VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

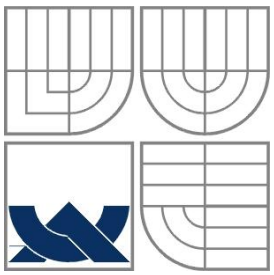
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

API DATOVÉHO ÚLOŽIŠTĚ PRO PRÁCI S VIDEEM A OBRÁZKY

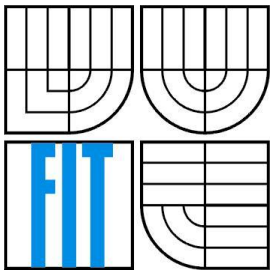
DIPLOMOVÁ PRÁCE

AUTOR PRÁCE Bc. VOJTĚCH FRÖML
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

API DATOVÉHO ÚLOŽIŠTĚ PRO PRÁCI S VIDEEM A OBRÁZKY

THE API OF A VIDEO AND IMAGE DATASTORE

MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH FRÖML

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. JAROSLAV ZENDULKA, CSc.

BRNO 2013

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací rozšíření databázového rozhraní VTApi, které je vyvíjeno v rámci projektu MV ČR „Metody a nástroje pro zpracování obrazu a videa pro boj s terorismem“ řešeného na FIT VUT. Toto rozhraní poskytuje podporu pro reprezentaci, správu a indexaci multimediálních dat a s nimi souvisejících popisných metadat využívaných analytickými aplikacemi založenými na počítačovém vidění. V současné době užívá jako datové úložiště SŘBD PostgreSQL. Práce popisuje základní techniky zpracování obrazových dat, koncept VTApi a navrhuje a implementuje jeho úpravy pro možnost použití různých datových úložišť. Jako příklad variantního úložiště integruje do VTApi podporu pro užití databáze SQLite.

Abstract

This master’s thesis proposes and implements an extension of the database interface VTApi which is being developed as a part of the MV ČR project “Tools and methods for video and image processing for terrorism prevention” at FIT VUT. This interface provides support for representation, management and indexation of multimedia data and related descriptive metadata used by analytic applications based on computer vision. It currently uses DBMS PostgreSQL as its default datastore. Paper describes basic techniques for processing image and video data, VTApi concept and proposes and implements its modifications for the purpose of supporting multiple types of datastores. As an example of an alternative datastore, support for usage of a SQLite database is integrated into VTApi.

Klíčová slova

zpracování videa, zpracování obrazu, počítačové vidění, multimédia, metadata, API, databáze

Keywords

video processing, image processing, computer vision, multimedia, metadata, API, database

Citace

Fröml Vojtěch: API datového úložiště pro práci s videem a obrázky, diplomová práce, Brno, FIT VUT v Brně, 2013

API datového úložiště pro práci s videem a obrázky

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Ing. Jaroslava Zendulky Csc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Vojtěch Fröml
20. května 2013

Poděkování

Rád bych poděkoval Doc. Ing. Jaroslavu Zendulkovi, Csc. za jeho doporučení, rady a čas, které mi věnoval.

Za spolupráci při vývoji projektu bych chtěl poděkovat databázovému týmu VTApi, zejména Ing. Petru Chmelařovi.

Obsah

1 Úvod

V dnešní době se metody zpracování multimediálních dat uplatňují v široké škále odvětví lidské činnosti. Rozvoj technologií ve sférách médií, průmyslu, služeb, zábavy a veřejného sektoru znamená nové výzvy pro vývoj a aplikace analytických algoritmů nad těmito daty. Pod pojmem „multimediální data“ si můžeme představit kombinaci různých forem obsahu (např. text, zvuk, obrázky, video, interaktivnost) do jedné platformy. Multimédia typicky obsahují enormní množství informací, které jsou často snadno rozpoznatelné a získatelné lidským úsilím. Přesná analýza rozsáhlého souboru takových dat by však byla spoléháním se na lidský faktor výrazně limitována. Moderní způsob zpracování velkého množství multimediálních informací představují domény informačních technologií zabývající se vyhledáváním na základě obsahu, získáváním znalostí (*knowledge discovery*) a dolováním (z) dat (*data mining*).

Termín „**dolování dat**“ ve své nejširší definici znamená „detekce něčeho nového“. Často je tedy mylně zaměňován s jakoukoliv formou zpracování dat, nicméně vhodně o něm lze uvažovat jako o kolekci (polo)automatizovaných úkonů sloužící k objevování nových vzorů (shluky, anomálie, asociace apod.) ve velkém množství dat. Tato metodologie hraje důležitou roli v komerční sféře, např. v analýzách trhů, efektivity marketingu, návyků zákazníků nebo správy lidských zdrojů. Další uplatnění nalézá ve vědě a výzkumu – např. v oblastech klinické medicíny, bioinformatiky, genetiky (výzkum lidského genomu), efektivity vzdělávání či analýzy rozvodu elektrické energie. Dolování geografických dat umožňuje v nasazení s geografickými informačními systémy (GIS) získávání nových zeměpisných vzorů, informací a modelů z mapových podkladů. Moderní aplikací je uplatnění této technologie ve sféře bezpečnosti pro účely boje s kriminalitou a prevence terorismu.

Jednu takovouto aplikaci představuje projekt „*Metody a nástroje pro zpracování obrazu a videa pro boj s terorismem*“ programu bezpečnostního výzkumu MV ČR, který je od roku 2010 řešen na Fakultě informačních technologií VUT v Brně [1]. Jeho cílem je výzkum metod a algoritmů zpracování obrazu a videa uplatňovaných při analýzách takových dat. Jedním z konkrétních výstupů projektu bude funkční vzorek systému, který bude zahrnovat sadu nástrojů na podporu zpracování a řešení analytických úloh. Součástí systému bude i úložiště a správa jak zdrojových dat, tak metadat z nich extrahovaných a využívaných při analýzách. Jedná se tedy o projekt spojující několik různých oborů informačních technologií – **databáze** pro reprezentaci a správu dat, **počítačové vidění** pro jejich analýzu a návrh celkové **hardwarové architektury** systému.

Tato diplomová práce je součástí vývoje a realizace části funkčního vzorku, které se týkají úložiště obrazových a video dat a metadat z nich extrahovaných. Hlavním produktem skupiny řešitelského týmu projektu, která tuto problematiku řeší, je databázového rozhraní **VTApi** (*VideoTeror Application Programming Interface*) s přidruženou metodologií [2]. Tento systém slouží k ulehčení a podpoře vývoje komplexních procesů získávání znalostí z obrazových dat (vyhledávání,

dolování). Databázové rozhraní zaobaluje přístup a správu samotných obrazových dat a metadat popisujících jejich obsah, zjednodušuje tedy jejich použití nástroji pro zpracování a analýzu.

V souladu se zadáním diplomové práce popisuje kapitola 2 základní techniky zpracování obrazových dat, zejména extrakci relevantních metadat a získávání znalostí pomocí technik klasifikace a shlukování.

