Avšak ani toto skórovanie dokumentov nie je úplne správne. Pre zriedkavé termy, chceme vyššiu váhu než pre tie časté. Predpokladáme totiž, že nesú väčšiu mieru informácie. Zriedkavosť termu chápeme v rámci kolekcie. Teda dokumentová frekvencia termu df_t vyjadruje počet dokumentov, v ktorých sa term nachádza. Na tomto základe definujeme potom inverznú dokumentovú frekvenciu:

$$\mathrm{idf}_t \;=\; \log \frac{N}{\mathrm{df}_t} \tag{2.7}$$

kde N je počet dokumentov v kolekcií. Základ logaritmu je irelevantný. Logaritmus používame len pre to, aby sme znížili efekt idf_t. Často sa tak volí číslo 10, príp. e. Samozrejme, táto technika má zmysel len pre dvoj- a viac slovné dotazy. Zvyšuje váhu málo početných slov a naopak znižuje váhu veľa početných slov v kolekcií.

Podobne, ako df_t definujeme aj frekvenciu termu t v kolekcií: cf_t. Tento rozmer však nemá takú vyjadrovaciu silu ako df_t. Predstavme si, že dve slová majú rovnakú frekvenciu v kolekcií, ale rozdielnu dokumentovú frekvenciu. Ktoré slovo chceme zvýhodniť, priradiť mu väčšiu váhu? Je zrejmé, že slovo s nižšou dokumentovou frekvenciou sa musí v týchto dokumentoch viackrát opakovať. To však znamená, že tieto dokumenty sa pravdepodobne daným termom zaoberajú a preto ich chceme zvýhodniť.

Teraz môžeme obe tieto váhovania spojiť. Získame tak veľmi dobré váhovanie, ktoré sa zvyšuje spolu s počtom výskytov termov a zvýhodňuje menej časté termy:

$$\mathrm{w}_{t,d} \;=\; \mathrm{tf}_{t,d} \,.\, \mathrm{idf}_t \;=\; (1 + \log \mathrm{tf}_{t,d}) \,.\, \log \frac{N}{\mathrm{df}_t} \tag{2.8}$$

Toto váhovanie je zatiaľ jedno z najlepších a najpoužívanejších.

2.6 Vector Space model

Hlavnou myšlienkou je reprezentovať dokumenty ako vektory reálnych čísel tf–idf váh. Máme $|V|$ –rozmerný priestor, kde V je množina termov. Termy sú osy tohto priestoru a každý dokument predstavuje vektor v tomto priestore. Avšak tieto vektory sú riedke - veľká časť zložiek je nulová. Dotaz je potom tiež vektorom v tomto priestore a skóre dokumentu k dotazu je určené podobnosťou vektora dotazu a vektora k príslušnému dokumentu. Situáciu znázorňuje obrázok 1.3. Ako však určiť podobnosť dvoch vektorov? Hoci sa zdá byť vzdialenosť koncových bodov vektorov dobrý nápad, nie je tomu tak. Predstavme si neprázdny dokument. Ten nech má vektor $\vec{d_1}$. Teraz za tento dokument pridajme ten istý dokument. Vektor $\vec{d_1}$ sa tak zdvojnásobí a vzdialenosť nového vektora a pôvodného vektora tak nie je nulová, hoci by sme to tak chceli, pretože oba dokumenty sú sémanticky totožné. Nulový je však uhol medzi týmito vektormi. Dôležité je si všimnúť, že čím je uhol vektorov menší, tým väčší je jeho kosínus. Je teda jedno, či zoradíme dokumenty vzostupne podľa uhla, alebo zostupne podľa kosínusu uhla. Aby sme však mohli kosínus vypočítať, musíme vektory normalizovať tak, aby koncové body vektorov ležali na jednotkovej guli. Kosínus bude totiž potom skalárnym súčinom vektorov:

$$\cos(\vec{q}, \vec{d}) \;=\; \frac{\vec{q}.\vec{d}}{|\vec{q}|.|\vec{d}|} \tag{2.9}$$

Normalizácia vektorov – prevod na jednotkový vektor $\vec{x}$, je definovaná:

$$\vec{x} = \frac{\vec{y}}{|\vec{y}|} = \frac{\vec{y}}{\sqrt{\sum_i y_i^2}} \tag{2.10}$$

Miera podobnosti, t.j. skóre dokumentu bude potom dané:

$$\text{skóre}(\vec{q}, \vec{d}) = \cos(\vec{q}, \vec{d}) = \frac{\vec{q}.\vec{d}}{|\vec{q}|.|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\ \sqrt{\sum_{i=1}^{|V|} q_i^2}} \tag{2.11}$$

kde q_i označuje tf–idf váhu termu i v dotaze a d_i označuje tf–idf váhu termu i v dokumente.

Algoritmom implementujúcim Vector Space Model je algoritmus 2.10.

Algorithm 2.10 Kosínusové skóre(q)

```
float skore[N]
VynulujPole(skore)
for all termy t z q do
  získaj postings zoznam pre t
  spočítaj w_{t,q}
  for all pár (d, tf_{t,d}) v postings zozname do
    skore[d] ← skore[d] +w_{t,d}.w_{t,q}
  end for
end for
for all dokument d v kolekcii do
  skore[d] ← skore[d] / SpocitajDlzkuVektora(d)
end for
return skore
```

V tabuľke 2.3 môžeme vidieť spomínané váhovania.

slovo	dotaz					dokument			váha
	tf	$w_{t,d}$	df	idf	tf-idf	tf	$w_{t,d}$	norm	
the	1	1	74478	0.13	0.13	0	0	0	0
game	1	1	274	2.56	2.56	1	1	0.52	1.33
theory	1	1	1708	1.77	1.77	2	1.3	0.68	1.2
set	0	0	3868	1.41	0	1	1	0.52	0

Tabulka 2.3: Príklad váhovaní termov pre dotaz „the game theory“ a dokument „game theory set theory“. tf udáva počet výskytov, $w_{t,d}$ je váhovaný počet na základe rovnice (2.5), df dokumentová frekvencia, idf inverzná dokumentová frekvencia (2.7), tf-idf (2.8). norm je vektor po normalizácií. V pravom stĺpci je potom výsledná váha pre termy.

V tejto tabuľke je na príklade ukázané váhovanie pre dotaz a dokument. Všimnime si, že $\sqrt{0^2 + 0.52^2 + 0.68^2 + 0.52^2} \approx 1$. Stĺpec *norm* tak vlastne obsahuje súradnice jednotkového vektora. Vstupom prevodu bol vektor $w_{t,d}$ s dĺžkou $\sqrt{0^2 + 1^2 + 1.3^2 + 1^2} \approx 1.92$. Výsledné skóre dokumentu k zadanému dotazu teda je $1.33 + 1.2 = 1.53$.

2.6.1 Implementácia

Aby sme mohli efektívne počítať skóre, potrebujeme v indexe aj údaje o $\mathrm{tf}_{t,d}$ pre každý term a dokument. Ak máme pozičný index, vieme túto informáciu z postings zoznamu spočítať. Efektívnejšie však bude, spočítať frekvenciu už počas indexovania a uložiť ju k postings záznamom. Čas zodpovedania dotazu je totiž kritickejší než čas naindexovania dokumentu. Navyše, ak budeme ukladať priamo $\mathrm{tf}_{t,d}$ a nie $w_{t,d}$, teda celé čísla namiesto desatinných, pri použití vhodného kódovania nie je tento prístup náročný na miesto.

Ako však získať prvých k najvyššie hodnotených dokumentov? Ak by sme chceli spočítať hodnotenie pre všetky dokumenty a následne ich zoradiť, určite by tým utrpela odozva systému. Lepšie je použiť binárnu haldu, t.j. binárny strom kde hodnota uzla je menšia než hodnota jeho potomkov. Vytvorenie takéhoto stromu má logaritmickú zložitosť $\Theta(\log n)$ a získanie prvých k prvkov $\Theta(k \log k)$ [25]. Radenie potom prebieha takto:

1. získaj minimum min z haldy ($\Theta(1)$)
2. $s' \leftarrow$ skóre dokumentu d
3. ak je $min \leq s'$ pokračuj ďalším dokumentom
4. ak je $min > s'$ zmaž koreň ($\Theta(\log k)$)
5. pridaj <d,s'> do stromu ($\Theta(\log k)$)

Ďalšie zrýchlenie môže predstavovať používanie viacúrovňových indexov. Najskôr sa hľadá v prvom indexe. Ak sa nenájde všetkých k dokumentov, zvyšné sa hľadajú v indexe druhej úrovne, atď. Index prvej úrovne môže obsahovať napr. len nadpisy z dokumentov, na druhej úrovni už index obsahuje celé dokumenty.

Pri tomto modeli je však obtiažnejšie implementovať výhody dotazovacieho jazyka známeho z Booleovských modelov. Napríklad podpora regulárnych výrazov, špeciálne wildcards *. Oproti klasickému Booleovskému modelu musíme totiž počítať váhy pre jednotlivé termy, čo môže znamenať veľa výpočtového času.

2.7 Latentná sémantická analýza

Jeden z problémov vektorového modelu je okrem vysokého počtu dimenzií aj fakt, že stále sa pohybujeme na lexikálne úrovni. Dokumenty buď slovo obsahujú alebo nie, neuvažujeme však synonymá. V ideálnom prípade chceme nájsť tie dokumenty, ktoré sémanticky vyhovujú dotazu, sú mu čo najbližšie. Preto sa snažíme využiť nástroje lineárnej algebry, najmä *singulárny rozklad* (*Singular Value Decomposion*), ktorý zníži počet dimenzií avšak bez narušenia podobnosti dokumentov. Naviac môžeme odhaliť niektoré skryté (latentné) väzby medzi termami a dokumentmi, čo je prínos oproti vektorovým modelom. Preto túto metódu nazývame *Latentná Sémantická Analýza*, resp. *Latentné Sémantické Indexovanie.*

Na začiatok si pripomeňme dôležité pojmy. Nech C je matica $n \times m$. Hodnosť matice $h(C) \leq \min\{n.m\}$ je počet lineárne nezávislých riadkov. Diagonálna matica je štvorcová matica, ktorá ak má nenulové prvky, tak jedine na hlavnej diagonále. Vtedy je ich počet zároveň jej hodnosťou. Ak sú všetky jej prvky rovné 1, nazývame ju jednotkovou. Ortogonálnou nazveme takú štvorcovú maticu Q, ktorej transponovaná matica je zároveň inverznou $Q.Q^T = I$. Vlastné čísla štvorcovej matice A sú také komplexné čísla λ ktoré vyhovujú rovnici:

$$A.\vec{v} = \lambda.\vec{v} \tag{2.12}$$

pre nenulový vektor $\vec{v}$. Ak takéto číslo existuje, potom nazývame vektor $\vec{v}$ vlastným. S ich použitím dokážeme maticu rozložiť na súčin 3 matíc. Vyjdeme z teorému dekompozície matice. Ten tvrdí, že pre každú štvorcovú maticu S, s rozmermi $m \times m$ a m vlastnými vektormi existuje rozloženie:

$$S = U \Lambda U^{-1} \tag{2.13}$$

kde stĺpce z matice U sú vlastné vektory S, a

$$\Lambda = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{pmatrix}, \lambda_i \leq \lambda_{i+1}$$

Singulárny rozklad, Signular Value Decomposion je potom také rozloženie matice C o rozmeroch $m \times n$ na súčin matíc U,Σ a V^T:

$$C = U \Sigma V^T \tag{2.14}$$

kde

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_r \end{pmatrix}$$

r je hodnosť matice C a $\sigma_i = \sqrt{\lambda_i}$. U je ortogonálna matica $m \times n$ a reprezentuje maticu vektorov, V je ortogonálna matica $n \times n$ a reprezentuje maticu dokumentov. Analytické rozloženie matice na singulárny rozklad je náročná úloha a preto sa používa skôr numerická aproximácia.

Prvky matice $u_{i,j}$ určujú, ako veľmi je term i príbuzný k téme reprezentovanou „sémantickou“ dimenziou j. Prvky matice Σ majú klesajúcu tendenciu a vyjadrujú dôležitosť príslušnej dimenzie. A konečne prvky matice V^T $v_{i,j}$ určujú, ako veľmi náleží dokument i sémantickej dimenzií j.

Celý tento zložitý proces robíme preto, aby sme mohli nastavovať váhu jednotlivým sémantickým dimenziám. Tým nepodstatným nastavíme váhu na nulu (v matici Σ). Opätovným násobením matíc $C_2 = U \Sigma V^T$ tak získame inú maticu C_2 než bola pôvodná C, ktorá lepšie vyjadruje podobnosť dvoch dokumentov než C.

Majme nasledujúcu dokument–term maticu:

	d_1	d_2	d_3	d_4	d_5	d_6
auto	1	0	1	0	0	1
kamión	1	1	0	1	0	0
cesta	1	0	0	1	0	0
oceľ	0	0	1	0	1	0
železo	0	0	1	0	0	1

Tabulka 2.4: Príklad dokument–term matice

jej rozklad je nasledovný 2.5

Ak teraz zmeníme Σ na Σ_2 tak, aby sme potlačili dimenzie 3–6, čím zredukujeme priestor na dvojdimenzionálny, bude Σ_2 obsahovať na diagonále len 2 prvky: $2.45, 2.00$ a ostatné budú nulové. Opätovným vynásobením dostaneme maticu C_2 (2.6).

U	1	2	3	4	5
auto	0.63	0.29	0.36	-0.22	0.58
kamión	0.48	-0.59	-0.17	0.62	0.11
cesta	0.40	-0.44	-0.04	-0.67	-0.44
oceľ	0.25	0.38	-0.88	-0.08	0.05
železo	0.38	0.47	0.24	0.34	-0.67

Σ	1	2	3	4	5	6
1	2.45	0.00	0.00	0.00	0.00	0.00
2	0.00	2.00	0.00	0.00	0.00	0.00
3	0.00	0.00	1.15	0.00	0.00	0.00
4	0.00	0.00	0.00	0.69	0.00	0.00
5	0.00	0.00	0.00	0.00	0.44	0.00

V^T	d_1	d_2	d_3	d_4	d_5	d_6
1	0.62	0.20	0.52	0.36	0.10	0.41
2	-0.37	-0.30	0.57	-0.52	0.19	0.38
3	0.13	-0.14	-0.25	-0.18	-0.77	0.52
4	-0.39	0.89	0.06	-0.07	-0.12	0.18
5	0.56	0.24	-0.09	-0.75	0.11	-0.21
6	0.00	0.00	-0.58	0.00	0.58	0.58

Tabulka 2.5: Dekompozícia matice 2.4

C_2	d_1	d_2	d_3	d_4	d_5	d_6
auto	0.74	0.13	1.13	0.26	0.27	0.86
kamión	1.17	0.58	-0.07	1.03	-0.10	0.04
cesta	0.93	0.45	0.00	0.81	-0.07	0.07
oceľ	0.10	-0.10	0.76	-0.17	0.21	0.55
železo	0.22	-0.10	1.03	-0.16	0.28	0.75

Tabulka 2.6: Redukovaná matica C_2

Ak teraz skúsime vypočítať podobnosť dokumentov d_5 a d_6 získame: $0.27*0.86+(-0.10)*0.04+(-0.07)*0.07+0.21*0.55+0.28*0.75 \approx 0.55$ oproti pôvodnej hodnote 0.

Vhodná voľba čísla k udávajúceho koľko prvkov v novej matici Σ' ponecháme zatiaľ čo ostatné vynulujeme, zostáva otvorená. Chceme totiž redukovať priestor, ale príliš veľká miera redukcie (malé k voči hodnosti Σ) môže spôsobiť, že si niektoré dokumenty budú zrazu podobné, no malá redukcia (veľké k) nemusí byť zasa účinná.

Vedecké výskumy ukázali, že existuje korelácia medzi LSA a spôsobom, akým porovnáva dokumenty ľudský mozog [37]. LSA totiž berie do úvahy dokumenty, ktoré sú sémanticky podobné, zatiaľ čo tieto nemusia byť označené za podobné vo vektorovom modele - môžu obsahovať synonymá. Redukciou priestoru dosiahneme zvýraznenie podobnosti, takže môžeme riešiť problém synoným a sémantickej príbuznosti a to bez nutnosti poznať danosti nejakého jazyka, keďže metódy LSA sú striktne matematické.

