VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYHLEDÁVÁNÍ VE VIDEU

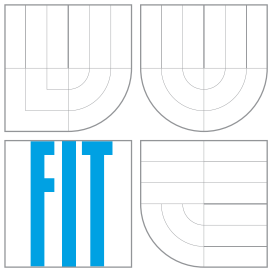
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR ČERNÝ

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYHLEDÁVÁNÍ VE VIDEU

SEARCHING IN VIDEO

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR ČERNÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR CHMELAŘ

BRNO 2012

Abstrakt

Tato práce shrnuje základní teorii týkající se vyhledávání informací, základy z oblasti relačního modelu dat a problematiky indexace dat v relačních databázových systémech. Práce se dále zabývá problematikou vyhledávání v multimediálních datech. Zahrnuje popisy základních principů automatické extrakce rysů multimediálního obsahu a indexace multidimenzionálních dat. Praktická část této práce se zabývá návrhem a implementací řešení, které má za úkol zvýšit efektivitu dotazů na podobnost multidimenzionálních vektorů rysů, které popisují jednotlivá videa. Závěr práce je věnován experimenty nad tímto řešením.

Abstract

This thesis summarizes information retrieval theory, relational model basic and focuses on data indexing in relational database systems. Thesis focuses on multimedia data searching. It includes description of automatic multimedia data content extraction and multimedia data indexing. Practical part discusses design and solution implementation for improving query effectivity for multidimensional vector similarit which describes multimedia data. Thesis final part discusses experiments with this solution.

Klíčová slova

Vyhledávání informací, Vyhledávání v multimediálních datech, Extrakce rysů, Relační model dat, Indexace dat, B-strom, Vyhledávací modely, Indexace multidimenzionálních dat, Analýza hlavních komponent, PostgreSQL, GSL, SPI, GiST, KNN GiST

Keywords

Information retrieval, Multimedial data retrieval, Features extraction, Relational data model, Data indexing, B-Tree, IR models, Multidimensional data indexing, Principal Component Analysis, PostgreSQL, GSL, SPI, GiST, KNN GiST

Citace

Petr Černý: Vyhledávání ve videu, diplomová práce, Brno, FIT VUT v Brně, 2012

Vyhledávání ve videu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Petra Chmelaře.

.....
Petr Černý
22. května 2012

Poděkování

Tímto bych rád poděkoval vedoucímu práce Ing. Petru Chmelařovi za jeho vztřícnost a pomoc při vypracovávání této práce.

© Petr Černý, 2012.
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

Kapitola 1

Úvod

Tento dokument vznikl jako technická zpráva k diplomové práci. Práce se zabývá problematikou vyhledávání informací, především se orientuje na problematiku vyhledávání v multimediálních datech a videu.

Jedním z cílů této práce je shrnout současné poznatky k problematice vyhledávání informací v obecných a multimediálních datech, používané principy, modely, struktury a metody a identifikovat hlavní problémy spojené s vyhledáváním informací, především pak v multimediálních datech.

Hlavním cílem práce je navhrnout a implementovat řešení vedoucí k odstranění, či zmírnění problému nepřiměřené doby provádění dotazů na podobnost multimediálních dat, který je způsoben visoce dimenzionálním charakterem popisných vektorů, na základě kterých se vyhledává. Tento cíl pak udává směr celé práce.

Druhá až čtvrtá kapitola této práce obsahuje shrnutí informací k problemtice vyhledávání informací a vyhledávání v multimediálních datech a videu. Tyto kapitoly pak tvoří teoretický základ pro kapitoly zaměřené na návrh a implementaci daného řešení. Pátá až sedmá kapitola jsou pak zaměřeny na vývoj řešení pro zvýšení efektivity provádění dotazů na podobnost multimediálních dat a provádění experimentů a zjištění vlastností daného řešení.

Druhá kapitola se zabývá problematikou vyhledávání informací jako celku. Definuje tradiční vyhledávací modely, jakými jsou klasický booleovský, pravděpodobnostní a vektorový model. Dále zahrnuje popis způsobu, jakým obvykle systémy pro vyhledávání informací fungují. V této kapitole jsou zmíněny myšlenky a metody používané při fulltextovém vyhledávání dokumentů, ze kterých pak vycházejí i některé metody používané při vyhledávání dokumentů multimediálních. Kapitola definuje obecné pojmy týkající se vyhledávání informací.

Třetí kapitola navazuje na základ položený v druhé kapitole a uvádí principy relačních databázových systémů, používané při vyhledávání v obecně libovolných datech. Značná část kapitoly je pak věnována indexačním strukturám, převážně pro skalární datové typy, které mají při vyhledávání dat v databázi velký vliv na dobu provádění dotazů.

Ve čtvrté kapitole se již práce úzce zaměřuje na podporu, principy, nástroje a metody pro vyhledávání v multimediálních datech, především ve vizuálních datech a videu. Poskytuje okrajový přehled o používaných metodách pro extrakci rysů z multimediálních dat a nastiňuje způsob využití těchto rysů při vyhledávání. Kapitola navíc obsahuje popis základních indexačních struktur používaných pro indexování multidimenzionálních dat, čímž rozšiřuje poznatky o indexování nabyté v kapitole druhé. Cílem této kapitoly je uvést do problematiky vyhledávání multimediálních dat a nastínit různé strukturované reprezentace

obecně nestrukturovaných multimediálních dat.

V páté kapitole bude diskutován návrh nástroje pro zefektivnění stávajícího řešení, které je po několik let vyvíjené na Fakultě informačních technologií Vysokého učení technického v Brně. Tato kapitola stručně popisuje současný stav tohoto systému, data využívaná pro vývoj a testování. V další části kapitoly je pak popsán návrh řešení, umožňující zkrácení délky provádění dotazů pomocí snížení dimenzionality vektorů rysů a použití vhodné indexační struktury. Významná část kapitoly je věnována analýze hlavních komponent, která je ve zbytku práce používána pro snížení dimenzionality dat.

Šestá kapitola obsahuje popis postupů, nástrojů a problémů spojených s implementaci daného řešení. Kapitola obsahuje stručné seznámení s vývojem uložených funkcí pro databázový systém PostgreSQL v jazyce C a použití vědecké knihovny GSL. Dále popisuje implementační potíže a postupy, které byly aplikovány. Na závěr kapitoly je zmíněno rozšíření operací datového typu `cube` tak, aby umožnil podobnostní vyhledávání s využitím KNN GiST.

Sedmá kapitola obsahuje definice metrik, které byly experimentálně sledovány, postup při provádění experimentů a zjištěné vlastnosti tohoto řešení.

Kapitola 2

Problematika vyhledávání informací

Tato kapitola se zabývá vyhledáváním informací jako disciplínou. Popisuje základní modely vyhledávání informací, postup při extrakci index termů a metodu fulltextového vyhledávání.

2.1 Motivace

Vyhledávání informací je velice stará oblast lidského bádání. S nástupem počítačů se však tento obor významně rozvinul a jeho rozvoj stále probíhá. Množství sbíraných dat se stále zvyšuje, proto je třeba tato data rozumným způsobem organizovat tak, aby se v nich dalo efektivně vyhledávat.

V dnešní době jsou finanční instituce (produkční data), akademická prostředí (data o studiu, vědecké poznatky, ...) či běžní lidé (fotografie, video, zábavné, vzdělávací a jiné materiály) doslova zaplaveni obrovským množstvím dat a informací. Současně je v dnešní době velice důležité umět s těmito informacemi efektivně pracovat. Navzdory tomu je stále těžší se v tomto nepřeberném množství dat vyznat a vyhledávat v nich informace, které jsou pro nás právě důležité.

Z těchto důvodů je vyvíjen velký tlak na tvůrce databázových systémů, on-line vyhledávačů, knihovních a jiných systémů. Tlak je směřován ke zdokonalování těchto systémů v oblasti efektivity, přesnosti a rychlosti vyhledávání informací ve velkých objemech dat.

V dnešní době je dostupné veliké množství vyhledávacích nástrojů, ať již obecně (například webové vyhledávače) či úzce zaměřených (např. knihovní systémy nebo katalogy zboží). Velice často jsme při vyhledávání informací zaplaveni obrovským množstvím výsledků, které jsou pro nás buď naprosto irelevantní nebo nedůvěryhodné. V takových záplavách informací se pak velice snadno ztrácí námi požadované informace. Tento fakt úzce souvisí se skutečností, že pro automatizované systémy je často velice složité určit relevantnost dat. To je zapříčiněno tím, že informacím obsažených v těchto datech prostě nerozumí, jelikož jejich formát není vhodný pro strojové pochopení obsahu, ale umožňuje efektivní interpretaci člověkem a jeho smyslovým vnímáním.

Dalším významným problémem při vyhledávání informací je skutečnost, že v informačních systémech máme často obrovské množství dat, ze kterých je ovšem velice obtížné získat nějaké netriviální znalosti. Pokud například máme produkční systém v prodejně, pravděpodobně bude obsahovat obrovské množství dat z nákupů našich zákazníků. Jistě bude velice snadné z těchto dat zjistit jaké zboží se prodává nejvíce, ale taková data v sobě

ukrývají daleko více informací, než je na první pohled znát. Dá se z nich například vyčíst jaké skupiny zboží se prodávají nejčastěji spolu (na základě čehož můžeme například zboží vhodně rozmístit po obchodě) a další informace, jejichž získání je často velice netriviální, nicméně můžou mít obrovský význam pro vedení podniku. Touto problematikou se zabývá obor získávání znalostí z databází (pro více informací, viz [24]).

Zásadním problémem při vyhledávání informací a dat je časová složitost vztažená k objemu ve kterém se vyhledává. S tím je spojená problematika vhodného indexování dat pro efektivní přístup.

2.2 Vyhledávání informací

Vyhledávání informací (Information retrieval) je multidisciplinární věda o vyhledávání v dokumentech, obsažených informacích (či znalostech) a jejich popisech ve formou metadat, založená na poznatcích z matematiky (statistika, pravděpodobnost, diskrétní matematika a další), počítačové vědy, knihovnictví, psychologie, lingvistiky a jiných vědních oborů [8]. Tato disciplína se zabývá efektivním přístupem k požadovaným informacím a snaží se co nejvíce minimalizovat tzv. „informační přetížení[1]“.

Jak již bylo řečeno dříve, v dnešní době existuje velké množství systémů pro vyhledávání informací. Takovéto systémy v drtivé většině vyhledávají požadované dokumenty v lokálních či sdílených databázích. Typickými zástupci jsou například vyhledávače článků, obrázků, videí, knihovní systémy a webové vyhledávače, bez kterých si dnes většina uživatelů internetu nedokáže práci s počítačem ani představit. Důležitým faktem přitom je, že vyhledávače typicky neposkytují přímo požadované informace, nýbrž se snaží nabídnout nějakou množinu zdrojů (dokumentů), ve kterých by se (v ideálním případě) měl uživatel požadovaných informací dopátrat.

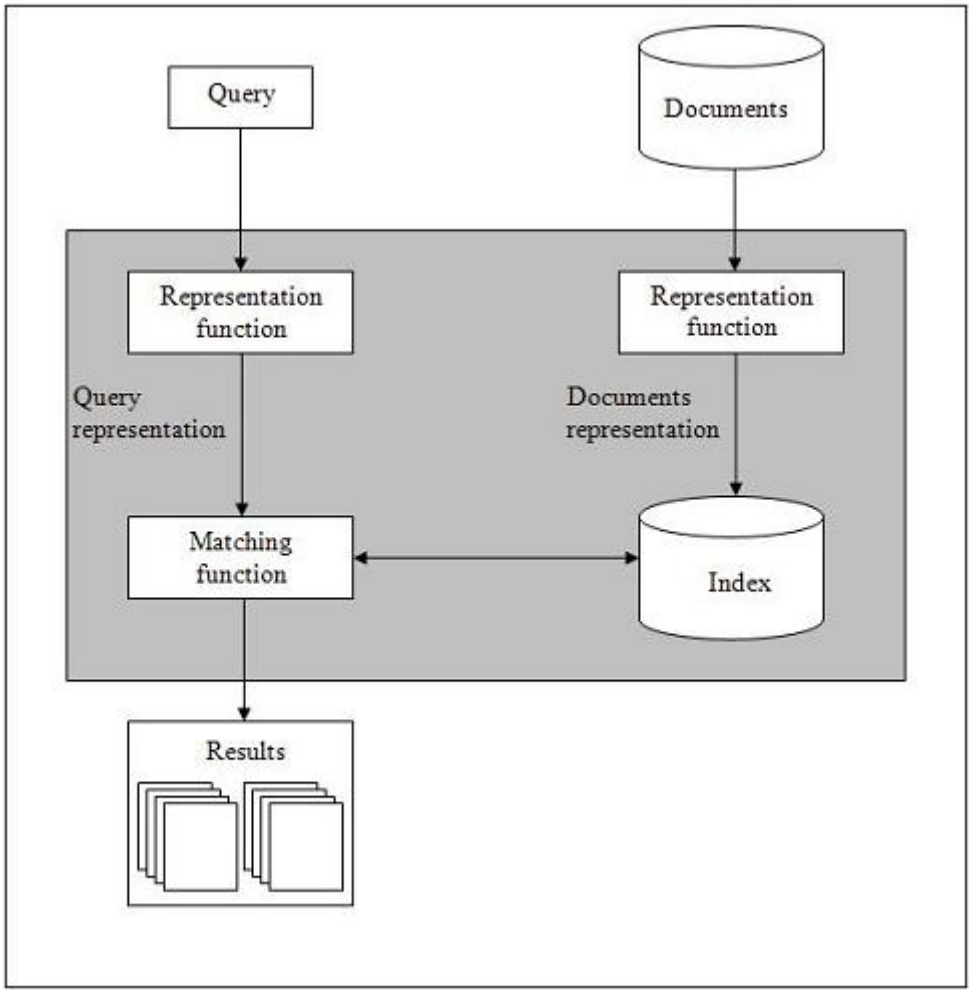
Základními problémy při tvorbě takového systému pro vyhledávání informací jsou vhodné uložení specifikací dokumentů (pro pozdější vyhledávání), použití vhodného vyhledávacího modelu a návrh mechanismů dotazování na požadované informace. Další důležitou vlastností takových systémů je způsob popisu jednotlivých dokumentů, který je pak typicky používán při jejich pozdějším vyhledávání (systém z důvodů efektivity nemůže analyzovat všechny dokumenty v době vyhledávání, ale musí tento proces provádět ve fázi předzpracování). Typický model použitý ve většině vyhledávacích systému je vyobrazen na obrázku 2.1. Jako vstupy systému slouží dotazy na dokumenty obsahující požadované informace a uložiště, ve kterém se tyto dokumenty nacházejí[2]. Výstupem systému jsou pak výsledky vyhledávání. Na obrázku jsou dále vidět další prvky systému, jako například reprezentační funkce pro dotazy, což je prvek systému, který se stará o zpracování dotazu od uživatele a jeho „překlad“ do jazyka vhodného pro strojové dotazování. Dalším prvkem systému je reprezentační funkce pro dokumenty, čímž rozumíme prvek systému, který se stará o poskytnutí popisu dokumentů metadaty vhodnými pro dotazování. Systém obsahuje Index dokumentů, který se stará o ukládání takovýchto popisů ve struktuře vhodné pro efektivní vyhledávání. Posledním prvkem systému je funkce určující míru shody, která pak určuje, které dokumenty jsou pro dotaz relevantní. Relevantní dokumenty jsou pak obsaženy ve výsledku vyhledávání.

Samotné vyhledávání pak probíhá v následujících krocích:

[1] Termín volně popsatelný heslem: někdy méně je více (příliš velké množství zbytečných informací bývá méně užitečné než malé množství těch podstatných)

[2] Jistá forma databáze dokumentů

1. Zadání dotazu – uživatel zadá dotaz na vyhledávané dokumenty
2. Reprezentace dotazu – systém přeloží dotaz do srozumitelné formy pro vyhledávání
3. Vyhledání dokumentů – prohledání indexu a hledání požadovaných dokumentů (odpovídajících dotazu)
4. Vrácení výsledku – výsledek pak obsahuje seznam všech relevantních dokumentů (nebo seznam těch nejvíce odpovídajících)



Obrázek 2.1: Obecné schéma vyhledávacích systémů [14]

2.3 Ukládání dat potřebných pro vyhledávání

Data určená pro vyhledávání jsou nejčastěji ukládána do nejrůznějších databázových systémů[3], které jsou postaveny na principu poskytování centralizovaného (případně i distribuovaného) uložiště dat s efektivními mechanismy vyhledávání. Zde je vhodné znovu zdůraznit, že tato data často nejsou tvořena samotnými dokumenty, které obsahují vyhledávané

[3]SŘBD – Systémy řízení báze dat

informace, ale pouze jejich vhodné popisy, které co nejpřesněji vystihují obsah daného dokumentu (jedná se tedy o formu metadat[4]). Pro účely vyhledávání se pak používají takové datové struktury, které jsou vhodné především pro čtení (nikoli pro efektivní řešení transakcí, na které je optimalizována většina podnikových operačních databází). Tomuto faktu je pak často potřeba přizpůsobit návrh použitého modelu ukládání.

Pro zefektivnění vyhledávání těchto metadat se pak hojně využívá tzv. indexů, což jsou nástroje databázových systémů, umožňující efektivní přístup k datům. O indexech bude v dalším textu řečeno mnohé, takže pro tuto chvíli postačí říci, že se jedná o dodatečné datové struktury vystavěné nad danou množinou dat, které zohledňují podle čeho se data budou vyhledávat. Pomocí indexů se pokoušíme co nejvíce optimalizovat přístup k těmto datům na základě zadaných kritérií (více o indexech naleznete v sekci 3.3).

2.4 Extrahování metadat pro popis dokumentů

Při procesu vyhledávání se často nevyhledává v dokumentech samotných, ale v databázi jejich popisů. Tím se výrazně zvýší efektivita vyhledávání, ale v některých případech se může výrazně snížit schopnost vyhledání relevantních dokumentů pro určité typy dotazů (především pokud jsou nevhodně zvolené popisy dokumentů). Aby toto bylo možné, musí samotnému hledání dokumentů předcházet fáze předzpracování, ve které se provádí extrakce takovýchto popisů, které pak uložíme ve vyhledávací databázi. Tohoto můžeme dosáhnout několika způsoby:

1. Vytvoření ručního popisu dokumentu
2. Automatická extrakce metadat dokumentu
3. Kombinovaný přístup (využívající jak ruční popis, tak automatickou extrakci)

První přístup využívající obsluhu, která ručně popisuje požadované dokumenty, je sice nejjednodušší, je však velice pomalá a často vyžaduje osobu znalou problematiky obsažené v popisovaných dokumentech. Pokud se obsluha v problematice vyzná, může (ale nemusí) se jí dařit vytvářet velice kvalitní popisy daných dokumentů (například pomocí množiny opravdu podstatných klíčových slov u textových dokumentů, ale i vysokoúrovňových popisů multimediálních dokumentů, kterých by automatický systém extrakce jen těžko dosáhl). Pro rozsáhlé databáze je však tento přístup naprosto nevhodný a často i nepoužitelný. Přes všechny nevýhody se však stále často používá například pro popis fotografií (autorem), hudby, filmů, nebo například ve vědeckých pracích (formou abstraktu a seznamu klíčových slov).

Druhý přístup využívající automatické extrakce metadat (který bude okrajově popsán v dalším textu) je velice složitá procedura, která využívá poznatků z nejrůznějších vědeckých disciplín[5]. Výsledné popisy se často diametrálně liší od popisů vytvořených člověkem a problém vytváření popisů vhodných pro vyhledávání je závislý i na formě vstupních dotazů (obzvláště u vyhledávání multimediálních dat je velký rozdíl, jestli je popis vhodný pro porovnávaní na podobnost s jinými multimediálními daty, nebo jestli je vhodný na vyhledávání pomocí textového popisu od uživatele). Obecně lze extrahované rysy rozdělit na nízkoúrovňové (snadněji detekované počítačem, ne vždy to, čeho si všimne člověk) a vysokoúrovňové (snadno rozpoznávané člověkem, avšak obtížně rozpoznatelné strojově). Snahou

[4] Metadata jsou data popisující strukturu a obsah skutečných dat.

[5] V závislosti na typu dokumentů – například z lingvistiky, knihovnictví či rozpoznání obrazu nebo řeči

při automatické extrakci dat je přijít na takové rysy, které jsou rozpoznatelné (a proto důležité) pro člověka (to ovšem nijak nesnižuje význam nízkoúrovňových rysů například pro strojové porovnání dokumentů na podobnost).

Třetí přístup je kombinací obojího a dá se použít například v systémech, které uchovávají informace o zpětných vazbách od uživatele.

2.5 Mechanismy dotazování

Dotazování na požadované dokumenty se tradičně provádějí pomocí zadávání několika klíčových slov obohacených o jisté syntaktické konstrukce pomáhající zadat složitější podmínky dotazu (např. nechceme dokument, který obsahuje slova „Karel“ a „Danko“, ale dokumenty, kde se nachází celé jméno „Karel Danko“ jako jeden celek). Tradiční dotazovací jazyky tedy fungují především na principu zadávání klíčových slov (*index termů*), které má dokument obsahovat. Takováto klíčová slova je potřeba z dokumentu nejdříve vyextrahovat tak, jak bylo popsáno dříve (viz 2.4).

Dalším široce využívaným způsobem dotazování jsou tzv. *fulltextové dotazy*. Takové dotazy umožňují pokládat intuitivnější požadavky, které jsou bližší uživateli jako člověku (tento způsob dotazovaní je více ergonomický). Fulltextové dotazy jsou pak složitěji analyzovány (například na synonyma, fráze, překlepy apod.) a jsou celkově více orientovány na snadné ovládání uživatelem. Takovýto model dotazování je dnes používán ve většině webových vyhledávačů.

V dnešní době se stále více používá vyhledávání na základě podobnosti obsahu dokumentů (uživatel zadá dokument a očekává dokumenty, které jsou nějakým způsobem podobné, nebo mají podobné některé vlastnosti), obzvláště pak v případě vyhledávání multimediálních dokumentů. Porovnání dokumentů na podobnost vyžaduje v mnoha ohledech odlišný přístup než např. porovnávání na shodu některých klíčových slov. Například pokud si vezmeme video jako typ dokumentu, tak z hlediska počítačového formátu může být tatáž nahrávka zcela odlišná z hlediska datové reprezentace a jistě je nesmysl, aby na dotaz na podobné video nahrávky nejdřív naši nahrávku shlédla nějaká obsluha, vytvořila popis a pak jej použila pro vyhledávání, tudíž musíme mít plně automatizovaný aparát na vytvoření vhodného popisu obsahu dat. Vyhledávání dokumentů na základě podobnosti obsahu je tedy obecně velice složitý problém, který vyžaduje sofistikované nástroje pro automatické extrahování rysů vhodných pro porovnávání tohoto druhu. Je dobré si také uvědomit, že termín obsah dokumentu je zde často chápán na sémantické úrovni (např. 2 nahrávky stejné písně jsou chápány jako obsahově velice podobné bez ohledu na to, jak rozdílný je datový formát, do kterého byly zakódovány).

2.6 Modely pro vyhledávání informací

Modely pro vyhledávání informací řeší strukturu a procesy vyhledávání informací. Každý model vyhledávání informací má definováno, jakým způsobem se rozhoduje, jestli je daný dokument pro zadaný dotaz relevantní či nikoli (případně do jaké míry). V této sekci se budeme zabývat pouze tradičními modely vyhledávání informací (Booleovský model, Pravděpodobnostní model a Vektorový model). Alternativními modely, mezi které patří například neuronové sítě, fuzzy množiny apod. se v této práci zabývat nebudeme.

2.6.1 Klasický booleovský model

Jedná se o jeden z nejstarších vyhledávacích modelů (ačkoli v minulosti hojně využívaných, dle [11]), který je jednoduše pochopitelný pro svoji přirozenou podstatu (jeho logika je blízká logice koncového uživatele). Tento model je založen na Booleově algebře a teorii množin. Dotazovací jazyk takového modelu má nejčastěji formát nějakého dialektu booleovského výrazového jazyka (operace `and`, `or`, `not`, závorkování, slova/řetězce jako fakta, která mají být v dokumentu hledána).

Booleovský model se vyznačuje několika specifickými vlastnostmi [11]:

1. Dokument je reprezentován konečnou množinou všech index termů, které obsahuje
2. Dotaz má formu booleovského výrazu nad index termy
3. Binární hodnocení relevance dokumentu (0 – dokument odpovídá všem podmínkám, 1 – dokument neodpovídá všem podmínkám)
4. Efektivní vyhodnocování
5. Neřeší řazení dle relevance (nemá smysl)
6. Někdy najde příliš mnoho (nerelevantních) dokumentů, jindy příliš málo (relevantních)

Základní myšlenkou booleovského modelu je, že dotaz je specifikován logickými podmínkami pro daný dokument, které můžeme chápat jako konjunkci podvýrazů. Slova ve výrazu pak můžeme chápat jako požadované (nebo naopak nechtěné) *index termy*[6]. Každý dokument v databázi je pak zaindexován pomocí množiny index termů, které se v dokumentu vyskytují. Pokud množina index termů dokumentu splňuje podmínku dotazu, dokument je relevantní. Pokud (byť jediný) index term neodpovídá specifikované podmínce (je v dokumentu a nemá, respektive není v dokumentu a má tam být), je dokument vyhodnocen jako nerelevantní.

Index takového modelu si pak můžeme představit tak, že klíčem je index term a hodnotou je seznam odkazů na dokumenty, kde se index term nachází. Pokud tedy hledáme konjunkci výskytů některých termů, hledáme průnik množin dokumentů, kde se nacházejí tyto termy (naopak sjednocení pro disjunkci, rozdíl pro konjunkci termu s negací druhého termu, apod.). Pochopitelně se tímto způsobem dá napsat i velice neefektivní dotaz typu $\neg(strukturalni \wedge indukce \wedge v \wedge ceske \wedge kinematografii)$, který by pravděpodobně vrátil téměř všechny dokumenty a vůbec by nevyužil indexu (pouzek identifikaci dokumentů, které uživatel nechce získat).

2.6.2 Vektorový model

Vektorový model je (na rozdíl od booleovského modelu) schopen rozhodovat o míře relevance jemněji než binárně (dle [11]).

Základní myšlenkou je přiřazení váhového vektoru všem dokumentům v databázi i zadanému dotazu. Podobnost dokumentu s dotazem je pak dána pomocí nějaké „vzdálenostní funkce“ těchto vektorů. Čím větší je pak vzdálenost mezi vektorem popisujícím dokument

[6] Klíčová slova

a vektorem dotazu, tím méně relevantní dokument je. Uspořádaný seznam nejvíce relevantních dokumentů se pak vytvoří vzestupným seřazením dokumentů podle hodnoty vzdálenosti jejich vektorů od vektoru dotazu (většinou je seznam omezen nějakou prahovou konstantou, která říká jaká je maximální vzdálenost dokumentu, který ještě lze považovat za relevantní).

Za předpokladu, že dokument je tvořen n rozdílnými slovy (index termy), bude tento dokument popsán n-rozměrným vektorem, jehož velikost složky každé dimenze je přímo úměrná frekvenci výskytu takového slova v dokumentu (viz [11]).

Nechť $\vec{d_j} = w_{1,j}, w_{2,j}, \ldots, w_{n,j}$ je váhový vektor pro j-tý dokument obsahující n různých slov a nechť $\vec{q} = w_{1,q}, w_{2,q}, \ldots, w_{n,q}$ je váhový vektor dotazu q, kde $w_{i,j}$ resp. $w_{i,q}$ jsou váhy odpovídající index termu t_i v dokumentu d_j, resp. v dotazu q. Potom míra relevance dokumentu d_j vzhledem k dotazu q je nepřímo úměrná ke vzdálenosti těchto vektorů. K výpočtu vzdálenosti dvou vektorů můžeme použít některou z následujících funkcí:

$$dist(\vec{q}, \vec{d_j}) = \frac{\vec{q} \cdot \vec{d_j}}{\sqrt{|\vec{q}|} \cdot \sqrt{|\vec{d_j}|}} = \frac{\sum_{i=1}^{n} w_{i,j} \cdot w_{i,q}}{\sqrt{|\sum_{i=1}^{n} w_{i,j}^2|} \cdot \sqrt{|\sum_{i=1}^{n} w_{i,q}^2|}} \tag{2.1}$$

$$dist(\vec{q}, \vec{d_j}) = (\sum_{i=1}^{n} (w_{i,q} - w_{i,j})^p)^{\frac{1}{p}} \tag{2.2}$$

Váhové vektory, které se k výpočtům používají se vypočítají podle metody „TF-IDF“ následovně: Nechť $freq_{i,j}$ je frekvence výskytu index termu t_i v dokumentu d_j Pak

$$tf_{i,j} = \frac{freq_{i,j}}{\max_l(freq_{l,j})}$$

nazýváme (normalizovaná) „frekvence termu“ t_i v dokumentu a d_j a

$$idf_i = log \frac{N}{n_i},$$

kde n_i je počet dokumentů, ve kterém se vyskytuje slovo t_i a N je celkový počet dokumentů, nazýváme „inverzní frekvencí dokumentu“ pro i-té slovo. Váhu pro i-tý term dokumentu d_j spočítáme takto:

$$w_{i,j} = tf_{i,j} \times idf_i = tf_{i,j} \times \log \frac{N}{n_i} \tag{2.3}$$

Vektorový model je velmi jednoduchý a rychlý a na rozdíl od ostatních modelů nepovažuje index termy za navzájem nezávislé. Vektorový model poskytuje velice dobré výsledky při vyhledávání. Další předností tohoto modelu je fakt, že zjišťuje míru relevance již při vyhledávání dokumentů. V současné době je vektorový model široce využíván (říká [27]).