Popis současného stavu projektu VTApi je k nalezení v kapitole 3. V aktuální verzi poskytuje knihovna implementována v jazyce C++ veškerou základní funkčnost dle jejího návrhu, nicméně pokročilejší požadavky na optimalizaci a univerzálnost systému jsou stále předmětem vývoje.

Hlavní praktický přínos této práce spočívá ve vypracování návrhu nové verze VTApi, která se vyznačuje větší flexibilitou z hlediska jejího nasazení. Především se tedy jedná o dosažení nezávislosti na užívaném typu úložiště pro reprezentaci obrazových a popisných dat. Využití knihovny VTApi v aktuální verzi znamená závislost na databázi PostgreSQL, což s sebou přináší často časově a technicky náročné požadavky instalace a správy databáze, nehledě na nutnost instalace klientských knihoven. Je tedy zapotřebí modifikovat celý systém, aby umožňoval implementaci rozšíření knihovny pro různá datová úložiště.

Řešení toho problému představuje zavedení určité úrovně abstrakce do konceptu knihovny. Návrhem nové verze systému VTApi se věnuje kapitola 4. Původní návrh byl zcela přepracován a novou abstraktní vrstvu reprezentuje pět tříd shlukujících funkcionalitu závislou na typu datového úložiště. Jedná se o správu připojení k databázi, konstrukci SQL příkazů a dotazů, získávání výsledků dotazů, správa datových typů a dynamické nahrávání klientských knihoven za běhu programu. Také byl navržen princip realizace těchto tříd pro různé konektory a jejich inicializace.

Kapitola 5 popisuje realizaci tohoto abstraktního rozhraní pro dva typy datových úložišť – SŘBD PostgreSQL a bez-serverovou souborovou databázi SQLite. Tato dvě úložiště byla zvolena z toho důvodu, že jsou ve svém konceptu a architektuře diametrálně odlišná. VTApi bylo původně nasazeno s databází PostgreSQL, jakožto poměrně robustního a výkonnostně optimalizovaného otevřeného systému bohatého na funkcionalitu a s možností jeho rozšíření pro podporu vícedimenzionálních datových typů. Přínos této práce spočívá v integraci podpory SQLite, jakožto light-weight přenositelného řešení výrazně ulehčujícího vývoj aplikací používajících VTApi.

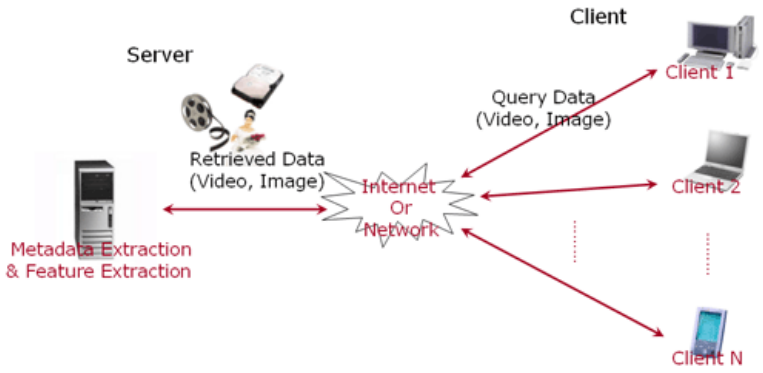
Testy výkonnosti databázových operací provedené nad implementací obou datových úložišť jsou popsány v kapitole 6. Na závěr je shrnut přínos této diplomové práce a navrhnuty možnosti dalšího pokračování vývoje VTApi.

2 Zpracování obrazových dat

V běžném životě jsme obklopeni nesmírným množstvím obrazových informací. Říká se, že obrázek vydá za 1000 slov a výzkumné studie prokazují, že vizuální vjemy představují více než 80% lidského učení [3]. Z toho plyne přirozená snaha o vývoj technologií, které budou obrazové informace snímat, ukládat, analyzovat jejich obsah a z těchto dat umožňovat získávání co největšího množství znalostí. Základním požadavkem pro takovýto systém je zavedení popisných metadat, která k obrazovým datům přidají sémantiku a umožní v nich vyhledávání.

Podle charakteru metadat lze uvažovat o dvou typech vyhledávání:

- **Vyhledávání dle popisu dat** (*description-based retrieval*):
  - Datům je přiřazena určitá množina předem definovaných popisných charakteristik, např. název souboru, klíčová slova, jméno autora či textový popis. Jinými slovy, jedná se o metadata, která je většinou nutno zadat manuálně pomocí lidského úsilí, což je typicky velmi složité, často nepřesné a neefektivní.
  - Množství znalostí získaných pomocí tohoto typu vyhledávání je relativně malé. Vyznačuje se však jednoduchostí, v praxi jde o provedení jednoduchého dotazu oproti uloženým metadatům.
- **Vyhledávání dle obsahu dat** (CBR - *content-based retrieval*):
  - Dotazování pomocí této metody je založeno na analýze samotných obrazových dat, nikoli jejich předdefinovaných metadat. V případě obrázků tedy mezi prohledávané informace patří např. histogram barev, rozmístění hran, textur, ploch, popis obličeje či jiné biometrické informace.
  - Metoda je typicky plně automatizovatelná nad velkým množstvím dat a podporuje algoritmy strojového učení. Aplikace užívající tuto metodu jsou tedy flexibilnější a snadněji škálovatelné. Její velkou výhodou je tedy poměrně slušná spolehlivost a kvalita získaných znalostí, nicméně za cenu vyšších výpočetních nároků.



Obr. 2.1: Schéma vyhledávání dle obsahu dat (převzato z [4])

Pro účely systémů zpracovávajících obrazová data typicky požadujeme objektivní a automatické algoritmy pro popis obsahu multimédií. Je zřejmé, že z tohoto pohledu je mnohem efektivnějším přístupem metoda vyhledávání dle obsahu dat [5].

Obr 2.1 znázorňuje jednoduché síťové schéma aplikace vyhledávání dle obsahu dat. V současné době se počet aplikací této metody stále zvyšuje. Nalézá využití v medicíně, např. pro detekci anomálních tkání v různých typech snímků, dále v meteorologii (předpověď počasí) či výzkumu vesmíru. Stálým předmětem výzkumu a vývoje je internetové vyhledávání obrázků na základě jejich obsahu či podobnosti. Podstatnou doménu aplikace představují biometrické systémy rozpoznávající lidské vnější charakteristiky pro účely autentizace či prevence a boje s kriminalitou.

2.1 Typy obrazových dat

Pro uchovávání a organizaci digitalizovaných dat je nezbytné standardizovat jejich souborový formát. Tato práce se zabývá pouze obrazovými daty, pomineme tedy reprezentaci jiných typů dat, např. textových a zvukových. Obrazová data můžeme dělit na statické obrázky a video (animaci), které k sekvenci obrázků přidává časovou dimenzi. Kontejnerem nazýváme soubor multimediálních dat a příslušných metadat v jednom formátu.