Kapitola 3

Prehľad a porovnanie existujúcich nástrojov

Rôzne softvérové spoločnosti, ktoré poznáme z prostredia internetového vyhľadávania, poskytujú nástroje na lokálne vyhľadávanie. Teda vyhľadávanie v rámci jednej pracovnej stanice. Ide o produkty:

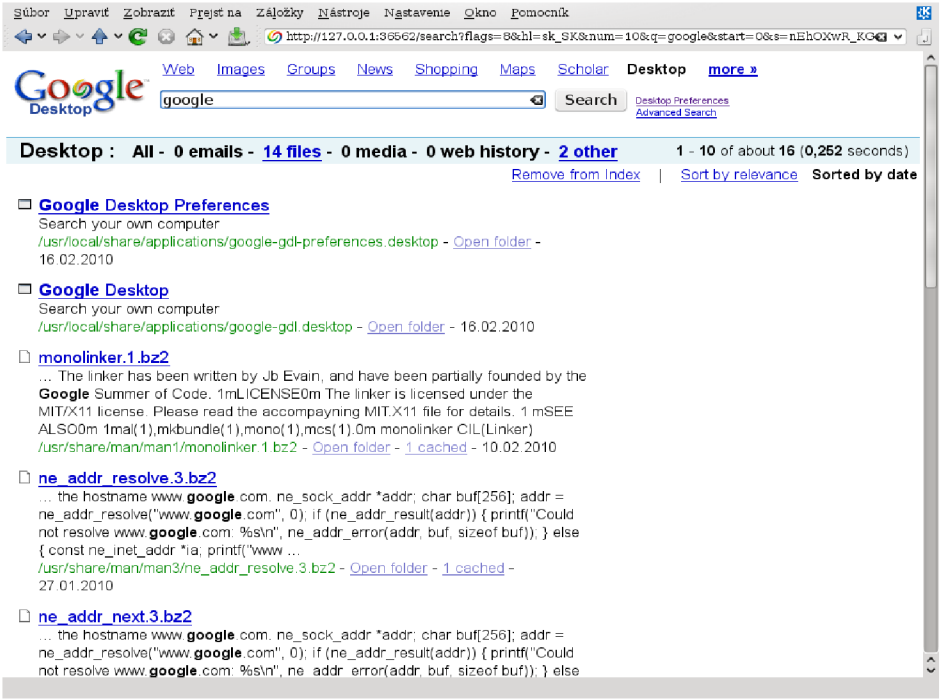
1. Google Desktop Search
2. Yahoo Desktop Search
3. Microsoft Desktop Search

Tieto vyhľadávače sú rôznou mierou zakomponované do operačných systémov, Microsoft Desktop Search je priamo súčasťou systému Windows, poskytujú rôzne výsledky, rôzne možnosti nastavenia priorít indexovania, rozsahu indexovania a iné. Medzi vyhľadávače, ktoré však nie sú zastrešené žiadnou softvérovou firmou, patrí aj Beagle, ktorý je však opensource.

3.1 Google Desktop Search - GDS

je produktom firmy Google [7]. Existujú verzie pre 3 najpoužívanejšie operačné systémy: Windows, Linux, Mac OS X. Okrem klasického vyhľadávania dokumentov integruje aj vyhľadávanie v histórií navštívených stránok a elektronickej pošte Gmail. Medzi podporované formáty súborov patrí: textové súbory, PDF, PS, HTML, OpenOffice, Microsoft Word, Excel a PowerPoint, obrázky a hudba. Tento zoznam sa dá rozšíriť. GDS totiž umožňuje doinštalovať pluginy pre rôzne formáty a prípadne aj naprogramovať nové. Pri vyhľadávaní sa potom zobrazuje zoznam výsledkov, ktorý môžeme filtrovať na konkrétny typ - napr. email. Indexovanie prebieha na pozadí, takže s počítačom môžeme naďalej pracovať. V závislosti od množstva dát, indexácia môže trvať aj niekoľko hodín. Akonáhle je však hotová, sleduje zmeny súborov, novo navštívené stránky a prichádzajúce e-maily a indexuje ich, takže index zostáva aktuálny. Vyhľadávanie cez GDS prebieha cez internetový prehliadač - rozhranie je podobné klasickej internetovej stránke 3.1.

GDS vytvára kópie verzií súborov, čo umožňuje hľadať predchádzajúce verzie prípadne náhodne zmazané súbory. Podporuje hľadanie presných fráz. GDS bohužiaľ neumožňuje nastavovanie priority indexovania. Indexuje síce s nižšou prioritou, takže nespotrebováva



Obrázek 3.1: Ukážka Google Desktop Search

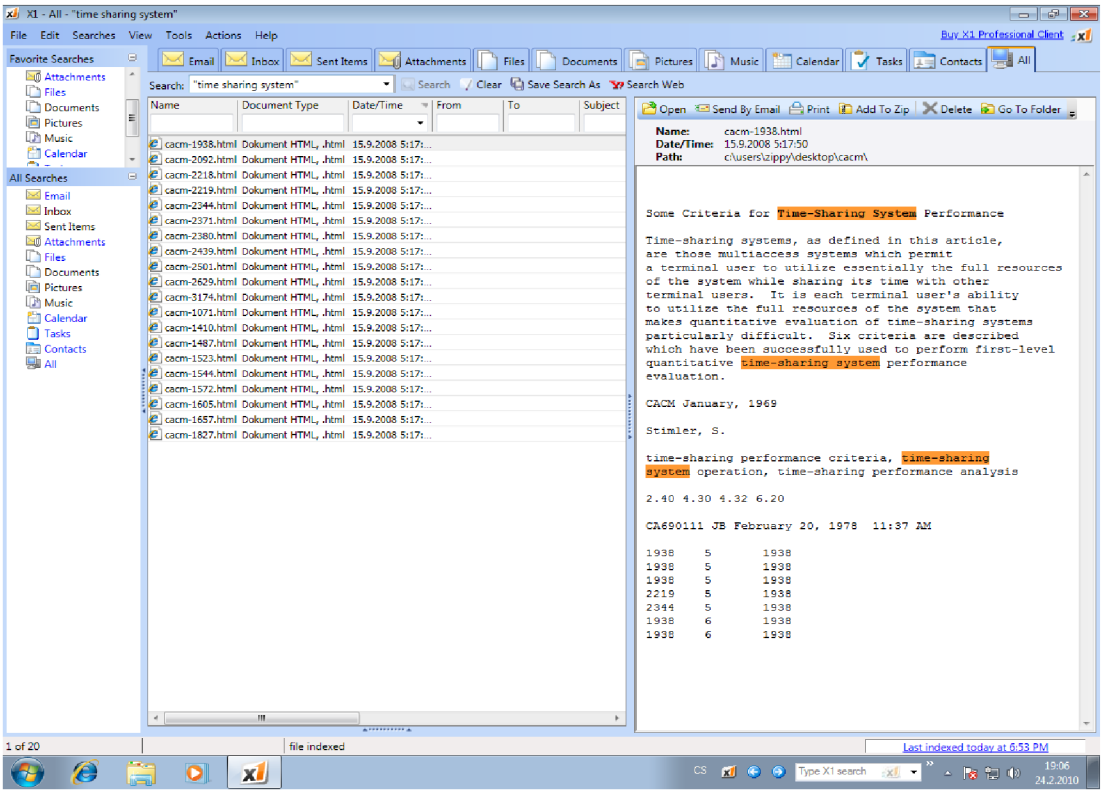
systémové prostriedky, avšak v prípade, že chceme urýchliť indexovanie, nemáme tú možnosť. Neumožňuje pridávať podporu pre ďalšie formáty súborov, dokonca ani textové súbory s inou príponou než txt. Z hľadiska samotného vyhľadávania informácií neumožňuje použitie regulárnych výrazov, dokonca ani logické operátory AND a OR, detekciu a opravu preklepov, stemming. Hľadané výrazy navyše musia byť písané s diakritikou, inak ich GDS nenájde. Istou nevýhodou pre používateľov rôznych „exotických“ distribúcií Linux môže byť aj fakt, že GDS sa dodáva len v 2 typoch balíčkov: RPM a DEB.

3.2 X1 Professional Client - XPC

bol pôvodne vyvíjaný pod názvom Find. Firma, ktorá softvér vyvíjala sa premenovala na X1 Technologies, čo bolo nasledované zmenou názvu produktu. Neskôr licenciu odkúpila spoločnosť Yahoo!, ktorá ho distribuovala zadarmo pod názvom Yahoo! Destkop Search [16]. Teraz je však tento softvér platený. Na domovských stránkach [14] sa dá stiahnuť skúšobná verzia, ktorá počas 14 dní poskytuje plnú funkčnosť. Po uplynutí tejto doby sa aplikácia zablokuje a ak ju chceme ďalej používať, musíme si ju kúpiť.

Z užívateľského pohľadu aplikácia prekvapí príjemným prostredím. Okno je rozdelené do niekoľkých častí. V hornej sú záložky umožňujúce hľadať len v istej skupine dát (e-maily, súbory, obrázky, hudba, . . .). Pod nimi sú dve časti. V prvej sa zobrazuje zoznam nájdených súborov a v druhej konkrétny súbor, ktorý sme zo zoznamu vybrali kliknutím. V tejto sa zvýrazňujú hľadané výrazy tak, ako to vidíme na obrázku 3.2.

Počet podporovaných formátov je veľký, vyše 400. Samozrejmosťou sú súbory Microsoft Office, OpenOffice, PDF, ZIP a RAR súbory vrátane ich obsahu, obrazové formáty vrátane



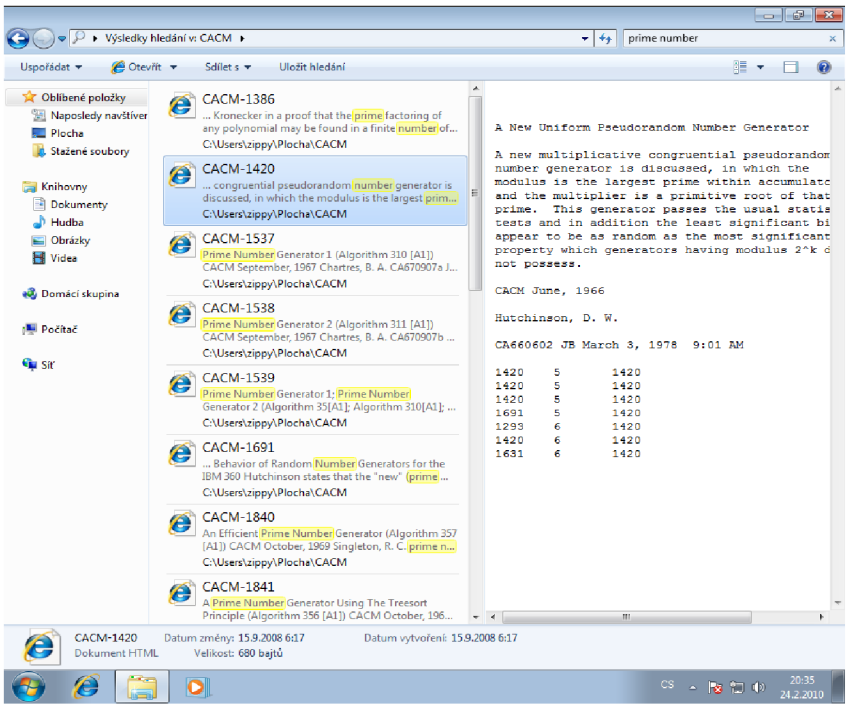
Obrázek 3.2: X1 Professional Client

metadát, vektorovú grafiku a mnohé iné. XPC je možné integrovať do Microsoft Outlook, vďaka čomu môžeme prehľadávať aj e-maily vrátane príloh, kontakty, udalosti v kalendári. XPC je postavený na platforme .NET, čo vyúsťuje do vyšších pamäťových nárokov. Dobrou vlastnosťou je možnosť nastavenia rôznych parametrov indexácie. Od priority indexovania, cez umiestnenie indexu, zdieľanie indexu s inými užívateľmi, až po manuálne spustené indexovanie, prípadne opakované podľa plánu. Vyhľadávanie je rýchle a prebieha už počas zadávania dotazu. XPC podporuje vyhľadávanie presných fráz, stemming. Dotazovací jazyk je veľmi pestrý, okrem logických spojok a ich skrátených verzií (napr. *-slovo* pre *NOT slovo*), máme možnosť špecifikovať veľkosť súboru alebo dátum (prijatia e-mailu, vytvorenia, zmeny súboru). Zaujímavou vlastnosťou je možnosť použitia zátvoriek. Máme tak plnú kontrolu nad platnosťou operandov.

3.3 Microsoft Desktop Search - MDS

tiež ako Windows Desktop Search je vyhľadávač od Microsoft-u pre ich operačný systém Windows[12]. Nejedná sa však o pôvodný, sekvenčný vyhľadávač, tak ako ho poznáme z dôb Windows 95, ale o vyhľadávač s plnou podporou indexu (invertovaného) pre systémy XP a vyššie. V najnovšom, Windows 7, je dokonca plne integrovaný. Pre XP a Vista je dodávaný ako zvlášť aplikácia. Ihneď po nainštalovaní vytvorí index implicitne všetkých pevných diskov, je však možné nastaviť aj indexovanie sieťových. Značnou nevýhodou je však obmedzenie indexácie na celý disk. Na výber má užívateľ len 2 možnosti - buď bude indexovať

celý logický disk, alebo nie. Nedajú sa vybrať len niektoré položky stromovej štruktúry adresárov, len rozšírené indexovanie metadát. Podporovaných súborov je veľké množstvo, chýbajú však otvorené formáty (napr. OpenOffice)[13].



Obrázek 3.3: Windows Desktop Search

Dotazovací jazyk, nazývaný *Advanced Query Syntax* je však na rozdiel od iných rozšírený o spracovanie prirodzeného jazyka. Logické spojky, filtrovanie metadát a zástupný znak (*) sú podporované. Tak ako sa nedá vybrať jeden adresár, jedna zložka na indexovanie, takisto sa nedá nastaviť priorita indexovania a ani žiadne iné parametre.

3.4 Beagle

je open-source projekt určený pre lokálne vyhľadávanie informácií na platforme Linux [1]. Je napísaný v jazyku C# a na správu indexu používa *Lucene.Net* [9], čo je C# verzia *Lucene*[8]. Spektrum podporovaných súborov je naozaj široké a navyše Beagle umožňuje rozšírenie o ďalšie typy bez nutnosti programovania (stačí úprava súboru *external-filters.xml*):

1. Office dokumenty: OpenOffice (odt, ods, odp), Microsoft Office (doc, xls, ppt), Rich Text Format (rft), PDF, ...
2. Textové dokumenty: HTML, zdrojové súbory (C, C++, C#, Java, JavaScript, PHP, Pascal, Phyton, Ruby, Shell skripty, ...), LaTeX, BibTex
3. Dokumentácia: manuálové stránky, info stránky, súbory pomocníka systému Windows
4. Obrázky: JPEG, PNG, BMP, TIFF, GIF, SVG; extrahovanie EXIF metadát.

5. Audio: MP3, OGG, FLAC, MPC, M4A, AAC, WMA, ...
6. Video: MPEG, ASF, WMV, MP4, ...
7. Archívy: zip, tar, gzip, bzip2 a ich obsah

Indexovanie môže prebiehať v jednom z 2 módov. V prvom na indexovanie využíva voľnú kapacitu – proces beží na najnižšej priorite. V druhom je indexovanie rovnocenné s ostatnými procesmi – hodí sa najmä na zrýchlené indexovanie, keďže značná práca s diskom a procesorom znižuje odozvu desktopového systému.

Beagle bohužiaľ rozumie len veľmi limitovaným regulárnym výrazom (len *wildcard* *). Naproti tomu umožňuje vyhľadávanie fráz, špecifikovanie slov, ktoré výsledok nemá obsahovať (*-slovo*), použitie logickej spojky OR, filtrovať typ súboru, dátum vytvorenia alebo zmeny súboru (na, pred, po, medzi), uviesť adresár, kde hľadať, používať metadáta. Beagle má aj web rozhranie, ktoré však negeneruje presne odkazy na hľadané pozície. Nevýhodou je, že Beagle vytvára veľké log súbory (v závislosti od veľkosti indexovaných dát to však môžu byť aj gigabajty za deň). Pri vyhľadávaní rozdelí Beagle výsledky do skupín (napr. obrázky, videá, hudba, zložky, ...), pokiaľ sme však nešpecifikovali konkrétnu kategóriu v ktorej hľadať.

3.5 Porovnanie vyhľadávačov

V tejto časti porovnáme vyhľadávače z hľadiska úspešnosti vyhľadávania.