2.6.3 Pravděpodobnostní model

Pravděpodobnostní model vyhledávání se snaží problematiku vyhledávání informací pojmout jako pravděpodobnostní problém (podle [1]). Je založen na myšlence, že uživatel, který chce vyhledat nějakou informaci ji nejdříve specifikuje pomocí zápisu dotazu. Dokumenty v databázi mají taktéž svůj popis ve formě index termů dokumentu. Pravděpodobnostní model se pokusí najít takovou množinu dokumentů u nichž je nejvyšší pravděpodobnost, že jsou relevantní vzhledem k informacím, které z nich chce uživatel zjistit (seřazené od nejrelevantnějšího). Uživatel pak popis dokumentu upravuje tak, aby systém

nalezl množinu dokumentů přesněji odpovídající tzv. *ideální množině*, což je množina obsahující všechny dokumenty, které jsou pro uživatele (vzhledem k vyhledávané informaci) relevantní a neobsahující žádný dokument, který by byl nerelevantní.

Iterativním opakováním tohoto procesu by se měl popis postupně zdokonalovat až bude téměř přesně odpovídat popisu ideální množiny dokumentů. Z tohoto pohledu je tedy proces vyhledávání informací s využitím pravděpodobnostního pohledu procesem hledáním co nejpřesnějšího popisu ideální množiny [1].

Při postupném zjišťování míry relevance dokumentu se bere v potaz především pravděpodobnost, že se index term nachází v náhodně vybraném dokumentu z množiny relevantních dokumentů. Při první iteraci je třeba tuto pravděpodobnost odhadnout, jelikož ještě není známa žádná množina relevantních dokumentů (úzké hrdlo celého modelu). Při dokončení iterace a vybrání relevantních dokumentů z výsledku, jsou znovu vypočteny zmíněné pravděpodobnosti pro jednotlivé index termy, například jako podíl počtu relevantních dokumentům obsahujících daný index term a počtu všech relevantních (vybraných) dokumentů.

Více informací o pravděpodobnostním vyhledávacím modelu lze nalézt v knize [1].

2.7 Fulltextové vyhledávání

Systémy pro vyhledávání textových dokumentů jsou i v dnešní době stále nejpoužívanějšími vyhledávacími systémy. Je tomu tak nejen z historických důvodů, kdy po dlouhou dobu nebyly počítače dostatečně výkonné, aby mohly v reálném čase vyhledávat v multimediálních datech či jejich multidimenzioálních popisech, ale především také proto, že práce s texty na počítači je pro lidi pravděpodobně nejzažitější (zanášení poznatků, příběhů, zákonů a celkově informací do písemné podoby). Zpracování informací reprezentovanými textovými daty je také pro počítač nejjednodušší a nejefektivnější.

I když již bylo (z výkonového hlediska) možné použít počítače k ukládání a vyhledávání velkých kolekcí textových dokumentů, bylo pro efektivní vyhledávání nutné dokumenty reprezentovat ve vyhledávacích strukturách nějakým úsporným a přitom dostatečně výstižným způsobem. Nejjednodušším, avšak pro vyhledávání většinou naprosto nedostačujícím způsobem, je vhodné pojmenování souboru dokumentu. Dalším (daleko přínosnějším) způsobem byl popis dokumentu pomocí vhodných klíčových slov a například zařazení dokumentu do určité kategorie (jistá forma *klasifikace*). V dnešní době se běžně uchovávají všechna slova z dokumentu, která jsou před tím automaticky vyextrahována a případně prošla různými úpravami tak, aby výsledná reprezentace byla pro vyhledávání co nejvhodnější a vypovídala o obsahu dokumentu co nejvíce (přestože je dokument zpracován automaticky).

V dnešní době asi nejpoužívanější vyhledávače informací jsou tzv. *fulltextové vyhledávače*, které se vyznačují tím, že pro vyhledávání dokumentů obsahujících požadované informace používají dotazování formou přirozeného jazyka uživatele. Systémy pro fulltextové vyhledávání dokumentů navíc používají kromě standardních index termů tzv. *fulltextovou reprezentaci dokumentu*, což je seznam všech slov (případně frází), které se v dokumentu nacházejí. Pokud je tedy zadán nějaký běžný dotaz s několika výstižnými termíny, je vyhledáváno pomocí index termů avšak pokud uživatel zadá něčím neobvyklý požadavek, který by vyhledávání na základě index termů z nějakého důvodu nedokázalo zpracovat, použije se jiná reprezentace dokumentu (například zmíněná fulltextová).

2.7.1 Extrakce metadat textových dokumentů

Jak již bylo mnohokrát zmíněno, v typických systémech pro vyhledávání ve velkých kolekcích dokumentů je třeba každý dokument nejdříve předzpracovat (zaindexovat) dříve, než jej bude možné vyhledávat. Tomuto procesu předzpracování se říká *extrakce metadat* dokumentu. Může se obecně jednat o metadata *strukturální* (zaměřená na strukturu dokumentu) či *sémantická* (zaměřená na obsah dokumentu). V této podsekci se budeme zabývat výhradně druhou variantou.

Extrakce metadat probíhá typicky v několika krocích, během kterých vznikají upravené reprezentace původního dokumentu. Je jen na tvůrcích systému, jestli danou reprezentaci použijí jen jako vstup do další fáze extrakce metadat, nebo jestli uchovávají třeba všechny tyto reprezentace a využívají jich při vyhledávání na základě nějakých speciálních požadavků.

Jedním z nejběžnějších automaticky prováděných kroků, je tzv. vytvoření fulltextové reprezentace dokumentu, což je vyextrahování všech slov z dokumentu, případně přidání informací o četnosti či frekventovanosti výskytu v dokumentu. Jelikož takovýchto slov může být v dokumentu obrovské množství a pouze omezený počet z nich má skutečně nějakou vypovídající hodnotu o obsahu dokumentu, provádí se následně často redukce *stop slov*, což jsou slova, která se vyskytují prakticky v každém dokumentu a proto s sebou obvykle nenesou žádnou informační hodnotu (rozumějme informační hodnotu o obsahu dokumentu). Takováto redukce typicky významně sníží počet slov či termů, které jsou používány při vyhledávání [15].

Dalším krokem často bývá redukce slov pomocí *identifikace podstatných jmen*. Výstupem tohoto kroku je seznam podstatných jmen, což jsou typicky slova nejvíce vypovídající o obsahu dokumentu (všechna jména, názvy, termíny, ...).

Další procedura, kterou textová reprezentace dokumentu často prochází je redukce slov na jejich kořeny (tzv. *steaming*) případně úprava skloňování.

2.7.2 Použití sémantických slovníků

Při vyhledávání v textových dokumentech se často využívá tzv. *sémantických slovníků*, což jsou nástroje, které pomáhají částečně zmírnit dopad nedostatků přirozeného jazyka. Mezi nejběžnější zástupce patří například tzv. *významové slovníky*, které obsahují termíny a jejich popis pomocí několika málo slov. Dalším typem sémantických slovníků jsou slovníky typu *tezaurus*, která obsahují vazby mezi příbuznými slovy (například synonyma, antonyma, vztahy typu nadřazená a podřazená slova, apod.)[15].

Sémantické slovníky jsou použitelné jak na straně uživatele, který je může používat k přeformulování dotazu (pokud mu vyhledávací systém na původně zvolená slova nechce odpovědět, případně vrací nerelevantní dokumenty), tak na straně vyhledávacího systému, který pak může poskytovat vyhledávání nejen přímo na základě index termů, nýbrž i na základě slov, či dokonce frází, které jsou s nimi spojeny (systém je pak daleko více orientován na význam daných slov, než na jejich textový tvar).

Kapitola 3

Vyhledávání dat v databázích

Problematika vyhledávání dat je v mnoha ohledech značně odlišná od problematiky vyhledávání informací. Vyhledávání dat je dnes již velice zaběhlá a díky dlouhodobému rozvoji i poměrně zvládnutá disciplína. Základním rozdílem mezi daty a informací je ten, že informace je sémantickým významem nějakých skutečností, zatímco jsou data jejich strukturované reprezentace vhodné pro ukládání v elektronické podobě a následné zpracování. Z tohoto pohledu je zřejmé, že chápání dat je pro strojové zpracování daleko přirozenější (přeci jen za tímto účelem data vznikla) a proto je i problematika ukládání a vyhledávání dat pomocí počítače o poznání jednodušší úlohou. Algoritmy a systémy pro vyhledávání dat jsou velice často založené na vyhledávání dat přesně odpovídajících daným kritériím (predikátům). Tyto dotazovací predikáty jsou typicky tvořeny již s ohledem na strukturu dat a způsob jejich uložení (často i s ohledem na očekávanou složitost vyhledávacího algoritmu).

Od 60. let dvacátého století se pro ukládání a následné vyhledávání dat začaly používat databázové systémy, které poskytují nejen prostředky pro spolehlivé ukládání *perzistentních* (trvalých) dat aplikací, kterým pomáhají zajistit celkovou *konzistenci* pomocí *integritních omezení*, ale navíc umožňují efektivní přístup při vyhledávání ve velkých objemech dat. V minulosti se využívaly více méně paralelně databázové systémy založené na *síťových* a *hierarchických* modelech. Během sedmdesátých let dvacátého století je však prakticky beze zbytku nahradily databázové systémy postavené nad *relačním databázovým modelem* (podle [26]).

Tato kapitola nejdříve stručně popisuje relační model dat a následně vysvětluje problematiku vyhledávání dat v relačních databázích[1]. Zaměřena je především na pochopení významu *indexace* při přístupu k datům pro čtení.

3.1 Relační databázový model

Tato práce není přímo zaměřená na relační databázový model, proto v této sekci pouze popíši nejdůležitější rysy a principy relačního modelu dat.

Relační model dat je (na rozdíl od předešlých databázových modelů) nejstarším široce používaným databázovým modelem plně založeným na formálním aparátu diskrétní matematiky [26]. Data jsou v relačních databázích uchovávány v *relacích* tak, jak je chápe matematika (podmnožina *kartézského součinu doménových množin* jednotlivých *atributů*). Sémanticky tyto relace odpovídají tabulkám obsahujícím data organizována do řádků (*uspořádané n-tice*) a sloupců (hodnoty sloupců jsou prvky z Doménové množiny daného sloupce).

[1] Databázových systémech založených na relačním modelu dat

Relační model dále definuje *normální formy* pro odstranění *redundance dat* v modelu a sadu operací, které se dají nad relacemi provádět. Důležitou vlastností relačních databází z pohledu dotazování je, že odpovědí na dotaz je zpravidla relace (tabulka). Nad relacemi v databázi lze provádět jak množinové operace (sjednocení, průnik, rozdíl a kartézský součin), tak některé speciální operace (projekce, selekce) [7].

Důležitým rysem relačních databází je striktní požadavek na *atomicitu* (nedělitelnost) dat, které se do relačních tabulek ukládají. Tento fakt je důležitý především proto, že pro velké množství moderních aplikací je tento požadavek natolik omezujícím, že se dnes prakticky žádný moderní databázový systém postavený nad relačním modelem neobejde bez aparátu pro práci s neatomickými daty ať už multimediálního, objektového, prostorového či jiného charakteru (takovýmto databázím se pak říká multimediální, objektové či objektově-relační, prostorové, nebo obecně *postrelační*). V dalším textu, pokud budu zmiňovat relační databázové systémy, budou se tím myslet i systémy postrelační (pokud nebude explicitně řečeno jinak).

Většina relačních databázových systémů používá (pro popis struktury dat, práci z daty a dotazování se na data) standardizovaný jazyk SQL, případně jeho dialekty (každý tvůrce relačního databázového systému si vytváří svoje rozšíření tohoto jazyka). Proto se relačním databázovým systémům často říká, trochu nepřesně, *SQL databáze*.

3.2 Vyhledávání dat v relačních databázích

Nejtriviálnějším způsobem vyhledávání v relačních databází je sekvenční procházení řádků tabulek a hledání takových, jejichž sloupce odpovídají vlastnostem stanovených dotazem. Tento přístup je však značně neefektivní vzhledem k často obrovskému množství řádků, které se v tabulkách nacházejí. Pro efektivní vyhledávání v relačních databázích bývají použity vhodné datové struktury navržené tak, aby byla dotazovaná data nalezena v co nejkratším čase bez nutnosti procházení celých tabulek.

Nejjednodušší formu efektivní datové struktury si můžeme představit u *skalárních typů*. Pokud je uložíme seřazené, můžeme použít *binární vyhledávání*, které je velmi efektivní (z původní složitosti $O(n)$ se dostáváme a složitost $O(\log_2 n)$ což představuje, při hledání v tabulkách s velkým množstvím řádků, velmi výraznou úsporu). U takovéhoto uspořádání jsme navíc schopni rychle zodpovídat dotazy na existenci, případně (neexistenci), dat s požadovanými vlastnostmi. Pro získání těchto výhod můžeme využít dvou přístupů (z [26]):

1. Uchovávat data seřazená
2. Vytvářet sekundární seřazené struktury pro tato data

Přestože první přístup je jistě intuitivní, můžeme data uchovávat seřazená pouze podle jediného kritéria bez ohledu na to, jak je složité (neuvažujeme možnost uchovávání kopií identických dat z důvodu nepřijatelné redundance). Tento přístup sám o sobě je tedy pro univerzální databázový systém nedostačující.

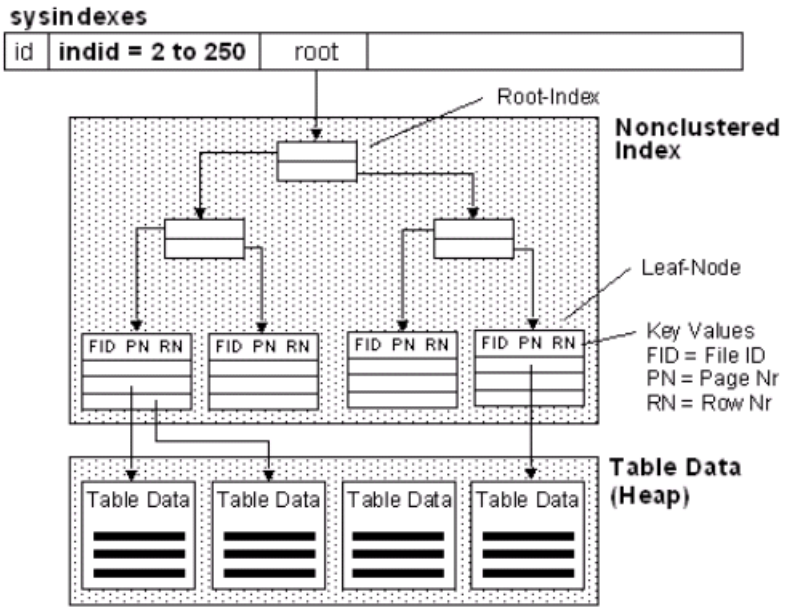
Vytváření sekundárních seřazených struktur nám naopak umožňuje mít k dispozici hned několik struktur zajišťujících virtuální uspořádání nad stejnou množinou dat. Takové struktury pak duplikují omezenou podmnožinu původních dat a obsahují koncept tzv. *záložek* (odkazů na původní data).

Vyhledání pomocí primárně seřazených dat je jednoduché, pokud vyhledáváme pomocí položek podle kterých jsou data seřazená. V takovém případě můžeme použít některý op-

timalizovaný algoritmus. Naopak pokud hledáme podle jiných položek, které neodpovídají seřazení primární struktury, tak nezbývá než použít lineární prohledávání.

Sekundární struktura má výhodu v tom, že jich může existovat několik nad stejnou množinou dat. Samotné získání finální množiny záznamů poté využívá nepřímou adresaci. Ze sekundární struktury (indexu) přečteme adresu, či jiný identifikátor, podle kterého jsou záznamy uloženy v primární struktuře, a ty vyčteme. Takovéto čtení je samozřejmě o něco méně efektivní, jelikož čteme několik struktur, avšak je pořád výrazně efektivnější než sekvenční průchod daty[2].

První varianta (ukládání seřazených dat podle indexu) se, v současných databázových systémech, vyskytuje pod názvem *clusterové indexy* (clustered indexes – Microsoft SQL Server) či *tabulka organizovaná indexem* (index organized table – Oracle) a určuje primární seřazení dat jedné tabulky na disku. Sekundární datové struktury se pak, v prostředí relačních databází, typicky nazývají *neclusterované indexy* (non clustered index) či nepřesně přímo „indexy“. Typická struktura takovýchto neclusterových indexů je ukázána na obrázku 3.1.



Obrázek 3.1: Struktura neclusterového indexu [20]

3.2.1 Vlastnosti indexů v relačních databázích

Indexy v relačních databázích se dále dají rozdělit na následující typy:

Unikátní index – hodnoty atributů, které takový index popisují musejí být unikátní v rámci celé tabulky (speciálním případem je primární klíč)

Jednoduchý index – index postavený nad jediným sloupcem tabulky

Složený index – index postavený nad větším množstvím sloupců tabulky

[2]Toto však není pravda úplně vždy – především u obzvláště malých tabulek

ASC index – index optimalizovaný pro průchod vzestupným (Ascending) směrem

DESC index – index optimalizovaný pro průchod sestupným (Descending) směrem

3.3 Indexační struktury

Základní a logickou strukturou pro indexování je samozřejmě seřazení skalárních hodnot. Tento způsob umožňuje efektivní vyhledávání. Problematické je zde ovšem aktualizování dat. Takováto aktualizace dat může vyžadovat přesun či přeindexování téměř celé datové struktury (např. změna počátečního prvku) a je tedy velmi neefektivní. Můžeme samozřejmě provádět optimalizaci pomocí různých padingů a mezer (podobně jako to dělají souborové systémy při snaze zabránit fragmentaci dat), ale principiální problém zde zůstává (takováto struktura není vhodná pro aktualizaci).

V dnešních databázových systémech se můžeme setkat s několika různými typy indexů. Jsou to indexy *bitmapové*, *husté* (dense) a *řídké* (sparse). Tyto struktury budou popsány v následujícím textu.

3.3.1 Bitmapové indexy

Do této chvíle jsme se bavili pouze o tzv. *uspořádaných indexech* tj. indexech, které nám nějakým způsobem uspořádávají záznamy v tabulkách. Bitmapové indexy jsou však poněkud odlišné struktury, které netvoří nad daty v tabulkách žádné uspořádání, nýbrž tvoří popis výskytů jednotlivých hodnot z domény daného vyhledávacího klíče pro jednotlivé řádky. Takový popis má potom podobu *bitmapy*. Každé hodnotě klíče, jež se nachází v tabulce je přiřazen bitový vektor o délce $m = count(rows)$, jehož hodnota bitu na obecně n-té pozici říká, jestli má indexovaný sloupec na n-tém řádku hodnotu daného klíče (platí, že $n \in 1 \ldots m$).

Tento druh indexu má hned několik výrazných nevýhod. Především je použitelný pouze pro ty sloupce, které nabývají pouze omezeného počtu hodnot. Pro příliš rozmanité (co do počtu možných hodnot) sloupce, případně pro sloupce nabývajících spojitých hodnot je tento index nevhodný.

Nad bitmapovým indexem je možné efektivně hledat na základě shody (či neshody) klíče s hodnotami indexovaných sloupců, efektivně vypočítat funkci *count(*)* nad potenciálními výsledky a kombinovat dotaz pomocí logických spojek pro konjunkci, disjunkci a negaci za pomoci jednoduchých bitových operací nad vektory.

Princip vyhledávání je následující. Pokud hledáme ty řádky, jejichž hodnota sloupce je (případně není) shodná s naším klíčem, najdeme si nejdříve v seznamu (sekvenčně) bitový vektor odpovídající požadované hodnotě klíče. Hodnoty bitů na jednotlivých pozicích ve vektoru nám pak říkají, jestli odpovídající řádek tabulky má (případně nemá) požadovanou hodnotu klíče.

3.3.2 Husté indexy

Husté indexy jsou takové indexy, které obsahují položku pro každou hodnotu, která se vyskytuje v tabulce. Pokud se jedná o index clusterový, obsahuje index pro každou hodnotu vyhledávacího klíče odkaz na první hodnotu v tabulce, která této hodnoty nabývá. Odkaz pouze na první hodnotu je dostačující proto, že v clusterovém indexu jsou řádky se stejnou hodnotou klíče vedle sebe a proto stačí nalézt první a k ostatním se již dá dostat sekvenčně bez procházení indexační struktury (což je efektivnější). U neclusterového hustého indexu

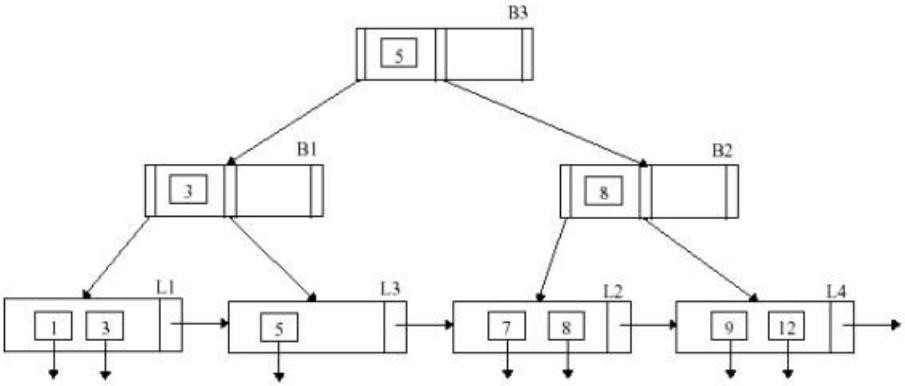
musí index obsahovat odkaz na každý řádek s odpovídající hodnotou klíče, jelikož se řádky se stejnou hodnotou vyhledávacího klíče nemusejí nacházet vedle sebe (ale třeba na opačných koncích tabulky) (dle [26]).

3.3.3 Řídké indexy

Řídké indexy jsou takové indexy, které (na rozdíl od hustých indexů) obsahují pouze určitou podmnožinu hodnot, kterých může vyhledávací klíč nabývat. Takovým indexem může být pouze index clusterový, jelikož lze využít faktu, že jsou data v tabulce uspořádaná. Řídké indexy typicky obsahují hodnoty klíčů pro první záznamy v indexových blocích, takže pak obsahují odkaz na začátek bloku, který je potřeba sekvenčně prohledat. Z tohoto je patrné, že procházení řídkého souboru je kombinací indexového a sekvenčního přístupu (viz [26]).

3.3.4 B^+ strom

Stromové indexové struktury jsou velmi efektivní pro vyhledávání dat, avšak jejich slabou stránkou je modifikace dat. Při přidávání, odebírání a aktualizaci dat v tabulce je vždy potřeba upravit strukturu stromového indexu, aby stále plnil svoji funkci. Důležité, pro efektivní vyhledávání ve stromových strukturách, je udržování stromové struktury ve vyváženém stavu. V databázových systémech je však taktéž důležité, aby režie spojená s údržbou stromových indexů během vkládání, odebírání a editace řádků byla co možná nejmenší. Mezi nejčastěji využívané stromové indexační struktury, které toto kritérium splňují, patří tzv. B^+ *stromy*, jejichž struktura je znázorněna na obrázku 3.2.



Obrázek 3.2: Struktura B^+ stromu [2]

Struktura (obecně n-árního) B^+ stromu má následující vlastnosti (z [26]):

1. Každý vnitřní uzel stromu má minimálně $\lceil \frac{n}{2} \rceil$ a maximálně n následníků
2. Pokud kořen není současně listem, má nejméně 2 následníky
3. Každý list obsahuje nejméně $\lceil \frac{n-1}{2} \rceil$ a nejvýše $n-1$ hodnot vyhledávacího klíče
4. Listové uzly jsou uspořádány do jednosměrně vázaného seznamu

Jak je vidět na obrázku 3.2, každý nelistový uzel B^+ stromu obsahuje až n odkazů na potomky a až $n-1$ hodnot vyhledávacího klíče. Obecně lze říci že odkaz na potomka je

současně odkazem na kořenový uzel podstromu, který má tu vlastnost, že všechny hodnoty klíče, které se v podstromu nacházejí, jsou menší než hodnota klíče, která následuje za odkazem na tento podstrom v rodičovském uzlu. Analogicky všechny hodnoty vyhledávacího klíče v podstromu jsou větší, než hodnota vyhledávacího klíče předcházející odkazu na tento podstrom v rodičovském uzlu. Tímto způsobem je v B^+ stromu zajištěno uspořádání hodnot vyhledávacího klíče.

Listy stromu obsahují odkazy na řádky v tabulkách, které odpovídají požadované hodnotě vyhledávacího klíče. Fakt, že sousední listy jsou propojeny do jednostranně vázaného seznamu pak umožňuje efektivní sekvenční procházení řádků tabulky ve směru uspořádání definovaném stromem.

Jak je vidět z vlastností B^+ stromu, vnitřní uzly stromu nemusejí být zaplněné (obsahovat všech n, resp. $n-1$ odkazů na potomky, resp. hodnot vyhledávacího klíče), musejí být však zaplněné alespoň z poloviny. Tato vlastnost znamená že B^+ strom často zabírá větší paměťový prostor, než je nutné, nicméně tím zvyšuje pravděpodobnost, že při vložení nové hodnoty klíče se nebude muset strom přeuspořádat. Snižuje se tak režie pro vkládání záznamů (dle [2]).

Operace vyhledávání B^+

Z důvodu zaměření této práce nejvíce popíši operaci vyhledávání nad B^+ stromem a ostatní operace vysvětlím pouze okrajově.

Při vyhledávání řádků s požadovanou hodnotou vyhledávacího klíč postupujeme tak, že směrem od kořene k listům procházíme sekvenčně uzly stromu, dokud nenarazíme na první hodnotu klíče vyšší, než je hodnota vyhledávaného klíče, případně než narazíme na konec seznamu klíčů. V případě že jsme klíč s vyšší hodnotou nalezli, přejdeme na uzel potomka na kterého vede odkaz bezprostředně před nalezeným klíčem. V případě, že jsme prohledali celý uzel, ale větší hodnotu klíče jsme nenalezli, přejdeme na uzel potomka odkazovaného posledním naplněným odkazem v uzlu. Vyhledávání většího klíče uvnitř uzlu je možné optimalizovat použitím binárního vyhledávání (klíče jsou uvnitř uzlu uspořádané). Takto procházíme přes všechny úrovně nelistových uzlů až k uzlům listovým. Na úrovni listů se vyhledává analogicky hodnota vyhledávaného klíče (nyní již stejná). Pokud je hodnota klíče nalezena, odkaz předcházející této hodnotě odkazuje na hledaný záznam v tabulce [2].

Záznam s odpovídající hodnotou vyhledávacího klíče se tímto postupem nalezne nejhůře v $\lceil \log_{\lceil \frac{n}{2} \rceil} K \rceil$ krocích, kde K je počet všech hodnot vyhledávacího klíče v uzlech stromu.

Jednou z mnoha výhod B^+ stromu je, že v něm lze efektivně vyhledávat podle rozsahu hodnot vyhledávacího klíče. Stačí nalézt záznam odpovídající počáteční hranici (chápáno ve směru uspořádání) rozsahu a následně lze již jednoduše sekvenčně procházet sousední hodnoty až dokud nenarazíme na záznam odpovídající koncové hranici.

Operace vkládání

Při vkládání nového záznamu do tabulky, nad kterou je vytvořen index ve formě B^+ stromu se vždy nejdříve vytvoří list na sekvenční úrovni. Úprava stromové struktury pak sestává ze dvou částí. Nejdříve je třeba nalézt vhodný uzel, do kterého by se hodnota vyhledávacího klíče pro nový záznam dala vložit, což provedeme pomocí operace vyhledávání tak, jak je popsáno výše. Pokud nalezneme uzel do kterého klíč patří, můžou nastat dvě situace. První situace je taková, že je v uzlu ještě volné místo pro nový klíč. V takovém případě stačí do uzlu tento klíč vložit a vytvořit odkaz na list. Pokud ale v uzlu již dost místa pro

novou hodnotu klíče není, přichází na řadu *rozštěp uzlu*, kdy se vybere prostřední hodnota vyhledávacího klíče a vloží se do rodičovského uzlu a uzel se rozpadne na dva z poloviny zaplněné (jeden bude odkazovaný stejným odkazem jako původní uzel, pro druhý se vytvoří odkaz bezprostředně za původně prostřední hodnotou v rodičovském uzlu). V rodičovském uzlu pak může nastat stejná situace a postupně se mohou uzly takto štěpit až ke kořenu, což může mít za následek zvýšení stromu o jednu úroveň (viz [2]).