2.1.1 Obrázky

Formáty statických 2D obrázků se dělí dle typu reprezentace obrazové informace:

Rastrové (bitmapové)

Rastrové formáty popisují obrazová data jako mřížku jednotlivých barevných a jasových hodnot bodů (pixelů). Velikost souboru tedy narůstá s velikostí obrázku, čemuž se předchází použitím vhodného typu komprese:

- **Ztrátová** komprese může výrazně zmenšit velikost souboru za cenu nezachování jeho původní reprezentace. Typickým příkladem ztrátového formátu je JPEG.
- **Bezztrátová** komprese zmenšuje velikost souboru při zachování kompletnosti původní informace. Bezztrátovou kompresi podporuje např. formát PNG.

Příklady rastrových formátů jsou např. JPEG, PNG, GIF, TIFF, BMP, PCX, PSD, ICO.

Vektorové

Vektorově reprezentované obrázky mohou mít libovolnou velikost bez vlivu na velikost souboru. Jedná se o kolekci geometrických vektorů, nejčastěji přímek.
Formáty vektorové grafiky jsou např.: SVG, CVG, PS, EPS, CDR, AI, ODG.

2.1.2 Video a kontejnery

Video jsou audiovizuální data s temporálním charakterem, tedy časované. Formáty neomezené na obraz a zvuk se nazývají kontejnery. Soubor kontejneru tedy slouží k identifikaci a prokládání velmi různých typů dat. Pokročilejší formáty kontejnerů mohou podporovat jejich velké množství – více audio a video proudů, titulky, informace o kapitolách, anotace a synchronizační informace nutné pro jejich současné přehrání.

Video data obvykle obsahují mnoho redundantních informací, které neúměrně navyšují jejich požadavky z hlediska uložení. Proto je nezbytné použít pro reprezentaci video dat vhodnou kompresi jednotlivých snímků i pohybovou kompresi přechodů mezi nimi.

Mezi nejpoužívanější standardy komprese videa patří H.264 a MPEG-4. Rozšířenými formáty kontejnerů jsou např. 3GP, AVI, FLV, MKV, MPEG-TS, MP4, RM.

2.2 Multimediální databázové systémy

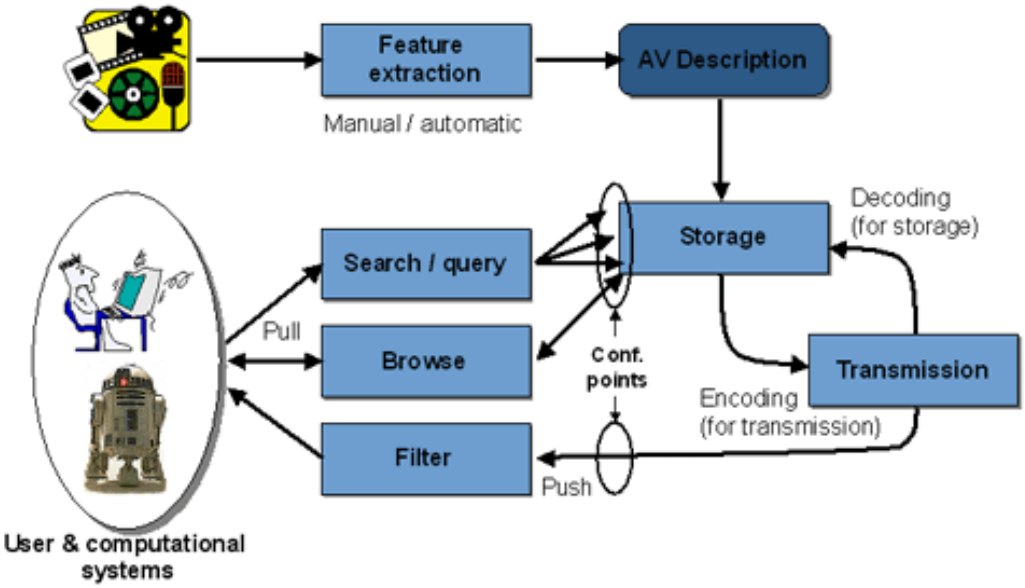
Obvyklým řešením ukládání velkého množství rozmanitých dat jsou samozřejmě databáze. Kromě zaručení konzistence, integrity, bezpečnosti a dostupnosti dat poskytují databázové systémy také funkce pro jednoduchou správu a dotazování významově hodnotných informací. Konstrukci multimediálních systémů řízení báze dat (MMDBMS) určují faktory vycházející z povahy multimediálních dat, tedy rozmanitost obsahu, typů a formátů, složitost reprezentace a subjektivní interpretace, rovněž jejich velikost a temporální povaha [6].

Kromě těchto vlastností klademe na MMDBMS stejné požadavky jako na databázové systémy klasické, tedy modelování, definice, a vytváření slovníku dat, manipulace s nimi, zajištění nezávislosti, bezpečnosti a integrity dat, zotavení po chybách, souběžný přístup, distribuované zpracování a zajištění co nejvyšší výkonnosti [7]. Kvalitní MMDBMS se vyznačuje následující funkcionalitou:

- správa různých typů vstupních, výstupních a úložných zařízení
- manipulace s množstvím různých datových a kompresních formátů
- integrace a koexistence datových modelů
- podpora různých platforem a operačních systémů pro koncové uživatele
- datové proudy audia a videa mají temporální charakter
- jednotná prezentace výsledků
- ochrana autorských práv
- nabídka množství jednoduchých dotazů vhodných pro různé typy dat, rychlé a přesné vyhledání požadované informace
- odlišení subjektivní (např. obličej) a exaktní (např. jméno) povahy dat

- vestavěné podobnostní vyhledávání (podle obsahu) korespondující s lidským vnímáním a pojetím podobnosti
- podobnostní vyhledávání ve smyslu časoprostorových dat
- existence modelu dat, tj. vztah mezí syntaxí (popis obsahu, low-level) a sémantikou (význam, high-level) audiovizuálních dat

Obrázek 2.2 ilustruje schéma procesů v multimediální databázi. První fází zpracování obrazových dat je **extrakce** jejich popisných **rysů/příznaků** (*Feature extraction*). Tyto tvoří tzv. **deskriptor** (*AV Description*), který se společně s obrazovými informacemi uloží do **databáze** (*Storage*). Uživatelé a klientské aplikace databázi **dotazují** (*Search / query*), výsledky mohou **filtrovat** (*Filter*) a následně **prohlížet** (*Browse*).



Obr. 2.2: Schéma procesů MMDBMS (převzato z [4])

2.3 Extrakce rysů z obrázku

Předchozí kapitoly se věnovaly technikám uložení obrazových dat. Dalším krokem k jejich zpracování podle obr. 2.2 je transformace vstupního obsahu média na **vektor rysů** (deskriptor, signaturu).