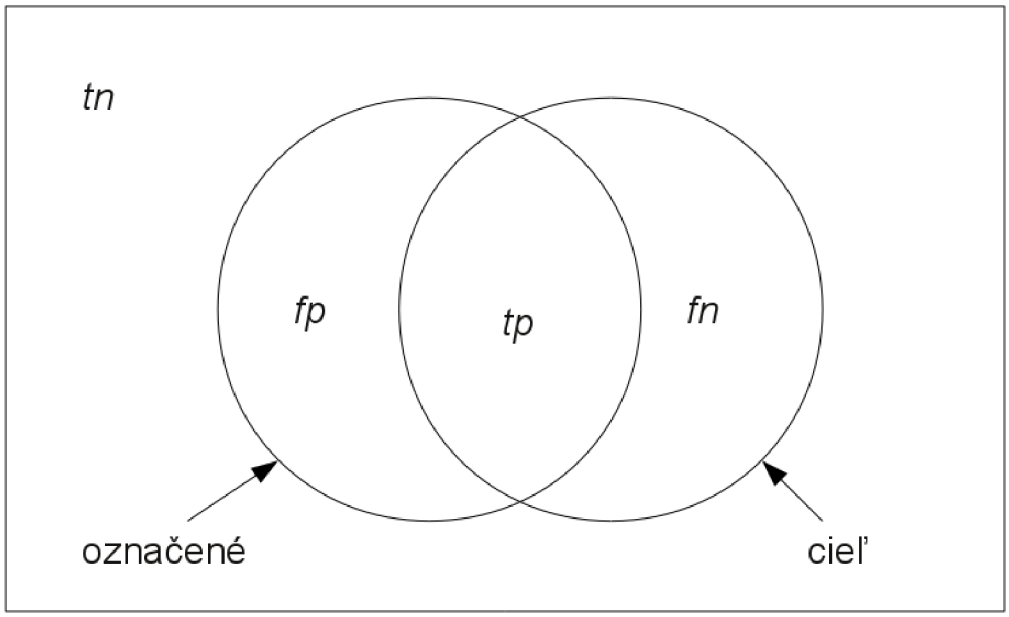
3.5.1 Úvod do metrík

Aby sme mohli porovnať dva alebo viac IR systémov, potrebujeme zaviesť metriku. IR systém pre zadanú požiadavku vráti dokumenty, ktoré, podľa neho, túto spĺňajú. Naproti tomu je množina, ktorá skutočne danú podmienku spĺňa. Napríklad: V dokumente chceme vyhľadať všetky vety, ktoré obsahujú určité slovo. Množinu týchto viet, ktoré nás zaujímajú, ktoré hľadáme, označme A. IR systém vráti množinu B - teda označené vety o ktorých si myslí, že dané slovo obsahujú. Označme:

- $tp = A \cap B$ množinu prvkov, ktoré IR systém správne označil a majú hľadanú vlastnosť.
- $fp = B \setminus A$ množinu prvkov, ktoré boli zle označené - nemajú hľadanú vlastnosť.
- $fn = A \setminus B$ množinu prvkov, ktoré neboli označené, ale mali byť
- $tn = U \setminus (A \cup B)$ množinu prvkov, ktoré neboli označené a ani nemali byť, pretože sú irelevantné.

U je univerzum, ktoré prehľadávame. V našom príklade sú to všetky vety v dokumente. Názorne sú tieto množiny na nasledujúcom obrázku:

Množina tp sa nazýva tiež True Positives. Spolu s množinou tn True Negatives tvoria správnu časť odpovede IR systému na našu požiadavku. Zle označené prípady fp False Positives sa označujú ako chyby druhého typu. False Negatives fn sú prípady chyby prvého typu, kedy systém mal označiť tieto prvky, ale neoznačil.



Obrázek 3.4: Diagram znázorňujúci možné kombinácie označenia dokumentov do 4 skupín: označené *positive*, neoznačené *negative*, (ne)označené správne *true* alebo nesprávne *false*.

Presnosť výsledku IR systému *precision* je definovaná:

$$precision = \frac{tp}{tp + fp} \tag{3.1}$$

Je to bezrozmerné číslo, ktoré udáva pravdepodobnosť, že získané prvky z IR systému sú relevantné (sú to, čo hľadáme).

Recall sa definuje nasledovne:

$$recall = \frac{tp}{tp + fn} \tag{3.2}$$

a vyjadruje pravdepodobnosť, že všetky relevantné prvky sú v odpovedi získanej z IR systému.

Tieto dva parametre sú protichodné. Môžeme napr. označiť všetky dokumenty, čím získame 100%-ný recall, ale presnosť výsledku bude minimálna. Ak naopak označíme len pár relevantných dokumentov, precision bude vysoká, ale recall bude malá. Niekedy chceme tieto dve metriky skombinovať do jednej, vyjadrujúcej celkovú výkonnosť. Taká metrika existuje a volá sa F a je definovaná:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \tag{3.3}$$

kde P označuje precision, R recall a α je váhovanie umožňujúce dať väčší dôraz na P alebo R. Často sa však volí $\alpha = 0.5$. Vtedy je $F = \frac{2PR}{P+R}$.

Existuje aj metrika *accuracy* a *error*, kde prvá uvádza počet prípadov, kde IR systém správne označil alebo neoznačil dokumenty $(tp + tn)$ a druhá počet prípadov, kde systém

chyboval ($fp + fn$). Tieto však nie sú presné, keď uvážime veľkosť množiny(čísla) tn, vecí ktoré nehľadáme. Accuracy by sa tak stala „necitlivou“ na síce malé, ale dôležité číslo tp.

Menej používanou metrikou je *fallout*:

$$fallout = \frac{fp}{fp + tn} \tag{3.4}$$

Určuje mieru zle vybraných dokumentov a ukazuje, ako ťažké je vyrobiť systém, ktorý by produkoval málo false positives.

V prípade, že IR systém dokáže zoradiť výsledky podľa relevancie, musíme tieto metriky rozšíriť a prípadne aj doplniť, pretože tie doterajšie nebrali do úvahy mieru relevancie. V tabuľke 3.1 sú výsledky 3 rôznych systémov. Každý z nich vrátil tých istých 10 dokumentov (d1,d2,...,d10) avšak s iným ohodnotením (ranking). Dokumenty sú zoradené zhora nadol podľa rankingu. V tomto prípade je precision pre všetky 3 systémy s 10 vrátenými dokumentami $precision_{10} = 50\%$. Vidíme však, že prvý systém je lepší než druhý, pretože na prvých pozíciach sú dokumenty relevantné – ak prechádzame výsledky zhora dolu. Ak by sme teda brali do úvahy len prvých 5 dokumentov, dostaneme tak už veľmi odlišné výsledky ($precision_5$).

	Ranking 1	Ranking 2	Ranking 3
	d1: ✓	d10: ×	d6: ×
	d2: ✓	d9: ×	d1: ✓
	d3: ✓	d8: ×	d2: ✓
	d4: ✓	d7: ×	d10: ×
	d5: ✓	d6: ×	d9: ×
	d6: ×	d1: ✓	d3: ✓
	d7: ×	d2: ✓	d5: ✓
	d8: ×	d3: ✓	d4: ✓
	d9: ×	d4: ✓	d7: ×
	d10: ×	d5: ✓	d8: ×
$precision_5$	1	0	0.4
$precision_{10}$	0.5	0.5	0.5
UAP	1	0.3544	0.5726
IAP	1	0.5	0.6440

Tabulka 3.1: Príklad troch rôznych usporiadaní dokumentov. Značka ✓značí relevantné a × nerelevantné dokumenty. $precision_{10}$ je pre všetkých 10 dokumentov, $precision_5$ len pre prvých 5. UAP je Uninterpolated Average Precision, IAP Interpolated Average Precision pre 11 bodov.

Táto technika sa nazýva *cutoff*.

Uninterpolated Average Precision spája rôzne precision pri rôznej miere cutoff. Pre každý jeden riadok, kde je relevantný dokument sa spočíta precision od počiatku až po daný riadok. Takto získane precision sa potom spriemerujú. Napríklad pre ranking 1 z tabuľky 3.1 je precision 1 pre d1, d2, d3, d4 a d5, pretože tieto sú jedine relevantné a sú na začiatku zoznamu. Pre ranking 3 sú jednotlivé precision: 1/2 (d1), 2/3 (d2), 3/6 (d3), 4/7 (d5) a 5/8 (d4). Priemer je potom 0.5726.

Interpolated Average Precision vychádza z recall. Je definovaná ako maximálna precision pre akýkoľvek vyšší alebo rovný recall:

$$IAP(r) = \max_{r' \geq r} p(r') \tag{3.5}$$

kde r je recall level, $p(r')$ je funkcia vracajúca precision pre recall r'. Podľa granularity s akou volíme r, hovoríme o bodových priemeroch. Pri 11 bodovom priemere volíme r postupne 0%, 10%, 20%, 30%, ... , 90%, 100%. Graf funkcie $p(r')$ sa nazýva precision-recall krivka. Štandardom v TREC komunite je *Mean Average Precision* (MAP) [42]. Bolo ukázané, že má veľmi dobrú popisovaciu schopnosť a stabilitu. Average Precision je priemer precision získaných z množiny top k dokumentov existujúcich po každom získanom relevantnom dokumente. Čiže ak pre vyhľadávania $q_j \in Q$ získame množinu relevantných dokumentov $\{d_1, d_2, \ldots, d_{m_j}$ a R_{jk} je množina ohodnotených dokumentov od začiatku až po dokument d_k, potom:

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk}) \tag{3.6}$$

MAP hodnota pre súbor testov je aritmetický priemer priemerných precision pre jednotlivé dotazy.

3.5.2 Metodika

Na porovnávanie jednotlivých vyhľadávačov bol použitý korpus CACM[3]. Jeho výhodou je, že je voľne dostupný spolu so 64 ukážkovými vyhľadávaniami a množinou k nim relevantných dokumentov. Korpus je kolekciou abstraktov článkov z časopisu Communications of the ACM medzi rokmi 1958 a 1979. Obsahuje 3204 súborov vo formáte HTML o celkovej veľkosti 12 952 KB.

Dôležité je určiť, podľa čoho chceme nástroje porovnávať. Určite podľa metrík popísaných v predošlej sekcií. Rozhodujúcim faktorom v prípade rovnakej úspešnosti môže byť rýchlosť indexovania, teda počet indexovaných dokumentov za jednotku času. Svoju rolu však zohráva aj veľkosť indexu, rýchlosť vyhľadávania a možnosti dotazovacieho jazyka (napr. podpora binárnych operátorov, regulárnych výrazov ai.), možnosť rozšíriť podporované formáty či otvorenosť kódu.

Pre minimalizovanie vplyvov prostredia bol pre každý vyhľadávač vytvorený virtuálny počítač. Pre GDS a Beagle: Ubuntu 9.10 *Karmic Koala* s 512MB RAM, a 8GB diskom, pre XPC a MDS: Windows 7 s 512MB RAM a 8GB diskom. Každý vyhľadávač mal pochopiteľne svoj vlastný virtuálny počítač avšak s rovnakými parametrami. Virtuálne počítače boli vytvorené a bežali vo VirtualBoxe 3.0.12 na 2.8GHz procesore Intel Pentium D 820, 3GB RAM a 7200 otáčkovom disku. Na hosťovskom počítači bol nainštalovaný Linux v distribúcií Gentoo s verziou jadra 2.6.31-gentoo-r6. Na virtuálne počítače bol nainštalovaný príslušný operačný systém a prevedená aktualizácia. V domovskom adresári bola vytvorená zložka `CACM`, do ktorej bol rozbalený korpus. Potom boli nainštalované vyhľadávače a zrušené indexovanie prednastavených priečinkov (napr. Beagle primárne indexuje $HOME adresár) a vytvorený snapshot pre každý zo 4 virtuálnych počítačov. To preto, aby sme mohli previesť viacero meraní rýchlosti indexovania a nástroje začínali vždy z rovnakej pozície. Celkovo som rýchlosť indexovania meral 3 krát pre každý systém. Keďže Beagle umožňuje prácu aj z príkazovej riadky, použil som príkaz `time`. GDS však nič také neumožňuje, takže som čas strávený indexovaním zmeral stopkami. Podobne aj XPC. Pri MDS nie je užívateľ nijako informovaný o stave indexácie, takže sa nedá ani zistiť či indexuje alebo nie. Preto túto vlastnosť nebudeme pri MDS uvažovať. Výsledky sú v tabuľke 3.2.

Z CACM korpusu, resp. ukážkových dotazov bolo vybraných desať tak, aby podľa možnosti čo najviac pokryli oblasť tém indexu. Vzhľadom na možnosť zadať dotaz viacerými spôsobmi (napr. vyhľadávanie presnej frázy verzus vyhľadávanie slov obsiahnutých v doku-

mente) bol každý zadaný viacerými spôsobmi, ale do výsledku sa započítal len ten, ktorý obsahoval najviac správne vybraných dokumentov (t.j. mal najvyšší recall). Konkrétne boli vybrané dotazy: 1, 2, 6, 10, 29, 39, 40, 44, 49, 63. Výsledky jednotlivých hľadaní boli zaznamenané. Na výpočet precision-recall kriviek a iných metrík (napr. MAP) bol použitý program trec_eval [11]. Tento program je používaný na získavanie TREC výsledkov podľa NIST (National Institute of Standards and Technology) procedúr. Na prevod výsledkov do vstupného formátu programu [29] som napísal program. Trec_eval je program používaný TREC komunitou na testovanie IR systémov, dokáže okrem spomínaných kriviek spočítať aj neinterpolovanú priemernú precision (UAP), R-precision (precision po R získaných dokumentoch, kde R je počet dokumentov relevantných k dotazu). Program berie do úvahy aj usporiadanie dokumentov IR systémom, takže tento fakt som pri testovaní a písaní programu na konverziu formátu neopomenul. Ako sme už videli v tabuľke 3.1, poradie dokumentov je veľmi dôležité. Výsledky sú v tabuľkách 3.3 3.4 a na obrázku 3.5.

3.6 Výsledky

Porovnanie rýchlosti indexovania sú v tabuľke 3.2. Keďže pre MDS nebolo možné tieto údaje získať ani ich neuvažujeme.

č. merania	Beagle	GDS	XPC
1	0:28	37:30	2:00
2	0:31	36:56	1:56
3	0:30	37:23	1:57
priemer	**0:29.67**	**37:16.33**	**1:57.56**

Tabulka 3.2: Porovnanie rýchlosti indexácie. Údaje sú v tvare [mm:ss].

	Beagle	**GDS**	**MDS**	**XPC**
num_ret	220	197	215	229
num_rel	127	112	124	124
num_rel_ret ↑	57	32	47	50
MAP ↑	0.2783	0.1575	0.2418	0.2407
R-precision ↑	0.3313	0.2158	0.3067	0.3012

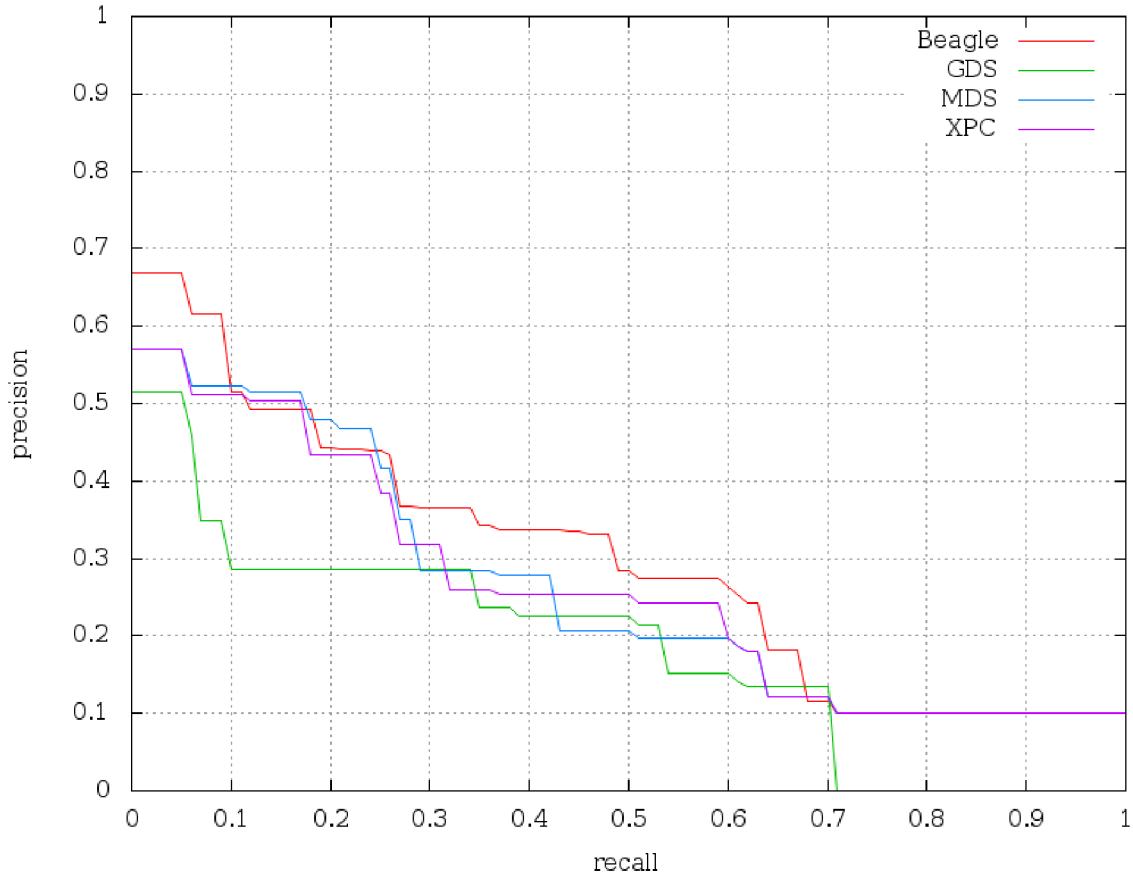
Tabulka 3.3: Výsledky porovnávania vyhľadávačov. num_ret – počet získaných dokumentov celkovo (za všetky dotazy dokopy), num_rel – celkový počet relevantných dokumentov (definovaných, nie získaných), num_rel_ret – koľko zo získaných dokumentov bolo relevantných. Šípka ukazuje, či je lepšie väčšie alebo menšie číslo.