Operace rušení

Rušení záznamu probíhá taktéž ve dvou krocích. Nejdříve je opět nutné nalézt uzel s hodnotou klíče shodnou s požadovanou. Tato hodnota klíče (spolu s odkazem na odpovídající řádek tabulky) je zrušena. Opět mohou nastat dvě různé situace. První situace je taková, že v je v uzlu stále není menší počet odkazů, než $\frac{n}{2}$, operace končí. V opačném případě je nutné provést *slévání* (coalescence), případně *redistribuci* (redistribution) uzlů, při kterých se mohou rušit hodnoty klíče v rodičovském uzlu, případně se může měnit hodnota nejmenší hodnoty vyhledávacího klíče v rodičovském uzlu. Obě tyto změny se mohou taktéž postupně šířit až ke kořenu, kde mohou zapříčinit až snížení stromu o jednu úroveň (viz [2]).

3.3.5 Ostatní používané indexační struktury

Kromě B^+ stromu se pochopitelně v databázových systémech využívají i jiné indexační struktury (často vycházející z B-Stromu, stejně jako B^+ strom). Některé z nich bych zde rád okrajově zmínil.

Samo-vyvažující se binární vyhledávací stromy

Samo-vyvažující se binární vyhledávací stromy (self balancing binary search trees) jsou varianty *binárních vyhledávacích stromů* (binary search trees – BST), které jsou založené na uspořádání indexové struktury do stromu, jejíž uzel má vždy nanejvýše dva potomky. Levý podstrom pak obsahuje pouze nižší hodnoty vyhledávacího klíče a pravý podstrom vyšší (nebo naopak, záleží opět na směru uspořádání indexu). Samo-vyvažující se binární vyhledávací stromy navíc řeší největší problém obecného binárního vyhledávacího stromu a sice jeho výraznou tendenci stávat se nevyváženým (v extrémním případě může dojít až k situaci, že struktura stromu degraduje na strukturu srovnatelnou s lineárním, jednosměrně vázaným, seznamem). Nejznámějšími implementacemi jsou následující (viz [17]):

- Červeno-černý strom (Red-black tree)
- AA strom (AA tree)
- AVL strom (AVL tree)
- Scapegoat strom (Scapegoat tree)
- Splay strom (Splay tree)

Hašované indexy

Hašované indexy (Hash based indexes) jsou indexační struktury (pro neclusterové indexy) založené na využití *hašovací funkce* (hash function), které efektivním a vhodným způsobem přepočítá hodnotu vyhledávacího klíče na nepřímou adresaci do seznamu adres jednotlivých

záznamů (z [26]). Jinými slovy hašovací funkce nám řekne, kde najdeme skutečné umístění záznamu, kterému hodnota vyhledávacího klíče odpovídá.

Celý princip hašovaných indexů stojí a padá na existenci vhodné hašovací funkce, která bude hodnoty klíčů co nejrovnoměrněji přepočítávat na adresy *sektorů* ve kterých se pak naleznou odkazy na záznamy tabulky. Pokud hašovací funkce nedokáže rozdělovat podle klíčů data do sektorů rovnoměrně, vznikne nepřiměřené využití *přetokových oblastí* (rozšíření sektoru o další prostor, ve kterém se pak vyhledává sekvenčně).

Pokud existuje vhodná hašovací funkce pro daný vyhledávací klíč, je tato indexační struktura obzvláště efektivní.

Indexy vícerozměrných dat

V této sekci jsme si popisovali výhradně indexační struktury a metody pro efektivní přístup k dobře strukturovaným jednorozměrným datům. Existuje však i mnoho technik pro indexaci vícedimenzionálních dat. Některé tyto techniky budou nastíněny v sekci 4.5 a v této sekci se jimi dále zabývat nebudeme.

Kapitola 4

Vyhledávání v multimediálních datech

V současné době se stále více rozmáhá využívání multimediálních dat na počítači. Multimediální data (MMD) se přitom používají v podstatě ve všech odvětvích lidské činnosti, při které se používají počítače. Audiovizuální data se stále více využívají nejen v zábavním, reklamním a grafickém průmyslu, ale také v ostatních průmyslových odvětvích, akademickém prostředí (za účelem výuky), a podobně. V dnešní době velkých kapacit paměťových uložišť a vysokorychlostních sítí jsou multimediální data stále více upřednostňována před tradičními textovými daty (například studenti se raději dívají na záznam přednášky, než by četli studijní opory, lidé si raději pouštějí „televizní“ reportáže, než aby četli noviny, atp.). Pro lidské vnímání jsou informace podávané v audiovizuální podobě často lépe pochopitelná a vyžadují menší soustředění (lépe využívají schopností smyslového vnímání lidí).

Následkem toho jsou multimediální data široce využívána a vznikají požadavky na jejich efektivní organizaci, správu a možnost vyhledávání. V dnešní době existuje velké množství internetových portálů, které se zabývají ukládáním a přehráváním audiovizuálních záznamů přímo skrze rozhraní webového prohlížeče. I tradiční vyhledávací systémy (např. Google) poskytují nástroje pro efektivní vyhledávání multimediálních dat ať už na základě podobnosti, nebo (častěji) na základě textového popisu požadované nahrávky.

Tato kapitola se zabývá přehledem typů multimediálních dat, jejich formáty, formáty jejich metadat a problematikou vyhledávání v takovýchto datech. Kapitola je obzvláště zaměřená na vizuální data a video, což je v zásadě nejsložitější forma multimediálních dat, která spojuje prvky ostatních typů v jeden celek.

4.1 Typy a formáty multimediálních dat

Pod pojmem „multimediální data“ (MMD) si v dnešní době může člověk představit ledacos. Z tohoto důvodu zde uvedu stručný přehled základních typů metadat a jejich stručnou charakteristiku. U každého uvedeného typu zmíním způsob, jakým je na ukládán v prostředí počítače, jak se interpretuje a některé zástupné formáty (z důvodu omezeného rozsahu této práce se zde nebudu podrobně zbývat detaily jednotlivých formátů). Informace pro tuto sekci jsem čerpal z [3].

Obrázky

Obrázky jsou dvoudimenzionální vizuální data, která mají statický charakter (jsou v neměnné v čase). Obrázky se dělí na rastrové a vektorové. Rastrové obrázky jsou reprezentovány dvourozměrnou maticí bodů (pixelů). Jednotlivé body pak mají různou barvu a jas. Takové obrázky jsou pak buď pořízeny pomocí nějaké fototechniky (fotoaparát, scanner, ...) nebo v rastrových kreslících nástrojích. Mezi základní nevýhody rastrových obrázků patří jejich velká paměťová náročnost a fakt, že rastrové obrázky přicházejí o svoji kvalitu při škálování. Mezi typické formáty rastrových (nebo-li bitmapových) obrázku patří JPEG, PNG, GIF, BMP nebo TIFF. Různé formáty používají různé metody (ztrátové či neztrátové) komprese tak, aby se snížila paměťová náročnost reprezentace.

Vektorové obrázky jsou reprezentovány pomocí barevných geometrických objektů (přímky, křivky, plochy, polygony, ...), jejich reprezentace má malé paměťové nároky a při změně velikosti nepodléhá ztrátě kvality. Při vykreslování je však nutné tyto obrázky rasterizovat (vychází to z rastrové podstaty zobrazovacích zařízení). Mezi typické zástupce rastrových formátů patří například EPS, SVG, PS, WMF, DVI a další.

Audio

Audiem většinou rozumíme zvuková multimediální data. Tato data vznikají převodem zvuku (mechanického vlnění) do digitální podoby (pomocí analogově digitálních převodníků) pomocí vzorkování v čase. Výsledná datová reprezentace je pak tvořena zakódovanými vzorky. Vzorky je pak možno komprimovat (většinou pomocí metod ztrátové komprese) tak, aby lidské ucho pokud možno nepoznalo rozdíl, avšak výsledná reprezentace měla nižší paměťové nároky a nižší datový tok při přehrávání. Mezi nejrozšířenějšími formáty pro ukládání audia patří mp3, AIFF, gsm, wav, wma, a další.

Video

Videem rozumíme audiovizuální data s temporálním charakterem. Temporálním charakterem se myslí časová proměnlivost zvuku a obrazu videa. Multimediální formáty, které obsahují taková data (navíc často přidávají další složky, především textové titulky) se nazývají multimediální kontejnery (media container). Video typicky obsahuje obrovské množství (často redundantních) informací, které tvoří základní reprezentaci videa extrémně paměťově náročnou. U videa je tedy komprimace velice důležitým faktorem. Video se komprimuje pomocí odstraňování redundance nejen v rámci jednotlivých snímků, ale i napříč nimi (snímkem rozumíme obrazový obsah v jednom časovém momentě). Přední kompresní algoritmy pro video jsou vyvíjené skupinou MPEG (Moving Picture Experts Group). Kromě formátů MPEG (především MPEG-1, MPEG-2, MPEG-4) jsou velice rozšířené formáty a kontejnery AVI, ASF, Ogg, QuickTime, 3GP či WMV. Video je interpretováno pomocí přehrávače využívajícího potřebné kodéry-dekodéry (codec).

4.2 Multimediální Metadata

Metadata (neboli data o datech) tvoří určitý druh popisu skutečných (v tomto případě multimediálních) dat. Metadata mají typicky strukturovaný charakter a dají se rozdělit na následující typy (z [3]):

Popisná metadata – Informace popisující zdroj dat, které jsou vidět uživateli a jsou určena především pro vyhledávání. Mezi takováto data patří například jméno autora, název díla, shrnutí obsahu, bibliografické informace, a jiné informace, které umožňují snadnější zařazení a pozdější vyhledávání v kolekcích multimediálních dat (např. klíčová slova).

Administrativní metadata – Data nezbytná pro uchování, správu a interpretaci. Jedná se o popisy kódování obsahu, formátu, velikosti apod. Vytváření těchto metadat je typicky věcí použitých formátu a multimediálních kontejnerů. Při vyhledávání mají jen malý význam.

Strukturální metadata – Popisují vnitřní strukturu objektů, ze kterých jsou data tvořena. Jedná se například o vztahy mezi objekty typu kompozice, agregace nebo asociace. Například kategorické uspořádání snímků, jejich zařazení ve videu, apod. Tato metadata jsou používána při vyhledávání na základě podobnosti.

Multimediální data, případně kontejnery mohou obsahovat v hlavičce další informace (kromě jejich typu, kódování, rozlišení a jiných, povětšinou administrativních, metadat). Za zmínku stojí například tyto formáty popisů (z [3]):

ID3 – Popis používaný v hlavičkách dat ve formátu MP3 (MPEG-1 a 2), obsahuje název díla, jméno interpreta, rok vydání, album, žánr, pořadové číslo v albu a jiné.

Exif – Standard používaný výrobci digitálních fotoaparátů pro přidávání metadat v době pořizování snímků. Tato metadata obsahují především informace o nastavení fotoaparátu, čase a místě pořízení. Používají se u formátů TIFF a JPEG.

XMP – Otevřený formát pro popis obrázků a dokumentů ve formátu PDF. Tento formát je vyvinutý společností Adobe Systems a je založen na XML. Kromě jiného přebírá informace z formátu Exif a dalších.

Dublin core – Otevřený standard. Obsahuje informace jako název, tvůrce, popis, vydavatele, typ, formát, identifikátor, zdroj, jazyk a jiné. Taktéž je založen na XML.

4.2.1 MPEG-7

MPEG-7 neboli Multimedia Content Description Interface je rozhraní pro popis multimediálních dat. Poskytuje množinu standardních deskriptorů (popisovačů) obsahu různých multimediálních typů. Účelem těchto deskriptorů je efektivní vyhledávání multimediálních dat ve velkých kolekcích (databázích). Deskriptor ve formátu MPEG-7 lze přiřadit multimediálním datům bez ohledu na jejich způsob uložení.

V samotném standardu jsou definovány tyto části (z [12]):

Deskriptory – atributy multimediálního obsahu založené na bibliografických údajích, syntaxi, sémantice a technologii použité při pořízení.

Jazyk pro definici popisů – definuje jazyk pro vytváření deskriptorů a popisových schémat.

Systémové nástroje – nástroje podporující tvorbu a přenos popisů.

4.3 Vyhledávání v MMD podle obsahu

Vyhledávat v multimediálních datech je vzhledem k jejich nestrukturovanému charakteru a neznámé sémantice možné pouze na základě deskriptorů rysů, případně jiných popisů, které jsme získali ve fázi předzpracování. V těchto deskriptorech je již možné poměrně jednoduše vyhledávat, přestože se jedná o výrazně složitější proces oproti např. vyhledávání informací z textových dokumentů a to především pro, na první pohled, ne zcela transparentní sémantiku deskriptorů a jejich multidimenzionální charakter.

Dalším závažným problémem při vyhledávání multimediálních dat podle obsahu je způsob, jakým je zadán dotaz. Pokud bychom zadávali subjektivní dojmy toho, co se v multimediálních datech nachází tak, jak to vnímá člověk, nemůžeme očekávat, že systém bude obsahovat automaticky detekované rysy média na takové úrovni, aby byl schopen vyhodnotit, že daný multimediální dokument odpovídá tomuto popisu. Jelikož jsou však popisované rysy ve většině případů automaticky extrahované (z důvodu vysoké časové a finanční náročnosti ručního popisu), je často potřeba zvolit jinou metodu zadávání dotazu, než slovní popis. Zadávání dotazu formou textového popisu je typicky používané pro vyhledávání podle textových metadat (například v hlavičkách multimediálních formátů, viz 4.2) zadaných lidmi (název, autor, příležitost pořízení apod.).

Problém vhodné reprezentace dotazu na očekávaný obsah vyhledávaných multimediálních dat je často řešen vyhledáváním na základě podobnosti s předloženým multimediálním vzorem. U některých druhů multimediálních dat je také možné jako vstup zadávat náčrtek či třeba zabroukat melodii, která se pak má vyhledávat (z [3]). Tento přistup je sice intuitivní, ale výsledky vyhledávání často nejsou příliš přesvědčivé.

Ruční zadávání požadovaných rysů je ve většině případů nemyslitelné, především pro obtížnou interpretaci jejich významů uživatelem.

Většinou se tedy multimediální data hledají podle jejich textového popisu či na základě podobnosti se zadaným vzorem. Jelikož první varianta je analogická např. pro fulltextové vyhledávání v textu, které bylo popsáno dříve, nebudeme se jím v této sekci dále zabývat a zaměříme se pouze na podobnostní vyhledávání. Podobnostní vyhledávání je typicky založené na určení podobnosti vektorů rysů multimediálních dat uložených v databázi a předloženého vzoru. Extrakce rysů dat v databázi probíhá ve fázi předzpracování poté, co jsou data do databáze nahrána. Extrakci rysů předloženého vzoru je pak nutné provést po zadání dotazu (více o extrakci rysů naleznete v sekci 4.4). Je úkolem návrhu databáze, aby byla multimediální data uspořádána a oindexována vhodným způsobem tak, aby se při vyhledávání podobných dat procházela co nejomezenější množina rysů dat z databáze.

Vyhledávání multimediálních dat na základě podobnosti je možný provádět obecně dvěma způsoby (z [3]):

Pomocí tříd podobnosti – Tento způsob předpokládá, že máme nějaký klasifikační model, který je schopný multimediální data, která jsou si podobná zařadit do stejných tříd. Pokud tedy klasifikátor rozhodne o zařazení našeho vzoru do nějaké třídy podobnosti znamená to, že výsledkem vyhledávání by měly být všechna multimediální data, která se v této třídě nachází (jedná se tedy o jistou formu booleovského vyhledávacího modelu). Relace podobnosti, pomocí které se provádí rozklad na třídy podobnosti je relací symetrickou a reflexivní.

Pomocí vzdálenosti rysů – Tento způsob se snaží zjistit podobnost dokumentů na základě hodnoty některé vzdálenostní funkce aplikované na vektory rysů těchto multimediálních dat. Míra podobnosti dokumentů je pak nepřímo úměrná velikosti vzdá-

lenosti mezi vektory rysů. Klíčovými prvky jsou zvolená vzdálenostní funkce a rysy, které jsou z dat vyextrahovány (pokud nebudou odpovídat rysům, které jsou člověkem vnímány jako důležité, nebudou se mu ani výsledné dokumenty, které jsou od sebe málo vzdálené podle funkce, příliš podobné).

4.4 Extrakce rysů z MMD a videa

Tato sekce stručně popisuje některé principy a metody používané při extrakci rysů multimediálního obsahu dat.

Extrakcí rysů nazýváme proces předzpracování dat, který slouží k získání deskriptorů multimediálního obsahu, které tento obsah co nejlépe popisují nějakou strukturovanou formou. Tím z nestrukturovaných multimediálních dat získáme jejich strukturovaný popis, který můžeme využít při indexaci medií a jejich následném vyhledávání. Cílem extrakce rysů by pak měla být redukce dimenzionality dat [28].

Proces extrakce rysů z multidimenzionálních dat je typicky plně automatizovaným procesem, na jehož výstupu je vektor strukturovaných popisů multimediálního obsahu [3].

U vizuálních dat rozlišujeme tzv. *globální rysy* (tj. rysy týkající se obsahu celého obrázku, případně např. celého záběru ve videu) a *lokální rysy* (tj. rysy týkající se částí obrázku, případně časoprostorové části části snímku ve videu).

Extrahované rysy se dělí do několika úrovní podle míry abstrakce:

Vysokoúrovňové rysy – rysy na úrovni lidského chápání sémantiky dat (tato úroveň se někdy nazývá také *konceptuální*).

Rysy střední úrovně – rysy na úrovni lidského vnímání tvarů, rozmístění objektu apod. (tato úroveň se nazývá také *geometrická*).

Nízkoúrovňové rysy – rysy na fyzikální úrovni

V následujících sekcích si popíšeme několik metod extrakce rysů nízké a střední úrovně, přičemž se zaměříme výhradně na extrakci z vizuálních dat. Je dobré zmínit, že pro extrakci vysokoúrovňových rysů se používá klasifikačních modelů a rysů nižších tříd.

4.4.1 Extrakce nízkoúrovňových vizuálních rysů

Extrakce popisu barev

Extrakce barevných rysů jsou jednou z nejpoužívanějších a nejzákladnějších metod popisu vizuálních dat (jak obrázků, tak videí). Hlavními barevnými rysy vizuálních dat jsou histogram barev, dominantní barva nebo barevná struktura. Pro extrakci barevných rysů vizuálních dat není vhodné použití běžných barevných modelů RGB a CMYK, jelikož příliš neodpovídají smyslovému vnímání barev člověkem. Jako vhodnější se jeví barevné modely HSV a HLS, ale i model YC_bC_r může být dostačujícím (některé metody pro detekci rozložení struktury barev jsou na tomto modelu dokonce postavené).

Extrakce popisu textur

Textura je druh povrchové vlastnosti reálného objektu, která je promítnutá do dvourozměrného obrazu. Textury v 2D obrazech mají některé specifické rysy vhodné k detekci. Jsou to například *kontrast*, *hranová frekvence* nebo *pravidelnost*. Textury se skládají z opakujících se obrazových částí, kterým říkáme *primitiva* (z [5]).

Jedním z nejefektivnějších způsobů popisu textury je extrakce frekvenčních rysů získaných pomocí *rychlé Fourierovy transformace* nebo *diskrétní Kosinové transformace* a následného použití Gaborových filtrů s různými orientacemi a měřítky. Tímto způsobem můžeme získat vektor deskriptorů nehomogenní textury.

Pro extrakci deskriptoru homogenní textury získáme například pomocí histogramu hran. Princip je takový, že se obraz rozdělí na pod-obrazy a v každém se snaží detekovat jeden z pěti typů hran (horizontálně podélná, vertikálně podélná, příčná vzrůstající, příčná klesající a nesměrová). Poté vytvoříme histogram výskytů jednotlivých hran v obrazu. Tato metoda dává velice dobré výsledky při porovnávání vizuálních dat na podobnost.

4.4.2 Extrakce vizuálních rysů střední úrovně

Metody pro extrakci vizuálních rysů střední úrovně typicky stavějí na existujících nízkoúrovňových popisech vizuálních dat. Mezy rysy střední úrovně patří především tvary a pohyb.

Extrakce popisu tvarů

Při detekci tvarů je třeba obraz nejdříve rozdělit na oblasti se společnou barvou či texturou, případně oblast uzavřenou nějakou množinou detekovaných hran (z [3]). Hlavním problémem detekce tvarů (především u statických obrázků) je, že dva i více objektů zobrazených na obrázku mohou sdílet jisté vlastnosti a prostorově se překrývat (v 2D prostoru budou splývat), jelikož se může jednat o objekty, pro člověka, známého tvaru, lze tyto objekty rozpoznat. Počítač však takové objekty však bude chápat jako jediný a slije je v jediný tvar. Analogicky to může platit pro obrácený případ, kdy jediný objekt (jako celek) má měnící se strukturu (myšleno strukturálně nikoliv časově) a proto jej počítač analyzuje jako několik různých tvarů. V prostorových multimediálních datech a ve videu se tomuto dá vytvořit extrakce trojrozměrného tvaru ze sady dvojrozměrných tvarů. Nejběžnější reprezentací trojrozměrných tvarů je trojrozměrná mřížka.

Extrakce popisu pohybu

Jako pohyb ve videu chápeme buď přemisťování vizuálních objektů v čase, posouvání kamery, rotace kamery či objektu, přibližování, oddalování, případně perspektivní deformaci. Pohyb kamery lze popsat pomocí fundamentální matice, kterou můžeme odhadnout pomocí detekce shody určitého množství náhodných bodů v po sobě jdoucích snímcích.

Pohyb obecně je možné popisovat detekovat pomocí perspektivní matice. Nejjednodušším typem pohybu je pohyb v rovině založený pouze na posunu, případně rotaci. Takový se z pohledu kamery příliš nemění a pohyb takových objektů lze detekova pomocí odečítání pixelů v po sobě jdoucích snímcích. Pokud jsme navíc schopni kompenzovat pohyb kamery (odečítáme vektor pohybu kamery), můžeme pohyblivé objekty ohraničit a pak snadno identifikovat pohybující se objekty a určovat jejich tvar (z [3]).

4.4.3 Extrakce lokálních rysů

Extrakce lokálních rysů v časoprostorové části snímku se provádí ve dvou základních krocích. Prvním krokem je detekce takzvaných *regionů zájmu* (regions of interest – ROI), což mohou být různé hrany, rohy či barevně spojité oblasti. Jakmile máme takovéto regiony zájmů detekované, extrahujeme jejich rysy (včetně jejich okolí). Důležitou vlastností těchto

rysů je pak jejich znovupoužitelnost s ohledem na změnu úhlu pohledu a jiné geometrické a fotometrické podmínky, případně výskyt šumu. Základní metody pro extrakci lokálních obrazových rysů jsou následující (z [5]):

MSER (Maximaly Stable Extremal Regions) – metoda detekce barevně spojitých částí obrazu podle ve vhodně (experimentálně) prahovaném obrazu. Všechny body uvnitř spojité části pak musejí mít vyšší či nižší intenzitu světla, než okolí.

SIFT (Scale Invariant Feature Transform) – metoda pro vytvoření vektoru popisujícího okolí regionů zájmu tak, aby byl invariantní vůči otáčení, měřítku, geometrickým nepřesnostem a částečně i změnám intenzit světla. Metoda vrací vektor histogramů lokálně orientovaných gradientů. Dá se například použít k vytvoření popisu regionů zájmu získaných pomocí metody MSER.

SURF – rychlá metoda pro detekci a popis mnohých nízkoúrovňových rysů částečně založená na SIFT. Metoda je založená na celočíselná aproximaci determinantu Hessovy matice z integrálního obrazu. Sestavení popisu je založeno na Haarově vlnkové transformaci. Taktéž se dá použít pro vytvoření popisu regionů zájmu získaných pomocí MSER.

4.5 Indexace MMD a videa

Indexace multimediálních dat se děje výhradně nad jejich popisy. To je způsobeno tím, že v databázových systémech jsou multimediální data ukládána ve své nestrukturované podobě a proto prakticky není možné vytvořit indexační strukturu pro efektivní přístup přímo nad těmito daty jednoduše proto, že jim databázový systém nerozumí. V této sekci jsou popsány některé základní indexační principy pro multidimenzionální data. Informace pro tuto sekci jsem čerpal především z [3]. Sekce volně navazuje na sekci 3.3, ve které jsme si popisovaly běžné indexační struktury používané pro atomická, jednorozměrná data.

Nejtriviálnější indexace multimediálních dat je pomocí indexace popisu klíčovými slovy. Nejznámější strukturou používanou pro tento druh indexace je tzv *frekvenční tabulka* (převzato z [3]).

V některých případech lze pro indexaci multimediálních dat podle obsahu použít i tradičních indexačních struktur databázových systémů, je však nutné, aby byl vektor rysů vektorem sklalárních hodnot o pevné délce (jednotlivé pozice ve vektoru pak lze chápat jako sloupce tabulky). Typicky však máme k dispozici multidimenzionální vektory, které je třeba do takovéto podoby převést. Tomuto přístupu říkáme *jednodimenzionální řazení* (one-dimensional ordering, viz [16]) a využívá se při něm funkcí křivek pro vyplnění prostoru (například z-order).

Pro indexování multimediálních dat na základě jejich multidimenzionálních vektorů rysů je potřeba použít indexy pro dělení prostoru. Takové indexy budou popsány dále.

4.5.1 K-D Strom

K-D Stromy jsou stromové indexační struktury, které slouží k indexaci k-dimenzionálních bodových dat (odtud K-D). Princip K-D stromu je založen na dělení prostoru při vkládání nových bodů. Při postupném vkládání bodu do stromu rozděluje K-D strom prostor podle jednotlivých dimenzí (z [3]). Na jednu stranu pak budou uloženy body s nižší hodnotou souřadnice této dimenze a na druhé s vyšší. Dimenze podle kterých se rozděluje se střídají.

Důležité je říci, že se vždy rozděluje pouze vymezená část prostoru, do kterého je bod vkládán. Postupně se tedy vytváří binární vyhledávací strom.

Mezi největší výhody tohoto modelu patří snadná pochopitelnost a jednoduchá implementace. Hlavní nevýhodou však je, že tento strom je z principu nevyvážený a nerozděluje prostor rovnoměrně. K-D Strom je speciální případ tzv. *BSP stromu* (binary space partitioning tree – BSP tree), který slouží k rozdělování prostoru na dva konvexní podprostory pomocí jediné roviny.

4.5.2 Quad Strom

Quad stromy využívají podobné myšlenky jako K-D strom (vycházejí z nich). Hlavním rozdílem je to, že Quad stromy rozdělují prostor ve všech směrech (z [3]). Typickým použitím Quad stromů je rozdělování dvourozměrného prostoru pomocí vkládání bodů (ve 2D prostoru se rozděluje rovina na čtyři podroviny – odtud Quad Strom). Oproti K-D stromu má lepší časovou složitost při vyhledávání, cenou za to je však špatná časová složitost při odstraňování bodů.

Existuje několik variací na Quad strom, které se snaží tuto strukturu ještě vylepšit pro specifické účely. Jedním z takových struktur je například MX-Quad strom, který dodržuje pevnou granularitu. MX-Quad stromy se snaží rozdělovat prostor tak, aby jednotlivé podprostory byly stejně velké.

4.5.3 R-Strom

R-Stromy jsou obdobou B-Stromů pro multidimenzionální data (z [3]). R-stromy jsou založeny na ukládání obdélníkových struktur. Některé obdélníky představují reálné objekty vložené do databáze. Další sdružují větší množství obdelníků do jednoho (tvoří index). Stejně jako je tomu i u B-stromů, všechny nelistové uzly kromě kořene musejí být minimálně z poloviny zaplněné. Pro neobdélníkové objekty z databáze se hledá minimální ohraničující obdélník (Minimum Bounding Rectangle, MBR).

Stejně jako je tomu u B-Stromu, R-strom má hned několik modifikací, které se snaží eliminovat jeho nedostatky. Například M-strom je modifikace R-stromu, která sestavuje index pomocí seskupování do kulových obálek (místo obdelníkových), což je sice výpočetně náročnější, avšak vhodné pro určování Euklidovských vzdáleností. Jinou modifikací je R^+-strom, který na rozdíl od obecného R-stromu vylučuje překrývání jednotlivých obdelníků. Cenou za tuto vlastnost je pak fakt, že reálný objekt může být rozdělen do několika listů.

R-stromy jsou nejvhodnějším algoritmem pro velké objemy dat. Problémem je, že se obdélníky mohou překrývat. Při vyhledávání potom nelze vyloučit určité větve, ale musí se prohledávat větší část stromu. To řeší lépe až jejich modifikace. R^+-stromy vylučují překrývání, ale obdélník reálného objektu může být zařazen do několika listů. Běžnější variací je R^*-strom, která volí vhodnou míru překrývání s ohledem na celkovou rychlost přístupu.

4.5.4 Další Multidimenzionální indexační metody

Pochopitelně pro indexování multidimenzionálních dat existuje velké množství vyvinutých struktur. V této kapitole byly zmíněné pouze základní indexační struktury, na kterých ostatní indexační struktury staví. Pro zájemce, kteří se chtějí dozvědět o indexování multidimenzionálních dat více, doporučuji si přečíst [16].