Za **rys** můžeme považovat jednotlivou charakteristickou vlastnost získanou z kolekce obrazových informací. Prvním krokem extrakce je tedy výběr vhodných vlastností, které budou reprezentovány rysy. Vhodně vybrané charakteristiky se budou vyznačovat tím, že co nejpřesněji reprezentují sémantiku obrazových informací s ohledem na jejich užití analytickými procesy, a tím výrazně redukují objem jejich vstupních dat. Rysy člověka jsou např. výška, pohlaví, barva kůže, očí a vlasů, dále tvar obličeje a úst, vzdálenost očí, délka končetina apod.

Rysy získávané ze statických obrázků lze kategorizovat do dvou hlavních skupin [8]:

- **Nízkoúrovňové rysy** představují informace o významných bodech, hranách a plochách v obrázku bez interpretace jejich významu např. vzhledem ke tvaru.
- **Vysokoúrovňové rysy** popisují obrazová data např. na základě tvaru obsažených objektů či detekce pohybu.

2.3.1 Nízkoúrovňové rysy

Tyto rysy můžeme definovat jako charakteristiky, které lze automaticky extrahovat z obrázku bez jakékoli informace o tvaru. Pro tyto rysy nejsou příliš vhodnými barevnými modely RGB a CMYK. Je rozumné použít model, který lépe odpovídá lidskému chápání barev, např. HSV, HLS nebo alespoň YCbCr [9]. K extrakci nízkoúrovňových (*low-level*) rysů se používají např. následující techniky:

Detekce rohů

Roh je bodem obrázku, který nese významnou informační hodnotu. Lze jej definovat jako průsečík dvou hran či jako bod mající ve svém okolí dvě dominantní a různé vektory hran. Za roh považujeme i samostatný bod, konec hrany nebo křivky. Obrázek 2.3 ukazuje průběžné výsledky algoritmu detekce rohů.

Obr. 2.3: Postupný proces detekce rohů (převzato z [10**]**)

Mnoho algoritmů detekce rohů je založeno na vylepšení těchto základních algoritmů:

- **Moravcův detektor** [11] [12]
  - Jeden z nejstarších algoritmů k detekci význačných bodů. Definuje význačný bod jako oblast s malou sebepodobností vůči okolí. Algoritmus provádí horizontální, vertikální a dvakrát diagonální posun čtvercového okna po obraze a počítá změnu jasu mezi posunutými okny.

- Je-li v aktivním okně **plocha**, změna posuvů je minimální. V případě **hrany** nastane výrazný rozdíl změn dvou kolmých posuvů. **Bod** je detekován při výrazných hodnotách všech posuvů.

- **Harrisův detektor** [12]
  - Vylepšení Moravcova algoritmu, odstraňuje jeho nedostatky (čtvercové binární okno, jeho posuv pouze o úhly v násobcích 45°). Gradienty obrazu se počítají pomocí derivací v bodě rohů v závislosti na uvažovaném směru pohybu.

Detekce hran

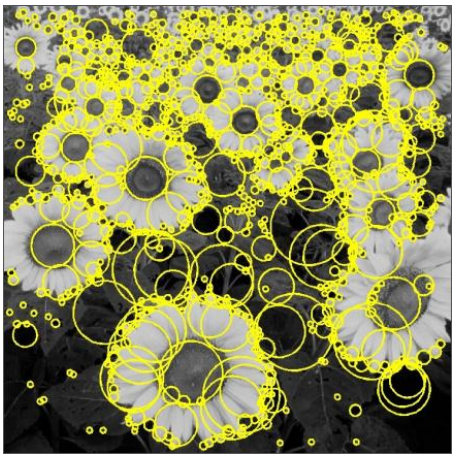
Za hranu v obrazových datech považujeme nespojitost v hloubce, orientaci povrchu, změny ve vlastnostech materiálů a variace v osvětlení scény. Obr. 2.4 zobrazuje jednoduchou detekci hran.

Obr. 2.4: Detekce hran (převzato z [19])

Mezi detektory hran řadíme např. algoritmy Cannyho, derivační a Sobelův operátor.

Detekce ploch

Plochou je oblast obrázku lišící se od svého okolí svými vlastnostmi, jako je jasnost či barva (viz obr. 2.5). Jejich detekci se věnují např. algoritmy FAST, DoG (*Difference of Gaussians),* DoH (*Determinant of Hessian*) či MSER (*Maximally stable extremal region*).



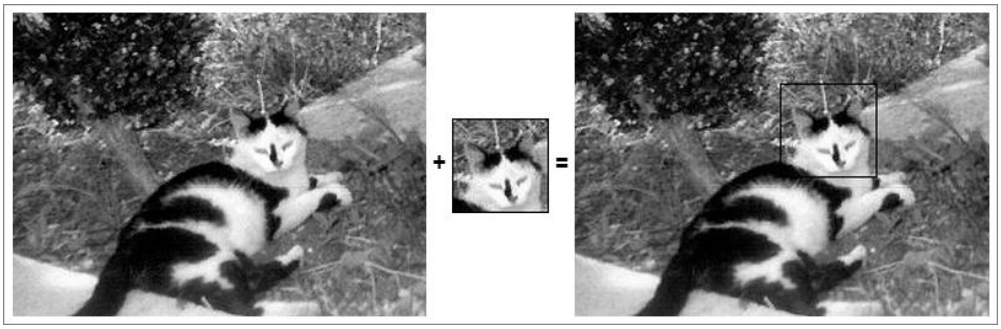
Obr. 2.5: Detekce ploch v černobílém obrázku (převzato z [20])

2.3.2 Rysy na základě statických tvarů

Vyhledávání tvarů v obrazových datech už lze považovat za extrakci rysů vysoké úrovně. Například pro automatické rozpoznávání tváří je zapotřebí zkombinovat výsledky mnoha dílčích algoritmů (rozpoznávání očí, nosu, úst, uší apod.) a opatřit je vhodnou sémantikou. Základními technikami získávání těchto rysů jsou [5]:

- **Prahování** (*thresholding*)
  - Nejjednodušší metoda segmentace obrazu, v kombinaci s technikou **odečítání** poskytuje binární výstup tvaru v obrazových datech (viz obr. 2.6).
- **Porovnávání šablon** (*template matching*)
  - Metoda používá vstupní šablonu a postupným porovnáváním s oblastmi obrázku zjistí případný výskyt podobné oblasti (viz obr. 2.7).
- **Houghova transformace** [13]
  - Metoda používaná k nalezení parametrů objektu v obraze. Tento objekt musí být popsatelný parametrickou rovnicí, typickým použitím tohoto algoritmu je tedy hledání přímek, křivek, kružnic, elips, ale i jednoduchých objektů.
  - Výhodou metody je její odolnost vůči šumu, nepravidelnostem a překrytí či jiné nedokonalosti zobrazení hledaného objektu.

Obr. 2.6: Fáze prahování (zleva - vstup, po odečtení pozadí, prahování) (převzato z [21])



Obr. 2.7: Extrakce rysů metodou porovnávání šablon (převzato z [22])

2.4 Detekce pohybu

V kontextu video dat či animace je pohybem myšlena změna umístění objektů s ohledem na následující snímky videa. Dále také jejich perspektivní transformace – přiblížení, oddálení, otočení, deformace či jiná změna jejich velikosti.