Na grafe 3.5 vidíme precision-recall krivky pre testované vyhľadávače. Ktorá krivka je však lepšia? Určite chceme najvyššiu možnú precision pre každú úroveň recall. Preto je Beagle z tohto hľadiska lepší než všetky ostatné programy. Najhoršie je na tom GDS. Je to tak práve kvôli jeho obmedzenému dotazovaciemu jazyku, ktorý neumožňuje bližšie špecifikovať dotaz a preto aj pri vyšších recall hodnotách ma stále nízku precision. Zaujímavým prekvapením je vyrovnanosť XPC a MDS. V podobnom teste, ktorý však prebehol pred 4 rokmi, GDS tiež skončil podobne [40]. V tabuľke 3.5 vidíme porovnané niektoré možnosti nastavenia indexácie a podporovaných typov súborov. Treba však podotknúť, že až na GDS podporujú ostatné aplikácie nepreberné množstvo typov (XPC vyše 500), takže

recall	**Beagle**	**GDS**	**MDS**	**XPC**
0.00	0.6690	0.5144	0.5695	0.5695
0.10	0.5157	0.2860	0.5219	0.5112
0.20	0.4427	0.2860	0.4788	0.4345
0.30	0.3660	0.2860	0.2843	0.3174
0.40	0.3371	0.2251	0.2776	0.2529
0.50	0.2835	0.2251	0.2062	0.2529
0.60	0.2639	0.1522	0.1961	0.1961
0.70	0.1149	0.1340	0.1206	0.1206
0.80	0.1000	0.0000	0.1000	0.1000
0.90	0.1000	0.0000	0.1000	0.1000
1.00	0.1000	0.0000	0.1000	0.1000

Tabulka 3.4: Precision-recall pre rôzne úrovne recall. Vyššie číslo (vyššia precision) je lepšia.

boli vybrané len tie najčastejšie používané. Všetky aplikácie, až na XPC umožňujú rozšírenie o nové typy: GDS a MDS doinštalovaním zásuvných modulov (tzv. pluginov), Beagle úpravou súboru `external-filters.xml`. V prípade, že modul pre nejaký typ súboru neexistuje, môžeme naprogramovať modul pomocou výrobcom dodávaného vývojárskeho kitu. Z hľadiska možností nastavenia indexácie je na tom najlepšie XPC, ktorý dovoľuje nastaviť snáď každý aspekt. Hoci to nie je v tabuľke, umožňuje aj nastavenie priority indexovania. Beagle síce neumožňuje manuálne indexovanie alebo jeho plánovanie, ale keďže sa dá ovládať aj z príkazovej riadky je možné za použitia štandardných Linuxových nástrojov (napr. cron) tento problém obísť. Avšak pre bežného užívateľa nemusí byť toto akceptovateľné riešenie.



Obrázek 3.5: Precision-recall graf pre porovnávané systémy.

	Beagle	GDS	MDS	XPC
Testovaná verzia				
Indexácia				
Možnosť rozšíriť o nové formáty	✓	✓	✓	×
Indexácia spúšťaná zmenou súboru	✓	✓	✓	✓
Indexácia môže byť naplánovaná	×	×	×	✓
Indexácia môže byť pozastavená	×	✓	×	✓
Manuálne spustená indexácia	×	✓	×	✓
Dotazovací jazyk				
AND	✓	×	✓	✓
OR	✓	×	✓	✓
NOT	×	×	✓	✓
*	✓	×	×	×
?	×	×	×	×
Frázy	✓	✓	✓	✓
Poddotazy	×	×	✓	✓
Synonymá	×	×	×	×
Rozsah dátumov	×	×	✓	✓
Rozsah čísel	×	×	×	✓
Dotazy v prirodzenom jazyku	×	×	✓	×
Citlivý na veľkosť písmen	×	×	×	×
Podporované formáty				
Office	OpenOffice, MS Office, RTF, PDF, PS, AbiWord, Thunderbird	OpenOffice, MS Office, PDF, PS, Gmail, Outlook, Thunderbird	MS Office, PDF, PS, RTF, Outlook	OpenOffice, MS Office, PS, RTF, PDF, Lotus, Outlook, Thunderbird, ...
Text	HTML, zdrojové texty, Latex, Bibtex, txt	HTML, txt	HTML, zdrojové texty, Latex, txt	HTML, zdrojové texty, Latex, Bibtex, txt
Audio	mp3, ogg, flac, wma, m3u, pls, ...	mp3, wav, wma, ...	mp3, midi, wav, wma, m3u, ...	mp3, wav, wma
Video	mpeg, asf, mp4, wmv, avi, ...	avi, mpeg, wmv	mpeg, wmv, avi, asf, mp2, ...	mpeg, wmv, avi, flash, asf, ...
Obrázky	jpeg, png, bmp, tiff, gif, svg, EXIF	jpeg, png, bmp	jpeg, png, bmp, tiff, gif ...	jpeg, png, bmp, tiff, gif, svg, EXIF, ...
Archívy	zip, tar, gzip, bzip	zip	zip, tar, gzip, cab,	zip, rar, gzip, tar,
iné	Info a Man, rpm, ebuild	vcard, reg		MS Access, DBase

Tabulka 3.5: Porovnanie vlastností jednotlivých produktov.

Kapitola 4

Návrh

V tejto kapitole navrhneme základné komponenty vlastného IR systému, ktorý v nasledujúcej implementujeme.

Ako vieme z predošlých častí, základným stavebným kameňom je invertovaný index. Jeho implementácia nie je triviálna a je spojená s rôznymi problémami, ktoré boli rozoberané vyššie. Našťastie, existuje niekoľko kvalitných knižníc, ktorého implementujú a to v rôznych mierach. Napríklad knižnica pre jazyk Java Lucene [8] podporuje nielen postings zoznamy, stop slová alebo stemming, ale aj vektorový model.

Druhou, nemenej dôležitou časťou je stemmer, prípadne lematizér. Požiadavky na neho kladené sú v podstate veľmi presné a strohé. Ich implementácia je však o to náročnejšia.

V neposlednom rade si treba ujasniť, aké formáty bude aplikácia podporovať, ako príjemné bude jej ovládanie, keďže v konečnom dôsledku je to práve táto časť, s ktorou budú užívatelia prichádzať do styku. Tu však nemáme na mysli estetickú hodnotu dizajnu aplikácie, ale akúsi nápomocnosť užívateľského prostredia. Sem spadá intuitívne ovládanie, jednoduchý dizajn, aby prácu s aplikáciou zvládol aj neskúsený užívateľ.

Rozoberme teraz jednotlivé prvky podrobnejšie.

4.1 Indexer a klient

Vzhľadom na úvod kapitoly je prirodzené rozdelenie na dve hlavné časti: *Indexer* a *klient*. Prvý z menovaných bude mať na starosti vkladanie dokumentov do indexu, udržiavanie aktuálnosti indexu a vôbec – akúkoľvek prácu spojenú s procesom indexovania alebo vyhľadávania v indexe. Druhý bude sprostredkovať služby človeku. Získa hľadané termy, prepošle ich Indexeru, ktorý dotaz vyhodnotí, výsledok prepošle späť klientovi, ktorý ich nejako (najlepšie graficky) reprezentuje. Z tohto rozdelenia však vyvstáva požiadavok na rýchlu a efektívnu komunikáciu, ktorá navyše ak sa nebude obmedzovať len na jedno spojenie bod–bod, umožní prácu viacerým klientom. Iste, pri lokálnom vyhľadávaní tento požiadavok ustupuje do úzadia – hoci, v niektorých prípadoch môže byť dôležitý (napr. ak by sme chceli spustiť viac inštancií klienta). Zvolený model jedného „servera“ a jedného alebo viacerých klientov navyše umožňuje lepšie prispôsobiť vykonávanie požiadavkov aktuálnemu stavu systému, prípadne serializovať niektoré typy požiadavkov, ktorých paralelné vybavovanie by mohlo byť pomalšie, napr. sekvenčné vybavovanie požiadaviek na vyhľadávanie, čo sa v konečnom dôsledku pretransformuje na prácu s pevným diskom, bude určite rýchlejšie než súbežné, pri ktorom by sa neúmerne prejavila réžia v podobe prístupovej doby disku, t.j. vystavenie hlavičiek, dotočenie platní ai.

4.2 Komunikácia

Požadovaný komunikačný kanál, teda medzi procesová komunikácia, je zložitý problém sám o sebe. Chceme, aby bol asynchrónny. Vykonanie niektorých operácií (napríklad prechod určitej vetvy stromu adresárov) môže trvať veľmi dlho a je nežiadúce, aby bol klient medzi tým blokovaný. Snáď najvhodnejším príkladom je samotný proces indexovania, pri ktorom chceme mať možnosť ho asynchrónne pozastaviť a zasa obnoviť. Ideálnym prípadom by bolo, keby indexer čakal na prichádzajúce spojenia (počas tohto čakania samozrejme aj vykonával nejakú užitočnú činnosť) a po prijatí požiadavku o spojenie vytvoril zvlášť kanál, po ktorom by komunikoval už len s jedným konkrétnym klientom. Zároveň však, ťažisko komunikácie bude v rámci jedného počítača, dokonca rámci jedného sedenia (*desktop session*), hoci ak bude použitý protokol podporovať aj sieťovú komunikáciu, môže to byť prínosom. Tu vyvstáva otázka, či vytvárať vlastné riešenie alebo použiť niečo už vytvorené, čo by však spĺňalo požiadavky.

4.3 Indexer

Jeho základná činnosť je jasná: prejsť stromovú štruktúru adresárov, vybrať tie položky, ktoré užívateľ chce naindexovať, vyextrahovať z nich text a metadáta a tieto vložiť do indexu. Ak však pôjdeme do hĺbky, nie je táto činnosť už taká priamočiara. Ako vybrať tie položky, ktoré sa nakoniec naindexujú? Ako vyextrahovať ich obsah alebo metadáta? Niektoré formáty súborov totiž umožňujú uchovávať aj rôzne pomocné informácie, napr. PDF umožňuje uložiť meno autora nezávisle na texte dokumentu. Lenže čo s tými formátmi, ktoré takúto vlastnosť nemajú? Ďalšou vlastnosťou indexera, ktorú sme doteraz príliš neuvažovali, je udržiavanie jeho aktuálnosti. Predstavme si totiž situáciu, kedy je konfiguráciou aplikácie daná nejaká zložka, ktorej obsah indexujeme, napr. domovský adresár užívateľa. Táto však vôbec nemusí byť statická (a častokrát nie je), nové položky vznikajú, iné zanikajú. Iste chceme, aby sa indexer tejto situácií prispôsobil a tiež sa aktualizoval. Vidíme, že indexer sa bude skladať z veľa častí, ktorých činnosť musíme popísať podrobnejšie. Mnohé moduly vzídu až ako riešenie už načrtnutých.

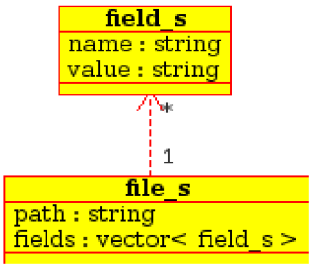
4.3.1 Index

Táto časť indexera bude mať na starosti index, a pri použití vhodnej knižnice bude vlastne „len“ jej použitím. To znamená, na začiatku otvorí index, čo je vlastne zložka so súbormi, prípadne sa vytvorí nový, prázdny. Kde je tento index, ak existuje, umiestnený, alebo kam ho ešte len umiestniť? Pevne definovaná cesta v programe by asi nebol užívateľsky prívetivý spôsob. Lepšie by bolo, ak by táto cesta bola uložená v konfiguračnom súbore, odkiaľ by sa pri spustení programu načítala a podľa toho, či by existovala, resp. by na jej konci existoval validný index, by sa aj vykonala príslušná akcia.

Teraz prichádza na rad prechádzanie stromu adresárov a výber položiek vhodných na indexovanie. Tu zasa využijeme konfiguračný modul, ktorý z kofiguračného súboru zistí všetky užívateľom definované cesty, ktoré treba prejsť. Ich zoznam, rovnako ako index, nemôže byť statický, ale chceme pridávať a odoberať položky z tohto zoznamu dynamicky, bez nutnosti reštartovať aplikáciu. V tomto bode je treba spraviť rozhodnutie, či pridávať dokumenty do indexu ihneď pri prechádzaní stromovej štruktúry adresárov (tzv. *traverzovanie*), alebo budeme jednotlivé položky ukladať do nejakého interného zoznamu odkiaľ ich budeme potom neskôr po jednej vyberať a vkladať do indexu.

Prvá možnosť má výhodu v menšej pamäťovej náročnosti, sťažuje však niektoré operácie popísané ďalej (napr. pridávanie užívateľom definovaných metadát). Ďalšia jej nevýhoda sa prejaví pri behu aplikácie. Najskôr si popíšme, ako traverzovanie prebieha. Dostaneme cestu k zložke. Tú otvoríme a začneme čítať položku po položke. Ak aktuálna položka je zložka, vnoríme sa do nej a celý cyklus sa opakuje. Pri hlbokom strome adresárov budeme mať veľa otvorených deskriptorov, keďže každý proces má limit pre počet simultánne otvorených deskriptorov. S týmto súvisí aj možnosť pozastavenia indexovania. Ak dáme užívateľovi možnosť pozastaviť indexovanie (pretože napr. zrovna potrebuje výkon počítača na niečo iné), zostali by sme zanorení v niektorej vetve stromu s alokovanými systémovými prostriedkami (otvorenie súboru, resp. zložky si vyžaduje nemalú asistenciu jadra operačného systému), čím by sme neefektívne využívali pridelené prostriedky. Preto bude druhá možnosť lepšia, udržiavať si vlastný zoznam dokumentov čakajúcich na indexovanie. Tento zoznam možno navyše ponať ako prioritnú frontu, kde sa budú môcť niektoré súbory „predbiehať“, napr. pri aplikácií už spomínaných, užívateľom definovaných metadát, čo vyústi z pohľadu užívateľa na rýchlejšiu odozvu aplikácie. Rýchlosť odozvy je vo všeobecnosti jedna z najžiadanejších vlastností dnešných desktopových systémov (nielen IR).

V tomto momente prichádza na rad extrahovanie obsahu a metadát zo súborov. Rôznych formátov súborov existuje veľmi veľa a ku každému sa pristupuje inak. Vyberieme však tie najpoužívanejšie a podporu pre ne implementujeme priamo. Ak navyše umožníme užívateľovi dodefinovať podporu pre iné formáty, než tie, ktoré priamo implementujeme, rozšíri to množinu podporovaných formátov. Na extrahovanie dát a metadát použijeme zvlášť modul, ktorý popíšeme neskôr. Na teraz postačí, že tento modul máme a pracuje. Čo je však jednotkou internej fronty súborov čakajúcich na vloženie do indexu? Je to akákoľvek informácia o súbore, ktorú chceme uložiť do indexu. Vzniká tak pojem atribút súboru, teda príznak nesúci informáciu o vlastnostiach súboru. Pri traverzovaní získame atribúty veľkosť, čas modifikácie a podobne, v module pre extrakciu metadát zasa autora dokumentu alebo názov organizácie, kde dokument vznikol. Všetky tieto atribúty chceme uchovať, pretože ak potom umožníme užívateľovi podľa nich aj vyhľadávať, môže hľadanú informáciu bližšie špecifikovať. Pochopiteľne, nemá význam tieto atribúty stemovať, ba dokonca by to mohlo byť skôr na škodu, pretože by mohlo prísť k pozmeneniu mena autora. Jednotkou je teda súbor s atribútmi, pretože rôzne atribúty získavame v rôznych moduloch. Jedinou vínimkou je obsah súboru, ktorý sa nechápe ako atribút súboru, pretože môže byť veľmi veľký, čo by znamenalo mrhanie pamäťou. Naproti tomu sa obsah získava tesne pred vložením do indexu. Koncept jednotky môžeme vidieť na obrázku 4.1.