Kapitola 5

Návrh

V předešlých kapitolách jsme si popsali základy problematiky vyhledávání informací nejdříve obecně a pak s užším zaměřením na vyhledávání multimediálních dat v databázi.

V této kapitole budou popsány principy a dílčí součásti návrhu řešení, jehož výsledkem by mělo být zmírnění jednoho z hlavních nedostatků současného řešení a sice časovou náročnost dotazů. V této kapitole si stručně popíšeme charakteristiku databázového systému PostgreSQL na kterém je řešení postaveno, popíšeme použitou datovou sadu, demonstrační aplikaci, indexační techniku použitou pro vyhledávání a především zde bude popsána analýza hlavních komponent, která je v navrhovaném řešení klíčová pro snížení dimenzionality dat.

5.1 Volba databázového sytému

Implementace a experimenty této diplomové práce jsou z větší části spojené s databázovým systémem PostgreSQL. Důvodem zvolení tohoto systému byl fakt, že práce do značné míry rozšiřuje již existující řešení vyvíjené na fakultě informačních technologií Vysokého učení technického v Brně (dále FIT), které využívá právě tento databázový systém. Dalším důvodem byl fakt, že tento databázový systém nativně podporuje implementaci uložených funkcí v jazyce C a poskytuje vhodné prostředky použitelné právě pro téma této diplomové práce (především se jedná o možnost indexace pomocí GiST). V této sekci stručně popíši základní vlastnosti databázového systému PostgreSQL.

5.1.1 Základní charakteristika SŘBD PostgreSQL

PostgreSQL je objektově-relačním systémem pro řízení báze dat (SŘBD) vyvíjený jako open-source projekt pod licencí PostgreSQL license. Jedná se o databázový systém primárně vyvíjený pro unixové platformy, nicméně v současné době, existuje podpora i pro operační systémy MS Windows. Systém poskytuje dnes již běžnou podporu ACID transakcí a referenční integrity, nástrojů pro obnovení po havárii či výpadku apod. Jedná se tedy o plnohodnotný databázový systém.

Základním rysem systému PostgreSQL je možnost definovat si libovolný datový typ s operátory a funkcemi, které jsou s datovým typem vztaženy a integrovat jej do systému, včetně podpory indexace, přetypovávání či použití řetězcových literálů v kódu. Výrazným prvkem systému je podpora různých typů indexačních struktur, jakými jsou R-strom, B-strom, hashovaný index, GIN nebo GiST (více, viz [22]). Součástí systému jsou již přede-

finované datové typy (a související operace) pro geometrická/geografická primitiva a práci s prostorovými daty.

Systém umožňuje využití velkého množství procedurálních jazyků pro definice uložených procedur a funkcí, včetně Perlu, Pythonu, C/C++, Javy, Ruby, Tcl a vlastního jazyka PL/pgSQL, který vychází z jazyka PL/SQL konkurenčních databázových systémů od společnosti Oracle.

Další typickou vlastností databázového systému PostgreSQL je poměrně striktní dodržování standardů SQL (např. standardu ANSI-SQL:2008).

Databázový systém PostgreSQL je robustním řešením, jehož tabulky mohou přesahovat i velikost několika TB. Z tohoto důvodu je tento systém poměrně hojně využíván, především v akademickém prostředí nebo v open-source projektech vyžadujících ukládání složitějších datových struktur (např. geografických dat při použití rozšíření PostGIS). Naproti tomu v komerční sféře zatím příliš často k vidění není. Mezi open-source SŘBD je druhým nejpoužívanějším hned po MySQL.

5.2 Datová sada

Návrh, implementace a experimenty této diplomové práce byly založené na datových sadách TRECVid z let 2008 a 2009. TRECVid (TREC Video Retrieval Evalution) je série konferencí sponzorovaná NIST (National Institute of Standards and Technology) s podporou vlády Spojených států, která má za cíl podpořit výzkum v oblasti vyhledávání a analýzy obsahu digitálního videa, pomocí otevřených a měřitelných hodnocení.

Za tímto účelem NIST uvolňuje rozsáhlé sady dat, aby usnadnila výzkum škálovatelných a efektivních metod pro analýzu a vyhledávání ve velkých objemech multimediálních dat. V letech 2008 a 2009 byly využívány datové *sady Sound and vision*, *BBC rushes*, *TRECVID 2008 surveillance video*, přičemž tato práce pro své experimenty využívá datové sady *Sound and vision* (více viz [13]).

5.2.1 Sound and Vision

Jedná se o datovou sadu používanou pro výzkum v letech 2007 – 2010 obsahující video data v součtu čítající přibližně 400 h videa.

Tato video data byla poskytnuta Nizozemským institutem pro zvuk a obraz (The Netherlands Institute for Sound and Vision). Z většiny obsahují záznamy ze zpravodajských, vědeckých, vzdělávacích a dokumentárních pořadů obohacené o některá archivní videa. Sada pro rok 2007 čítala 100 h. V roce 2008 se data z minulého roku použila jako trénovací a dalších 100 h bylo využito pro testování vytvořených modelů (všech 200 h se použilo navíc pro detekci rysů vhodných pro vyhledávání). V roce 2009 bylo pro vývoj modelů a trénování použito opět původních 100 h z roku (2007) a zbylých 100 h (z roku 2008) obohacených o dalších 180 h nových dat bylo určeno pro testování modelů.

V této práci se využívají data tak, jak byly dostupné pro rok 2009 (viz [13]). Data z let 2007 a 2008 tvoří jednu datovou sadu a novější data z roku 2009 tvoří sadu druhou. Tyto sady jsou pak v databázi rozdělené do dvou separátních schemat „F08“ a „trecvid09“.

V rámci dřívějších kvalifikačních prací a iniciativy FIT v účasti na konferencích TRECVid byla pro tuto datovou sadu vytvořeno databázové schema uchovávající informace o jednotlivých videích, záběrech a snímcích, které tato videa obsahují. Přitom byly, na základě některých metod extrakce rysů, vyextrahovány vektory rysů jednotlivých záběrů, které později sloužili ke klasifikaci a vyhledávání v těchto datech. Základním problémem

je přitom vysoká dimenzionalita těchto vektorů rysů, kterou se snaží vyřešit tato diplomová práce. V následujících podsekcích budou popsány použité metody extrakce rysů a databázová schemata používající tuto datovou sadu.

5.2.2 Extrakce rysů

V minulé podsekci jsme si stručně popsali datovou sadu *Sound and Vision*, která bude tvořit základ navrhovaného systému pro experimenty zpracovávaného řešení. V této podsekci si popíšeme extrahované rysy, které máme k dispozici pro tuto datovou sadu a způsob jakým byly získány. Základy extrakce rysů (metod, které zde budou odkazovány) byly diskutovány v 4.4.

V rámci iniciativy FIT v konferencích TRECVid v letech 2009 a 2008 bylo na datovou sadu *Sound and Vision* aplikováno množství metod extrakce rysů (jak nízké, tak vysoké úrovně), jejichž výstupy mají podobu vektorů těchto rysů (v 4.4 nazývány také jako deskriptory). Jednotlivé vektory rysů jsou vždy vztaženy k nějakému klíčovému snímku (stav obrazu videa v konkrétním momentu), ze kterého byly extrahovány. Obsah této podsekce byl čerpán z [6] a [5].

Histogram barev

Tento vektor rysů obsahuje statistické informace o rozmístění barev ve smyslu frekvence odstínů a sytosti barev ve snímku. Lepšího prostorového popisu pak bylo dosaženo pomocí rozdělení snímku do několika částí, které pak byly zpracovávány samostatně [6].

Víceúrovňová distribuce gradientů

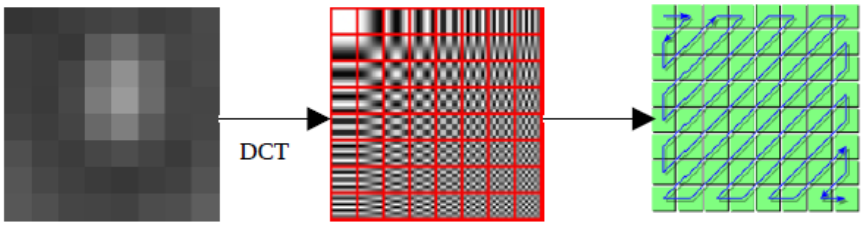
Jako další prvek vektorů rysů byl zvolen histogram orientací gradientů. Nejdříve se spočítají gradienty snímku. Potom, každý gradient přispívá do některého *koše* histogramu podle své orientace. Příspěvky jsou váženy pomocí významnosti gradientu. Jednotlivé gradienty jsou počítány pro několik rozlišení, takže méně významné struktury se také podílí na výsledném vektoru rysů [6].

Barevné rozložení

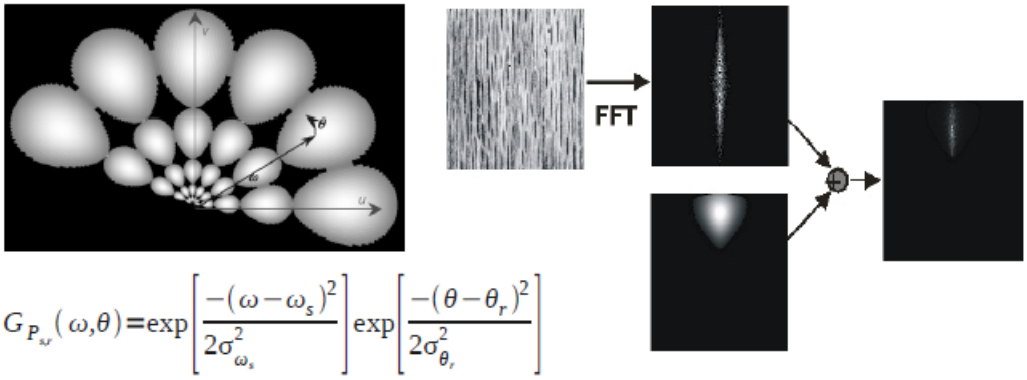
Vypočet barevného rozložení je založeno na technikách komprese JPEG. Nejdříve je obrázek převzorkován do 8×8 pixelového prostoru za použití barevného modelu YC_bC_r. Poté na každý kanál aplikujeme *diskrétní kosinovou transformaci*. Koeficienty deskriptoru jsou pak vyextrahovány průchodem „křížem krážem“, viz 5.1. Z vektoru je pak vyextrahováno 20 koeficientů pro jas (Y) a 15 pro chrominační komponenty C_b a C_r, čímž získáváme 50 koeficientů pro vektor [6].

Gaborovy textury

Pomocí Gaborových filtrů na frekvenční doméně můžeme rozdělit prostor, vytvořený např. pomocí Fourierovy transformace, do pásem (viz 5.2). Pro konstrukci deskriptoru pak použijeme první momenty energie z vyfiltrovaných 30-ti pásem (6-ti úhlových a 5-ti radiálních) [6].



Obrázek 5.1: Postup popisu barevného rozložení [6]



Obrázek 5.2: Postup popisu Gaborových textur [6]

Lokální rysy

O extrakci lokálních rysů jsme se zmínili již v sekci 4.4.3, proto zde zmíníme, že pro extrakci byly použity metody MSER, SIFT a SURF.

Detekce obličejů

Velmi cennou informací pro vyhledávání a extrakci rysů vysoké úrovně je informace o přítomnosti lidí na snímku. Jedním ze způsobu, jak zjistit přítomnost lidí na snímku je detekce obličejů. Existuje velké množství metod, jak rozpoznat obličej v obraze (v reálném čase).

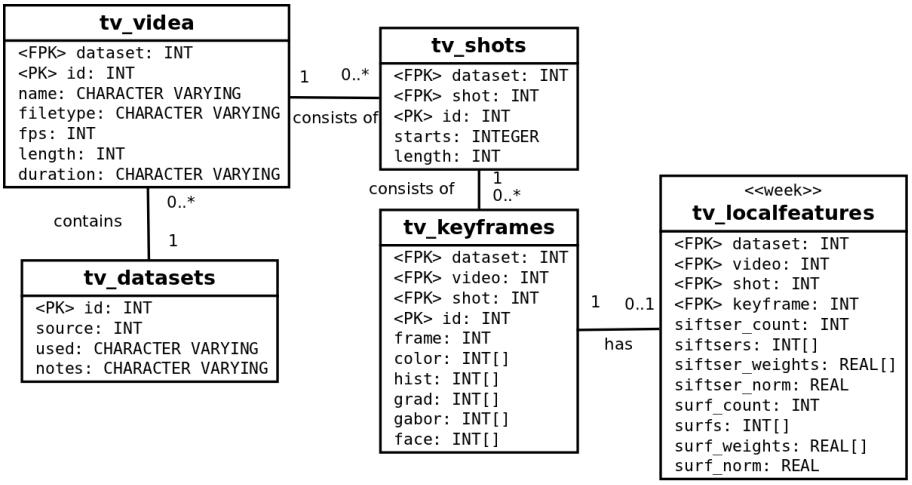
Detekcí obličejů jsou získány celkem 4 deskriptory pro každý snímek. Prvním dekriptorem je počet všech obličejů na snímku. Dalšími vektory jsou počet malých, středních a velkých obličejů, jelikož vypovídací schopnost velikosti obličejů a jejich počet je pro vyhledávání a extrakci rysů vysoké úrovně vyšší než např. u jejich pozice na snímku [6].

5.2.3 Původní schema datové sady

V předešlých letech v rámci hned několika kvalifikačních prací byla vyvinuta a využívána datová sada TRECVid uchovávaná ve schematu znázorněném na obrázku 5.3. V této podsekci si uvedeme význam jednotlivých tabulek a jejich sloupců.

Nejdříve si uvedeme základní termíny potřebné pro pochopení významu jednotlivých datových struktur a jejich vzájemných návazností.

Data v uvedeném schematu jsou rozdělena do jednotlivých *datových sad* tak, jak byly zveřejňovány pro konferenci TRECVid. Každá datová sada se skládá z určitého množství



Obrázek 5.3: Původní schema datové sady

video záznamů. Jednotlivá videa jsou pak rozdělena na sadu *záběrů*, což jsou krátké úseky video záznamů, jejichž délka intervalu se blíží přibližně době mezi dvěma střihy. Každý takový záběr se skládá z množiny *snímků* (videozáznam je obecně sekvencí snímků).

Původní schema se tedy skládalo z následujících tabulek:

`tv_dataset` – tabulka obsahující informace o jednotlivých datových sadách. Primárním klíčem této tabulky je identifikační číslo datové sady, sloupec `dataset`.

`tv_videa` – tabulka obsahující základní informace o jednotlivých videích. Příslušnost videa do některé datové sady je zajištěna referenční integritou cizího klíče mezi tabulkami `tv_videa` a `tv_dataset`. Primární klíč tabulky `tv_videa` tvoří spojení cizího klíče `dataset` a identifikátoru vido záznamu, sloupce `video`.

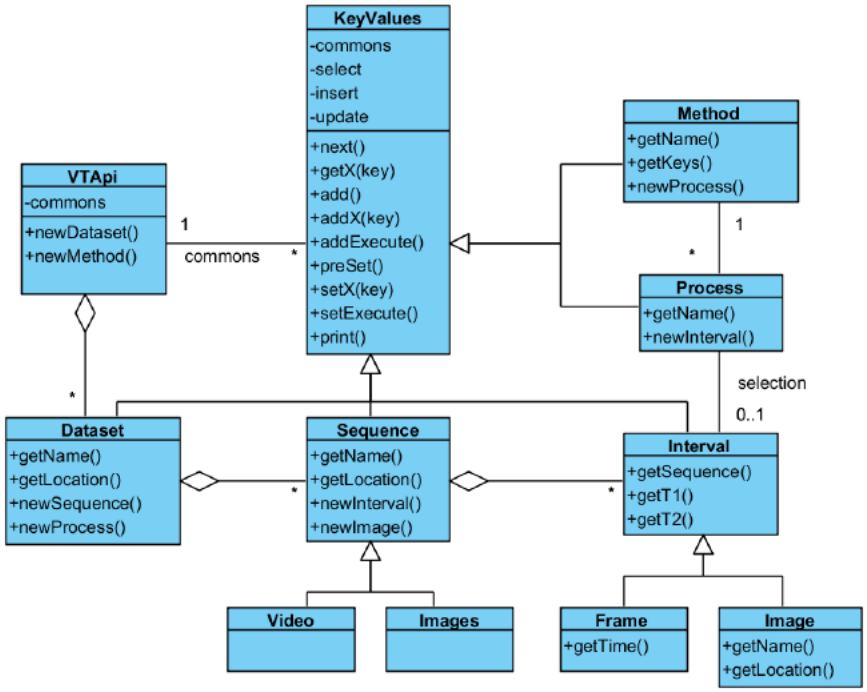
`tv_shot` – tabulka obsahující vymezení jednotlivých záběrů. Příslušnost k videu je zajištěna referenční integritou cizích klíčů `dataset` a `video`. Identifikátorem je spojení těchto sloupců a identifikátoru záběru, sloupce `shot`. Tabulka navíc obsahuje informace o začátečním času tohoto záběru ve videu a jeho celkové délce.

`tv_keyframe` – tabulka klíčových snímků v záběru. Příslušnost do daného záběru je dána referenční integritou cizích klíčů `dataset`, `video` a `shot`. Primárním klíčem je pak spojení těchto sloupců a identifikátorem klíčového snímku `id`. Kromě těchto identifikačních informací obsahuje tabulka navíc i vektory globálních rysů snímku (histogram barev, Gaborovy textury, gradienty a detekce obličejů).

`tv_localfeature` – tabulka obsahující vektory lokálních rysů (SURF, SIFT) vyextrahovaných z daného snímku. Snímek, kterému jsou rysy přiřazeny, je určen kombinací cizích klíčů `dataset`, `video`, `shot`, `keyframe`. Jelikož se jedná o *slabou entitní množinu* ve vztahu 1:1, primárním klíčem je právě tato sada cizích klíčů.

5.2.4 Nové schema datové sady

V rámci této diplomové práce bylo nutné transformovat původní datové schema (viz 5.2.3) do nového, které je jednodušší a je založené na schematu databáze používaným např. v aplikačním rozhraní *VTApi* (viz [23], [4]) projektu „Nástroje a metody zpracování videa a obrazu pro boj s terorismem“[25]. Zjednodušený objektový model používaný ve *VTApi* je vyobrazen na obrázku 5.4.



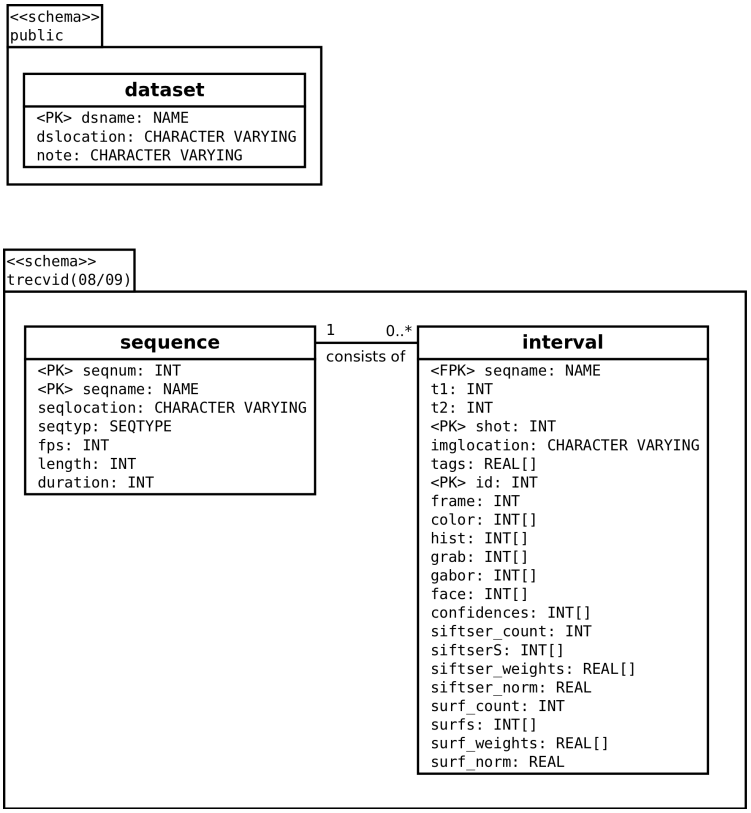
Obrázek 5.4: Zjednodušený objektový model *VTApi* [23]

Pro účely této diplomové práce bylo schema výrazně zjednodušeno nejvyšší přijatelnou mírou tak, aby schema bylo alespoň kompatibilní se schematem *VTApi*. Jelikož uvedené objekty modelu VTApi neobsahují položky pro data získaná z extrakce rysů, je potřeba o tyto položky naše schema rozšířit. Výsledný návrh nového databázového schematu je vyobrazeno na obrázku 5.5.

5.3 Aplikace TRECVid Search

TRECVid Search je desktopová aplikace vyvíjená na FIT, která slouží převážně k demonstraci pokroku v oblasti vyhledávání ve videu. Tato aplikace je vytvořená v Jazyce Java a proto je přenositelná na úrovni spustitelné aplikace pro většinu současných operačních systémů.

Grafické uživatelské rozhraní aplikace je založené na Swing frameworku, který je součástí aplikačního rozhraní standardní Javy. Jedná se o poměrně robustní framework umožňující vytvářet i velice složitá grafická uživatelská rozhraní desktopových aplikací při minimální námaze při vývoji (významná část grafiky se dá vytvořit v GUI Builderu obsaženého ve vý-



Obrázek 5.5: Schema nové datové sady

vojovém prostředí NetBeans, přičemž je vygenerován kód nezbytný pro sestrojení grafických objektů).

Pro připojení do databázového systému PostgreSQL využívá aplikace standardního JDBC rozhraní (verze 3) bez jakékoli nadstavby (např. JPA či jiné formy objektově relačního mapování), což je důsledkem předchozího vývoje. Z toho plyne poměrně silná závislost aplikace na schematu databáze.

Aplikace kromě jiného implementuje vyhledávání v databázi na základě podobnosti popisu (různých úrovní a typů) snímků, shlukovací metody pro klasifikaci snímků na základě rysů (a vyhledávání na základě této klasifikace), extrakci vysokoúrovňových rysů a jiné.

Jelikož je aplikace původně vyvinuta pro starší verzi schematu databáze (viz 5.2.3), je zapotřebí upravit aplikaci, tak aby alespoň klíčová funkcionalita fungovala nad novějším schematem (viz 5.2.4). Aplikace má i další problémy, jako například nekorektní volání databázových operací přímo z vlákna, které obstarává události (jak pro ovládání, tak pro např. překreslování objektů), což má za následek i několikaminutová odmlčení aplikace, kdy nereaguje na uživatelské ani systémové události. Z tohoto důvodu si aplikace vyžaduje alespoň částečnou údržbu, aby byla nadále využitelná (mimo jiné k demonstraci úprav, jež jsou předmětem této diplomové práce).

5.4 Analýza hlavních komponent

Analýza hlavních komponent je matematická metoda, která umožňuje redukci dimenzionality vektorů vzájemně korelovaných proměnných, pomocí vyjádření v jiném vektorovém prostoru o nižší dimenzionalitě, kde spolu jednotlivé dimenze nekorelují. Jedná se tedy o jistou formu transformace zdrojových vektorů z jednoho prostoru do druhého za použití ztrátové komprese. Pro další text této práce se jedná o významnou metodu a proto bude dále rozebrána.

[9] o analýze hlavních komponent píše, že hlavním smyslem je redukce dimenzionality datové sady obsahující vysoký počet vnitřně závislých proměnných tak, aby byla (pokud možno) zachována míra rozptylu hodnot z této sady. Toho je dosaženo pomocí transformace do nové množiny proměnných, hlavních komponent, které jsou vzájemně nekorelované. Tyto proměnné jsou pak seřazeny tak, že první proměnná zachovává nejvyšší míru rozptylu hodnot ze všech původních proměnných a poslední proměnná má na zachování rozptylu hodnot nejmenší vliv.

5.4.1 Definice důležitých pojmů

V této sekci si uvedeme základní matematické definice a termíny důležité pro analýzu hlavních komponent. Je zde uvedené jen nezbytné minimum pojmů, které jsou převážně vztaženy na použitou variantu analýzy hlavních komponent založenou na kovarianční matici.

Základní definice

Mějme datovou sadu $D = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}$ obsahující m vektorů náhodných veličin dimenzionality n ve tvaru:

$$\vec{x}_k = x_{k,1} \dots x_{k,n}, \text{ kde } k \in \{1 \dots m\}$$

a řekněme, že kovariance a rozptyl hodnot veličin z jednotlivých dimenzí jsou významné. Dále mějme aritmetický průměr μ_i hodnot jednotlivých dimenzí $i \in 1 \dots n$ definován takto:

$$\mu_i = \frac{1}{m} \cdot \sum_{k=1}^{m} x_{k,i} \tag{5.1}$$

Za předpokladu, že všechny hodnoty náhodné veličiny v dimenzi i mají stejnou pravděpodobnost, můžeme rozptyl σ_i^2 dimenze i definovat následovně:

$$\sigma_i^2 = \frac{1}{m-1} \cdot \sum_{k=1}^{m} (x_{k,i} - \mu_i) \tag{5.2}$$

Pro výpočet kovariance dvou dimenzí i a j, kde $i, j \in 1 \dots n$ použijeme následující vztah:

$$cov(x_i, x_j) = \frac{1}{m-1} \cdot \sum_{k=1}^{m} ((x_{k,i} - \mu_i) \cdot (x_{k,j} - \mu_j)) \tag{5.3}$$

Kovarianční maticí C pak nazýváme matici o rozměrech $n \times n$, ve tvaru:

$$C = \begin{pmatrix} \sigma_1 & cov(x_1, x_2) & \cdots & cov(x_1, x_n) \\ cov(x_2, x_1) & \sigma_2 & \cdots & cov(x_2, x_n) \\ \vdots & \vdots & \ddots & \dots \\ cov(x_n, x_1) & cov(x_n, x_2) & \dots & \sigma_n \end{pmatrix} \tag{5.4}$$

Kovarianční matice tedy obsahuje hodnoty rozptylu jednotlivých dimenzí na hlavní diagonále a hodnoty kovariancí dvou dimenzí mimo hlavní diagonálu. Můžeme si všimnout že matice je čtvercová a symetrická.

Vlastní hodnoty a vektory matice

Nechť M je matice ve tvaru:

$$M = \begin{pmatrix} \vec{m}_1 \\ \vdots \\ \vec{m}_n \end{pmatrix} \tag{5.5}$$

Pak *vlastním vektorem* matice m rozumíme takový vektor $\vec{e} = 1 \dots n$, kdy existuje hodnota λ taková, že:

$$M \times \vec{e} = \lambda \cdot M \tag{5.6}$$

Hodnota λ se pak nazývá *vlastní hodnota*.

5.4.2 Postup při analýze hlavních komponent

V předešlé podsekci jsme si popsaly nezbytné pojmy potřebné pro výpočet analýzy hlavních komponent. V následujícím textu popíši postup při výpočtu analýzy hlavních komponent. V rámci této práce se zaměříme na analýzu hlavních komponent založenou na kovarianční matici datové sady.

Sestavení matice datové sady

V prvním kroku si z datové sady sestavíme vstupní matici tak, že jednotlivé vektory náhodných proměnných budou tvořit řádky matice a jednotlivé proměnné napříč vektory budou tvořit sloupce matice (jinými slovy každý sloupec je tvořen jednou dimenzí vstupních vektorů). Pro naše účely takovou matici budeme nazývat DS.

$$DS = \begin{pmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,n} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \cdots & v_{m,n} \end{pmatrix} \tag{5.7}$$

Spočtení průměru a kovarianční matice

V tomto kroku sestavíme vektor průměrných hodnot všech dimenzí $\vec{\mu} = \mu_1 \dots \mu_n$ kde průměry spočítáme podle vztahu 5.1 a sestavíme kovarianční matici C podle 5.4. Přestože v tomto kroku není nic překvapivého z matematického hlediska, je nutné podotkout, že pouze tyto struktury mají (v součtu) paměťovou složitost $O(n \cdot (n+1)) = O(n^2)$.

Nalezení vlastních hodnot a vektorů

Když máme k dispozici kovarianční matici, můžeme vypočítat její vlastní hodnoty a vlastní vektory. Zde je důležité, že chceme jednotkové vlastní vektory, což nám naštěstí většina používaných nástrojů umožňuje (tvrdí [19]). Dalším důležitým faktem je, že pro n-dimenzionální prostor můžeme vypočítat n vlastních hodnot a n vlastních vektorů dimenzionality n. Proto všechny vlastní hodnoty a vektory můžeme vyjádřit následovně:

$$\vec{\lambda} = \lambda_1 \dots \lambda_n \tag{5.8}$$

$$E = \begin{pmatrix} \vec{e}_1 \\ \vdots \\ \vec{e}_n \end{pmatrix} \tag{5.9}$$

Kde $\vec{\lambda}$ je vektor vlastních hodnot kovarianční matice C a E je matice, kde na každém řádku je jeden vlastní vektor kovarianční matice C.