Perspektiva kamery se tedy obecně pohybuje v šesti stupních volnosti – pohyb v obou směrech po horizontální a svislé ose a zoom. Detekce pohybu kamery nemusí být jednoduchým úkonem, nicméně v počítačovém vidění se často jedná o žádanou funkci [6]. Pohyb kamery lze reprezentovat fundamentální maticí, kterou je možno odhadnout pomocí testování shody náhodných bodů v po sobě následujících snímcích.

Tuto matici se můžeme pokusit dopředu odhadnout např. pomocí algoritmu **RANSAC** (*Random Sample Consensus*) [14]. Tato metoda rozděluje data na tzv. *inliners* (náležící modelu) a *outliners* (nenáležící modelu). Odhad matice je iterativně počítán pouze z *inliners*, což umožňuje predikci pohybu kamery i ve velmi zašuměných datech.

Výpočet perspektivní transformace, která se běžně používá např. při přenosech fotbalových zápasů, je o trochu jednodušší, než výpočet a predikce fundamentální matice [15]. Nicméně není dostačující, chceme-li zjistit reálnou polohu objektu ve scéně, což je nezbytný požadavek např. bezpečnostních aplikací provádějící zpracování dat kamerových systémů.

Nejjednodušším typem pohybu je pohyb v rovině, kdy nepřipouštíme zásadní změny tvaru, vzhledu a velikosti objektu. Tělesa lze pak ve video datech najít pomocí technik pro odhad pohybu. V principu nejsnazším přístupem pro odhadování je odečtení hodnoty pixelů následujících snímků. Výsledný obraz se pak v bloku pixelů či v ohraničené oblasti prochází a snaží se najít její nový výskyt. Takto se zjistí jednoduchý vektor pohybu. Ten může velmi dobře fungovat i prediktivně za použití korekčního **Kalmanova filtru** [15]

Kompenzujeme-li pohyb kamery (odečtení pohybového vektoru) a ohraničíme dále se pohybující oblasti (seskupení bloků), jsme schopni **identifikovat** pohyblivé **objekty** a určit jejich tvar. To je velmi ceněná vlastnost při anotaci videa [6]. Identifikované vizuální těleso lze poté obalit konvexní obálkou, zapsat jeho pozici do databáze a doplněním časového razítka zaznamenat i jeho časoprostorové umístění. Máme-li k dispozici doplňující geografické informace, můžeme objekt označkovat v kontextu reálného prostředí a uchovávat v (časo)prostorové databázi.

2.5 Vyhledávání

Každý kvalitní MMDBMS musí samozřejmě podporovat vyhledání relevantních informací v obrazových datech. Použití samotného obsahu média je však pochopitelně neúměrně náročné, algoritmus vyhledávání se tedy zaměřuje výlučně na popisná metadata. Je důležité, aby tato metadata byla vhodně zvolena (vysoká míra popisnosti, snadná interpretace), extrahována a indexována.

Klientská část vyhledávání je neméně důležitá, zejména co se týče konstrukce dotazů a interpretace dat (např. v obrázcích 2.7 vlevo a vpravo znamená rozdíl v hodnotě obrazových bodů výrazně jinou informaci).

V úvodu kapitoly 2 byly popsány dva typy vyhledávání – na základě popisu a obsahu (CBR). Jednou z nejrozšířenějších aplikací CBR je vyhledávání na základě podobnosti.

Podobnostní vyhledávání

Porovnávání podobnosti věcí v reálném světě je nepochybně velice subjektivní pojem. V kontextu zpracování obrazových dat se však jedná o poměrně jednoduchý princip. Předpokládá, že pro vyhledávanou informaci se prostřednictvím dotazu dodá MMDBMS nějaký vzorek multimediálního obsahu. Z tohoto obsahu se pak extrahuje stejný vektor rysů jako při uložení do databáze.
Tento deskriptor je pak využit pro vyhledávání. Běžně se používají dvě podobnostní funkce:

- **Pomocí tříd podobnosti**
  - Máme-li množinu objektů, na které je podobnost reflexivní (prvky podobny samy sobě) a symetrickou (mezi prvky) relací, jedná se o třídu podobnosti. Žádný její objekt nesmí být však prvkem jiné množiny.
- **Pomocí vzdálenosti rysů**
  - Založena na měření vzdálenosti (např. Eukleidovské) mezi rysy reprezentovanými hodnotami ze spojitých číselných intervalů. Pro vektory v_1, v_2 o N počtu rysů vyjadřuje podobnost jako váženou vzdálenost d:

$$d(v_1, v_2) = \sqrt{\sum_{i=1}^{N} \alpha_i \left(v_2[i] - v_1[i]\right)^2}$$

Rovnice 2.1: Podobnostní funkce vyhledávání pomocí Eukleidovské vzdálenosti rysů

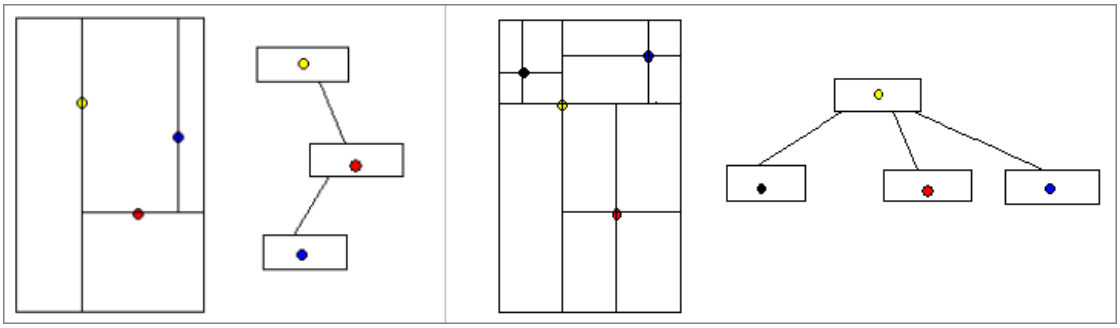
Indexace

Index je prvkem databáze zajišťujícím rychlý přístup k datům a jejich vyhledání. V případě obrazových dat lze indexovat multimediální obsah i popisná metadata. Použití existujícího indexu určuje na základě databázového dotazu plánovač. Cílem je nezpracovávat data, která zcela jistě nejsou třeba či momentálně nejsou dostupná.