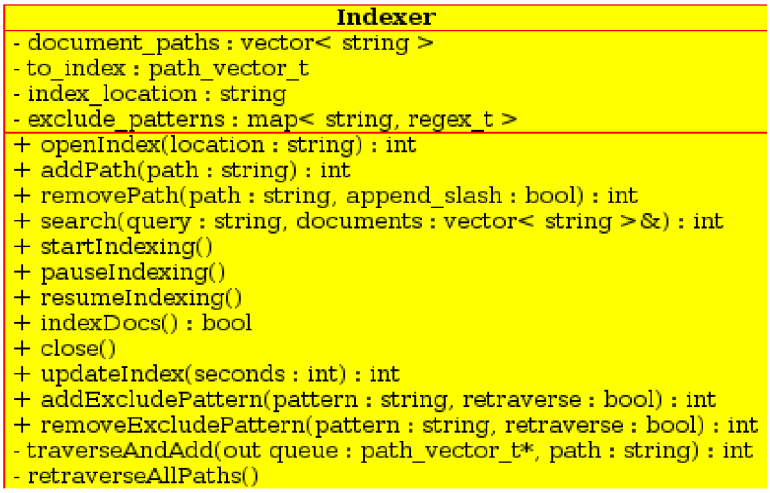


Obrázek 4.1: UML koncept internej reprezentácie súboru čakajúceho na indexovanie.

Do tejto časti aplikácie ešte patrí aj udržiavanie aktuálnosti indexu, teda sledovanie zmien v definovaných častiach adresárového stromu. Tu sa priam ponúka podpora jadra

operačného systému, konkrétne služba *inotify* (ktorá nahradila staršiu *dnotify*), ktorá poskytuje notifikácie o udalostiach v súborovom systéme a to na ľubovolnom mieste. Nemusí sa tak prechádzať celý súborový systém, ale hneď vieme, ktorý súbor akciu vyvolal a aká akcia to vlastne aj bola. K preindexovaniu tak môže prísť veľmi rýchlo a s malou réžiou. Nás budú zaujímať akcie: vytvorenie súboru, vtedy vložíme nový súbor do indexu, zmazanie súboru, vymažeme súbor z indexu, a zmena obsahu súboru, kedy znova naindexujeme zmenený súbor. Podpora pre tento subsystém jadra bola pridáná vo verzií 2.6.13 [39]. Ak teda zistíme zmenu, vykonáme príslušnú akciu a súbor vložíme do fronty.

Celkový návrh triedy Indexer môžeme vidieť na obrázku 4.2. Je v ňom zachytená ešte jedna funkcia, ktorú sme doteraz opomínali. Nie je totiž vôbec samozrejmé, že ak aj užívateľ definuje adresáre, ktoré sa majú naindexovať (vrátane podadresárov), že chce zahrnúť všetky súbory v nich obsiahnuté. Napríklad ak indexuje svoj domovský adresár, kde má uložený aj index, rozhodne ho treba z procesu vylúčiť. Okrem toho, že by to viedlo na cyklické znovuindexovanie (index sa zmenil, čo vyvolá zmenu, ktorá sa však musí znova zahrnúť do indexu), informačná hodnota vkladaná do indexu by bola nulová. Musíme teda poskytnúť užívateľovi možnosť nejakým spôsobom označiť tie časti (či už uzly alebo listy) adresárového stromu, ktoré si neželá indexovať. Najvohdnejším spôsobom sa javí vulučovať tieto položky na základe toho, aké majú atribúty. Môže ísť o názov súboru alebo ceste k nemu, ktorá vyhovuje zadanému regulárnemu výrazu alebo tzv. *shell pattern*, teda vzorom zástupných znakov používaných v príkazovom riadku, alebo veľkosť súboru, čas modifikácie, a podobne.

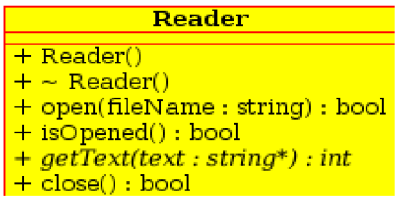


Obrázek 4.2: UML koncept triedy Indexera.

4.3.2 Reader

Táto trieda bude mať na starosti získavanie dát a metadát zo zadaného súboru. Najkôr však určí jeho typ, aby sme zvolili správnu metódu extrakcie. V Linuxe sa však typ súboru určuje inak než v systéme Windows. Teda nie na základe prípony súboru (s čím je spojených niekoľko problémov ako dvojitá prípona) ale na základe tzv. magického čísla. Pôvodne tento termín označoval prvé dva bajty súboru, ktoré obsahovali identickú sekvenciu znakov,

ktorá odlišovala typy súborov. Neskôr sa obmedzenie dĺžky zrušilo. Napríklad súbory typu GIF vždy začínajú sekvenciou GIF87a alebo GIF89a. Binárne súbory su takto veľmi ľahko rozpoznateľné, pretože častokrát je v záujme ich tvorcov, aby dokázali identifikovať ich typ a predišli tak situácií, keď im niekto „podsunie“ iný formát, čo by znamenalo nesprávne chovanie ich aplikácie, pretože dáta by nedávali zmysel. Pri textových súboroch je situácia ťažšia. Napríklad HTML súbor môže začínať tagom `<html>`, `<?xml` alebo `<!DOCTYPE`, či prípadne prázdnymi riadkami alebo komentármi. Existuje však knižnica libmagic [46], ktorá dokáže tieto anomálie rozoznať a spoľahlivo určiť nielen typ dokumentu, ale aj jeho kódovanie, či verziu formátu. Keď už vieme formát súboru, zvolíme príslušnú triedu, napr. XMLReader pre XML súbory, ktorá rozparsuje obsah a zistí aj prípadné metadáta. Rodičom týchto tried je trieda Reader, ktorá definuje základné rozhranie (viď obrázok 4.3).



Obrázek 4.3: UML koncept triedy Reader.

V tomto module sa dajú vhodne využívať voľne dostupné a slobodné knižnice pre prácu s dokumentmi daného typu, napr. libexpat [17] pre prácu s XML.

Ako sme už spomenuli, dávame užívateľovi možnosť pridať podporu pre priamo nepodporované formáty tak, že si sám túto podporu dodefinuje. Definovanie bude prebiehať tak, že užívateľ špecifikuje pár typ súboru – akcia. Pod akciou sa rozumie príkaz, ktorý prevedie súbor to textovej podoby (napr. program `pdftotext`). Tento príkaz sa potom nad príslušným súborom vykoná a zachytí sa jeho výstup. Problémom zostáva interpretácia typu súboru, ktorá by bola zrozumiteľná pre človeka a aj počítač. Javí sa najvhodnejšie použiť MIME typ. No pri bližšom preskúmaní nie je dostatočný. Pretože dva rozdielne formáty súborov (najmä binárne) môžu mať rovnaký MIME typ - napr. `application/octet-stream`. Ak k tomu však pripojíme aj príponu súboru, dokážeme typ veľmi dobre špecifikovať.

Interne teda môžeme pracovať s magickými číslami, navonok však musíme tieto nejako unifikovať a prispôsobiť užívateľovi. Preto musia existovať dva spôsoby určovania typu súboru. Akcia môže byť buď interná – vtedy sa tento modul pokúsi na základe magického čísla vybrať najvhodnejší prístup, alebo externá, keď modul zistí či sa zhoduje MIME typ, prípona súboru alebo oboje a vykoná užívateľom definovaný príkaz. Zachytí jeho výstup, ktorý potom interpretuje ako obsah súboru.

Extrakcia metadát môže prebiehať na úrovni aplikačných knižníc, keď napríklad knižnica poppler [19] ponúka prácu nielen s obsahom PDF súborov, ale aj s ich metadátami ako napr. autor, kľúčové slová, tvorca a mnohé iné. Tieto chceme samozrejme uchovať a umožniť aj podľa nich vyhľadávať. Tu sa naplno prejavuje modulárnosť aplikácie, kedy podpora pre nové formáty vznikne vytvorením novej triedy odvodenej od triedy Reader, ktorá definuje základné rozhranie pre extrakciu dát.

4.3.3 Konfigurácia

Táto časť aplikácie bude mať na starosti načítanie nastavení z prednastaveného súboru, prípadne zadaného ako argument pri spustení. Tak ako je dobrým zvykom v operačných

systémoch typu Unix, konfigurácia je vlastne textový súbor, kde môžu byť okrem samotných nastavení aj komentáre vysvetľujúce ktorý aspekt chovania aplikácie to ktoré nastavenie ovplyvňuje. Túto vlastnosť chceme zachovať. Ktoré všetky vlastnosti aplikácie sa budú uchovávať v konfigurácií? V akom formáte? Aká bude syntax?

Syntax sa bude vyvíjať spolu s tým, ako budú pribúdať jednotlivé nastavenia uchovávané v konfiguračnom súbore. Ktoré to však sú určíme tak, že začneme na začiatku, pri spustení aplikácie a budeme pre každom nastavení uvažovať či má alebo nemá byť užívateľsky nastavované. Takto dostaneme, pozíciu indexu, definované adresáre alebo súbory, ktore sa majú indexovať, pravidlá pre vylúčenie súboru z indexovania, definície akcií vykonávaných nad typom súborov. Mierne predostriem aj iné vlastnosti: priorita indexovania, prípadne maximálna záťaž systému pri ktorej sa indexovanie pozastaví.

Niektoré nastavenia môžu obsahovať viac položiek - napríklad cesty zahrnuté v indexe, ale niektoré nie (pozícia indexu, priorita indexovania). Vhodnou syntaxou sa javí zápis: `<vlastnosť> = {<atribút> [, <atribút>, <atribút>, ...]};`. Tento je veľmi jasný, pochopiteľný, ľahko strojom spracovateľný. Aby sa predišlo nejednoznačnosti, znakové reťazce (napr. cesty k adresárom) sa navyše uvádzajú v dvojitých úvodzovkách. Spracovanie takto popísanej syntaxe sa môže potom diať cez zásobníkový automat (keďže pravidlá tvoria bezkontextovú gramatiku).

V prípade, že však žiadny konfiguračný súbor nebude zadaný a prednastavený nebude existovať, treba tieto hodnoty nastaviť predvolenými. Sledované vlastnosti sa môžu dynamicky meniť za behu programu, treba ich preto pred ukončením programu zapísať, upraviť.

4.3.4 Stemmer

Tento modul bude v logickej postupnosti zaradený za Reader. Jeho tvorba nie je pri morfologicky bohatých jazykoch triviálna. V slovenčine existujú 3 rody podstatných mien a každý má 4 vzory podľa ktorých sa skloňuje. Existujú však vínimky. Stemmer teda môže fungovať takto: určí rod podstatného mena, a na základe toho vyberie správne pravidlo pre odstránenie prípony, ktorú nahradí za príponu nominatívu. Ak má teraz stem na konci príponu plurálu, nahradí ju za príponu nominatívu singuláru. Tento spôsob však prináša dve problémy - pri niektorých vzoroch sa zhodujú prípony pre (vo všeobecnosti) rôzne pády. Tieto vzory však majú rozdielnu príponu v nominatíve. Výber vhodného pravidla je tak obtiažny. Druhým, ťažším, je určenie rodu vyskloňovaného podstatného mena a vôbec, určenie slovného druhu. Ku každému slovu tak existuje množina pravidiel, ktoré sa naň dajú aplikovať, no len aplikovaním niektorých z nich získame správny koreň. Táto množina môže byť navyše prázdna. Ak však postupne budeme aplikovať pravidlá, získame stem, ktorý potom môžeme porovnať so slovníkom. Ak príde k zhode, je veľmi pravdepodobné, že aplikované pravidlo bolo to správne a získali sme koreň. Treba sa však nejako vyrovnať s tým, že hľadaný stem nemusí existovať v slovníku a tak ho nenájdeme. Vtedy ho pridáme do slovníka.

Pre vytváranie stemmera dokonca existuje programovací jazyk Snowball [21], v ktorom sa zapisujú pravidlá. Kompilátor tohto jazyka potom dokáže tieto previesť do programu v jazyku C alebo Java.

4.3.5 Ostatné

Sem spadajú všetky ostatné časti, ktoré sú príliš malé na to, aby boli samostatne oddelené, alebo poskytujú len podpornú funkcionalitu. Konkrétne sa jedná o nastavovanie priority indexovania, čo je vlastnosť, ktorú paradoxne nemá veľa produktov (ako sme mali možnosť vidieť v teste). Pri lokálnom indexovaní je dôležité spotrebovávať čo najmenej systémových

prostriedkov tak, aby užívateľ mohol naďalej pracovať s počítačom, čiže odozva na vstup od užívateľa bola rýchla. Naštastie, priamo jadro Linuxu meria zátaž systému a dokáže ju exportovať pre čítanie do užívateľského režimu. Túto hodnotu môže ktokoľvek čítať a nachádza sa v súbore `/proc/loadavg`. Príklad jeho výstupu: `0.05 0.03 0.00 2/263 29646`.

Prvé 3 čísla znamenajú využitie procesora a I/O subsystému za jednu, päť a desať minút. Štvrté číslo, zlomok, určuje podiel bežiacich procesov ku všetkým procesom v systéme. Posledné čislo určuje PID procesu, ktorý obsah súboru čítal [18].

Ak teda dáme užívateľovi možnosť pozastaviť indexovanie pri dosiahnutí ním definovanej hranice záťaže systému (*load*), môže určiť ktorá z antagonistických požiadaviek je pre neho dôležitejšia. Či je to rýchlosť indexovania (množstvo naindexovaných dokumentov za jednotku času), alebo je to možnosť naďalej pracovať s počítačom bez toho, že by bol obmedzovaný. Samozrejme, táto hranica sa bude dať dynamicky meniť, prípadne úplne vypnúť. Dokonca, samotné indexovanie bude zvyšovať záťaž, takže sa po dosiahnutí hranice pozastaví, počká kým záťaž neklesne a potom obnoví indexovanie. Stojí za povšimnutie, že sledovanie záťaže systému a priorita indexovacieho procesu sú dve rozdielne veci, ktoré sa v rozdielnych situáciach prejavia inak. Ak by mal indexovací proces nižšiu prioritu než všetky ostatné užívateľské procesy, stále môže vyťažovať systém (najmä pri práci s pevným diskom, kedy samotný proces nebeží, ale je uspaný až do chvíle, keď operačný systém pripraví dáta) a ovplyvňovať tak rýchlosť reakcie systému na podnety od užívateľa.

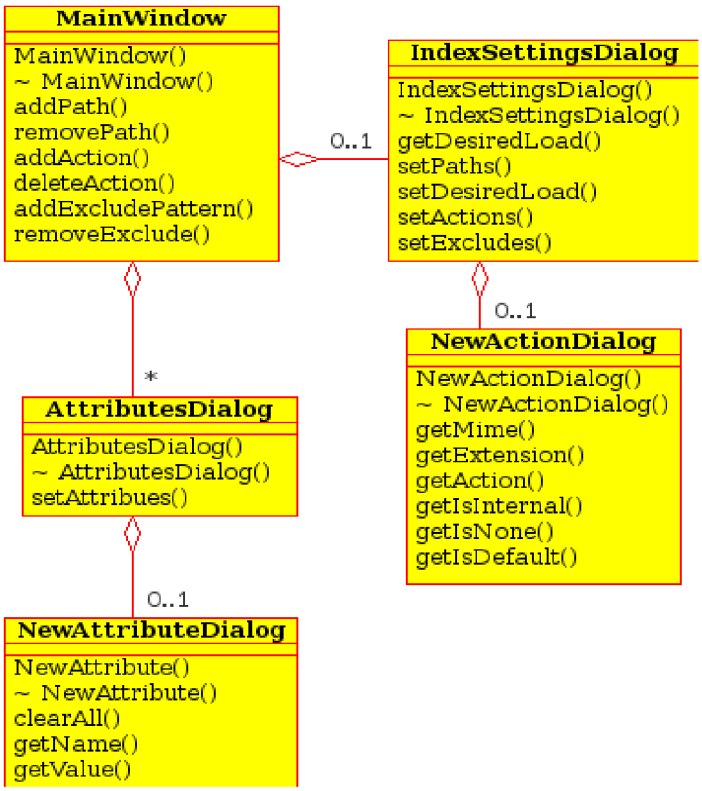
4.4 Klient

Jeho hlavnou úlohou je poskytovať služby indexera užívateľovi. Mal by prezentovať výsledky vyhľadávania, poskytovať rozhranie pre nastavovanie všetkých spomenutých vlastností indexera.