Seřazení podle vlastních hodnot a sestavení vektoru energií

V tomto kroku pouze seřadíme současně vektor vlastních hodnot a matici vlastních vektorů tak aby vektor vlastních hodnot tvořil klesající posloupnost (seřadíme sestupně) a matice vlastních vektorů obsahovala řádky s vlastními vektory ve stejném pořadí v jakém jsou jejich související vlastní hodnoty (pokud vlastní hodnota z dimenze 3 je nyní na první pozici vektoru vlastních hodnot, pak vlastní vektor z řádku 3 bude nyní na prvním řádku, atp.). Získáme tedy data ve tvaru:

$$\vec{\lambda'} = \lambda'_1 \dots \lambda'_n, \lambda'_i < \lambda'_{i+1} \tag{5.10}$$

$$E2 = \begin{pmatrix} \vec{e'}_1 \\ \vdots \\ \vec{e'}_n \end{pmatrix}, e'_i = e_j \iff \lambda_i = \lambda_j \tag{5.11}$$

V tomto kroku navíc můžeme vytvořit vektor energií $\vec{en} = en_1 \dots en_n$ kde

$$en_i = \frac{\sum_{j=1}^{i} \lambda'_j}{\sum_{k=1}^{n} \lambda'_k} \tag{5.12}$$

Prvky tohoto vektoru mají charakter percentilu míry rozptylu. Jinými slovy, hodnota en_i říká kolik procent z původní míry rozptylu bude zachováno, pokud pro transformaci využijeme právě i dimenzí z matice $E2$. Toto je důležitý ukazatel kolik dimenzí ve výsledné matici zvolit tak, abychom měli pokud možno optimální míru rozptylu v novém prostoru při co nejmenším počtu dimenzí tohoto prostoru (například, při $en_1 = 60\%, en_2 = 100\%, en_3 = 100\%, \dots$ bude vhodnější zvolit dvě dimenze než více, jelikož více dimenzí stejně nezajistí zachování vyšší míry rozptylu).

Sestavení transformační matice a její aplikace

Pokud jsme zvolili dimenzionalitu výsledného prostoru n_2, kde $n_2 < n$ (např. na základě vektoru energií), výsledná transformační matice P bude mít tvar:

$$P = \begin{pmatrix} \vec{p}_1 \\ \vdots \\ \vec{p}_{n_2} \end{pmatrix}, \vec{p}_i = \vec{e'}_i, i \in 1 \dots n_2 \tag{5.13}$$

Jinými slovy omezíme počet řádků seřazené matice vlastních vektorů na n_2.

V této chvíli již máme k dispozici všechny prostředky pro transformaci libovolného vektoru $\vec{v}$ z datové sady DS na vektor $\vec{w}$ dimenzionality n_2. Vztah pro transformaci je následující:

$$\vec{w} = P^T \times \vec{v}_{\mu_0} \tag{5.14}$$

Zde P^T je transponovaná matice P a $\vec{v}_{\mu_0} = \vec{v} - \vec{\mu}$.

5.4.3 Návrh analýzy hlavních komponent

V předešlých podsekcích jsme si poměrně podrobně popsali matematické definice a postupy potřebné pro provedení analýzy hlavních komponent založené na kovarianční matici. V této sekci uvedu návrh datových struktur a operací, které budou potřebné pro implementaci analýzy hlavních komponent do databázového systému.

Volba implementačního jazyka

Do této sekce zařazuji i volbu implementačního jazyka, jelikož tato volba je důležitá již před plánováním datových struktur a operací. Po důkladném zvážení byl z repertoáru procedurálních jazyků podporovaných databázovým systémem PostgreSQL zvolen jazyk C (ve své základní neobjektové podobě) a to především z důvodu kvalitní podpory ze strany systému a především efektivita aplikací napsaných v tomto jazyce pramenící právě z faktu, že se jedná o jazyk nízké úrovně, který je velmi efektivně překládán do strojového kódu. Toto rozhodnutí s sebou pochopitelně nese i jisté nevýhody, jako například potřebu zvýšené pozornosti v oblasti práce s pamětí.

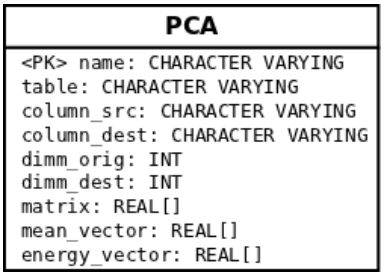
Datové struktury pro uchování výsledků analýzy

Jelikož výsledky analýzy chceme uchovávat dlouhodobě (jsou využitelné dokud se zásadně nezmění data ve zdrojové datové sadě), zvolil jsem možnost jejich uchovávání v databázi PostgreSQL. Za tímto účelem byla navržena tabulka zobrazená na obrázku 5.6, která slouží k uchování výsledků všech analýz hlavních komponent (resp. jejich nezbytných částí) v rámci dané databáze. Z tohoto důvodu se tabulka nachází ve schematu „public“, které se implicitně nachází v každé nově vznikající databázi systému PostgreSQL. Druhou výhodou tohoto umístění je fakt, že (např. i v našem případě, viz 5.2.4) v jedné databázi se mohou tabulky, na kterých chceme provádět analýzy hlavních komponent, nacházet v několika různých schematech.

Sloupce mají následující významy:

`name` – název jednoznačně určující danou analýzu

`table` – tabulka nad kterou se má analýza provádět



Obrázek 5.6: Tabulka výsledků analýz hlavních komponent

`column_src` – sloupec tabulky `table`, který obsahuje zdrojové vektory

`column_dest` – sloupec tabulky `table`, do kterého chceme ukládat transformované vektory

`dimm_orig` – dimenzionalita zdrojových vektorů

`dimm_dest` – požadovaná dimenzionalita transformovaných vektorů

`matrix` – transformační matice získaná z analýzy

`mean_vector` – vektor průměrných hodnot v dimenzích zdrojových vektorů získaný z analýzy

`energy_vector` – vektor energií získaný z analýzy

Prvních šest sloupců jsou v zásadě požadované vlastnosti analýzy a plní se před analýzou samotnou. Poslední tři sloupce jsou pak plněny během analýzy hlavních komponent. Sloupec `energy_vector` se je ukládán především pro poskytnutí přehledu o míře rozptylu v jendotlivých dimenzích, který může významně pomoci (během korekce) při volbě cílové dimenzionality transformovaných vektorů. Sloupce `matrix` a `mean_vector` jsou použity při samotné transformaci vektorů do cílového prostoru.

Na úrovni jazyka C pak (krátkodobě při provádění operací) bude použita následující datová struktura:

```
typedef struct {
    char *name;             // name
    char *source;           // table
    char *column;           // column_src
    char *column_dest;      // column_dest
    size_t dims_dest;       // dimm_dest
    size_t dims_src;        // dimm_orig
    float4 *matrix;         // matrix
    float4 *mean_vector;    // mean_vector
    float4 *energy_vector;  // energy_vector
} PCA;
```

Jednotlivé položky pak přesně odpovídají jednotlivým sloupcům databázové tabulky (jedná se tedy o obdobu „aktivního záznamu“).

Operace analýzy hlavních komponent

Pro práci s analýzou hlavních komponent jsou navrženy tyto operace[1]:

- `build_pca`
- `apply_pca`
- `apply_pca_on_table`

V dalším textu budou tyto operace podrobněji vysvětleny.

Operace `build_pca`

Tato operace slouží k samotnému provedení analýzy hlavních komponent. Rozhraní této operace je následující:

```
FUNCTION build_pca(CSTRING) RETURNS CSTRING
```

Funkce `build_pca` má jediný parametr, název identifikující odpovídající záznam v tabulce `PCA`. Funkce pak načte vstupy potřebné pro analýzu (zdrojovou tabulku, zdrojový sloupec, zdrojovou i cílovou dimenzionalitu) a provede samotnou analýzu, přičemž naplní položky `matrix`, `mean_vector` a `energy_vector`.

Operace `apply_pca`

Tato operace slouží k transformaci zdrojového vektoru za pomocí již existujících výsledků provedené analýzy hlavních komponent. Rozhraní této operace je následující:

```
FUNCTION apply_pca(CSTRING, float4[]) RETURNS float4[]
```

Funkce `apply_pca` má dva vstupní parametry. Prvním je řetězec obsahující název identifikující záznam v tabulce `PCA`. Druhým je zdrojový vektor, který chceme transformovat do cílového prostoru. Funkce pak načte vstupy potřebné pro transformaci (zdrojovou i cílovou dimenzionalitu, transformační matici a vektor průměrů) a provede samotnou transformaci. Pokud transformace proběhne úspěšně, funkce vrátí transformovaný vektor. Pokud transformace selže, vrátí prázdnou hodnotu (`NULL`).

Operace `apply_pca_on_table`

Tato operace slouží k transformaci všech vektorů ze zdrojového sloupce a uloží transformované vektory do cílového sloupce. K tomu použije již existujících výsledků provedené analýzy hlavních komponent. Rozhraní této operace je následující:

```
FUNCTION apply_pca_on_table(CSTRING) RETURNS BOOL
```

Funkce `build_pca` má jediný parametr, název identifikující odpovídající záznam v tabulce `PCA`. Funkce pak načte vstupy potřebné pro transformaci (zdrojovou tabulku, zdrojový sloupec, cílový sloupec, zdrojovou i cílovou dimenzionalitu, transformační matici a vektor průměrů) a provede samotnou transformaci, přičemž naplní sloupec `column_dest` transformovanými vektory. Pokud se transformace vektorů provede úspěšně, funkce vrátí TRUE.

[1] všechny mají charakter funkcí databázového systému PostgreSQL implementované v jazyce C

5.5 Indexace

V předešlých sekcích byla popsána datová sada *Sound and Vision* (viz 5.2.1), která je pro tuto diplomovou práci použita, stručný popis extrahovaných rysů a návrh databázového schematu 5.2.4 pro uložení strukturovaných dat o videích v databázi. Dále jsme si uvedli metodu pro snížení dimenzionality dat, analýzu hlavních komponent 5.4. V této sekci si popíšeme principy indexačních a vyhledávácích metod, které budou použity pro zefektivněn přístupu k vyextrahovaným informacím.

Jak již bylo zmíněno, základním problémem extrahovaných rysů je jejich vysoká dimenzionalita, která znesnadňuje indexaci a vyhledávání těchto dat. V sekci 5.4 jsme si popsali jakým způsobem snížit dimenzionalitu těchto dat. To je sice klíčové pro zvýšení efektivity vyhledávání, ale i po aplikaci této analýzy máme k dispozici vektory v multidimenzionálním prostoru (byť o výrazně nižší dimenzionalitě).

Z výše uvedeného důvodu není možné používat běžných indexačních struktur a metod, jakými jsou například B^+ strom a jeho jednodimenzionální varianty (viz 3.3.4). Je tedy třeba využít některou indexační metodu, která je adaptovatelná na multidimenzionální data a ideálně je již implementovaná v databázovém systému tak, abychom ji nemuseli implementovat sami.

Dalším faktorem důležitým pro indexování multidimenzionálních vektorů rysů je, aby zvolená indexační metoda podporovala vyhledávání na základě *podobnosti vektorů*. Jinými slovy, pokud použijeme vyhledávací dotaz, který např. vybírá n nejpodobnějších vektorů, je důležité aby plánovač databáze sestrojil takový plán dotazu, který bude využívat náš index (v opačném případě by při těchto dotazech docházelo k sekvenčnímu procházení celé tabulky, což je z důvodu efektivity nepřijatelné).

Podobností multidimenzionálních vektorů přitom většinou rozumíme *vzdálenost* těchto vektorů, kterou získáme pomocí některé *vzdálenostní funkce*. Prinipy vzdálenostních funkcí přitom uvedeme v této sekci

Dále v této sekci zmíníme několik technik vhodných pro indexaci a vyhledávání v multidimenzionálních datech, které jsou navíc již implementovány v databázovém systému a jsou vhodné pro podobnostní vyhledávání.

5.5.1 Vzdálenostní funkce

Jak již bylo řečeno, pro určení podobnosti vektorů slouží *vzdálenostní funkce*. Pro úplnost si uvedeme základní definici takové funkce.

Mějme n-dimenzionální vektory $\vec{v}$,$\vec{w}$ a $\vec{z}$. Řekněme, že platí $\vec{v}, \vec{w} \in \Re^n$, kde $\Re^n$ je množina všech n-dimenzionálních vektorů. Pak $d : \Re^n \times \Re^n \longmapsto \Re$ je vzdálenostní funkcí, jestliže pro libovolné $\vec{v}, \vec{w}, \vec{z} \in \Re^n$ platí:

$$d(\vec{v}, \vec{w}) \geq 0 \tag{5.15}$$

$$d(\vec{v}, \vec{w}) = 0 \iff \vec{v} = \vec{w} \tag{5.16}$$

$$d(\vec{v}, \vec{w}) = d(\vec{w}, \vec{v}) \tag{5.17}$$

$$d(\vec{v}, \vec{w}) \leq d(\vec{v}, \vec{z}) + d(\vec{z}, \vec{w}) \tag{5.18}$$

Mezi nejznámější vzdálenostní funkce pak patří např. *Euklidovská*, *Manhattanská* nebo *Minkowského* vzdálenost. Pro úplnost si uvedeme jejich zjednodušené definice pro vektory $\vec{v}$,$\vec{w}$:

$$d_{euc}(\vec{v},\vec{w}) = \sqrt{\sum_{i=1}^{n}(v_i - w_i)^2} \tag{5.19}$$

$$d_{man}(\vec{v},\vec{w}) = \sum_{i=1}^{n} \mid v_i - w_i \mid \tag{5.20}$$

$$d_{min}(\vec{v},\vec{w}) = (\sum_{i=1}^{n}(\mid v_i - w_i \mid)^p)^{\frac{1}{p}} \tag{5.21}$$

5.5.2 GiST

Pod pojmem GiST[2] si můžeme představit stromovou indexační strukturu (neclusterový index) a množinu operací nad touto strukturou. Jedná se o strukturu vyváženého stromu, která implementuje „šablonované“ algoritmy pro navigaci stromovou strukturou a modifikaci této struktury pomocí rozdělování a mazání stránek[10].

Podobně jako jiné neclusterové stromové indexy, GiST ukládá dvojice (klíč, RID[3]) do listů. Vnitřní uzly struktury tvoří dvojice (predikát, následník), kde následníkem rozumíme uzel potomka a predikátem je booleovský predikát, který je pravdivý pro všechny uzly v podstromu, jehož kořenem je následník[10].

GiST vznikl jako generalizace tradičních stromových přístupových metod tak, aby byly použitelné pro libovolný datový typ bez nutnosti implementace veškeré potřebné funkcionality. Touto funkcionalitou máme na mysli například řešení souběžnosti, ukládání indexu na disk, obnovu indexu při pádu nebo implementaci vyhledávání, vkládání či odstraňování listu ve struktuře.

GiST implementuje základní algoritmy potřebné pro sestavování indexační struktury a samotné vyhledávání na základě dotazu. Tuto funkcionalitu poskytují operace `SEARCH`, `INSERT` a `DELETE`, které budou vysvětleny dále.

Operace `SEARCH`

Tato operace je standardní vyhledávací operací ve stromu. Vstupem této operace je predikát dotazu, který určuje které listové uzly indexu mají být vráceny. Algoritmus prochází rekurzívně ty podstromy, pro jejichž kořenové uzly vrací uživatelská funkce `consistent()` hodnotu `TRUE`[10].

Operace `INSERT`

Tato operace vkládá nový pár (klíč, RID) do některého listu struktury a přeuspořádává strom tak, aby byl stále vyvážený. Na rozdíl od B-stromů, GiST umožňuje překrývání predikátů podstromů, takže ve struktuře může být hned několik listů, do kterých je možné klíč vložit. Procházíme strom od kořene k listu tak, že volíme následníky podle výšky *postihu* za vložení klíče do daného podstromu. Výše tohoto postihu je dána hodnotou funkce `penalty()`) pro daný uzel následníka a klíč, který vkládáme. Koncepčně by tato hodnot měla reflektovat jakou cenu zaplatíme za to, že budeme rozšiřovat predikát daného podstromu. Hodnota má srovnávací význam.

[2] GiST – Zobecněný vyhledávací strom (Generalized Search Tree)

[3] RID – identifikátor záznamu (Record Identifier), neboli ukazatel na odpovídající záznam v datové stránce

Pokud list následkem vložení hodnoty přeteče hranice dané predikátem předka, musí se provést *split*. Pokud i předek přetéká, provádí se *split* rekurzívně. Funkce `pick_split()` určuje strategii, které záznamy se budou přesouvat do nového pravého sourozence během operace *split*.

Pokud rodičovský uzel neobsahuje nový klíč, je potřeba *rozšířit* predikát tak, aby všechny predikáty na cestě od kořenu k uzlu obsahovaly daný klíč. Funkce `union()` určuje *rozšířený* predikát jako sjednocení původního predikátu a nového klíče. Podobně jako u operace *split*, může operace *rozšíření* predikátu předchůdce způsobit rekurzívní šíření až k předku, který nepotřebuje rozšířit (pro novou hodnotu klíče), případně až ke kořenu[10].

Operace `DELETE`

Tato operace slouží k odstranění listu, který má požadovanou hodnotu klíče. Tato operace nejdříve, obdobně jako operace `SEARCH` prochází několik podstromů tak, aby nalezla požadovaný list. Jakmile je list zaměřen a klíč vyhledán, odstraníme z listu dvojici (klíč, RID) a pokud je to možné, zpřesníme predikát předchůdce. V tomto případě se použije funkce `union()` pro výpočet požadovaného predikátu jako sjednocení všech uzlů na dané stránce[10].

Rozhraní GiST

Rozhraní GiST umožňuje uživateli, aby definoval strategii, která bude pro jeho datový typ použita při jednotlivých operacích GiST indexu. Určuje tak, kam se při vkládání daný klíč uloží, jakým způsobem se budou stránky rozdělovat při operaci *split*, nebo jakým způsobem bude probíhat rozšiřování predikátu. Tím je efektivita přístupu stále v plné moci uživatele, který se rozhoduje, jaké strategie pro vyhodnocení těchto funkcí zvolí.

GiST tedy v implementaci operací `SEARCH`, `INSERT` a `DELETE` využívá několika generických metod, a umožňuje uživateli přímo dosadit konkrétní implementaci těchto funkcí a tím ovlivnit (či definovat) strategii specifickou pro jím zvolený datový typ. Sadě těchto generických metod se říká rozhraní GiST (GiST interface) a skládá se z následujících funkcí (čerpáno z [18]):

consistent – Funkce implementující tuto metodu bere ze vstupu uzel GiST stromu a dotaz a vrací `FALSE` v případě, že podstrom daného uzlu určitě neobsahuje žádný list, jehož klíč by odpovídal dotazu. V ostatních případech by měl vracet `TRUE`.

compress – Funkce implementující tuto metodu bere ze vstupu uzel GiST stromu a vrací odpovídající uzel GiST stromu se shodným ukazatelem na stránku a komprimovanou reprezentací predikátu.

decompress – Funkce implementující tuto metodu bere ze vstupu uzel GiST stromu a vrací odpovídající uzel GiST stromu se shodným ukazatelem na stránku a rozbalenou verzí predikátu.

union – Funkce implementující tuto metodu bere ze vstupu množinu uzlů GiST stromu a vrací predikát, který platí pro všechny listy, uzly všech podstromů s kořeny v dané množině.

pick_split – Funkce implementující tuto metodu bere ze vstupu množinu uzlů GiST stromu a vrací dvě množiny uzlů GiST stromu tak, že oba obsahují minimálně k následníků, kde k je minimální faktor zaplnění (minimum fill factor).

penality – Funkce implementující tuto metodu bere ze vstupu dva uzly GiST stromu a vrací hodnotu *postihu* za vložení druhého uzlu do podstromu uzlu prvního. Tato operace slouží k předejití `split` operací.

Zhodnocení

Implementací těchto metod tedy získáme plnohodnotný index, který na míru řeší sestavování stromového vyhledávací struktury nad množinou prvků našeho datového typu a vyhledávání prvků na základě dotazu. Pomocí indexu GiST lze relativně snadno implementovat tradiční indexační struktury, jako například B-strom a jeho varianty, R-strom a jeho varianty, nebo např. K-D strom.

GiST tedy umožňuje i indexaci multidimenzionálních dat, což byl náš základní požadavek na volenou indexační strukturu.

5.5.3 KNN GiST

KNN GiST[4] je podpora vyhledávání několika nejpodobnějších hodnot klíče v indexu GiST. Někdy se také označuje jako podpora pro řazení v GiST, což možná ještě lépe vyjadřuje o jakou úpravu se vlastně jedná. V zásadě jde o to, že oproti klasickému GiST pokud zadáváme dotaz na výsledky seřazené podle nějakého kritéria (typicky vzdálenosti dvou prvků), KNN GiST je vrátí již v tomto pořadí.

Z toho důvodu je není třeba znovu seřazovat (což mimo jiné vyžaduje průchod všemi vybranými záznamy, navíc je to z principu náročná operace, ať už časově či paměťově). Záznamy pak stačí jen omezit pomocí `LIMIT`, abychom vybrali k nejpodobnějších prvků. Kromě této úpravy v řazení je pro KNN GiST potřeba, aby měl datový typ, který chceme pomocí KNN GiST zaindexovat, definovaný operátor vzdálenosti dvou hodnot. K tomu je třeba použít některou vzdálenostní funkci (viz 5.5.1), případně definovat operátor pomocí funkce již existující.

Typický dotaz v jazyce SQL využívající KNN GiST má pak následující podobu:

```
SELECT point
,point <-> '(0, 0)'::point AS dist -- display point and distance
FROM public.points                  -- from points table
ORDER BY dist ASC                   -- in order from nearest points
LIMIT 10                            -- only 10 nearest are interesting
```

Díky tomuto rozšíření lze tedy poměrně jednoduše využít index KNN GiST pro podobnostní vyhledávání, čímž splňuje podmínku podpory pro podobnostní vyhledávání, přičemž je zachována možnost indexace multidimenzionálních dat. Z toho důvodu je KNN GiST vhodným indexem pro indexaci multidimenzionálních vektorů rysů.

5.6 Shrnutí

V této kapitole jsme si popsali základní návrh databázového schematu a popsali si datovou sadu a vektory rysů, které datová sada bude obsahovat.

Dále jsme si popsali základní principy analýzy hlavních komponent založené na kovariační matici, která nám pomůže snížit dimenzionalitu vektorů rysů, podle kterých chceme provádět vyhledávání na základě podobnosti.

[4] KNN GiST – K Nearest Neighbor GiST

V poslední sekci jsme si popsali princip indexu GiST a jeho varianty KNN GiST pro vyhledávání na základě podobnosti a vysvětlili jsme si, že KNN GiST je indexem dobře použitelným pro vyhledávání multidimenzionálních vektorů na základě podobnosti.

Teoreticky jsme si tedy uvedly nástroje, které umožní efektivnější vyhledávání ve videu na základě obsahu. Jelikož máme k dispozici nástroje pro automatickou extrakci vektorů rysů a nástroje, které nám umožní reprezentaci těchto rysů (jakožto dodatečnou informaci o snímcích videa) vhodnou pro indexaci umožňující vyhledávání na základě podobnosti těchto vektorů, máme k dispozici nástroje pro efektivní vyhledávání snímků na základě podobnosti (alespoň teoreticky, pochopitelně úspěšnost vyhledávání záleží především na kvalitě a vypovídajících schopnostech metod pro extrakci rysů a na míře ztrátovosti komprese při použití analýzy hlavních komponent).

V další kapitole si popíšeme nástroje a postupy a problémy týkající se implementace navrženého řešení pro vyhledávání ve videu.

Kapitola 6

Implementace

V předchozích kapitolách byly popsány obecné principy vyhledávání informací ať už v multimediálních či obecných databázích a byly vysvětleny základní principy indexace jak skalárních tak multidimenzionálních dat.

V předchozí kapitole byly provedeny základní návrhy řešení problému vysoké dimenzionality vektorů rysů, které byly vyextrahované z použité datové sady za účelem extrakce vysokoúrovňových rysů a vyhledávání videí na základě podobnosti. V této kapitole byla pak blíže popsána datová sada a velmi stručně byly popsány vektory rysů, které byly vyextrahovány z jednotlivých snímků. Nakonec jsme se zaměřili na možnost indexace multidimenzionálních dat pomocí indexu GiST a jeho variantu KNN GiST, která umožňuje využití indexu při vyhledávání vektorů na základě podobnosti.

V této kapitole si popíšeme jednotlivé kroky nezbytné při implementaci řešení, použité technologie a jejich aspekty důležité pro tuto práci. Diskutovány zde budou i základní problémy při implementaci řešení a jejich eliminace.

Nejdříve si popíšeme implementační kroky, které byly nezbytné pro započetí prací na zefektivnění dotazů na vyhledávání videí na základě podobnosti. Tyto budou obsahovat transformaci databázového schematu a úpravu demonstrační aplikace tak, aby byla schopná s novějším schematem pracovat.

V dalších sekcích budou popsány problémy spjaté s implementací nástroje pro využití analýzy hlavních komponent (dále PCA[1]) k transformaci vektorů rysů do prostoru o nižší dimenzionalitě. Budou poměrně podrobně popsány nástroje a knihovny, které byly při implementaci použity. Nakonec si uvedeme způsob, jakým byla data v databázi zaindexována. Informace z této kapitoly úzce staví na informacích, které byly uvedeny v kapitole předchozí.

6.1 Implementace nezbytných úprav

V této sekci budou popsány implementační práce, které předcházely implementaci samotného řešení, avšak byly nutnou podmínkou pro započetí prací na tomto řešení.

Pro dané řešení nemají však z pohledu efektivity vyhledávání ve videu příliš velký význam a proto je možné tuto sekci přeskočit (pro úplnost technické zprávy o diplomové práci je však nezbytná) a pokračovat sekci 6.2.

V rámci tohoto projektu bylo ve fázi přípravy na implementaci samotného řešení nutné provést dvě činnosti. První z nich byla transformace dat, získaných z datové sady *Sound and Vision* (viz 5.2.1) a následné extrakce rysů (viz 5.2.2), z původního databázového schematu

[1] PCA – Principal Component Analysis

do nového. Druhou činností pak byla úprava aplikace TRECVid Search (viz 5.3) tak, aby byla schopna využívat toto nové schema.

6.1.1 Transformace datové sady

Tato sekce pojednává o problémech a postupu transformace datové sady ze staršího schematu popsaného v sekci 5.2.3 na schema novější (viz 5.2.4). Jelikož je tato sekce zaměřená na implementační detaily transformace, kde termín `schema` vystupuje v podobě databázového objektu, budeme jej potřebovat rozlišit od termínu *schema* ve smyslu struktury databázových tabulek (jak byl chápán v předešlých částech textu). V této sekci tedy *schema* bude databázovým objektem `SCHEMA` a pod pojmem *databázový model*

Obě datové sady byly již popsány v kapitole 5, proto zde uvedeme jen několik informací důležitých při transformaci.

Z důvodu velikosti transformovaných dat bylo zapotřebí provést transformaci po menších částech tak, abychom plně nevytížili databázový server na neúměrně dlouhou dobu. Jako vhodný krok bylo zvoleno 10000 záznamů v každé dávce.

Na rozdíl od původního databázového modelu, který byl celý obsažen v jednom databázovém schematu (chápáno jako databázový objekt), v novém modelu obsahuje databáze jedno schema pro každou datovou sadu. V našem případě budou tedy data rozdělena do dvou databázových schemat `trecvid08` a `trecvid09`. Tabulka `datasets`, která slouží k popisu jednotlivých datových sad je pak přístupná ve schematu `public`. Z tohoto důvodu se transformovala data z jednotlivých sad odděleně.

Mapování mezi modely

V novějším databázovém modelu jsou v zásadě, pro každou datovou sadu, dvě tabulky. Jejich význam a vztah ke staršímu modelu jsou následující:

`sequence` – Tato tabulka má význam audiovizuální sekvence. V našem případě se jedná o tabulku, která má stejný význam jako tabulka `tv_videa` ve starším modelu. Náležitost do datové sady je přitom dána přímo databázovým schematem, ve kterém se tabulka nachází.

`interval` – Tato tabulka popisuje obsah jistého časového intervalu sekvence. V našem případě se jedná o spojení tabulek `tv_shot`, `tv_keyframe` a `tv_localfeature`, přičemž pro každý záběr (shot) vybíráme pouze jediný klíčový snímek (keyframe) s jeho lokálními rysy (local features).

Nyní, když víme, jaké jsou vztahy mezi tabulkami ze původního modelu a z novějšího modelu, můžeme si definovat vztahy jednotlivých sloupců. Tyto vztahy popisuje tabulka 6.1.

V této tabulce jsou popsány transformace sloupců rozčleněny na jednotlivé dvojice tabulek (levá je z nového modelu, pravá ze staršího), přičemž v levém sloupečku se nacházejí cílové sloupce transformace (v novém modelu), v prostředním sloupečku vidíme zdrojové sloupce (ze staršího modelu) a v pravém sloupečku je stručná charakteristika operace, která byla při transformaci použita (*equiv* je prosté přiřazení hodnoty beze změny, *const* je přiřazení nějaké konstanty, `NULL` je ponechání cílového sloupce prázdného, *cast* znamená použití transformační funkce a *conat* znamená nějaké spojení hodnot zdrojových sloupců s použitím řetězcové konkatenace).