Typickým požadavkem na MMDBMS je indexace metadat, tedy především anotovaných objektů. Tato data se běžně vyznačují vícerozměrností, pro jejich indexaci se tedy nejlépe hodí struktury založené na dělení prostoru:

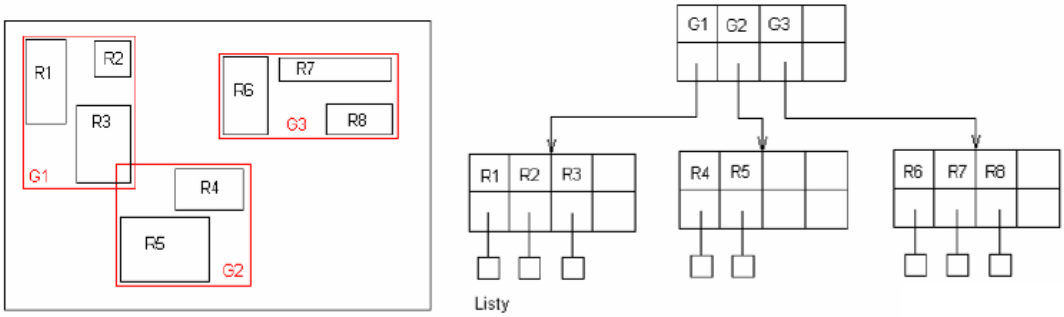
- **K-D stromy**
  - Slouží k ukládání bodových k-dimenzionálních dat. Dělí prostor střídavě v jedné dimenzi podle vkládaných bodů.
  - Výhodou této struktury je zejména snadná implementace.

- **Quad stromy**
  - Vycházejí z K-D stromů, ale vkládané body rozdělují prostor ve všech dimenzích.
  - Umožňují rychlejší vyhledávání, ale zesložiťují např. operaci rušení.



Obr. 2.8: Dělení prostoru pomocí K-D stromů (vlevo) a Quad stromů (vpravo) (převzato z [23])

- **R-stromy**
  - Ukládají obdélníkové struktury. Listové uzly reprezentují reálné objekty (příp. jejich minimální ohraničující obdélník), ostatní sdružují menší obdélníky do větších. Uzly stromů (kromě kořenového) musí být alespoň z poloviny zaplněné



Obr. 2.9: Dělení prostoru pomocí R-stromů (převzato z [23])

Nad velkými objemy dat dosahují nejlepších výsledků R-stromy či jejich modifikace R+ stromy (vylučují překrývání obdélníků) a efektivnější R* stromy, které se snaží překrytí minimalizovat. Jsou však nevhodné pro vysoký počet dimenzí.

2.6 Získávání znalostí

Cílem zpracování obrazových dat může být nabytí nových znalostí o jejich obsahu či vnitřních souvislostech. Toto získávání znalostí zpravidla představuje iterativní a interaktivní proces, který je průběžně vylepšován se snahou dosáhnout informačně hodnotnějších dat. Kompletní proces se skládá z několika fází – čištění, integrace a transformace dat (obsahuje extrakci popsanou výše), dále dolování dat, jejich vyhodnocení a prezentace.

Proces **dolování dat** slouží k objevování nových vzorů (shluky, anomálie, asociace apod.) ve velkém množství dat. Z pohledu obrazových dat se jedná především o nalezení vztahů mezi jejich

syntaxí (obsahem – změť barev) a sémantikou (významem – dav lidí). Aplikace lze z tohoto pohledů rozdělit na dva typy:

- **Obecné** aplikace nemají žádnou předchozí znalost o zpracovávaných datech. Jejich schopnost odvodit sémantiku je omezená.
- **Speciální** aplikace užívají zdrojová data s jasným účelem a ve známém kontextu. Mohou tedy lépe vytvářet svou znalostní bázi a poskytovat cílenější výsledky.

2.6.1 Klasifikace

Za klasifikaci považujeme proces, který hledá modely popisující a odlišující známé třídy dat tak, aby do těchto tříd bylo možno zařadit i neznámé objekty. Vyžaduje tedy jako vstupní znalostní bázi množinu klasifikačních tříd.

Bayesovská klasifikace

Jednoduchým příkladem klasifikačního procesu je naivní Bayesovská klasifikace. Předpokládejme následující množinu znalostí o výskytu objektů v obrázku:

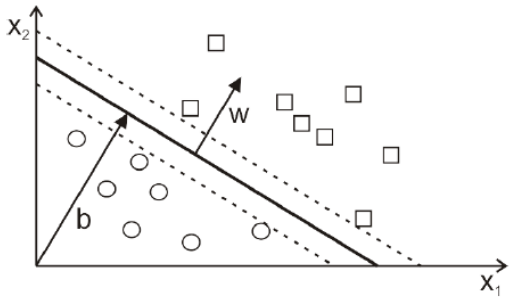
Tvar (x_1)	**Velikost (x_2)**	**Třída (y)**	**P(y)**	**P (y|x_1)**	**P (y|x_2)**
obdélník	*střední*	*pes*	*0,33*	*0,9*	*0,9*
kruh	*malá*	*pneumatika*	*0,33*	*0,45*	*0,7*
kruh	*velmi malá*	*hodiny*	*0,33*	*0,45*	*0,7*
?	*?*	*?*	*0,01*	*0,1*	*0,1*

Objekty psa, pneumatiky a hodin jsou pozorovány se stejným počtem výskytu. Hodnota v předposledním sloupci označuje pravděpodobnost, že objekt tvaru x_1 bude třídy y. Je-li tedy pozorován obdélník, dle této klasifikace se z 90% bude jednat o psa.

Dle této znalostní báze klasifikujeme nové neznámé objekty. Např. pro objekt *středně velkého kruhu* vynásobíme u všech tříd pravděpodobnosti, že nabývají těchto vlastností s celkovou pravděpodobností jejich výskytu (0,33). Maximalizací výsledků pro jednotlivé třídy získáme odhad třídy pro neznámý objekt, v našem případě *pes*.

SVM

Pro klasifikaci a regresi existuje množství modifikací, které neobsahují naivní předpoklad. Obecným algoritmem jejich aplikací jsou kernelové metody např. SVM (*Support Vector Machine*). Slouží ke generalizaci dat a lineární separaci jednotlivých tříd za pomocí separační hyper-roviny. Není-li možné data separovat, musí se použít transformační funkce, tzv. kernel.



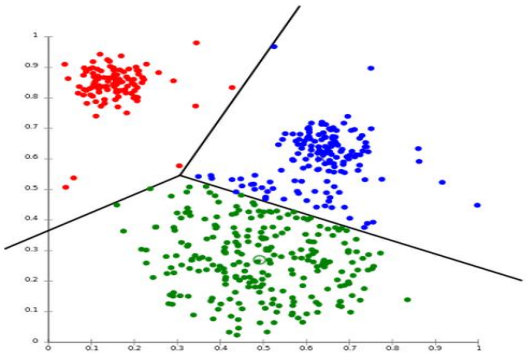
Obr. 2.11: Lineární separace tříd pomocí SVM

Ke klasifikaci obrazových dat lze dále použít klasické algoritmy učení – rozhodovací stromy, hrubé množiny, fuzzy logika či Markovovy skryté modely.

2.6.2 Shlukování a analýza

Shlukování (*clustering*) je technika seskupující objekty dle míry jejich podobnosti a zařazující neznámé objekty do nově vytvářených tříd zařazující neznámé objekty. Existuje několik typů shlukovacích metod dle jejich založení na hierarchii, rozložení, hustotě či centroidní modely.