Tak ako vidíme na obrázku 4.4 klient sa skladá z niekoľkých dialógov. Hlavné okno *MainWindow* obsahuje riadok, kam bude užívateľ zadávať dotaz a tabuľku, v ktorej sa budú zobrazovať výsledky. V dolnej časti je stavový riadok, kde sa zobrazujú chybové hlášky alebo dôležité upozornenia. Vhod príde aj funkcia na automatické doplňovanie hľadaných výrazov. Užívateľ začne písať slovo a už po prvých písmenkách sa mu zobrazia možné dokončenia. Z menu si bude môcť užívateľ vybrať zobrazenie dialógu na úpravu nastavení indexovania, pozastaviť prípadne obnoviť pozastavené indexovanie. Navyše, po označení súborov v tabuľke pre výsledky bude možné pridávať atribúty, prípadne ich odstraňovať (dialóg *AttributesDialog*, resp. *NewAttributesDialog*).

V dialógu pre nastavovanie vlastností indexovania *IndexSettingsDialog* sa budú dať pridávať a odoberať cesty pre indexovanie, akcie pre jednotlivé typy súborov, regulárne výrazy pre vylúčenie súborov z procesu indexovania, prípadne iné. už spomenuté vlastnosti.

Z menu bude možné vyvolať dialógové okno s nápovedou popisujúcou základné ovládanie a dotazovací jazyk.



Obrázek 4.4: UML návrh klienta

Kapitola 5

Implementácia

Implementácia indexera je v súlade s návrhom. Využíval som pri nej voľne dostupné a slobodné knižnice. Pre modul Indexer som použil knižnicu *CLucene* [4], ktorá je vlastne len verziou populárnej knižnice Lucene pre jazyk C++ v ktorom je napísaná celá aplikácia. Objektový prístup bol zvolený aj z dôvodu väčšej prehľadnosti kódu, možnosti používania predvolených argumentov. Už v sekcií 4.3.2, kde sme rozoberali modul Reader extrahujúci dáta zo súboru, sme vlastne nepriamo vynútili použitie objektovo orientovanej paradigmy. Jazyk C++ som zvolil preto, že s ním mám väčšie skúsenosti a kompilátor pre tento jazyk je dostupný skoro v každej distribúcií (zatiaľ čo prostredie Javy nemusí, prípadne až po doinštalovaní).

5.1 Indexovanie v CLucene

Pri výbere knižnice pre vytváranie a správu indexu som sa rozhodoval medzi CLucene a Xapian [15]. Xapian je veľmi flexibilná knižnica, ktorá však neumožňuje použitie vlastného stemmera priamo. Museli by sme teda najskôr každý vstup do indexu ošetriť vlastnou funkciou pre stemming. táto funkcia by však musela pracovať v dvoch režimoch. V prvom by pracovala pri vkladaní do indexu a stemovala by každé slovo. Druhý režim by sa aktivoval pri vyhľadávaní a aplikoval by sa na užívateľský vstup. Tento by však bol komplikovanejší – museli by sme zachovávať syntax a vyhýbať sa operátorom dotazovacieho jazyka, pretože tie stemovať nechceme. Naproti tomu CLucene umožňuje zahrnúť vlastné stemming funkcie priamo do procesu indexácie a vyhľadávania. Nemusíme sa tak starať o parsovanie dotazu. Rozdieľ je aj v spôsobe ukladania indexu. CLucene používa jeden zložený súbor, zatiaľ čo Xapian používa B-stromy.

CLucene na najnižšej úrovni používa invertovaný index. Postupne vytvára segmenty, ktoré spája s hlavným indexom, ak prekročia tzv. *MergeFactor*. Ide teda o BSBI (2.3.1). Pre ohodnocovanie dokumentov používa variantu kosínusovej vzdialenosti vektorov (dokumentu a dotazu) a umožňuje zvýhodniť niektoré termy alebo celé dokumenty cez tzv. *boost*. Táto knižnica navyše reprezentuje dokument ako súbor polí, teda vlastností dokumentu. Tie sa definujú pre každý dokument zvlásť a dokonca aj obsah súboru je chápaný ako pole. Každé pole navyše obsahuje príznaky, na základe ktorých sa bude vykonávať nad daným poľom stemming, komprimovať dáta, prípadne sa ponechajú bez akejkoľvek zmeny a len sa uložia. Veľkou výhodou CLucene je systém, ktorým polia spracováva. K dokumentu definujeme analyzátor, ktorý vytvára obálku pre tokenizátory. Analyzátor môže pracovať nad všetkými poliami alebo len nad vybranými. Ku každému poľu tak môžeme vytvoriť zvlášť

tokenizátor. Ten potom rozdelí obsah poľa na jednotlivé tokeny a prípadne nad každým vykoná stemming. K tokenizátoru možno pridať jeden či viacero filtrov, ktoré prevedú rôzne transformácie, napr. skonvertujú písmená na malé či veľké, zmenia kódovanie a podobne. Za zmienku stojí filter *StopFilter*, ktorý odfiltruje stop slová. Takto skonštruovaný analyzátor sa potom použije aj pri parsovaní dotazu od užívateľa. Dotaz je navyše možné po vykonaní odfiltrovať inou sadou filtrov, ktoré pri indexovaní nemajú sémantický význam, napr. *RangeFilter* pre vybratie hodnôt z rozsahu.

5.2 Komunikácia

Jedným z prvých problémov, ktoré bolo treba vyriešiť bola komunikácia medzi klientom a indexerom. Zvažoval som viaceré možnosti: od naprogramovania vlastného protokolu nad zdieľanou pamäťou, cez v prostredí POSIX–ových operačných systémov bežné prostriedky pre medziprocesovú komunikáciu (*IPC*) ako pomenované rúry alebo sockety, až po špecializované knižnice. Nakoniec som vybral poslednú možnosť, konkrétne knižnica D-Bus, pretože som chcel v klientovi použiť grafickú knižnicu Qt, ktorá D-Bus priamo podporuje. Presnejšie, v indexeri som použil aplikačnú nadstavbu *D-Bus Glib bindings*[5], pretože samotná knižnica D-Bus je veľmi nízkoúrovňová. D-Bus je moderný spôsob pre IPC (*Inter-Process Communication*) a je nástupcom veľmi úspešného DCOP systému. Ako už názov napovedá, modeluje zbernicový prístup, keď umožňuje vymieňať správy medzi dvoma konkrétnymi uzlami, prípadne vyslať správu alebo signál, ktorý1 bude doručená všetkým ostatným (*broadcast*). Program ponúkajúci svoje služby sa zaregistruje a čaká na prichádzajúce žiadosti o ich poskytnutie. Po prijatí takejto žiadosti sa vytvorí zvlášť komunikačný kanál, cez ktorý sa už komunikuje bod – bod. Výhodou tohto systému je, že sa neposiela správa celému programu, ale priamo objektu, ktorý sme zaregistrovali. Tieto objekty však treba nejako adresovať. Inak než adresou do pamäte, pretože každý proces má svoj adresový priestor. D-Bus adresuje objekty cestami, tak ako ich poznáme zo súborových systémov. Každý objekt má programátorom definované rozhranie: sadu metód a signálov s argumentmi. Navyše, toto rozhranie možno zapísať do XML súboru a nechať si vygenerovať „kostru“ programu či už pre knižnicu Qt alebo Glib. Systém D-Bus je teda na vysokej úrovni, splňuje všetky naše požiadavky a jeho použitie je perpektívne, keďže obe najpoužívanejšie desktopové prostredia KDE a Gnome ho používajú, resp. na neho prechádzajú ([45]).

5.3 Indexer

Po spustení zistí, s akými parametrami bol spustený. V prípade, že bol argumentom zadaný alternatívny konfiguračný súbor, aplikujú sa nastavenia z neho. Štandardne sa totiž načítava súbor zo adresáru `${HOME}/.lds/config`, kde sa hľadá súbor `config`. Každý užívateľ má teda svoju sadu nastavení, ktoré však môžu byť totožné. Nie je tak problém zdieľať index s viacerými užívateľmi v rámci jednej pracovnej stanice, ak bude samozrejme tento umiestnený tak, že k nemu budú mať prístup viacerí užívatelia. Syntax konfiguračného súboru sa dá popísať jednoduchou bezkontextovou gramatikou, čo umožňuje použitie nástrojov `lex` a `bison`. Prvý je generátor lexikálnych analyzátorov, druhý vytvára LALR(1) parsery (Look-Ahead LR) [2], teda syntaktické analyzátory. Práve počas syntaktickej analýzy sa nastavujú príslušné položky v internej konfiguračnej štruktúre. Iste, vykonáva sa aj základná sémantická kontrola, napr. číslo pre maximálny load nemôže byť záporné.

Po spracovaní parametrov, s ktorými bol spustený program, skúsi zistiť, či už iná inštancia indexera nebeží. Je totiž zbytočné, aby bežali dva indexery naraz, keďže by dochádzalo k prekrytiu komunikačného rozhrania a skôr spustený proces indexera, by sa nedal ovládať. Ak už teda existuje iný proces indexera, na základe parametra sa rozhodne, či sa tento pokúsime ukončiť, alebo sa ukončí práve spúšťaný proces. Táto prax je bežná aj pri iných aplikáciách (napr. Beagle). S tým súvisí obsluha signálov. Aby sme aj pri násilnom ukončení, t.j. pri zaslaní signálu SIGTERM, SIGQUIT, SIGABRT alebo SIGINT zanechali index v konzistentnom stave, treba vyriešiť príjem týchto signálov. Po ich prijatí musí aplikácia samozrejme ihneď prerušiť činnosť a regulérne sa ukončiť. Tento prístup zaručuje, že po zaslaní týchto signálov nedôjde k čiastočnému zápisu do indexu, čo by ho znehodnotilo (hoci by pravdepodobne zostal konzistentný).

Teraz traverzujeme cez definované adresáre a porovnávame ich obsah s indexom. Tento krok je nutný preto, lebo sa niektorý súbor mohol medzi ukončením aplikácie a jej spustením zmeniť a nie sme schopní to zistiť cez systém inotify, keďže aplikácia nebola spustená. Navyše, práve pri tomto traverzovaní vytvárame jednotlivé monitory pre adresáre. Najefektívnejším spôsobom je porovnať čas modifikácie aktuálne spracovávaného súboru s časom uvedeným v indexe. Ak sa nezhodujú, treba súbor znovu naindexovať. Všimnime si, že čas v indexe môže byť v niektorých prípadoch väčší (meriame ho v počte sekúnd od Unixovej epochy – teda presne tak, ako sa meria čas v Linuxe) než čas získaný z volania `fstat()`. Stať sa to môže pri prepísaní súboru jeho staršou verziou (napr. v SVN repozitári).

Potom vytvoríme dve nové vlákna. Jedno pre samotný indexer, v ktorom bude cyklicky volaná metóda triedy Indexer `indexDocs()`. Podrobnejšie sa jej budeme venovať neskôr. Metóda môže skončiť vo veľa rôznych prípadoch, napr. fronta súborov sa vyprázdnila, prekročili sme limit maximálnej záťaže systému, prípadne užívateľ indexáciu pozastavil. Navyše, Indexer ako taký nemá informáciu o skutočnosti, že niektoré jeho metódy bežia v separátnom vlákne. Preto, ak uspíme Indexer „zvnútra“, jediný spôsob, ako ho zvnovu obnoviť je nastaviť príslušný indikátor a nechať z tohto vlákna zavolať metódu `indexDocs()`. Tento indikátor je testovaný na jej začiatku a určuje, či je možné pokračovať v indexácií alebo nie. Uspanie vlákna je riešené cez klasické synchronizačné prostriedky POSIX–ových vlákien – zámky, resp. *mutexy*. Pokus o zamknutie už zamknutého zámku spôsobí uspanie vlákna a následné odomknutie (iným vláknom než uspaným) spôsobí opätovné prebudenie. Tak vzniká pohľad na metódu `indexDocs` ako na kritickú sekciu, čo je podporené aj konštrukciou knižnice CLucene, ktorá nedovoľuje súčasné vykonávanie niektorých operácií nad indexom (napr. vkladanie a vyhľadávanie). Preto, aj vo vnútri Indexera je každý prístup k tomuto (a iným) zdieľaným prostriedkom ošetrený vzájomným vylúčením.

Potreba druhého vlákna vnikla zo spôsobu, akým sa pracuje so subsystémom inotify. Notifikácie o akciách sú zasielané cez desktriptory (celočíselné indexy do tabuľiek v rámci jadra), avšak čítanie môže byť blokujúce. Vykonáva sa totiž funkciou `read()`, presne tak, ako keď čítame dáta zo socketu alebo súboru. Čiže ak nie je žiadná nová udalosť, funkcia `read()` bude blokovať, až pokiaľ nejaká nenastane. Toto blokovanie si však v prípade jednovláknovej aplikácie nemôžeme dovoliť, avšak v prípade, že čakáme v inom vlákne, nevadí nám to. Iste, existuje aj funkcia `select()` prípadne `pselect()`, ktorá toto blokovanie obmedzuje na dopredu nastavenú hodnotu, po ktorej uplynutí skončia a je možné v programe ďalej pokračovať. Lenže práve počas tohto čakania (hoc akokoľvek malého) môžeme vykonávať inú užitočnú činnosť (optimalizovať index pre vyhľadávanie, vyhľadávať dokumenty, ...).

Po vytvorení vlákien, nastáva fáza pripojenia k zbernici D–Bus, teda zaregistrovanie komunikačného rozhrania Indexera a čakanie na príchodzie volania metód. Čakanie je pasívne,

v cykle, ktorý ukončíme len po zavolaní metódy pre ukončenie Indexera. Po jej zavolaní získame z Indexera nastavenia a tieto zapíšeme do konfiguračného súboru.

Pod rozhraním Indexera si teda môžeme predstaviť dve rozličné veci. Prvou je rozhranie triedy Indexera, teda na úrovni programovacieho jazyka. Druhou je rozhranie na úrovni medziprocesovej komunikácie s okolím cez D–Bus.

Indexer sa skladá z niekoľkých metód, ktoré zabezpečujú základnú funkčnosť. Či už ide o pridanie cesty do indexu (`addPath()`), odstránenie časti stromu adresárov alebo konkrétneho súboru z indexu (`removePath()`), indexovanie (`indexDocs()`), vyhľadávanie (`search()`) alebo manipuláciu nastavení. Každá táto metóda je prístupná aj cez rozhranie D–Bus, čo prináša komplikácie, pretože tieto metódy musia byť *thread-safe*, aby mohli byť volané asynchrónne z iných vlákien bez rizika uviaznutia, prípadne poškodenia konzistencie vnútorných štruktúr. To znamená, že každý prístup k vnútorným prostriedkom (index, fronta dokumentov čakajúcich na indexáciu, zoznam indexovaných ciest a iné) je ohraničený vzájomným vylúčením. Aby sa predišlo uviaznutiu, snažíme sa porušiť aspoň jednu z podmienok, ktoré k nemu vedú. Konkrétne podmienku o výlučnom vlastníctve viacerých prostriedkov. Žiadna metóda teda nevlastní naraz dva alebo viac prostriedkov, t.j. nezamkla viac než jeden zámok. Tento problém obzvlášť vystupuje pri knižnici CLucene, kedy nemožno naraz zapisovať a čítať index. Riešenie spočíva vo „vypožičiavaní“ si IndexReadera (pre čítanie) alebo IndexWritera (pre zapisovanie). Ak chce nejaká operácia pristúpiť k indexu, zavolá príslušnú metódu (`getIndexWriter()`, `getIndexReader()`), vykoná požadovanú operáciu a vráti prostriedok. Ten sa neuvoľní ihneď, pretože môže byť vzápätí zase požadovaný. Preto sa uvoľní až pred vytvorením prostriedku iného typu. V prípade, že požadujeme prostriedok, ktorý je však momentálne nedostupný, je naša požiadavka pozdržaná do doby, než sa prostriedok uvoľní.