Cílový sloupec	Zdrojové sloupce	Metoda
sequence	tv_videa	
seqnum	id	equiv
seqlocation		const
seqname	name	equiv
seqtyp	filetype	cast
fps	fps	equiv
length	length	equiv
duration	duration	equiv
interval	tv_shot	
t1	from	equiv
t2	to	equiv
shot	id	equiv
imglocation	video, id	concat
tags		NULL
interval	tv_keyframe	
id	id	equiv
frame	frame	equiv
color	color	equiv
hist	hist	equiv
grad	grad	equiv
gabor	gabor	equiv
face	face	equiv
confidences	confidences	equiv
interval	tv_localfeature	
siftser_count	siftser_count	equiv
siftsers	siftsers	equiv
siftser_weights	siftser_weights	equiv
siftser_norm	siftser_norm	equiv
surf_count	surf_count	equiv
surfs	surfs	equiv
surf_weights	surf_weights	equiv
surf_norm	surf_norm	equiv

Tabulka 6.1: Tabulka mapování sloupců

6.1.2 Úprava aplikace TRECVid Search

Hlavní náplní úprav aplikace TRECVid Search byla adaptace na novější databázový model a oprava rozložení grafického uživatelského rozhraní, které bylo během let, následkem začlenění úprav od velkého množství studentů v rámci diplomových prací, výrazně poškozené.

Adaptace byla provedena více méně účelovým způsobem jen nahrazením jednotlivých dotazů dotazy novými. Významější refaktorizace kódu ve smyslu naimplementování složitějšího a flexibilnějšího modelu či použití nějaké pokročilejší technologie, jelikož se jednalo o činnost mimo přímé zaměření této diplomové práce.

Oprava rozložení uživatelského rozložení proběhla poměrně důkladně, jelikož bylo ve špatném stavu a oprava samotná nebyla natolik složitá (díky předchozím zkušenostem

s vývojem uživatelských rozhraní ve Swing frameworku) a časově náročná. Příklad pro porovnání rozložení před a po úpravě je ilustrován v příloze B.

6.2 Problémy spojené s implementací PCA

Hlavními problémy při implementaci tohoto modulu byly následující:

- Paměťová náročnost implementace a pochopení správy paměti v serveru PostgreSQL.
- Provádění dotazů nad databází přímo z implementovaných funkcí.
- Nástroj pro efektivní implementaci výpočtů vycházejících z lineární algebry.
- Překlad a dynamické načítání implementovaných funkcí do databázového systému.

Vzhledem k tomu, že některé operace (např. `build_pca`, viz 5.4.3) je potřeba provádět nad celou vstupní tabulkou v databázi, je třeba si uvědomit, že při špatném návrhu implementace velmi snadno dojde k nepřiměřenému či nepřijatelnému paměťovému vytížení serveru, které bude mít za následek pád procesu. Rozhodně například nemůžeme načíst celou tabulku do paměti a následně nad ní provádět operace. Dalším nepříjemným faktem je, že operace nad celou tabulkou jsou z principu vysoce časově náročné (přestože tato vlastnost se drží lineární časové složitosti).

Zatímco paměťovou náročnost řešení máme plně v rukou, časovou náročnost průchodu celé databáze nijak neovlivníme. Z tohoto důvodu je potřeba říci, že jelikož se v našem případě jedná o práci s databází vyhledávací (nikoli operační), ve které se příliš často nemění obsah, můžeme si jistou časovou náročnost dovolit. Zde zmiňované operace stačí provádět jen při opravdu významných změnách obsahu databáze, při kterých je třeba počítat s hned několika časově náročnými operacemi (např. extrakce rysů nebo zaindexování dat).

Pro implementaci operací definovaných v návrhu (viz 5.4.3) je zapotřebí nástroje, který nám zpřístupní objekty databáze jak pro čtení tak pro zápis. Uvedené operace vždy pracují se vstupy, které získají z tabulky `PCA` a některé i ovlivňují obsah některých tabulek. Přitom by bylo nanejvýš nevhodné použít nějaké klientské rozhraní poskytované pro běžné volaní dotazů do databáze (když pomineme fakt, že bychom na serveru používali klientské API[2], tak hlavním důvodem je především zbytečná komunikace po síti, či přes lokální síťové rozhraní, a s ní spojené časové nároky a režie). Naštěstí systém PostgreSQL nabízí právě takové aplikační rozhraní popsané v podsekci 6.4.4.

Jelikož implementace knihovny pro operace a struktury pro reprezentaci objektů lineární algebry nejsou náplní této diplomové práce, bylo zapotřebí najít nějakou volně dostupnou knihovnu, která již potřebnou funkcionalitu implementuje. Popis této knihovny (především součástí využitých v této práci) budeme diskutovat v sekci 6.3.

Ostatní problémy týkající se převážně implementace funkcí pro databázový systém PostgreSQL bude popsán v sekci 6.4.

6.3 Knihovna GSL

Pro efektivní implementaci některých částí analýzy hlavních komponent bylo potřeba použít nějakou programovou knihovnu, která řeší implementaci operací lineární algebry. Převážně

[2] Application Programming Interface – Aplikační rohraní

se jedná o operace nad maticemi a vektory. Velmi důležitými faktory pro výběr knihovny byla i možnost výpočtu vlastních vektorů a hodnot.

Po zvážení jsem vybral knihovnu GSL (GNU Scientific library), která využívá knihovnu BLAS (Basic Linear Algebra Subprograms) pro jednoduché vektorové a maticové operace. GSL obsahuje rozhraní pro vyhledávání vlastních hodnot a vektorů ze zadané matice. Tato knihovna je open-source řešením vyvíjeným organizací GNU v jazyce C. GSL implementuje velké množství matematických nástrojů, přičemž tato diplomová práce využije pouze operace a struktury pro práci s maticemi a vektory (moduly `(gsl_vector*)`, `(gsl_matrix*)` a podpora BLAS v GSL).

Důležitý rysem knihovny je možnost práce s maticemi a vektory pomocí tzv. „pohledů“, kdy pro již existující blok paměti, kterým reprezentujeme nějaký vektor nebo matici, můžeme vytvořit dočasnou datovou strukturu, která blok paměti obalí vhodným způsobem. Tato struktura je pak již vhodná pro operace knihovny a bez jakékoli dodatečné alokace paměti (tyto pohledy jsou tedy nezávislé na konkrétní použité technologii správy paměti) tak umožní provádět operace definované rozhraním GSL, případně BLAS, nad naším blokem dat. Tato knihovna se při použití dynamicky připojuje (linkuje) k aplikaci (lze zvolit i variantu se statickým připojením v době překladu).

Knihovna chápe matice reprezentované vícerozměrným polem, v lineárním bloku kódu, jako řádkově orientované (row-major), což se shoduje s vnímáním vícerozměrných polí v jazyce C (první index je řádek, druhý je sloupec, v paměti jsou řádky ukládány za sebou). Tím se knihovna odlišuje např. od široce využívané knihovny LAPACK (Linear Algebra PACKage), která vnímá vícerozměrná pole jako sloupcově orientovaná (dáno původem z jazyka FORTRAN – v paměti jsou za sebou ukládány sloupce).

Knihovna je stabilní a je aktivně používána například v GNU Octave projektu (open-source alternativa k MATLABu).

Informace pro tuto sekci jsem čerpal výhradně z referenční příručky [21] a osobních zkušeností s knihovnou.

6.3.1 Konvence pojmenování typů a jejich operací

V této podseckci bude popsána konvence pojmenování typů a operací v knihovně GSL. Popis konvence je důležitý, jelikož většina typů a rozhraní má větší množství variant, např. podle typu dat, která jsou v nich ukládána. Z důvodu omezeného prostoru budu vždy uvádět jen základní typy a operace, které pracují s datovým typem `double`. Pro ostatní datové typy jsou pak typicky všechny obdobné operace a typy taktéž dostupné (při dodržení pojmenovávací konvence).

V knihovně GSL většina názvů datových typů začíná prefixem `gsl_`, za kterým pak následuje samotný název datového typu (např. `matrix` nebo `vector`). Pokud v typu není řečeno jinak, implicitně pracuje s daty typu `double`, neboli s daty v plovoucí řadové čárce s dvojnásobnou přesností. Pokud například chceme typ, který pracuje s hodnotami typu `float` nebo `int`, následuje v názvu typu řetězec `_float`, resp. `_int` (např. `gsl_vector_float`, `gsl_matrix_int`). Toto pravidlo se vztahuje na všechny typy číselných dat (`ulong`, `char`, ...). Pokud se jedná o pohled (viz 6.3.2), tak přibyde v názvu ještě suffix `_view`.

Operace nad datovými typy pak začínají celým jménem typu, následovaným podtržítkem a pak názvem operace (např. `gsl_vector_int_view_array_with_stride`).

6.3.2 Maticové a vektorové pohledy

Knihovna GSL umožňuje alokaci paměti pro matice a vektory požadované velikosti pomocí funkcí ve tvaru:

```
gsl_vector* gsl_vector_alloc(size_t n);
gsl_vector* gsl_vector_calloc(size_t n);
```

Obě tyto funkce alokují paměť pro vektor s `n` dimenzemi. Tyto dvě funkce se liší pouze tím, že `gsl_vector_calloc` inicializuje všechny složky vektoru na 0. Pro uvolnění paměti se pak použije funkce:

```
void gsl_vector_free(gsl_vector* v);
```

Tato funkce je obdobou funkce `free` ze standardní knihovny `stdlib.h`, která je však poskytována knihovnou GSL pro uvolňování vektorů. Při použití těchto metod pak necháváme knihovnu GSL, aby se postarala o alokaci paměti, což nemusí být vždy optimální.

Alternativou k tomuto přístupu je vytvoření vlastního bloku paměti pomocí nástrojů nebo knihoven vhodných pro konkrétní případ a vytvoření pohledu nad tímto blokem. Pro příklad uvedu jednoduchý příklad kódu:

```
// own memory allocation
double *vec_block = my_alloc(VEC_DIMMS * sizeof(double));
gsl_vector_view view = gsl_vector_view_array(vec_block, VEC_DIMMS);
for (int i = 0; i < view.vector.size; i++) {
    gsl_vector_set(&view.vector, i, 10); // set each entry to 10
}
```

Jak je v příkladu ukázáno, nejdříve si alokuji vlastními prostředky blok paměti a pak následovně vytvořím pohled, který mi již zpřístupní GSL funkce implementující vektorové operace, přičemž tyto operace jsou prováděny přímo nad námi alokovaným blokem paměti. Vektorové pohledy mohou mít i charakter konstant (vektorů pouze pro čtení). Pro získání vektorových pohledů slouží tyto funkce:

```
gsl_vector_view gsl_vector_view_array (double * base, size_t n);
gsl_vector_view gsl_vector_view_array_with_stride (double * base,
            size_t stride, size_t n);

gsl_vector_view gsl_vector_subvector (gsl_vector * v,
            size_t offset, size_t n);
gsl_vector_view gsl_vector_subvector_with_stride (gsl_vector * v,
            size_t offset, size_t stride, size_t n);
```

První dvě funkce souží k vytvoření pohledu nad námi alokovaným polem o velikosti `n`, druhé dvě jsou již součástí repertoáru operací s vektory a vytvářejí nový pohled nad již existujícím vektorem, který reprezentuje podvektor původního vektoru. Je dobré mít na paměti, že jak původní vektor, tak nově vytvořená struktura jeho podvektoru spolu sdílejí paměť. Druhá funkce z páru vždy přidává možnost definovat `krok` ve vektoru. Jinými slovy umožňuje brát v potaz například každý n-tý prvek.

V GSL existují obdoby těchto operací pro vektory a matice všech číselných datových typů. V případě matic je jedna změna a sice tam, kde je u vektorů umožněno definovat krok, je u matic možné definovat velikost řádku v paměti (jinými slovy počet sloupců), tento údaj je pak použit při přístupu do jednotlivých polí matice.

6.3.3 Vektorové a maticové operace GSL

V minulé podsekci jsme si ukázali několik základních operací nad vektory knihovny GSL vysvětlili jsme si, k čemu jsou pohledy a jak je získat. V této podsekci si ukážeme několik jednoduchých operací nad vektory, které knihovna GSL nabízí. Jsou to operace definované následujícím výčtem funkcí:

```
int gsl_vector_add (gsl_vector * a, const gsl_vector * b);
int gsl_vector_sub (gsl_vector * a, const gsl_vector * b);
int gsl_vector_mul (gsl_vector * a, const gsl_vector * b);
int gsl_vector_div (gsl_vector * a, const gsl_vector * b);

int gsl_vector_scale (gsl_vector * a, const double x);
int gsl_vector_add_constant (gsl_vector * a, const double x);
```

Jedná se o jednoduché binární operace buď mezi dvěma vektory (první skupina), nebo mezi vektorem a konstantou (druhá skupina). Výsledný vektor je vždy uložen do do vektoru určeného prvním operandem.

První skupina operací slouží k sečtení, odečtení vynásobení nebo vydělení všech složek vektoru `a` a odpovídajících složek vektoru `b`.

Druhá skupina operací slouží k vynásobení všech složek vektoru konstantou (změna délky vektoru `x`-krát při zachování směru) a přičtení konstanty `x` ke všem složkám vektoru.

Pro matice opět existuje stejná množina operací. Jediným rozdílem v konvenci pojmenování jsou funkce:

```
int gsl_matrix_mul_elements (gsl_matrix * a, const gsl_matrix * b);
int gsl_matrix_div_elements (gsl_matrix * a, const gsl_matrix * b);
```

Jejich názvy jsou upraveny pro lepší vypovídací schopnost o tom, co operace delají (nenásobí/nedělí se matice mezi sebou, ale jen jejich odpovídající prvky navzájem).

Jak je vidět, nenacházejí se zde žádně operace pro násobení vektorů s maticemi a podobně. Tyto operace jsou implementovány právě v knihovně BLAS.

6.3.4 Výpočet vlastních hodnot a vektorů matice

Jednou z nejdůležitějších nástrojů, které byly po knihovně vyžadovány byla implementace vyhledávání vlastních hodnot a vektorů z matice. Knihovna GSL poskytuje pro tuto funkcionalitu velmi jednoduché rozhraní:

```
int gsl_eigen_symmv (gsl_matrix * A, gsl_vector * eval,
            gsl_matrix * evec, gsl_eigen_symmv_workspace * w);
```

Tato operace vyhledá všechny vlastní hodnoty a vektory symetrické matice `A` a uloží je do vektoru `eval` a matice `evec`. Posledním parametrem je pracovní datový typ, který je potřeba nejdříve alokovat pomocí funkce `gsl_eigen_symmv_alloc` a posléze uvolnit pomocí funce `gsl_eigen_symmv_free`.

Existují i obdoby těchto operací, jejichž názvy se odvíjejí od typu vstupní matice.

6.3.5 Podpora BLAS

Posledním modulem využitým v rámci vektorových a maticových operací je rozhraní, které GSL knihovna poskytuje pro volání funkcí BLAS. Pojmenovávací konvence je přibližně následující: (v potaz budeme brát pouze operace násobení matic a vektorů).

Každá BLAS operace násobení má v GLS prefix `gsl_blas_` přičemž následuje 5 nebo 6 znaků specifikující typ operandů a konkrétní operace. První znak identifikuje typ číselných dat (`s` pro `float`, `d` pro `double`, ...). Následují dva znaky identifikující typ matice (`ge` pro obecnou, `sy` pro symetrickou, ...). Poslední dva až tři znaky identifikují danou operaci. V našem případě (operace násobení) se jedná jen o dva symboly identifikující typy operandů (`mm`, resp. `mv` pro násobení matice maticí, resp. matice vektorem).

Následující příklad ukazuje BLAS operaci násobení obecné matice vektorem (v dvojnásobné přesnosti čísel s plovoucí řadovou čárkou):

```
int gsl_blas_dgemv (CBLAS_TRANSPOSE_t TransA, double alpha,
            const gsl_matrix * A, const gsl_vector * x,
            double beta, gsl_vector * y);
```

Jak je vidět z definice funkce, knihovna GSL poskytuje pro BLAS rozhraní, ve kterém již počítá s datovými typy knihovny GSL (např. zde `gsl_matrix` a `gsl_vector`).

6.4 Programování funkcí pro PostgreSQL

Jak již bylo řečeno dříve, implementace nástroje pro transformaci vektoru rysů do prostoru s nižší dimenzionalitou byla provedena v jazyce C (v jeho základní, neobjektové, podobě) formou implementace uložených funkcí do databázového systému PostgreSQL.

Tato sekce bude pojednávat o možnostech, nástrojích a problémech spjatých s implementací funkcí v tomto prostředí. Pro tuto práci se jedná o sekci opravdu klíčovou a proto jí bude věnován značný prostor, nicméně popsány budou pouze ty nejdůležitější rysy. Informace pro tuto sekci byly čerpány převážně z [22].

6.4.1 Načítání funkcí

Uložené funkce (nebo procedury, chcete-li) systému PostgreSQL implementované v jazyce C jako rozšíření portfolia funkcí dostupných po instalaci serveru je potřeba samostatně přeložit pomocí nějakého standardizovaného překladače jazyka C (např. GCC[3]) jako dynamicky načítanou, neboli sdílenou knihovnu (shared library).

Takovou knihovnu je pak možné načíst přímo z SQL konzole serveru pomocí absolutní cesty (příkaz `LOAD 'absolute_path';`), nebo je možné knihovnu přemístit do adresáře ve kterém PostgreSQL svoje knihovny hledá automaticky a v příkazu `LOAD` uvést relativní cestu k souboru s knihovnou. Standardní přípona souboru by měla být `.so`.

6.4.2 Základní datové typy

Základní datové typy používané při programování uložených funkcí pro PostgreSQL se dělí na tři základní druhy:

[3] GCC – GNU C Compiler

Předávané hodnotou s fixní délkou – pouze hodnoty s velikostí 1, 2, 4 (případně 8, pokud je `sizeof(Datum)` rovno 8-mi na daném systému). Je potřeba definovat datové typy opatrně tak, aby měli na různých strojích stejnou velikost (jazyk C je známý tím, že jeho standard neurčuje striktně velikost základních datových typů, ale nechává ji na překladači tak, aby zvolil optimální velikosti pro danou platformu).

Předávané odkazem s fixní délkou – Oproti typům předávaným hodnotou mohou mít libovolnou velikost (přestože fixní). Tato varianta se používá například pro rozsáhlejší heterogenní struktury.

Předávané odkazem s proměnlivou délkou – Oproti předchozím typům, v tomto případě může být celková velikost datové položky pokaždé zcela jiná. Takový datový typ pak začíná přesně čtyřmi bajty obsahujícími délku datového typu (zbylou) za níž pak následuje blok paměti obsahující hodnotu nebo hodnoty daného datového typu.

Příklady definic datových typů jednotlivých druhů mohou být následující:

```
/* fixed length by value passed 4B integer */
typedef int32 int4;

/* fixed length by reference passed 12B type */
typedef struct {
    float4 x;
    float4 y;
    float4 z;
} Point3d;

/* variable length by reference passed type */
typedef struct {
    int4 length;
    char data[1];
} text;
```

[22] tvrdí, že abychom pak mohli používat takový typ o neznámé velikosti, použijeme trik, kdy alokujeme prostor o velikosti `length` přičemž nastavíme hodnotu prvních čtyřech bajtů tohoto prostoru tak, aby reflektovaly skutečně alokovanou velikost paměti. Dále již (pomocí specializovaných operací) pracujeme s položkou data jako s ukazatelem na dynamicky alokovanou paměť. Jedná se o poměrně využívaný trik, který využívá jistých nedostatků jazyka C, aby odstranil nedostatek jiný. Příklad použití je následující:

```
int4 size = VARHDRSZ + STRLEN * sizeof(char);
text *text = (text *) palloc(size);
SET_VARSIZE(text, size);
strncpy(text->data, source, STRLEN * sizeof(char));
```

V uvedeném příkladě jsou použita makra `VARHDRSZ` (makro definující velikost hlavičky – 4 B) a `SET_VARSIZE` které uloží na první 4 B paměti odkazované ukazatelem `text` číslo `size`. Při programování funkcí pro systém PostgreSQL se obecně potkáte s velkým množstvím maker, která usnadňují práci s jinak poněkud nepraktickými strukturami, které systém používá (především pro jejich univerzálnost).

6.4.3 Rozhraní pro jazyk C

Pro implementaci funkcí v jazyce C poskytuje systém PostgreSQL dvě možná rozhraní funkcí. Prvním takovým rozhraním je tzv. volací konvence „verze 0“ (Version 0 Calling Conventions). Druhým je pak tzv. volací konvence „verze 1“ (Version 1 Calling Conventions). Obě tyto konvence a jejich rozdíly si uvedeme v dalším textu.

Volací konvence verze 0

Jedná se o prakticky běžné definice funkcí, přičemž argumenty a návratový typ určujeme v definici funkce právě tak, jak je očekáváme. Například:

```
text *concat_texts(text *t1, text *t2) {
    text *result = (text *) palloc(t1->length + t2->length - VARHDRSZ);
    SET_VARSIZE(text, t1->length + t2->length - VARHDRSZ);
    strncpy(text->data, t1->data, t1->length - VARHDRSZ);
    strncat(text->data, t2->data, t2->length - VARHDRSZ);
    return result;
}
```

Po přeložení a načtení knihovny s funkcí do systému PostgreSQL, vytvoříme uloženou funkci následovně:

```
CREATE FUNCTION concat_texts(text, text) RETURNS text
    AS 'DIRECTORY/funcs', 'concat_texts'
    LANGUAGE C STRICT;
```

V této chvíli máme funkci dostupnou v prostředí PostgreSQL a můžeme ji volat např. pomocí SQL dotazů. Klíčové slovo `STRICT` říká, že funkce vrací automatický prázdný výsledek (`NULL`), pokud je alespoň jeden z argumentů prázdný. Také je dobré upozornit na alokaci paměti pomocí makra (nikoli funkce) `palloc()`. Jak bude popsáno dále6.4.4, jedná se o speciální volání poskytované systémem PostgreSQL pro alokaci paměti, která (na rozdíl od systémového volání `malloc()`) umožňuje databázovému systému hlídat a spravovat paměť (a např. předcházet „memory-leakům[4]“).

Přestože je tato konvence jednoduchá na použití, není příliš přenositelná a nese s sebou některé problémy týkající se např. obtížným předáváním prázdných výsledků. Z těchto důvodů je tato konvence označena jako zastaralá a je v systému přístupná pouze z důvodu zpětné kompatibility (tvrdí [22]).

Tato konvence je dnes již plně nahrazena konvencí verze 1.

Volací konvence verze 1

Oproti předchozí konvenci, v této mají funkce vždy stejné rozhraní:

```
PG_FUNCTION_INFO_V1(my_funcion);
/* my function description */
Datum my_funcion(PG_FUNCTION_ARGS) {
    ...
}
```

[4]Situace, kdy zahodíme ukazatel na paměť, kterou neuvolníme a paměť je nedostupná pro ostatní procesy

Zde je důležité, aby makro `PG_FUNCTION_INFO_V1` s názvem naší funkce bylo ve stejném zdrojovém souboru, jako definice této funkce.

Jednotlivé atributy funkce pak získáváme pomocí makra `PG_GETARG_XX(n)`, kde `XX` je název očekávaného datového typu velkými písmeny a `n` je pořadové číslo argumentu počítáno od 0. Pro všechny standardní typy systému PostgreSQL jsou tato makra již definována a pro uživatelsky definované typy je dobrým zvykem je dodefinovat. Pro vracení hodnot jsou pak definována makra `PG_RETURN_XX(val)` kde `XX` je opět název datového typu proměnné, kterou chceme vrátit a `val` je název této proměnné.

Za zmínku ještě stojí makra `PG_ARGISNULL(n)` a `PG_RETURN_NULL()` která slouží k detekci prázdné hodnoty argumentu, případně k vrácení prázdné hodnoty.

V dalším textu budeme uvažovat pouze tuto konvenci.

Práce s kompozitními typy

Pod pojmem *kompozitní typ* si můžeme představit řádek tabulky nebo pohledu ať už materializovaného, nebo sestaveného na základě dotazu. Například můžeme definovat funkci, která má za jeden z atributů řádek tabulky:

```
Datum
has_null_fname(PG_FUNCTION_ARGS) {
    HeapTupleHeader header = PG_GETARG_HEAPTUPLEHEADER(0);
    bool isnull;
    GetAttributeByName(header, "fname", &isnull); /* ignore val */
    PG_RETURN_BOOL(isnull);
}
```

Pro práci s kompozitními typy je k dispozici datový typ `HeapTupleHeader` a odpovídající sada maker a funkcí, která nám usnadňuje práci s ním. Například funkce `GetAttributeByName` vrátí hodnotu sloupce podle jména a nastaví flag `isnull` podle toho, jestli je hodnota tohoto sloupce prázdná či nikoli.

Pro funkce vracející kompozitní typ je situace o poznání složitější, jelikož musíme sestavit proměnnou typu `HeapTuple`. V zásadě máme dva způsoby, jak toho můžeme dosáhnout. Prvním způsobem je sestavení řádku z pole hodnot typu `Datum`, které naplníme hodnotami odpovídajících datových typů, nebo řádek sestavíme z hodnot typu `CSTRING`[5], pro které se pak provede konverze do odpovídajících typů. V tomto textu si popíšeme pouze první způsob (z prostorových důvodů).

Při sestavování `HeapTuple` musíme nejdříve sestavit *deskriptor* typu `TupleDesc`. Toho můžeme dosáhnout například tak, že zavoláme funkci `get_call_result_type`, která nám naplní deskriptor na základě očekávané výstupní struktury (která je dána definicí volané varianty funkce v SQL). To nám mimo jiné umožní jistou míru polymorfizmu[6] výstupu, takže tato funkce je vhodná i pro funkce vracející skalární hodnoty.

Když už máme deskriptor, můžeme nad ním zavolat funkci `BlessTupleDesc` a následně pro každý řádek výstupu zavoláme `heap_from_tuple` (kdy předáváme pole prvků typu `Datum` a pole prvků typu `bool` které říká, jestli je hodnota v daném sloupci prázdná).

[5] Standardní reprezentace řetězce ukončeného prázdným znakem

[6] Mnohotvarosti

Práce s poli

Na rozdíl od kompozitních typů, které jsou heterogenní a jejich struktura a délka je definována deskriptorem, pole je struktura homogenní, jehož délka je obecně proměnlivá a musí být distribuována spolu s Tímto polem. PostgreSQL však poskytuje rozhraní pro obecně vícedimenzionální pole, což implikuje nutnost specifikace délky každé dimenze zvlášť.

Funkce zpracovávající jednorozměrné pole může být definována například takto:

```
Datum
array_mean(PG_FUNCTION_ARGS) {
    ArrayType* inputArray = PG_GETARG_ARRAYTYPE_P(0);
    float4* values = ARR_DATA_PTR(inputArray);
    size_t length = ArrayGetNItems(ARR_NDIM(inputArray),
        ARR_DIMS(inputArray));
    double sum = 0.0;
    size_t i;
    for(i = 0; i < length, i++) {
        sum += values[i];
    }
    sum /= (double) length;
    PG_RETURN_FLOAT4((float4) sum);
}
```

Přístup k prvkům pole nám tedy usnadňují makra `ARR_DATA_PTR`, `ARR_DIMS` a `ARR_NDIM`, která nám poskytují ukazatel na samotná data v poli, ukazatel na pole obsahující počet prvků v každé dimenzi a počet dimenzí. Funkce `ArrayGetNItems` vrací počet prvků pole napříč všemi dimenzemi. Jelikož jsou data všech dimenzí ukládána v jednom bloku paměti, můžeme i vícedimenzionální pole procházet jako jednodimenzionální.

6.4.4 Rozhraní SPI

V předešlé podsekci jsme si popsali základní přístupy a datové typy používané pro programování funkcí systému PostgreSQL v jazyce C. V této podsekci si podrobně popíšeme práci s rozhraním SPI[7], které zpřístupňuje nástroje pro dotazování do databáze pomocí jazyka SQL přímo z prostředí námi definovaných databázových funkcí.

Rozhraní SPI je jakousi obdobou klientského rozhraní pro dotazování do databáze, avšak s tím rozdílem, že je určeno pro funkce, které běží uvnitř databázového serveru. Podle [22] SPI zjednodušuje přístup k parseru, plánovači (planner) a spouštěči (executor). SPI také řeší správu paměti (viz 6.4.4).

Správa paměti

Systém PostgreSQL alokuje paměť v rámci tzv. *paměťových kontextů* (memory contexts). Zničení takového kontextu má za následek uvolnění veškeré paměti, která v tomto kontextu byla alokována. Díky tomu není nezbytně nutné hlídat si uvolňování paměti abychom předešli memory leakům (tvrdí [22]).