Metoda K-Means [17] je vhodnou technikou pro aplikace zpracování obrazových dat. Požaduje manuální určení počtu shluků před spuštěním učení. Neefektivní může být metoda volby počátečních středů shluků (náhodně), vhodnější může být volba na základě gradientu [6].



Obr. 2.12: Ukázka shlukování metodou K-Means

Asociační analýza nachází asociační vztahy mezi analyzovanými daty. Výstupní znalostí je určení, které vlastnosti se vyskytují v dané množině dat společně a s jakou pravděpodobností. Užití v dolování z obrazových dat může mít následující užití:

- Asociace mezi umístěním objektů a popisem. Vyskytuje-li se např. v horní části obrázku modrá barva, požadovaným asociační pravidlo poskytne informaci o pravděpodobnosti, že se jedná o oblohu.
- Asociace bez určení umístění – např. „Vyskytuje-li se v obrázku objekt pes, bude s v něm vyskytovat i člověk“.

- Asociace vzájemných poloh objektů – pravidla určující vzájemný prostorový vztah objektů, např. „Cyklista se vždy bude nacházet mezi okraji silnice“.

Další skupiny technik pro získávání znalostí jsou např. **evoluční analýza** (odhalování trendů chování objektů) či **analýza anomálií** (*outliers*).

OpenCV

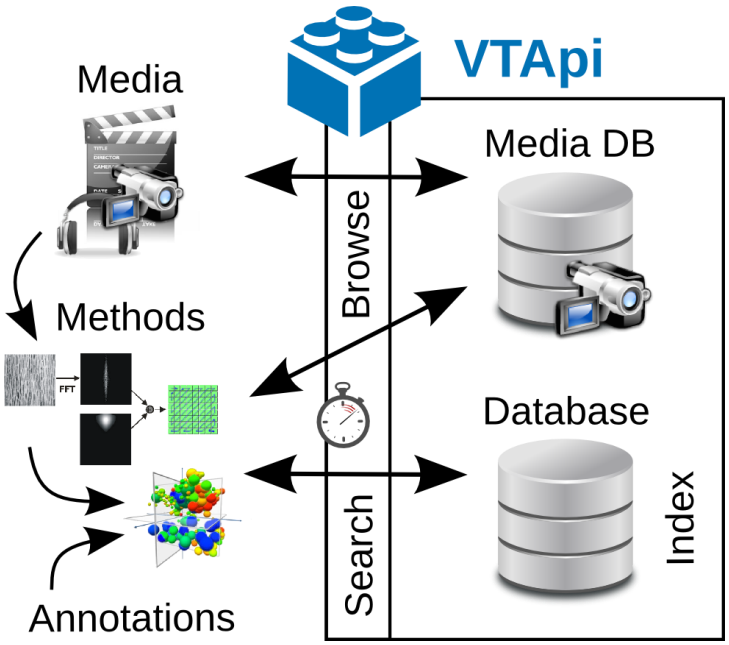
Nezbytným požadavkem na analytickou aplikaci založenou na počítačovém vidění je užití efektivních a vysoce optimalizovaných algoritmů zpracování obrazových dat. Za účelem poskytnutí široké škály funkcí implementujících tyto algoritmy byla vyvinuta open-source knihovna **OpenCV** (*Open Computer Vision*) [24]. Tato knihovna nalézá uplatnění např. v oblastech detekce pohybu, sledování trajektorií, identifikace objektů, rozpoznávání gest, tváří a mnoha dalších. Poskytuje také podporu pro statistické strojové učení – např. pro rozhodovací stromy, EM algoritmus, naivní Bayesovský klasifikátor, SVM a neuronové sítě.

3 Projekt VTApi

VTApi (*VideoTeror Application Programming Interface*) je databázové rozhraní a přidružená metodologie vyvíjené v rámci projektu „*Nástroje pro zpracování videa a obrazu pro boj s terorismem*“ na Fakultě informačních technologií VUT v Brně [2]. Toto rozhraní poskytuje efektivní podporu pro správu obrazových dat a souvisejících metadat užívaných analytickými aplikacemi založenými na počítačovém vidění. Hlavním cílem je tedy návrh úložiště těchto dat a implementace intuitivních metod rozhraní pro jejich vytváření, získávání a efektivní správu.

Takto navržené rozhraní umožňuje programátorům aplikací analyzujících obrazová data soustředit se pouze na vývoj samotných analytických metod. Metodologie VTApi umožňuje registraci a správu těchto metod sloužící k jejich různému jednotlivému použití či řetězení v komplexním analytickém procesu. VTApi je tedy primárně určeno pro zjednodušení a podporu vývoje komplexních analýz, vyhledávání či dolování z obrazových dat.

Databázové rozhraní VTApi je implementováno jako knihovna v programovacím jazyce C++ (dostupná také v jazyce Python). Jako hlavní datové úložiště slouží databáze PostgreSQL s rozšířeními PostGIS a GEOS pro podporu geometrických datových typů. Knihovna také obsahuje metody pro manipulaci s obrázky a videem za použití knihovny OpenCV.



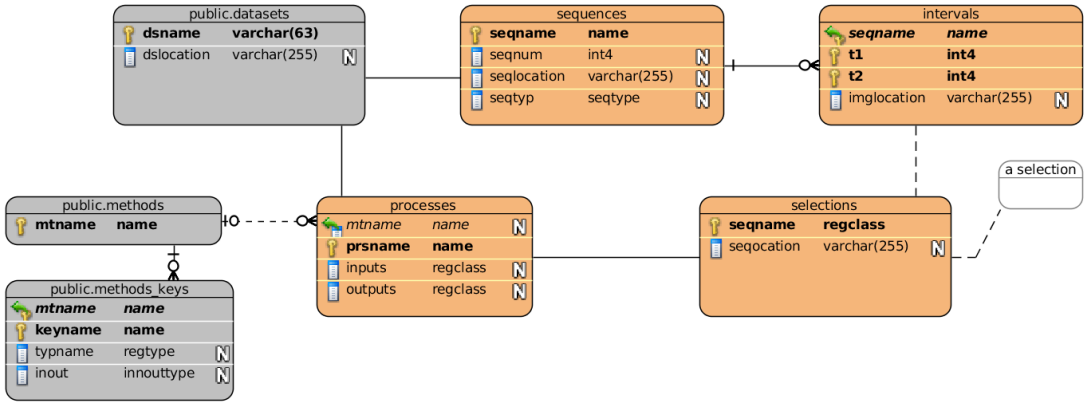
Obr. 3.1: Ilustrace funkcionality VTApi

Typické použití knihovny VTApi pro účely zpracování obrazových dat vypadá následovně:

- Nahrání informací o obrazových datech do úložiště (databáze). K tomuto účelu lze použít automatizovaný skript, který uloží např. celý adresář s videi.
- Alokace místa pro výstupy algoritmů (tabulky, sloupce v databázi).