Vlastné indexovanie dokumentov sa deje v metóde `indexDocs()`. Táto metóda, ako už bolo predostreté, na začiatku skontroluje, či môže niečo indexovať, teda či užívateľ nepozastavil indexáciu, alebo load systému nie je príliš veľký, alebo či vôbec je čo indexovať (fronta je neprázdna). Potom vyberie prvý prvok z fronty a prevedie ho do reprezentácie pre knižnicu CLucene. Tento proces v sebe obnáša určenie typu súboru, aby sme následne zvolili vhodný prístup a získali správne dáta. Rovnako získame metadáta. ktoré uložíme spolu s dokumentom. Takto získaný dokument sa pokusíme vložiť do indexu a v prípade úspechu tento dokument odoberieme z fronty. Skontrolujeme load systému (extrakcia obsahu je náročná operácia, ktorá vyťažuje procesor) a prejdeme na ďalšiu iteráciu. Knižnica CLucene však do toho vnáša jeden problém. Je ním spôsob vkladania dokumentu do indexu. Dokument sa v skutočnosti vloží do akéhosi pomocného indexu, no vyhľadávanie prebieha v hlavnom. Je to z dôvodu optimalizácie. Ak však chceme užívateľovi dať možnosť vyhľadávať aj „čerstvo“ naindexované dokumenty, musíme všetky pomocné indexy spojiť s hlavným. To je však náročná operácia (vyťažujúca najmä disk). Preto ju chceme volať nie po každom vložení dokumentu, ale len po vložení určitého počtu. Tento počet bol určený empiricky.

Získavanie obsahu dokumentov je založené na triede `Reader`, z ktorej sú odvodené jednotlivé triedy pre konkrétne formáty súborov. Obsah súboru je ešte pred samotným pridaním do indexu podrobený stemmingu. Aby sme však použili správny stemmer, teda odpovedajúci jazyku dokumentu, musíme tento najskôr určiť. To prebieha cez externý program *TextCat* v skriptovacom jazyku Perl [10], metódou hľadania N–gramov. Na identifikáciu postačuje pár kilobajtov textu. Zasa sú tu dve antagonistické požiadavky: veľkosť textu nepriaznivo ovplyvňuje množstvo času stráveného jeho spracovávaním a naproti tomu malé množštvo textu môže viesť k nesprávnej kategorizácií. Vzhľadom na používane externého

programu, zapíšeme kategorizovaný text do dočasného súboru, nad ktorým spustíme TextCat a zachytíme jeho výstup. Ten potom porovnávame a zvolíme príslušný stemmer.

Dokumenty v indexe sú vždy v kódovaní UTF-8, do ktorého ich musíme prípadne previesť cez funkciu `iconv()`. Predídeme tak nekonzistenciám. Navyše, na rozhraní s indexom je nutná konverzia z/do kódovania UCS-2. Na to som napísal špeciálne funkcie.

Potrebujeme tiež jednoznačný identifikátor súboru – aby sme vedeli určiť, či už súbor v indexe alebo fronte je, prípadne rôzne iné operácie, kedy pracujeme len s jedným konkrétnym súborom. Týmto identifikátorom síce môže byť číslo. Potom by sme však potrebovali jednotku, ktorá by dvom rozdielnym súborom (z hľadiska absolútnej cesty, nie obsahu) priradila rôzne čísla, a naopak dvom rovnakým súborom rovnaké čísla. A tak je týmto jednoznačným identifikátorom práve absolútna cesta. Tá samozrejme tiež nie je jedinečná, keďže v Linuxe môžeme pripojiť súborový systém v podstate do akéhokoľvek adresára. No na naše potreby postačí. Absolútnu cestu získame v dvoch krokoch. V prvom interpretujeme všetky zástupné znaky a symboly tak, ako ich poznáme z príkazového riadku (`wordexp()`) v druhom prevedieme takto získanú cestu na absolútnu (`realpath()`).

Pri implementácií podpory pre inotify a následnom testovaní som však veľmi rýchlo pocítil obmedzenie tohto systému. Je ním množstvo maximálnych možných monitorov pre užívateľa. Monitor sleduje udalosti len v rámci jedného adresára, dokonca ani nie podadresáre. Musíme ho preto vytvoriť pre každý adresár a podadresáre ktoré pri traverzovaní prejdeme. Toto obmedzenie je na jednej strane logické, keďže monitor spotrebováva pamäť jadra operačného systému, čo je vzácny zdroj, na druhej strane je prednastavená hodnota (8192 monitorov) dosť malá. Našťastie, sa dá táto hodnota zmeniť zápisom novej hodnoty do súboru `/proc/sys/filesystem/inotify/max_user_watches`. Jej nastavovanie je však veľmi citlivá operácia, ktorá je tak radšej prenechaná na užívateľa, pretože jej dynamické zväčšovanie podľa potrieb Indexera by mohlo viesť k zahlteniu systému, odhliadnúc od faktu, že tak môže učiniť len administrátor systému.

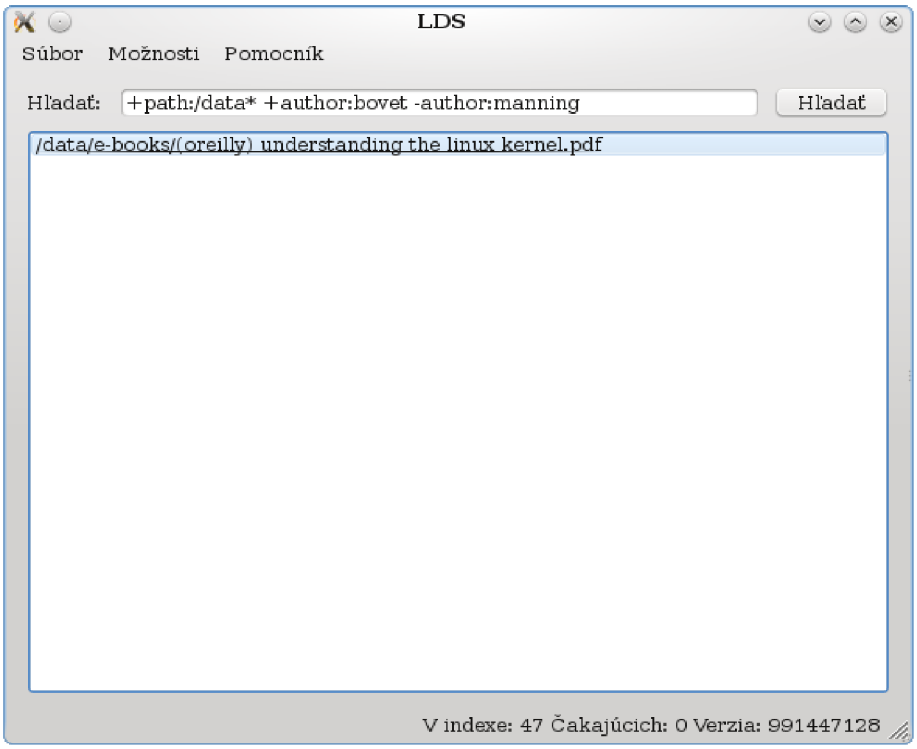
5.4 Klient

Klient je naprogramovaný tiež v jazyku C++ a pre grafické rozhranie využíva knižnicu Qt4, ktorá navyše poskytuje aj rozhranie pre komunikáciu cez D–Bus. Hlavné okno aplikácie, trieda `MainWindow`, je, v zhode s návrhom, rozdelené na niekoľko častí, tak ako to vidíme na obrázku 5.1.

Samozrejmosťou je automatické doplňovanie hľadaných výrazov na základe termov v indexe. K tomu sa dá využiť priamo knižnica Qt4, ktorá má túto možnosť. Jediné, čo musí programátor zabezpečiť, je dodať dáta vhodné na doplnenia k danému začiatku slova.

Pri implementácií je využívané rozšírenie jazyka C++ definované knižnicou Qt o tzv. *Meta Object Compiler*. Sú to špeciálne direktívy preprocesora, z ktorých špeciálny program *moc* vytvorí dodatočné zdrojové texty. Tento prístup sa využíva najmä v architektúre Signal – Slot, typickej pre knižnicu Qt. Táto umožňuje jednoduchým spôsobom reagovať na udalosti (signály), napr. stlačenie tlačidla emituje signál `activated()`. A ak tento signál privedieme práve pomocou spomínaného rozšírenia na niektorý zo slotov, čo sú vlastne metódy objektov, daná akcia sa spustí. Takto môžeme jednoduchým spôsobom definovať akcie spúšťané na akýkoľvek podnet. Oproti štandardne používaným *callback* funkciám, dochádza tu k typovej konzole a sloty so signálmi nie sú tak silne previazané ako callback funkcia s funkciou, ktorá ju volá.

Každý prvok má popis, ktorý sa zobrazí po podržaní kurzora myši nad ním (*toolTip*). Ak teda užívateľ nevie, čo môže ten či onen prvok meniť, akú ma funkciu, prípadne aká je



Obrázek 5.1: Hlavné okno klienta.

syntax dotazovacieho jazyka, stačí ak nad daným prvkom podrží kurzor. Vzápätí sa zobrazí text, vysvetľujúci jeho funkciu.

Vzhľadom na rozdelenie indexer – klient je nemožné s klientom pracovať, pokiaľ nebeží proces indexera, pretože klient samotný nevie nič o indexovaní. On len komunikuje s indexerom a zobrazuje výsledky v užívateľsky príjemnej forme. Preto každú sekundu kontroluje, či je na zbernici D-Bus dostupný indexer, pretože len vtedy majú niektoré jeho prvky zmysel. Ak indexer nebeží, poskytneme užívateľovi možnosť ho spustiť, prípadne, ak beží, ukončiť.

Klient sa skladá z niekoľkých tried, každé okno má svoju vlastnú, odvodenú od triedy grafického elementu, ktorý reprezentuje. Tak je okno pre pridávanie odvodené od triedy `QDialog` a trieda pre hlavné okno z triedy `QMainWindow`. Jednotlivé grafické prvky sú dotvárané dynamicky, t.j. počas kompilácie je z príslušného XML súbora vygenerovaný kód, ktorý sa vykoná v konštruktore danej triedy.

5.5 Porovnanie s ostatnými aplikáciami

Už pri návrhu a implementácií Linux Desktop Search – LDS, som bral do úvahy niektoré nedostatky porovnávaných produktov (kapitola 3) a snažil som sa im vyhnúť. Moja aplikácia tak umožňuje nastavovanie priority, čo naopak chýba Beagle-u alebo GDS. Pozastavenie indexácie pri istej záťaži systému je naopak vlastnosť, ktorú nemal žiaden z porovnávaných produktov. Podpora regulárnych výrazov je podobné ako v Beagle, teda len hviezdička zastupujúca akýkoľvek reťazec. S Beagle-om má LDS spoločných viacero vecí – vyhľadávanie v metadátach (napr. všetky dokumenty od uvedeného autora), vylúčenie slov, ktoré

výsledky nemôžu obsahovať (*-slovo*), hľadať v konkrétnej ceste (*path:<cesta>*).

Rozdielny od všetkých je dotazovací jazyk. Ten umožňuje vyhľadávanie presných fráz (uvedených v dvojitých úvodzovkách), poddotazy (uvedené v okrúhlych zátvorkách), používanie logických operátorov: pre AND pridáme pred oba operandy znak +, pre OR netreba operandy nijako meniť. Výraz *slovo1 slovo2* sa teda chápe ako *slovo1 OR slovo2* a *+slovo1 +slovo2* ako *slovo1 AND slovo2*. Pre operátor NOT, stačí pred slovo dopísať znak -. Samozrejme, tieto operátory možno použiť aj pred poddotazmi. Vtedy ich dopíšeme pred okrúhlu zátvorku.

Oproti prepracovanému XPC však možno LDS rozšíriť o podporu nových formátov a to dokonca priamo z grafického prostredia, no indexácia nemôže byť naplánovaná. Avšak tak ako pri iných programoch, aj tu je možné indexáciu manuálne spustiť alebo pozastaviť a je sledovaná zmena súborov.

Z hľadiska presnosti výsledkov je LDS lepší než GDS a v niektorých prípadoch dokonca lepší než Beagle či MDS. Je to spôsobené práve dotazovacím jazykom, keďže GDS má veľmi strohý jazyk a v podstate podporuje len vyhľadávanie fráz. Grafy precision–recall sú znázornené na obrázku 5.5, uzlové body sú zachytene v tabuľke 5.1.

recall	**Beagle**	**GDS**	**MDS**	**XPC**	**LDS**
0.00	0.6690	0.5144	0.5695	0.5695	0.6975
0.10	0.5157	0.2860	0.5219	0.5112	0.5883
0.20	0.4427	0.2860	0.4788	0.4345	0.4769
0.30	0.3660	0.2860	0.2843	0.3174	0.3102
0.40	0.3371	0.2251	0.2776	0.2529	0.2658
0.50	0.2835	0.2251	0.2062	0.2529	0.1908
0.60	0.2639	0.1522	0.1961	0.1961	0.1807
0.70	0.1149	0.1340	0.1206	0.1206	0.1539
0.80	0.1000	0.0000	0.1000	0.1000	0.1000
0.90	0.1000	0.0000	0.1000	0.1000	0.1000
1.00	0.1000	0.0000	0.1000	0.1000	0.1000

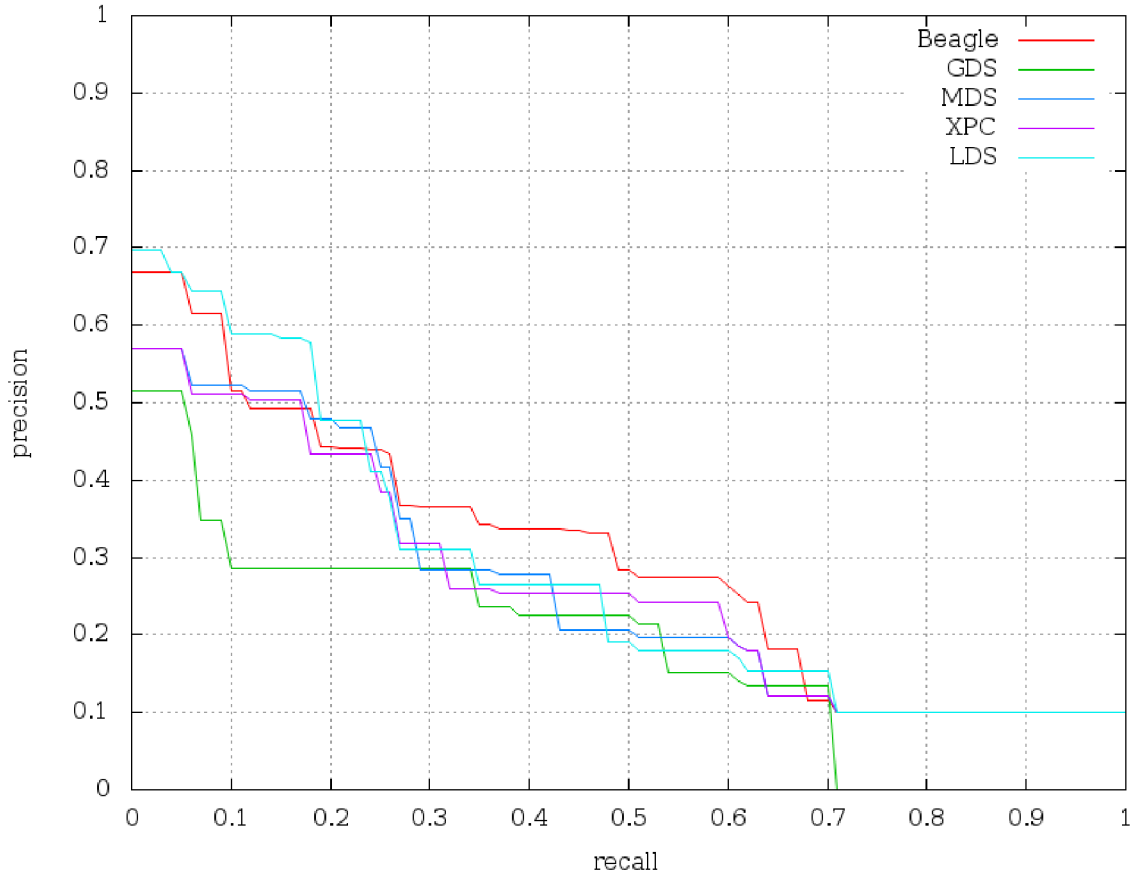
Tabulka 5.1: Precision-recall pre rôzne úrovne recall. Vyššie číslo (vyššia precision) je lepšia. Údaje pre Beagle, GDS, MDS a XPC prebraté z tabuľky 3.4

Rýchlosť indexovania je tiež veľmi dobrá. Užívateľovi poskytuje dostatočnú kontrolu, hoci by užívateľ pravdepodobne privítal možnosť naplánovania indexácie. Porovnanie rýchlosti indexovania je v tabuľke 5.2 a je rozšírením tabuľky 3.2.