Makro `palloc()` a související funkce `pfree()` a `repalloc()` pracují s *aktuálním kontextem*. SPI poskytuje funkce, které umožňují přepínání paměťového kontextu. Jsou to

[7] SPI – Server programming interface

funkce `SPI_connect()`, která vytvoří nový paměťový kontext a označí jej za aktuální, a `SPI_finish()`, která zničí aktuální kontext (uvolní všechnu paměť v něm alokovanou) a nastaví původní kontext jako aktuální.

SPI poskytuje vlastní funkce pro správu paměti a sice `SPI_palloc()`, `SPI_fpree()` a `SPI_repalloc()`. Tyto funkce využívají o původní kontext, který byl aktuální dříve, než byla zavolána funkce `SPI_connect()`. Hlavním využitím je tedy alokace paměti v původním kontextu, což potřebujeme například pro proměnné, jejichž hodnotu chceme vrátit (po zavolání `SPI_finish()`).

Pokud zavoláme `SPI_palloc()` dříve než `SPI_connect()`, bude mít volání identický efekt, jako bychom zavolali makro `palloc()`. To je způsobeno tím, že před připojením ke *správci SPI* (SPI manager) je aktuálním paměťovým kontextem *vyšší kontext spouštěče* (podle [22]).

Dotazování do databáze

Základní funkcionalitou SPI je umožnění volání SQL dotazů do databáze, čímž můžeme jednak získávat data která se v databázových objektech nacházejí, jednak nám SPI umožní upravovat obsah databázových objektů přímo v kódu naší funkce. Před dotazováním do databáze musí vždy předcházet volání `SPI_connect()`, které krom vytvoření nového paměťového kontextu vytvoří připojení k *SPI správci* (SPI Manager).

Pro dotazování do databáze nabízí SPI hned několik funkcí rozhraní. Stručně si je vyjmenujeme a vysvětlíme základní rozdíly:

`SPI_execute` – Základní podoba dotazu do databáze. Jako parametry dostává flag říkající, jestli se jedná o operaci jen pro čtení a počet řádků, které mohou být dotazem ovlivněny nebo vráceny.

`SPI_exec` – Identické volání jako `SPI_execute` s tím, že nezadáváme zmíněný flag a jeho hodnota je chápána jako nepravdivá (umožňujeme číst i zapisovat).

`SPI_execute_with_args` – Parametrizovaný dotaz do databáze. Kromě argumentů, které sdílí s `SPI_execute` má ještě možnost zadat hodnoty parametrů v podobě typu `Datum`.

`SPI_prepare` – Sestaví plán spuštění (execution plan) bez toho, aby se provedl. Jako argumenty si bere (kromě dotazu) počet argumentů a pole jejích typů. Tato varianta je velmi vhodná v případě, že několikrát voláme stejný dotaz pouze s různými argumenty, jelikož umožňuje opakovaně využít stejný plán spuštění a nenutí databázový systém, aby jej vždy sestavoval znovu.

`SPI_execute_plan` – Obdoba `SPI_execute`, ale s využitím již existujícího plánu spuštění.

`SPI_execute_plan_with_paramlist` – Jako předešlé volání ale s možností předání parametrů.

`SPI_execp` – Obdoba `SPI_execute_with_args`, ale s využitím již existujícího plánu spuštění a vždy s povoleným jak čtením, tak zápisem.

Zpracování výsledků dotazu

Všechny funkce SPI, které způsobují provádění SQL dotazu nebo příkazu, ovlivňují několik globálních proměnných. Jedná se především o proměnné `SPI_processed`, která říká kolik

řádků bylo dotazem ovlivněno, případně kolik řádků bylo vráceno v odpovědi na dotaz (v závislosti na tom, jestli se jednalo o dotaz `SELECT` nebo `INSERT/ UPDATE/DELETE`) a proměnnou `SPI_tuptable` což je ukazatel na strukturu obsahující vysledek dotazu v případě příkazu `SELECT`.

Proměnná `SPI_tuptable` obsahuje deskriptor tabulky, který říká na kterém indexu sloupce jsou data jakého typu a jak se sloupec jmenuje. Pro vyčítání dat z této struktury je pak definováno několik funkcí:

`SPI_fname` – Vrátí název sloupce podle zadaného indexu a deskriptoru řádku.

`SPI_fnumber` – Vrátí index sloupce podle zadaného názvu a deskriptoru řádku.

`SPI_getvalue` – Vrátí řetězcovou reprezentaci hodnoty na daném řádku a sloupci.

`SPI_getbinval` – Vrátí binární hodnotu na daném řádku a sloupci, nastavuje flag `isnull`.

`SPI_gettype` – Vrátí název datového typu sloupce podle indexu a deskriptoru řádku.

`SPI_gettypeid` – Vrátí Oid datového typu sloupce podle indexu a deskriptoru řádku.

Důležitou informací je, že data vrácená v dotazu se neuvolňují a proto, pokud je chceme uvolnit, musíme zavolat `SPI_freetuptable()`, které paměť alokovanou pro data uvolní. Některé funkce vracející názvy vytvářejí kopie původních dat. Pokud tedy provádíte nějaký iterační výpočet, kde se každý kus neuvolněné paměti nepříjemně projeví, doporučuji se obrátit na referenční příručku a zjistit si, jestli se v dané funkci vytváří kopie, nebo jestli se jen vytváří další reference.

Parametrizace dotazů

SPI umožňuje parametrizaci dotazů na binární úrovni, což je jednak výhodné pro předejití SQL injection a jiným hrozbám, ale navíc i velmi výhodné v kombinaci s použitím předvytvořeného plánu spuštění. V takové situaci máme již rozparsovaný a přeložený plán uchovaný a pro jednotlivá volání (kterých může být i velké množství) stejného dotazu neustále používáme již předvytvořený plán. Tím se výrazně sníží režie provádění dotazu a celý proces se výrazně urychlí.

Pro parametrizaci dotazů je typicky použitá následující množina argumentů:

`nargs` – Celkový počet parametrů dotazu

`argtypes` – Pole o velikosti `nargs` obsahující identifikátory (`Oid`) jednotlivých typů parametrů v pořadí, jak jdou v dotazu za sebou.

`values` – Pole o velikosti `nargs` obsahující hodnoty jednotlivých argumentů ve formátu datového typu `Datum`.

`nulls` – Pole o velikosti `nargs` obsahující booleovské hodnoty určující, jestli je na dané pozici parametr prázdný (pokud ano, je patřičná hodnota v poli `values` ignorována).

Při použití parametrizace dotazu je tedy především nutné převést hodnoty parametrů do datového typu `Datum`. Navíc je dobré si uvědomit, že při opakovaném volání dotazu (ať už s existujícím plánem spuštění či bez něj), se typicky mění pouze hodnoty v poli `values`.

Použití kurzoru

SPI poskytuje rozhraní pro práci s kurzory, čímž umožňuje jednoduché zpracování např. celé tabulky po jednotlivých řádcích nebo jejich sadách. Využití kurzorů je výhodné například v případech, kdy očekáváme velmi objemný výsledek dotazu a nemůžeme si jej dovolit načíst do paměti. Rozhraní SPI pro práci s kurzory je složeno z následujících funkcí:

`SPI_prepare_cursor` – Obdoba `SPI_prepare`, ale při použití kurzoru. Kromě argumentů společných s `SPI_prepare`, obsahuje ještа nastavení kurzorových voleb.

`SPI_is_cursor_plan` – Booleovská funkce, která vrací `TRUE` pokud je možné na daném plánu spuštění otevřít kurzor.

`SPI_cursor_open` – Nastaví kurzor pomocí existujícího plánu spuštění a předaných parametrů.

`SPI_cursor_open_with_args` – Nastaví kurzor za pomoci dotazu a parametrů.

`SPI_cursor_open_with_paramlist` – Nastaví kurzor za pomoci jeho názvu a parametrů.

`SPI_cursor_find` – Vrátí existující kurzor podle zadaného jména. Pokud takový kurzor není nalezen, vrací `NULL`.

`SPI_cursor_fetch` – Načte z kurzoru několik řádků.

`SPI_cursor_move` – Posune kurzor.

`SPI_scroll_cursor_fetch` – Vybere z kurzoru několik řádků.

`SPI_scroll_cursor_move` – Posune kurzor.

`SPI_cursor_close` – Uzavře otevřený kurzor.

Při používání kurzorů je obzvláště potřeba uvolňovat paměť alokovanou pro každý dílčí výsledek pomocí `SPI_cursor_fetch` a obdobný funkcí. Pro každý dotaz je po zpracování nutné zavolat `SPI_freetuptable()`.

6.5 Implementace analýzy hlavních komponent

V předchozích sekcích jsme si popsali nástroje které byly použité pro implementaci PCA. V této sekci si dále popíšeme konkrétní řešení, tak jak bylo postupně naimplementováno pomocí uvedených nástrojů. Implementovaný nástroj pro analýzu hlavních komponent, který bude popisován v této sekci, je postaven na návrhu, jež byl uveden v kapitole 5.

6.5.1 Funkce

V této podsekci budou popsány funkce implementující jednotlivé operace analýzy hlavních komponent, které byly definovány v sekci 5.4.3. Funkce jsou přitom založeny na existenci databázové tabulky, ilustrované na obrázku 5.6.

Implementace operace `build_pca`

Ze všech tří operací definovaných v 5.4.3, je operace `build_pca` nejsložitější. Hlavními problémy bylo vyřešení paměťové náročnosti implementace a přesnost metody. Jelikož je zapotřebí analyzovat data z celé tabulky, byla zvolena metoda postupného vyčítání záznamu pomocí *kurzoru*.

Tak jako v případě ostatních operací je nejdříve potřeba pomocí SPI vyčíst záznam z tabulky `PCA`, který definuje parametry pro budoucí analýzu. Sestavení tohoto dotazu probíhá na úrovni řetězcových operací tak, aby bylo možné používat dotazy nad nejrůznějšími schematy a nebyl nástroj příliš složitý na ovládání (pouze se v definici uvedou řetězcové názvy sloupce a tabulky, které tvoří vstup pro analýzu hlavních komponent). Jakmile jsou jednotlivé vstupy z tabulky `PCA` načteny, přejde se k samotné analýze.

Celá analýza probíhá dvoukrokově (tabulka se projde dvakrát). V prvním běhu pouze sestavíme *vektor průměrů* tak, jak je definován ve vztahu 5.1.

V druhém průchodu teprve můžeme provést výpočet *kovarianční matice*, jelikož pro výpočet jednotlivých kovariancí mezi dimenzemi potřebujeme znát konečné průměrné hodnoty těchto dimenzí. Jelikož si nemůžeme dovolit načíst celou tabulku do paměti tak, abychom mohli provést výpočet kovarianční matice najednou (například použitím knihovny GSL), byl implementován iterační algoritmus pro postupnou aktualizaci kovarianční matice na základě přibývajících hodnot. Pro ilustraci je algoritmus uveden v příloze C.

Z důvodu omezeného prostoru jsem v kódu ponechal pouze prvky potřebné pro ilustraci algoritmu. Algoritmus využívá myšlenky, že pro každou další hodnotu můžeme přepočítat průměr (kovariance má charakter aritmetického průměru nějakého součinu odchylek hodnot od průměrů) tak, jakoby všechny předešlé hodnoty byly „průměrné“. Tím dostáváme jakýsi vážený průměr dosavadní hodnoty kovariance (z iterace $n-1$) a nové hodnoty součinu, kdy dosavadní kovariance má $n-1$ krát vyšší váhu než nová hodnota.

Z algoritmu je vidět, že je s výhodou využívána vlastnost symetričnosti kovarianční matice, takže se vždy přepočítává pouze její polovina.

Po spočtení vektoru průměrů a kovarianční matice se pomocí knihovny GSL provede analýza vlastních vektorů a hodnot. Přičemž se soustavně uvolňují všechny alokované prostředky tak, aby byly v držení co možná nejkratší dobu. Pak již jen následuje oříznutí matice do požadované dimenzionality a uložení vektoru energií.

Posledním krokem je transformace všech tří získaných metrik do podoby vhodné pro uložení do databáze a následně je zavolána databázová operace `UPDATE`, která uloží dané metriky do odpovídajícího záznamu tabulky `PCA`.

Implementace operace `apply_pca`

Tato operace stejně jako operace `build_pca` začíná vyčtením odpovídajícího záznamu z tabulky `PCA`. Operace `apply_pca` však navíc vyžaduje, aby již byla analýza hlavních komponent provedena (jinými slovy sloupce `matrix` a `mean_vector` musí být nastaveny).

Po načtení záznamu z tabulky `PCA` se od vstupního vektoru rysů nejdříve odečte (pomocí operací GSL knihovny) *vektor průměrů*. Následně se provede součin transformační matice takto upraveným vektorem (pomocí operací knihovny BLAS).

Posledním krokem této operace je převedení nově získaného vektoru do struktury vhodné pro vrácení.

Implementace operace `apply_pca_on_table`

Tato operace je ze všech nejjednodušší. Pouze se sestaví (na úrovni řetězcových operací) příkaz v následujícím tvaru uvedeném níže a pustí se (za pomocí SPI).

```
UPDATE table
SET column_dest = apply_pca(name, column_src)
```

V tomto dotazu jsou hodnoty `table`, `column_dest` a `column_src` získány ze záznamu tabuly `PCA` a hodnota `name` je získána z argumentu.

6.5.2 Použití

V předchozí podsekci jsme si popsaly postup implementace funkcí pro transformaci vektorů rysů do prostoru o nižší dimenzionalitě založené na analýze hlavních komponent. V této podsekci si ukážeme příklady využití.

Pro ilustraci použití implementovaného nástroje pro transformaci založenou na PCA uvažujme následující tabulku:

```
CREATE TABLE public.test_pca (
    id INTEGER PRIMARY KEY,
    vector INTEGER[] NOT NULL,
    transformed INTEGER[]
);
```

V této tabulce je `id` primárním klíčem, `vector` je 200 dimenzionálním vektorem rysů, jehož jednotlivé dimenze jsou spolu nějakou mírou korelované, a `transformed` je transformovaný vektor (na začátku prázdný). Předpokládejme, že máme v tabulce velké množství řádků. Pro začátek řekněme, že zvolíme cílovou dimenzionalitu vektoru 4. Vložíme proto řádek do tabulky `public.PCA`:

```
INSERT INTO "PCA" (
    "name", "table", column_src, column_dest, dimm_orig, dimm_dest
) VALUES (
    'test', 'public.test_pca', 'vector', 'transformed', 200, 4
);
```

Nyní máme definované všechny potřebné vstupy pro provedení analýzy:

```
SELECT build_pca('test');
```

Když máme dostupné výsledky analýzy, můžeme si například vyzkoušet transformaci několika málo vektorů

```
SELECT vector, apply_pca('test', vector) as transformed
FROM public.test_pca
LIMIT 40;
```

Případně můžeme aplikovat transformaci na celou tabulku:

```
SELECT apply_pca_on_table('test');
```

6.6 Indexace vektorů

V předchozí sekci jsme si popsali, jakým způsobem byl implementován nástroj pro transformaci vektorů na základě PCA. V této sekci si řekneme, jakým způsobem byly tyto vektory zaindexovány. Tato sekce je založená především na využitá indexu GiST a jeho varianty KNN GiST, které byly popsány v kapitole 5.5.

Po použití nástroje popsaného v minulé kapitole máme k dispozici transformovaný vektor rysů (jeden či několik), který potřebujeme zaindexovat tak, abychom jej mohli využít při vyhledávání na základě podobnosti. Jelikož samotná implementace GiST indexu je nad rámec této diplomové práce, bylo třeba najít datový typ, který má již operace rozhraní GiST indexu implementované a vhodně jej upravit tak, aby se hodil pro námi uchovávaná data.

Pro tyto účely poměrně dobře posloužil datový typ `cube`, který již všechny funkce a operátory potřebné pro index GiST má implementované.

6.6.1 Datový typ cube

Tento datový typ slouží především k reprezentaci n-dimenzionálních kostek a kvádrů v prostoru. Zvláštním případem takové n-dimenzionální kostky je kostka s nulovým objemem, která reprezentuje bod v daném prostoru. Právě tato reprezentace je vhodná pro reprezentaci n-dimenzionálního vektoru, jaké máme k dispozici, protože jsou definovány právě jako bod v prostoru.

Důležitou vlastností datového typu `cube` je pak maximální dimenzionalita, která je omezena na 100 dimenzí. Tato vlastnost činí použití analýzy hlavních komponent ke snížení dimenzionality původních vektorů ještě důležitější.

6.6.2 Rozšíření rozhraní typu cube

Jelikož datový typ cube je obsažen v rozšiřujícím modulu *contrib* a nikoli v základním sortimentu datových typů systému PostgreSQL, nemá definované všechny metody potřebné pro KNN GiST. Konkrétně mu chybí definice vzdálenostního operátoru `<->` a definice funkce pro určení strategie vzdálenosti pro KNN GiST. Obě tyto chybějící operace můžeme zastoupit již existující funkcí `cube_distance()` po vytvoření wrapperu, který přizpůsobí rozhraní funkce tak, aby vyhovovalo požadovanému rozhraní KNN GiST indexu.

Po zadefinování těchto operátorů je již možné pro datový typ `cube` použít podobnostní vyhledávání za použití KNN Search.

Kapitola 7

Experimenty

V předešlých kapitolách byla popsána teorii vyhledávání informací, návrh nástroje a celkového řešení pro zefektivnění vyhledávání na základě podobnosti v těchto datech a nástroje a problémy a konkrétní řešení při jeho implementaci. Tato kapitola obsahuje popis experimentů nad tímto řešením, výsledky z těchto experimentů a popis měřitelných vlastností daného řešení.

7.1 Sledované metriky

V této sekci si popíšeme základní metriky, které byly u experimentů sledovány a jaký mají význam. Základními sledovanými metrikami jsou:

1. Celkové zrychlení podobnostního vyhledávání
2. Vliv použité metody na přesnost a úplnost vyhledávání

V dalším textu budou diskutovány významy těchto metrik a experimenty, které byly na tyto metriky zaměřené.

7.1.1 Přesnost a úplnost

Jedná se o základní metriku používanou v oblasti vyhledávání informací, která se snaží popsat kvalitu použitého vyhledávacího systému z hlediska relevance výsledků. Definice pojmů *přesnost* (precision) a *úplnost* (recall) jsou následující:

$$P = \frac{|D_{relevant} \bigcap D_{retrieved}|}{|D_{retrieved}|} \tag{7.1}$$

$$R = \frac{|D_{relevant} \bigcap D_{retrieved}|}{|D_{relevant}|} \tag{7.2}$$

Přesnost P je poměrem říkajícím jaký je poměr nalezených relevantních dokumentů, vůči všem nalezeným dokumentům. Jinými slovy je přesnost ukazatelem schopnosti nevybírat nerelevantní dokumenty. Úplnost R je pak poměr nalezených relevantních dokumentů vůči všem relevantním dokumentům. Jinými slovy je úplnost ukazatelem schopnosti vybrat všechny relevantní dokumenty.

Ve vztahu k implementovanému řešení má smysl tuto metriku sledovat, jelikož byla použita ztrátová komprese původních vektorů rysů, na základě kterých se provádí podobnostní vyhledávání. Tato ztrátová komprese je způsobena transformací vektorů do prostoru o nižší dimenzionalitě, s použitím analýzy hlavních komponent (viz 5.4).

7.1.2 Vliv řešení na rychlost podobnostního vyhledávání

Jedním z hlavních cílů této práce je nalezení řešení pro zefektivnění podobnostního vyhledávání multimediálních dat na stávajícím systému. Z tohoto důvodu jsou experimenty zaměřené na vliv tohoto řešení na efektivitu provádění dotazů nedílnou součástí této práce.

Základní metrikou je zde doba provádění jednotlivých dotazů, přičemž sledovanými parametry (které mohou mít vliv na efektivitu provádění) jsou následující:

1. Počet vyhledávaných záznamů
2. Dimenzionalita prohledávaného prostoru
3. Zavedení indexu nad sloupcem
4. Velikost prohledávané množiny záznamů a výkon serveru

Abychom analyzovali vliv jednotlivých parametrů na efektivitu vyhledávání, je nutné provádět experimenty vždy s ohledem na právě jeden parametr (ostatní by měli být v rámci experimentu konstantní). Jedinou výjimkou pak bude porovnání efektivity vyhledávání za použití kompletního řešení pro srovnání s výchozím stavem, před zavedením implementovaného řešení.

7.2 Příprava na experimenty

V rámci přípravy na experimenty bylo sestaveno několik testovacích tabulek, které pak byly využívány při konkrétních experimetech. Jednalo se o jednoduché tabulky, složené z primárního (a současně cizího) převzatého z tabulky `intervals` a dvou sloupců obsahujících vektory v podobě pole a datového typu `cube`, na který byl aplikován index `GiST` (tyto tabulky byly nadále využívány i v dalších experimentech, ve kterých se uplatňoval tento index). Vznikly tak v podstatě tabulky tvořící externí index do tabulky `intervals`, kdy při nalezení záznamu v těchto tabulkách se skrze operaci `JOIN` můžeme odkázat na daný řádek v tabulce `intervals`.

Taková jednoduchá testovací tabulka pak byla vytvořena pro každý z vektorových sloupců `hist`, `color`, `grad`, `gabor` a `face` z tabulky `intervals`. V další fázi předzpracování byly navrženy a sestaveny definice analýz hlavních komponent v tabulce `PCA`, tak, aby pro každý uvedený sloupec existovalo 5 různých definic (výjma vektorů `face`, který má pouze 4 dimenze), které slouží ke snížení dimenzionality při zachování 50 %, 75 %, 85 %, 95 % a 98 % rozptylu hodnot dimenzí původních vektorů (tuto míru zachování rozptylu zjistíme z *vektoru energií*, který je popsán v 5.4.3). Pro každou transformaci danou jednotlivými definicemi analýzy hlavních komponent v tabulce `PCA` pak taktéž byla vytvořena stejná testovací tabulka.

Následně byla vytvořena tabulka `points`, která obsahovala několik vybraných vektorů z pro každý typ vektoru z původní datové sady v `intervals` a všechny jejich transformace. Tato tabulka pak poskytovala zdroj vektorů, vůči kterým se v experimentech provádí podobnostní dotazy (měří se vzdálenost vůči těmto vektorům).

7.3 Experimenty zaměřené na přesnost a úplnost

Metriky přesnost a úplnost byly definovány v sekci 7.1.1. V této sekci bude popsán průběh experimentů týkajících se této metriky a budou zde shrnuty jejich výsledků.

V rámci přípravy na tyto experimenty byly (krom tabulek popsaných v sekci 7.2) vytvořeny ještě tabulky `result_orig` a `result_trans`, které slouží k uchovávání výsledků jednotlivých dotazů (ve formě cizích klíčů na záznamy tabulky `intervals`).

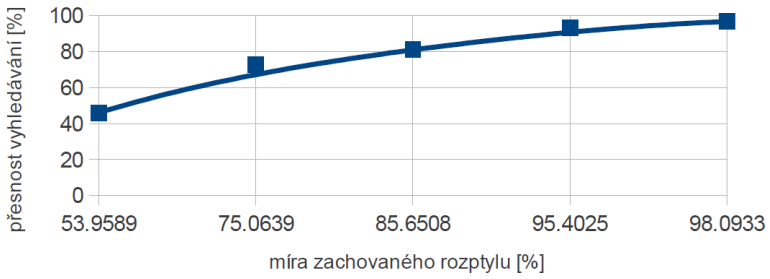
V rámci této sekce je důležité zdůraznit, že vzhledem k charakteru podobnostních výrazů je vypovídající schopnost těchto metrik poněkud zavádějící. Jelikož v takovýchto dotazech vždy specifikujeme počet požadovaných odpovědí, je třeba specifikovat tyto požadavky tak, aby nám metriky řekli o kvalitě systému co nejvíce.

V rámci sledování těchto metrik považujeme za relevantní všechny dokumenty (v našem případě intervaly nalezené na základě vektorů rysů), které jsou nalezeny pomocí stávajícího vyhledávání založeného na podobnosti. Jinými slovy tyto metriky vypovídají o míře nezávislosti výsledků vyhledávání na ztrátové kompresi způsobené využitím transformace založené na dané analýze hlavních komponent.

7.3.1 Experimenty zaměřené na přesnost

Tyto experimenty spočívaly v porovnání výsledků vyhledávání na základě původních vektorů a dříve definované vzdálenostní funkce s výsledky vyhledávání na základě odpovídajících transformovaných vektorů.

Experimenty zaměřené na přesnost probíhaly ve dvou variantách. První varianta využívala dotazy na stejné množství (konkrétně 1000) podobných vektorů z obou sad. Druhá varianta pak využívala dotaz na 10 krát nižší počet transformovaných vektorů, než vektorů původních.

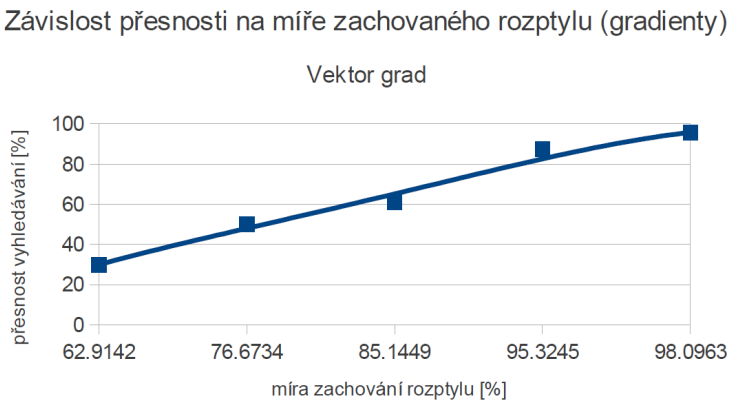


Obrázek 7.1: Závislost přesnosti na míře zachování rozptylu vektoru `color`

Obě varianty dotazů proběhly pro několik různých bodů (které byly nachystány v tabulce `points`), přičemž množství pro každý bod byly uloženy výsledky hledání do tabulek `result_orig` (pro vyhledávání nad původní množinou vektorů) a `result_trans` (pro vyhledávání nad množinou transformovaných vektorů). Následně byl proveden dotaz, který určil počet dokumentů, které byly nalezeny oběma dotazy (na základě shody cizích klíčů do tabulky `intervals`). Tato hodnota byla pak zprůměrována napříč všemi body a použita pro výpočet přesnosti tak, jak definován vztahem 7.1.

Tyto výpočty byly prováděny zvlášť pro všechny zvolené úrovně transformací nad daným typem vektoru. Výsledkem pak byly hodnoty přesnosti vyhledávání pro jednotlivé úrovně transformací, jejichž nejvyšší informační hodnota spočívá ve vztahu závislosti přesnosti na míře zachování rozptylu při použití dané transformace. Obrázky 7.1 a 7.2 ilustrují zjištěné závislosti pro vektory rysů `color` a `grad`.

Z obou uvededených charakteristik je vidět, že míra zachování rozptylu silně ovlivňuje



Obrázek 7.2: Závislost přesnosti na míře zachování rozptylu vektoru `grad`

přesnost vyhledávání. Pro udržení stávající kvality a přesnosti vyhledávání je tedy nutné zvolit takovou dimenzionalitu, aby byla míra zachování rozptylu co nejvyšší (ideálně 100 %).

Z charakteristik je dále vidět, že přesnost v nižších hodnotách roste, s vzrůstající mírou zachování rozptylu, prudčeji než míra zachování rozptylu. Je patrné, že pokud příliš snížíme dimenzionalitu výsledného vektorového prostoru, bude přesnost nižší, než míra zachování rozptylu.

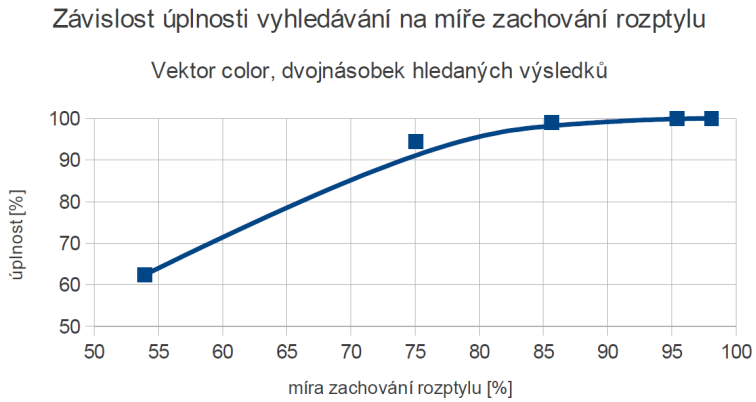
7.3.2 Experimenty zaměřené na úplnost

Experimenty zaměřené na úplnost byly obdobné experimentům zaměřeným na přesnost, přičemž využívaly stejných tabulek a dílčích výpočtů. Základním rozdílem v přístupu k těmto experimentům bylo vyhledávání většího množství transformovaných vektorů oproti vektorům původním.

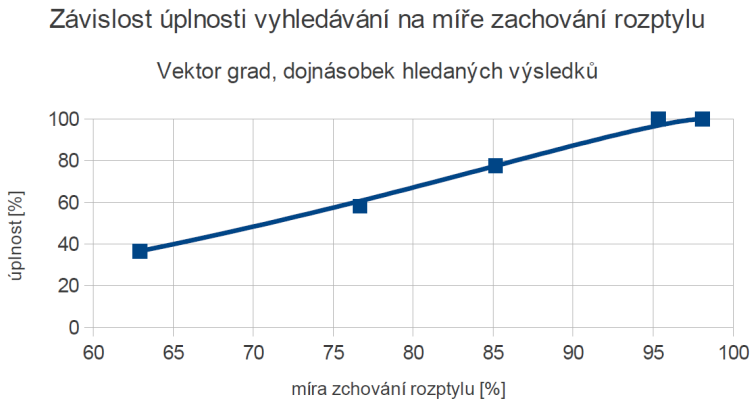
Experimenty taktéž probíhaly ve dvou variantách, kdy první se zaměřovala na úplnost při vyhledávání čtyřnásobně vyššího počtu transformovaných vektorů (oproti množství původních vektorů), zatímco druhá využívala dotazů na dvojnásobně vyšší počet transformovaných vektorů.

Z hodnot získaných obdobně, jako v případě experimentů zaměřených na přesnost, se pak podle vztahu 7.2 spočetla úplnost, opět pro jednotlivé úrovně transformací. Tím jsme získali hodnoty úplnosti vyhledávání ve vztahu k použitým úrovním transformací, jejichž nejvyšší informační hodnota (obdobně, jako je tomu u přesnosti) spočívá ve vztahu závislosti úplnosti na míře zachování rozptylu při použití dané transformace. Obrázky 7.3 a 7.4 ilustrují zjištěné závislosti pro vektory rysů `color` a `grad`. V tomto případě se jedná o charakteristiky pro variantu dotazů na dvojnásobný počet transformovaných vektorů.

Charakteristiky mají obdobný charakter jako charakteristiky získaně z experimentů zaměřených na přesnost. Opět je úplnost silně závislá na míře zachování rozptylu. Tato zkutečnost vyplývá z faktu, že počet vyhledaných dokumentů je v případě podobnostních dotazů zadaný (a tedy konstantní). Úplnost a přesnost jsou pak vzájemně lineárně závislé (uvedené charakteristiky pro úplnost a přesnost se vizuálně liší právě proto, že jsou spočtené z různých experimentů s jinými počty vyhledávaných dokumentů). Z tohoto důvodu nemá příliš velký význam uvádět charakteristiku vztahu úplnosti a přesnosti, jelikož má lineární průběh.



Obrázek 7.3: Závislost úplnosti na míře zachování rozptylu vektoru `color`



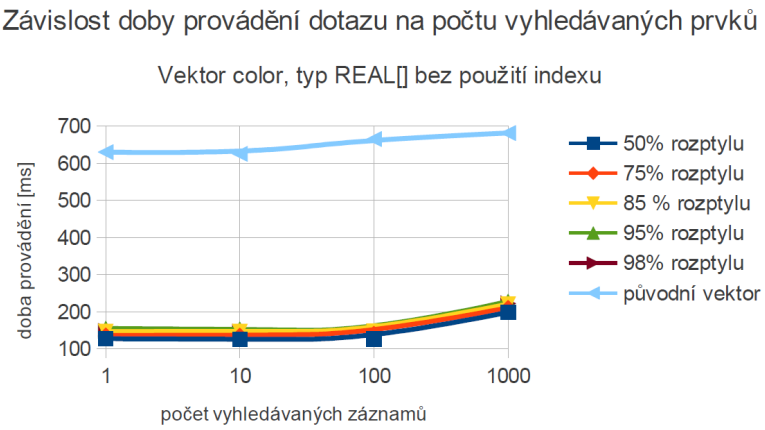
Obrázek 7.4: Závislost úplnosti na míře zachování rozptylu vektoru `grad`

7.4 Experimenty zaměřené na efektivitu provádění dotazů

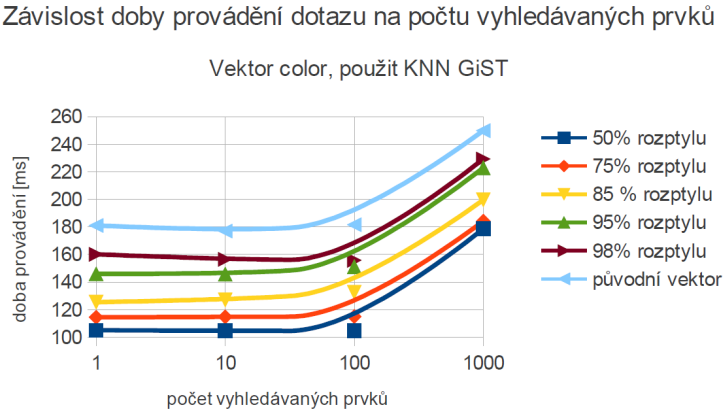
V rámci experimentů prováděných na testovací databázi byly použity testovací tabulky popsány v sekci 7.2. Při provádění experimentů byly postupně propočítávány doby přístupů při podobnostním vyhledávání vzhledem k náhodně zvoleným vektorům, jež byly uloženy do tabulky `points`. Každý typ vektoru byl testován zvlášť (včetně indexovaných `CUBE` variant, neindexovaných `REAL[]` a každé úrovně abstrakce). Přitom každý vektor byl podroben vyhledávání nejbližších 1, 10, 100 a 1000 vektorů. Obrázky 7.5 a 7.10 ilustrují příklad závislosti doby provádění (pro jednotlivé varianty transformovaných vektorů a původního vektoru) dotazu na počtu vyhledávaných záznamů.

Z těchto charakteristik je vidět, že dotazy prováděné nad zaindexovanými vektory jsou efektivnější než nad původními variantami. Také je vidět, že doba provádění do jisté míry závisí také na počtu dimenzí daného vektoru. Tuto závislost lépe ilustruje obrázek 7.7, který vyobrazuje závislost doby provádění dotazu nad jednotlivými transformacemi vektoru `gabor` na počtu dimenzí těchto transformací. Je patrné, že s narůstajícím počtem dimenzí vektoru pozvolna roste i doba provádění dotazů nad takovým vektorem.

Při současném pohledu na charakteristiku závislosti míry zachování rozptylu na počtu



Obrázek 7.5: Závislost doby provádění na počtu vyhledávaných záznamů

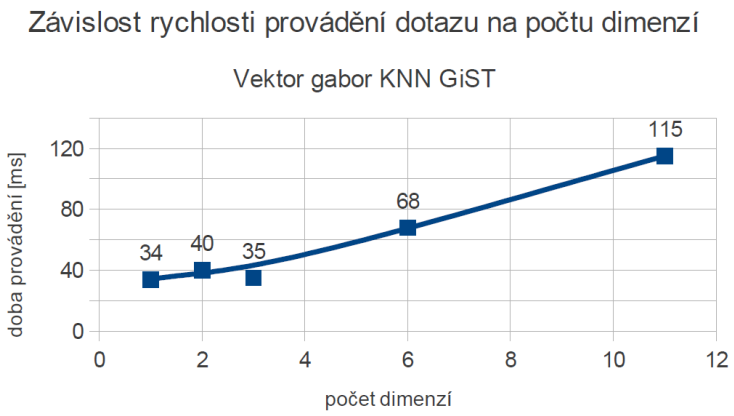


Obrázek 7.6: Závislost doby provádění na počtu vyhledávaných záznamů (za použití KNN GiST nad typem `CUBE`)

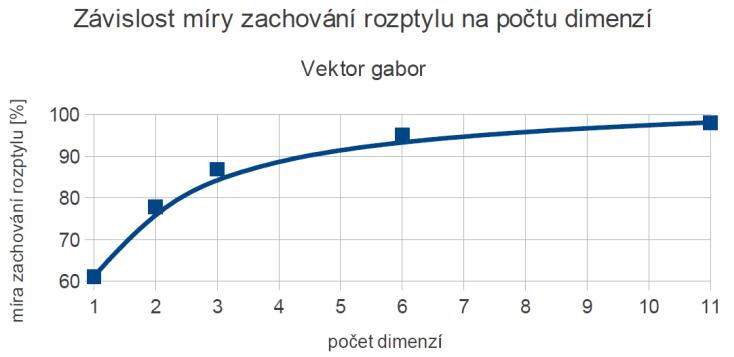
dimenzí, která je ilustrována na obrázku 7.8 navíc zjišťujeme, že při zisku pouhých 3 % z původního rozptylu musíme zvýšit počet dimenzí téměř dvojnásobně a současně nám prakticky stejnou mírou vzroste doba provádění dotazů nad takovým vektorem. V takové chvíli je nutné zvážit, jestli je takové zvýšení paměťové a časové náročnosti únosné vzhledem k získané míře rozptylu.

Obrázek 7.9 ilustruje vliv indexu na dobu provádění dotazu v případě zaindexování původního vektoru `grad` bez transformace. Z grafu je patrné, že použití indexu přispívá nejvyšší mírou ke snížení doby provádění dotazů (např. oproti snížení počtu dimenzí). Při použití indexu můžeme zkrátit dobu provádění až na pětinu.

Poslední charakteristika ilustruje případ, kdy pro zvýšení efektivity vyhledávání zvolíme optimální míru dimenzionality a následně zaindexujeme množinu transformovaných vektorů. V tomto případě volíme variantu, která zachovává přibližně 95 % rozptylu z původní sady vektorů a současně snižuje dimenzionalitu vektoru z původních 31 dimenzí na pouhých 6, což je méně než pětina. Po zaindexování transformovaných vektorů pak zís-



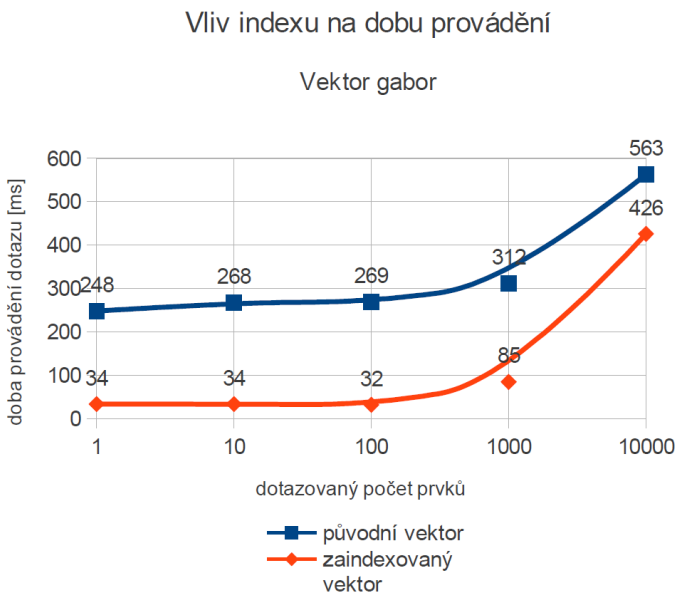
Obrázek 7.7: Závislost doby provádění na počtu dimenzí



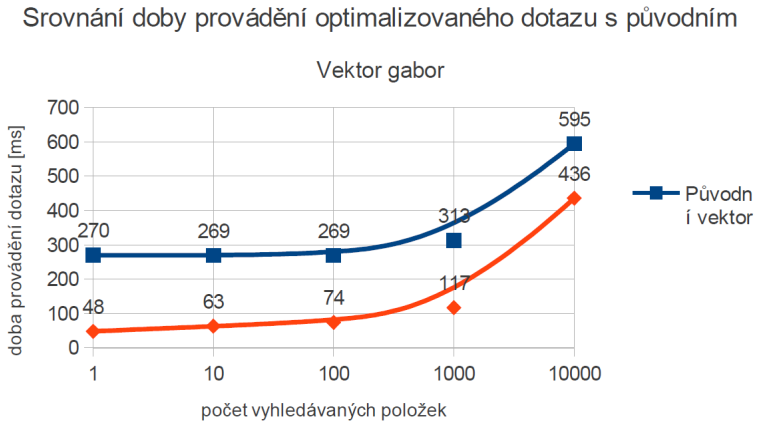
Obrázek 7.8: Závislost míry zachování rozptylu na počtu dimenzí

káváme sadu vyhledávacích vektorů o pětinových paměťových nárocích a s třetinovou až čtvrtinovou dobou provádění dotazů na omezené množství podobných vektorů (do řádu tisíců).

Z pohledu efektivity provádění dotazů i uchovávání dat, se jedná o velmi výraznou úsporu při zachování velké míry rozptylu.



Obrázek 7.9: Vlivu indexu na dobu provádění dotazu



Obrázek 7.10: Porovnání doby provádění dotazu nad optimálním vektorem oproti původnímu

Kapitola 8

Závěr

V současné době je stále větší zájem ne efektivním a kvalitním vyhledávání v multimediálních datech, především pak ve videu. Díky nestrukturovanosti těchto dat, se však jedná o velmi netriviální problém, který se snažíme řešit pomocí extrakce rysů a obencně metadat z dokumentu. Hlavním problémem je, že pokud mají být tyto vektory dostatečně popisující, maji typicky vysoký stupeň dimenzí a proto je práce s nimi nesnadná a časově náročná. Cílem této diplomové práce pak bylo shrnout dosavadní poznatky z oblasti vyhledávání informací a řešit tento problém pomocí implementace algoritmu pro snížení dimenizonality těchto vektorů a použití vhodného indexu.

V kapitolách 2 a 3 naleznete základní pojmy z oblastí vyhledávání informací a vyhledávání v databázích. Kapitola 3 navíc poskytuje úvod do základních principů indexů. Tato kapitola tvoří teoretický základ pro vyhledávání a problém indexace, které jsou diskutovány v dalším textu.

Práce se zaměřuje na multimediální data v kapitole 4, čímž doplňuje teoretické poznatky z kapitol 2 a 3. Kapitola nastiňuje problematiku extrakce rysů v multimediálních datech a jejich reprezentaci. Dále poskytuje přehled indexačních metod používaných pro vícedimenzionální data, jakými jsou například vektory vyextrahovaných rysů (případně různá prostorová data).

Pro snížení dimenziontality vektorů rysů byla použita *analýza hlavních komponent*(viz 5.4), která je diskutována v kapitole 5. Tato kapitola pak obsahuje popis stávajícího řešení systému pro vyhledávání vida vyvíjeného na Fakultě informačních technologií VUT v Brně a popisuje návrh rozšíření tohoto systému o struktury a operace využívající analýzy hlavních komponent ke snížení dimenzionality vektorů. Závěr kapitoly je pak věnován zvolenému indexu GiST a jeho rozšíření KNN GiST.

Implementaci rozšíření, použité nástroje a techniky jsou popsány v kapitole 6. Kapitola diskutuje nějvětší implementační potíže při implmentaci operací, jakými jsou například nutnost použití škálovatelných algoritmů nebo nezbytnost efektivní správy paměti. Zvláštní sekce je věnována principům a postupům při implementaci funkcí pro systém PostgreSQL za použití jazyka C (viz 6.4) tak, aby usnadnila čtenářům pochopení při budoucím vývoji takových funkcí.

V kapitole 7 jsou popsány jednotlivé vlastnosti daného řešení pomocí metrik. V kapitole se dozvíme, že použitím implementovaného řešení můžeme získat reprezentaci původních vektorů rysů s výraznou paměťovou úsporou, díky výrazné redukci dimenzionality těchto vektorů při zachování většiny míry rozptylu a při minimálním vlivu na přesnost a úplnost vyhledávání na základě těchto vektorů. Řešení je pak poměrně dobře použité pro všechny vektory, jejichž hodnoty jsou silně korelované.

Implementované řešení není použitelné jen pro vektory rysů v rámci projektu zaměřeného na vyhledávání ve videu, ale obecně pro zefektivnění vyhledávání v multidimenzionálních datech v prostředí databázového systému PostgreSQL.

V rámci rozšíření, by bylo zajímavé implementovat další varianty analýz hlavních komponent, rozšířit implementaci na větší portfolio datových typů a především implementace vlastní indexace vektorů rysů, která by byla nezávislá na ostatních modulech a přesně by reflektovala požadavky na efektivní podobnostní vyhledávání multidimenzionálních vektorů.

Literatura

[1] Bayeza-Yates, R.; Ribeiro-Neto, B.: *Modern information retrieval*. ACM Press, 1999, ISBN 0-201-39829-X.

[2] Caoying, Y.: B tree, B-tree & B+ tree. 2011 [cit.2012-01-03].
URL http://toyhouse.cc/profiles/blogs/b-tree-b-tree-amp-b-tree

[3] Chmelař, P.: Multimediální databáze [online]. 2006.

[4] Chmelař, P.: VTApi. 2010 [cit. 2012-05-04].
URL http://vidte.fit.vutbr.cz/vtapi.html

[5] Chmelař, P.; Beran, V.; Herout, A.; aj.: Brno University of Technology at TRECVid 2008. In *Proceedings of TRECVID 2008*, National Institute of Standards and Technology, 2008, str. 16.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8828

[6] Chmelař, P.; Beran, V.; Herout, A.; aj.: Brno University of Technology at TRECVid 2009. In *TRECVID 2009: Participant Notebook Papers and Slides*, National Institute of Standards and Technology, 2009, str. 11.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9148

[7] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. 1970.

[8] Frakes, W. B.; Baeza-Yates, R. A.: *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992, ISBN 0-13-463837-9.

[9] Jolliffe, I. T.: *Principal component analysis*. New York: Springer, 2002, ISBN 0-387-95442-2.

[10] Kornacker, M.: High-Performance Extensible Indexing. 1999.

[11] Krejčíř, T.: Vyhledávání v multimodálních databázích [online]. 2009 [cit. 2012-01-04], diplomová práce.
URL http://www.fit.vutbr.cz/study/DP/DP.php

[12] Martínez, J. M.: MPEG-7 Overview [online]. 2004 [cit. 2012-01-05].
URL http://mpeg.chiariglione.org/standards/mpeg-7/mpeg-7.htm

[13] Over, P.: Guidelines for the TRECVID 2009 Evaluation. 2009 [cit. 2012-12-04].
URL http://www-nlpir.nist.gov/projects/tv2009/tv2009.html

[14] Pedraza-Jimenez, M. V. R.: Natural Language Processing in Textual Information Retrieval and Related Topics [online]. 2007 [cit. 2012-01-03].
URL http://www.hipertext.net/english/pag1025.htm

[15] Salton, G.: Automatic Text Indexing Using Complex Identifiers. 1988.

[16] Samet, H.: foundation of multidimensional and metric data structures. 2006, iSBN 0-12-369-446-9.

[17] Seidel, R.; Aragon, C. R.: Randomized search trees. 1996.

[18] Sigaev, T.; Bartunov, O.: Introduction to GiST. 2002 [cit. 2012-05-04].
URL http://www.sai.msu.su/~megera/postgres/gist/doc/intro.shtml

[19] Smith, L. I.: A tutorial on Principal Components Analysis. 2002.

[20] Surkov, K.: SQL Server Optimization [online]. 2006 [cit. 2012-01-02].
URL
http://msdn.microsoft.com/en-us/library/aa964133%28v=sql.90%29.aspx

[21] Team, T. G.: GNU Scientific Library – Reference Manual. 2011 [cit. 2012-05-04].
URL http://www.gnu.org/software/gsl/manual/html_node/

[22] The PostgreSQL Global Development Group: PostgreSQL 9.1.3 Documentation. 2011.

[23] Volf, T.: The Application Interface for Video Databases. In *Proceedings of the 18th Conference STUDENT EEICT 2012*, vol. 3, Brno University of Technology, 2012, ISBN 978-80-214-4462-1, s. 415–419.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9931

[24] Zendulka, J.; Bartík, V.; Lukáš, R.; aj.: Získávání znalostí z databází. 2009.

[25] Zendulka, J.; Drahanský, M.; Zemčík, P.: Nástroje a metody zpracování videa a obrazu pro boj s terorismem. 2010 [cit. 2012-05-04].
URL http://www.fit.vutbr.cz/units/UIFS/grants/index.php.cs?file=%2Fproj%2F498&order=name&id=498

[26] Zendulka, J.; Rudolfová, I.: Databázové systémy. 2006.

[27] Čeloud, D.: Vyhledávání informací TRECVid Search [online]. 2010 [cit. 2012-01-04], diplomová práce.
URL http://www.fit.vutbr.cz/study/DP/DP.php

[28] Šabatka, P.: Vyhledávání informací [online]. 2010 [cit. 2012-01-04], diplomová práce.
URL http://www.fit.vutbr.cz/study/DP/DP.php

Příloha A

Obsah CD

pgPCA – adresář se zdrojovými kódy modulu pro PostgreSQL.

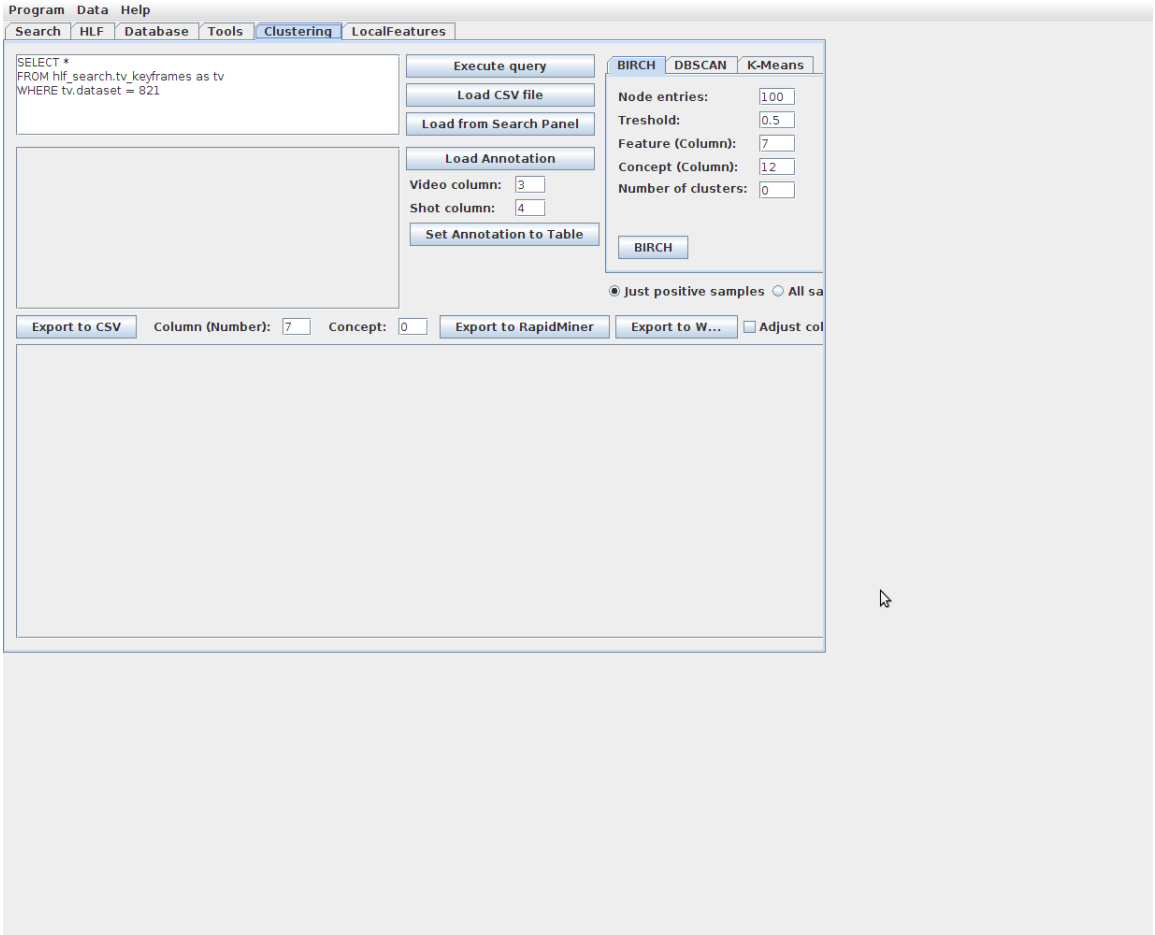
TRECVid Search – adresář se zdrojovými kódy aplikace TRECVid Search

skripty – adresář obsahující použité SQL scripty

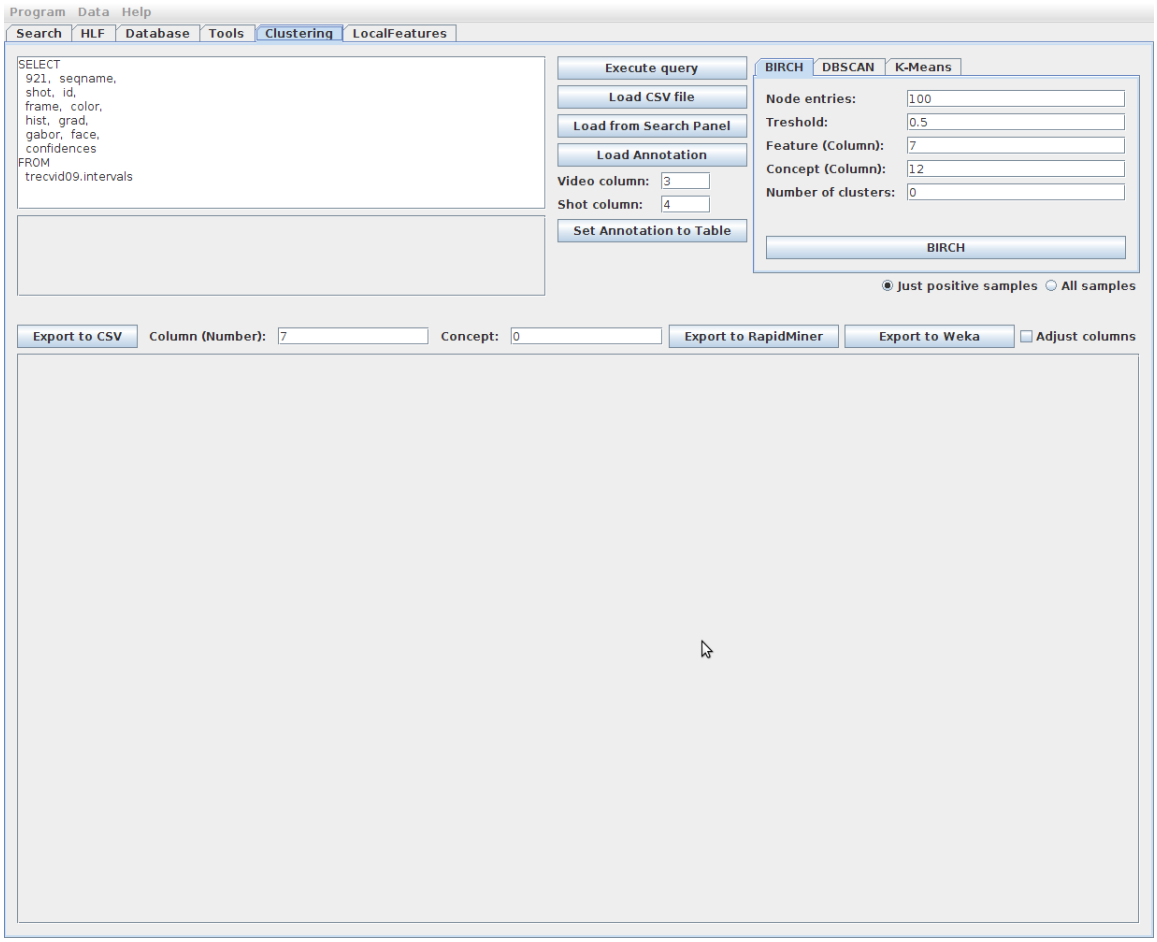
text_prace – adresář obsahující text práce ve formátu PDF

Příloha B

TRECVid Search – ilustrace úprav



Obrázek B.1: TRECVid Search – Příklad rozložení GUI před úpravou



Obrázek B.2: TRECVid Search – Příklad rozložení GUI po úpravě

Příloha C

Algoritmus iteračního výpočtu kovarianční matice

```
double covariance_iteration(double covariance, const float4 u1,
    const float4 u2, float4 value1, float4 value2, size_t n) {

    double new_product = (value1 - u1) * (value2 - u2);
    switch (n) {
        case 0: /* first time only the product is calculated */
            return new_product;
        case 1: /* second time new product is added */
            return covariance + new_product;
        default: /* apply the weighted mean principal */
            return (covariance * (n - 1) + new_product) / n;
    }
}

/* build covariance matrix from table */
double *get_covariance_matrix(PCA *pca, float4 *means) {
    ...
    get_row_from_db(pca, portal, &length, &row); /* get first row */
    ...
    do {
        size_t i, j;
        for (i = 0; i < dims; i++) {
            for (j = 0; j <= i; j++) {
                /* compute next iteration */
                double covariance = covariance_iteration(
                    gsl_matrix_get(&matrix_view.matrix, i, j),
                    means[i], means[j], row[i], row[j], index);
                /* fill covariance matrix symmetrically */
                gsl_matrix_set(&matrix_view.matrix, i, j, covariance);
                gsl_matrix_set(&matrix_view.matrix, j, i, covariance);
            }
        }
```

```
            SPI_freetuptable(SPI_tuptable); /* free memory */
            get_row_from_db(pca, portal, &length, &row); /* next row */
            index++;
        } while (row);
    }
    ...
}
```