č. merania	Beagle	GDS	XPC	LDS
1	0:28	37:30	2:00	0:27
2	0:31	36:56	1:56	0:29
3	0:30	37:23	1:57	0:30
priemer	**0:29.67**	**37:16.33**	**1:57.56**	**0:28.67**

Tabulka 5.2: Porovnanie rýchlosti indexácie. Údaje sú v tvare [mm:ss].

Ďalšie metriky porovnávajúce LDS s ostatnými vyhľadávačmi môžeme vidieť v tabuľke 5.3. Z nej je zrejmé, že z hľadiska metrík je LDS lepší než GDS a dokonca čiastočne lepší než MDS alebo XPC. V čom však LDS silne zaostáva za všetkými vyhľadávačmi je počet priamo podporovaných formátov. Podporuje totiž len textové súbory, XML a z neho



odvodené súbory (napr. HTML), PDF a ODF. Pri ďalšom vývoji LDS sa treba zamerať práve na tento aspekt a prípadne ho integorvať do desktopového prostredia tak, že by sa napríklad dalo vyhľadávacie okno vyvolať globálnou klávesovou skratkou.

Napriek rovnakým knižniciam – CLucene, resp. Lucene.Net – použitým na indexovanie v Beagle a LDS, dáva LDS v niektorých prípadoch lepšie výsledky, konkrétne vyššiu precision pri nízkej recall. Tento fakt je daný bohatším dotazovacím jazykom, v ktorom môžeme presnejšie zadať kritéria pre hľadaný dokument a nepoužívaním stop slov (na rozdiel od Beagle). V iných prípadoch, pri vyššej recall, dosahuje Beagle vyššiu precision pre lepší stemming než použitý Porter v LDS. V Beagle je totiž použitý Snowball, ktorý obsahuje vylepšenú verziu tohto algoritmu, označenú ako *Porter2*. Ten dokáže stemovať slová presnejšie, čo vyústi vo vyššiu precision. Hoci sú obe knižnice rovnaké z hľadiska použitých algoritmov, nie sú identické. Váhovanie v Lucene.Net berie štvorec idf, zatiaľ čo v CLucene sa idf neumocňuje. Tento fakt priamo vplýva na ohodnotenie dokumentov a teda aj na výslednú precision–recall krivku.

	Beagle	**GDS**	**MDS**	**XPC**	**LDS**
num_ret	220	197	215	229	123
num_rel	127	112	124	124	124
num_rel_ret ↑	57	32	47	50	44
MAP ↑	0.2783	0.1575	0.2418	0.2407	0.2564
R-precision ↑	0.3313	0.2158	0.3067	0.3012	0.3345

Tabulka 5.3: Výsledky porovnávania vyhľadávačov. Popisky viď tabuľka 3.3

Kapitola 6

Záver

V tejto dipolomovej práci som vytvoril modulárny IR systém. Úspešne som ho testoval na menších či väčších kolekciách. Jedinou nevýhodou bolo, že som metriky musel merať na malom korpuse, keďže väčšie korpusy buď neobsahujú oficiálné výsledky relevancie alebo sú tieto platené. Práve tieto výsledky spolu s prislúchajúcimi dotazmi potrebujeme, aby sme mohli zmerať spomínané metriky. Preto som všetky kvalitatívne merania prevádzal nad voľne dostupným korpusom CACM [3].

Z porovnávania v sekcií 5.5 vyšiel LDS obstojne, ak uvážime že porovnávané produkty vyvíjajú tímy odborníkov na plný úväzok. LDS má bohatý dotazovací jazyk, avšak ten by sa dal tiež ešte rozšíriť. Napríklad o podporu rozsahov čísel, pretože teraz sa čísla uvažujú len ako znakové reťazce. Tento fakt je však dôsledkom použitia knižnice CLucene, ktorá v použitej verzií nedokáže pracovať s inými typmi dát než sú reťazce znakov. Avšak v novej verzií je naplánované pridanie podpory aj pre iné typy, najmä dátumy. Tá je však dostupná len cez systém správy verzií GIT a je zatiaľ označená ako nestabilná.

Ďalším zlepšením by bolo zobrazovanie náhľadov súborov a podpora pre ich otváranie v príslušnom programe priamo z klientského programu. Znížil by sa tak čas potrebný na dohľadanie informácie s dokumente, čím by vzrástla efektivita práce s týmto programom. Za zváženie stojí aj vytvorenie klienta nad inou sadou knižníc pre grafické rozhranie než je Qt, napr. pre GTK+, či OpenMotif, ba dokonca vytvorenie jednoduchého programu pre vyhľadávanie priamo z príkazového riadku, čo by uľahčilo integrovanie do skriptov.

Z hľadiska vnútorného fungovania, by sa do aplikácie mohli doplniť stemre pre ostatné jazyky, spomínaný Snowball obsahuje stemre pre 18 indoeurópskych jazykov, prípadne prepracovať slovenský stemmer tak, aby pracoval so slovníkom (sekcia 2.2.4), alebo ho úplne nahradiť lemmatzérom. Ak by sme chceli zvyšovať efektivitu, mali by sme sa zamerať na kategorizáciu textu, ktorá je momentálne vykonávaná cez externý skript v jazyku Python. Tento by sa dal previesť do jazyka C a následne zakomponovať do aplikácie, takže by už táto nebola závislá na interprete jazyka Python a nemusela by sa zakaždým spúšťať nová inštancia príkazového riadku.

Veľmi zaujímavým rozšírením aplikácie by bola schopnosť zodpovedať dotazy v prirodzenom jazyku. Toto však už spadá do iného odboru v rámci NLP.

Literatura

[1] Beagle. http://beagle-project.org/, domovská stránka.

[2] Bison - GNU parser generator. Domovská stránka.
URL http://www.gnu.org/software/bison/

[3] CACM corpus. http://www.search-engines-book.com/collections/, domovská stránka.

[4] CLucene. Domovská stránka.
URL http://clucene.sourceforge.net/

[5] D-Bus GLib bindings - Reference Manual. Domovská stránka.
URL http://library.gnome.org/devel/dbus-glib/unstable/

[6] GNU General Public License, version 2.
URL http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

[7] Google Desktop Search. http://desktop.google.com/, domovská stránka.

[8] Lucene. http://lucene.apache.org/, domovská stránka.

[9] Lucene.Net. http://lucene.apache.org/lucene.net, domovská stránka.

[10] TextCat. Domovská stránka.
URL http://www.let.rug.nl/~vannoord/TextCat/

[11] TrecEval. http://trec.nist.gov/trec_eval/, domovská stránka.

[12] Windows Search. http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx, domovská stránka.

[13] Windows Search: Searchable File Types. http://www.microsoft.com/windows/products/winfamily/desktopsearch/technicalresources/filetypes.mspx.

[14] X1 Email Search, Application Search and eDiscovery. http://www.x1.com/, domovská stránka.

[15] Xapian. Domovská stránka.
URL http://xapian.org/

[16] Yahoo Licenses X1 Desktop Search Technology. http://www.lawlibtech.com/2004/12/yahoo-licenses-x1-desktop-search-technology.html, lawLibTech.

[17] The Expat XML Parser. 2010, domovská stránka.
URL http://expat.sourceforge.net/

[18] Manuál RHEL. 2010.
URL http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/en-US/Reference_Guide/s2-proc-loadavg.html

[19] Poppler. 2010, domovská stránka.
URL http://poppler.freedesktop.org/

[20] Slovenský národný korpus. http://korpus.juls.savba.sk/index.sk.html, 2010.

[21] Snowball. 2010, domovská stránka.
URL http://snowball.tartarus.org/

[22] Bancroft, B.: Word Count. http://www.blairbancroft.com/pdf/WordCount.pdf.

[23] Bialecki, A.: Stempel - Algorithmic Stemmer for Polish Language.
http://getopt.org/stempel/, domovská stránka.

[24] Clarke, C. L. A.; Cormack, G. V.: Dynamic Inverted Indexes for a Distributed Full-Text Retrieval System. Technická zpráva, 1995.

[25] Dvořák, T.: Metody návrhu efektivních algoritmů, složitost algoritmů. 2005, prednášky.
URL http://ksvi.mff.cuni.cz/~dvorak/vyuka/UIN009/

[26] Galamboš, L.: Egothor. http://www.egothor.org/~galambos/.

[27] Gilleland, M.: Levenshtein Distance, in Three Flavors. 19. května 2010.
URL http://www.merriampark.com/ld.htm

[28] Grossman; Frieder; Goharian: Database Systems II (lectures).
http://www.eng.auburn.edu/~gilbert/Comp7120/Concept-50/.

[29] Grossman, D. D.: Notes on Trec Eval. http://www.ir.iit.edu/~dagr/cs529/files/project_files/trec_eval_desc.htm.

[30] Humay, P.: Stemmer slovenského jazyka.
http://vi.ikt.ui.sav.sk/index.php?title=Projekty/Projekty_2008%2F%2F2009/Pavol_Humay_-_Stemmer_slovensk%C3%A9ho_jazyka_%285%29, 2008, semestrálny projekt.

[31] doc. Ing. Ján Paralič Ph.D.: Manažment znalostí.
http://people.tuke.sk/jan.paralic/mz.html, 2010, prednášky.

[32] prof. Javed A. Aslam: Information Retrieval.
http://www.ccs.neu.edu/home/jaa/CSG339.06F/Lectures/, 2006, prednášky.

[33] Knuth, D. E.: *The art of computer programming, volume 1 (3rd ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[34] Knuth, D. E.: *The art of computer programming, volume 3 (2nd ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[35] Krajči, S.; Novotný, R.; Turlíková, L.: Použitie lematizácie vo fulltextovom vyhľadávaní v slovenských dokumentoch. In *2nd Workshop on Intelligent and Knowledge oriented Technologies*, editace F. Babič; J. Paralič, FEI TU KE, 2007, ISBN 978-80-89284-10-8, s. 147–152.
URL http://web.tuke.sk/fei-cit/wikt2007

[36] Lampa, P.: Pokročilé operační systémy. 2010, prednášky.
URL https://www.fit.vutbr.cz/study/courses/POS/private/slides/

[37] Landauer, T. K.; Laham, D.: Learning Human-like Knowledge by Singular Value Decomposition: A Progress Report. 1998.
URL http://www.pearsonkt.com/papers/nips1998.pdf

[38] LaPaugh, A.: Information Retrieval, Discovery, and Delivery. 2009, prednášky.
URL http://www.cs.princeton.edu/courses/archive/spring09/cos435/schedule.html

[39] Love, R.: Kernel Korner. 2005, linux Journal.
URL http://www.linuxjournal.com/article/8478

[40] Lowe, M.: Evaluation of desktop search applications.
http://www.kalio.info/Desktop_Search_Comparison/, 2006, online.

[41] Manning, C. D.; Schtitze, H.: *Foundations of Statistical Natural Language Processing*. MIT Press, 1999, iSBN 0-262-13360-1.

[42] Manning, C. D.; Schtitze, H.; Raghavan, P.: *Introduction to Information Retrieval*. Cambridge University Press, 2008, iSBN 0521865719.

[43] O'Neill, E. T.; Lavoie, B. F.; Bennett, R.: Trends in the Evolution of the Public Web. *D-Lib Magazine*, ročník 9, č. 4, April 2003.

[44] Porter, M.: An algorithm for suffix stripping. *Program*, , č. 14, Júl 1980: s. 130–137.
URL http://tartarus.org/~martin/PorterStemmer/def.txt

[45] Vyskočil, M.: Co přináší KDE 4. 2007, abcLinuxu.
URL http://www.abclinuxu.cz/clanky/recenze/co-prinasi-kde-4-alfaverze-porty-a-d-bus

[46] Zoulas, C.: Fine Free File Command. 2010, domovská stránka.
URL http://www.darwinsys.com/file/

Příloha A

Manuál

Indexer akceptuje tieto prepínače s ktorými môže byť spustený:

-c, --config SÚBOR alternatívny konfiguračný súbor. Štandardne sa načítava konfigurácia
z `/.lds/config/config.txt`

-f, --foreground beh aplikácie na popredí.

-b, --background beh aplikácie na pozadí. Táto možnosť sa berie ako predvolená.

-v, --verbose Výpis práve vykonávanej aktivity do konzoly na štandardný výstup.

-e, --error SÚBOR Výpis chybových hlásení do tohto súboru, v prípade, ak je SÚBOR rovný znaku '-', vypisuje sa na štandardný chybový výstup.

-r, --replace Ak už beží nejaká inštancia indexovacieho démona, pokúsi sa ju ukončiť.

-q, --quiet Potlačí výpis informačného banera pri spustení.

-h, --help, --usage Vypíše pomocníka k prepínačom.

Ak sa pri spustení zistí, že už beží iná inštancia indexovacieho démona a program nebol spustený s prepínačom '-r', program sa ukončí.

Program je distribuovaný pod licenciou GNU GPL verzia 2 [6]. Umožňujú to aj použité knižnice, ktoré sú distribuované tiež s licenciou GPL alebo LGPL. Program vyžaduje na svoj beh tieto knižnice a programy:

- CMake (2.6)
- pthreads
- Qt4 (4.6.2)
- libmagic (5.3)
- libexpat (2.0.1)
- dbus-glib (0.82)
- poppler (0.12.3)

- libzip (0.9)
- clucene (0.9.21b)

Počas kompilácie sú potrebné tieto aplikácie:

- lex (2.5.35)
- bison (2.4.1)

Pre určovanie jazyka dokumentu sú potrebné tieto programy:

- TextCat (1.10)
- python (2.6.4)

V zátvorke je uvedená verzia, s ktorou bol program vyvíjaný. Všetky tieto sú obsiahnuté na priloženom CD-ROMe. Program obsluhuje niektoré signály, takže po ich prijatí dôjde ku korektnému ukončeniu aplikácie.

Příloha B

Konfiguračný súbor

Jeho syntax sa dá popísať bezkontextovou gramatikou s ľavou rekurziou. Graf jej pravidiel možno nájsť na stránkach http://www.stud.fit.vutbr.cz/~xprivo00/dip/ (rovnako ako programovú dokumentáciu, zdrojové texty, doplňujúce obrázky, ktoré nie sú uvedené v tejto práci, ...). Skladá sa z definícií jednotlivých vlastností a z prípadných komentárov. Komentáre sú všetky tie riadky, ktoré začínajú znakom mriežky '#' a pokračujú až do konca riadku. Definícia konkrétnej vlastnosti má syntax:

```
premenná = {hodnota [, hodnota, hodnota, ...]};
```

V prípade, že `hodnota` je typu reťazec znakov, je táto uzatvorená v dvojitých úvodzovkách. V prípade, že akési nastavenie nechceme uplatniť, resp. nechceme používať, stačí neuviesť hodnotu a ponechať tak zložené zátvorky prázdne. Ak však naopak, chceme uviesť viacero hodnôt, oddelíme tieto čiarkou. Jedinú výnimku tvorí premenná `filetypes`, kde pre lepšiu čitateľnosť oddeľujeme jednotlivé entity bodkočiarkou ';'.

Nastavovať sa dajú tieto vlastnosti:

index umiestnenie indexu

paths cesty, ktoré chceme zahrnúť do indexu

max_load hranica záťaže systému, pri ktorej sa indexovanie pozastaví

priority priorita indexovacieho procesu

index_hidden indexovanie skrytých súborov a/alebo adresárov.

exclude vylúčenie súborov a/alebo adresárov z indexu na základe mena ktoré spĺňa vzor

filetypes akcie vykonávané nad nepodporovanými typmi súborov pre získanie ich obsahu

langid_path cesta k text_cat [10] skriptu

Aby sa predišlo nejednoznačnostiam, odporúča sa uvádzať cesty v absolútnom tvare. Program sa síce pokúsi previesť relatívne cesty na absolútne, ale tento prevod nemusí vrátiť želaný výsledok.

Keďže LDS umožňuje používať externé programy pre extrahovanie obsahu, je dokonca možné použiť nástroj `beagle-extract-content`, ktorý je súčasťou Beagle a získava z dokumentov analyzovaný a tokenizovaný text. Stačí, ak prednastavenému pravidlu asociujeme akciu:

```
beagle-extract-content --analyze --tokenize \{}; | \
  grep -A 2147483647 "Content:" | tail -n +2 | head -n -4
```

Získame tak rovnakú podporu formátov súborov ako má Beagle. Nevýhodou tohto riešenia je jeho závislosť na Beagle.

Príklad konfiguračného súboru:

```
index {"~/.lds/index_dir"};

paths {"/usr/share/man", "/usr/share/doc", "/usr/portage"};

max_load {0.5};

priority {0};

index_hidden {1};

langid_path {"/usr/local/text_cat/text_cat"};

filetypes {
        default action=internal ;
        mime="application/x-bzip2" action="bzcat \{};" ;
};

exclude {"*.swp", "*tmp*", ".svn"};
```