- Registrace analytických algoritmů, tedy především formát jejich vstupů a výstupů. Provádí se manuálně.
- Implementace samotného analytického programu – procházení obrázků/videí dle uložených metadat a dosavadních výsledků analýzy; spouštění registrovaných algoritmů, vkládajících další data.
- Dotazování na výsledky procesů.

Obrázek 3.2 zobrazuje zjednodušený logický model databázového úložiště, poskytující zajištění této základní funkčnosti. Šedé objekty mají **globální charakter**, oranžové jsou **specifické pro** každou **sadu dat** (*dataset* – viz níže).



Obr. 3.2: Logický datový model VTApi

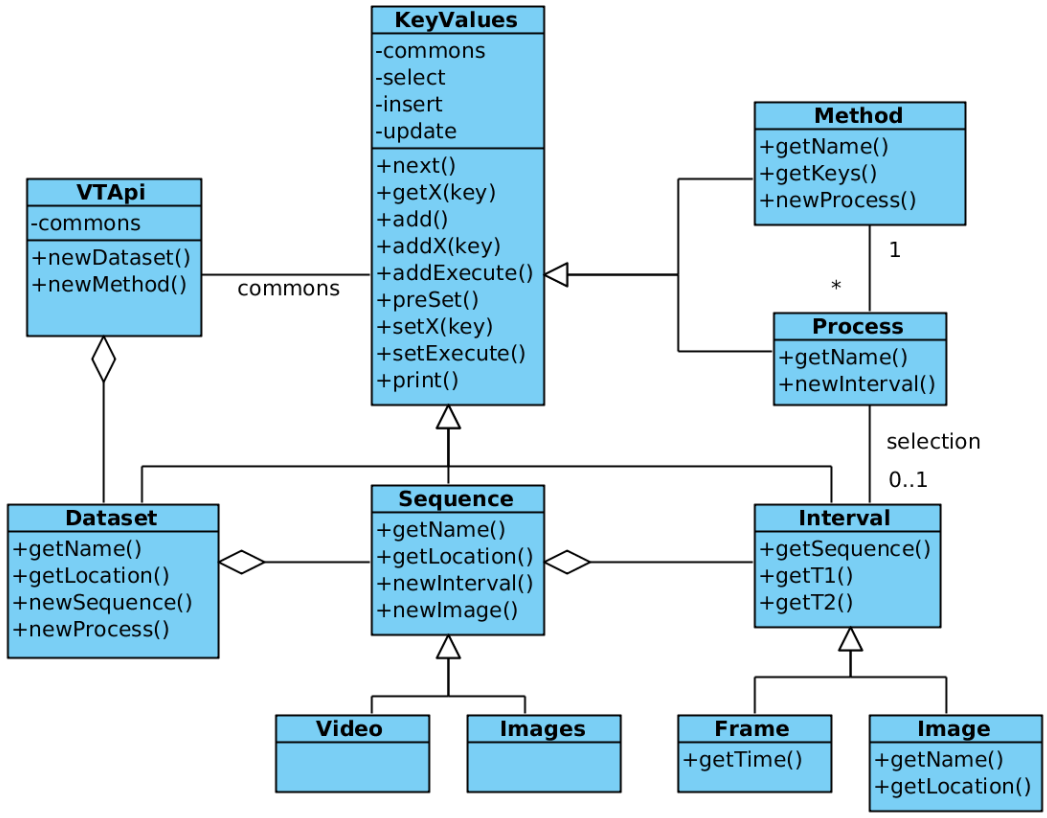
Bez podrobnější specifikace terminologie lze objekty modelu lze rozdělit do následujících kategorií:

- **Reprezentace obrazových dat**
  - **public.datasets**: obsahuje informace o různých sadách dat (*dataset*). Tyto sady jsou typicky určeny pro různé projekty. Každý dataset odpovídá jednomu schématu databáze (schéma *public* je základní).
  - **sequences**: informace o videích či složkách obrázků (*sequence*)
  - **intervals**: informace o jednotlivých obrázcích či částech videa (*interval*)
- **Reprezentace algoritmů**
  - **public.methods**: registrované algoritmy (*method*). Společné pro všechny sady dat. Strukturu vstupních a výstupních dat metody určuje kolekce objektů **public.methodkeys**.
  - **processes**: spuštěné instance metod nad jednou sadou dat (*process*)
- **Optimalizace**
  - **selections**: omezení výběru procesem zpracovávaných obrazových dat (*selection*) pro účely optimalizace

V následujících podkapitolách budou popsány mechanismy užití a implementace těchto objektů v samotném API.

3.1 Diagram tříd

Následující obrázek ilustruje s jistou mírou zjednodušení třídy API podstatné z hlediska koncového uživatele.



Obr. 3.3: Diagram tříd VTApi

Z diagramu lze jednoduše odvodit, že:

- třídy **Dataset**, **Sequence**, **Interval**, **Method** a **Process** přímo odpovídají objektům databázového modelu popsanému výše. Svými metodami zaobalují přístup k těmto objektům a prováděním databázových dotazů umožňují dotazování, vkládání, mazání a jejich aktualizaci.
- třídy **Video** (video), **Images** (sada obrázků), **Frame** (snímek videa) a **Image** (obrázek) jsou intuitivními specifikacemi tříd *Sequence* a *Interval* pro jejich užití ve vhodném kontextu.

Všechny tyto třídy dědí převážnou většinu své funkčnosti z obecné třídy **KeyValues** (klíč-hodnota sloupce v tabulce), která v sobě obsahuje funkce pro manipulaci s databázovými objekty prostřednictvím vnořených tříd **select**, **insert** a **update**.
Specifikaci a funkčnosti těchto tříd se věnují kapitoly 3.2 a 3.3.

Třída **VTApi** slouží jako vstupní třída API. Její hlavní funkcí je umožnění konfigurace připojení k databázi a dalších vstupních parametrů prostřednictvím vnořené třídy **commons**. Konfiguraci se věnuje kapitola 3.5.

3.2 Přístup k obrazovým datům

Aplikace na vyhledávání či získávání znalostí z obrazových dat typicky požaduje úložiště pro dva typy dat:

- *Vstupní multimediální data*: zpracovávané **obrázky** (či jejich sady) a **videa.**
    - Ukádány uživatelem do **diskových souborů**, logicky uspořádaným do adresářů
- *Vstupní / výstupní popisná data*: související metadata např. ve formě **anotací** či **klasifikací**.
    - Ukládány v datovém úložišti (**databáze**)

Koncept VTApi zaobaluje přístup a manipulaci s oběma těmito typy dat. Poskytuje metody pro reprezentaci uložených obrazových dat (místo uložení, logická hierarchie, popis multimediálního typu a další specifické informace). Tato funkčnost je reprezentována třídami **Dataset**, **Sequence** a **Interval**, které dědí společnou funkčnost ze třídy **KeyValues**.

Následující sekce podrobněji popisují, jak jsou tyto třídy hierarchicky uspořádány na úrovních zdrojových dat a databázových objektů včetně jednoduchých příkladů přístupu v C++. Praktičtější případy použití obsahuje kapitola 3.5.

3.2.1 Dataset

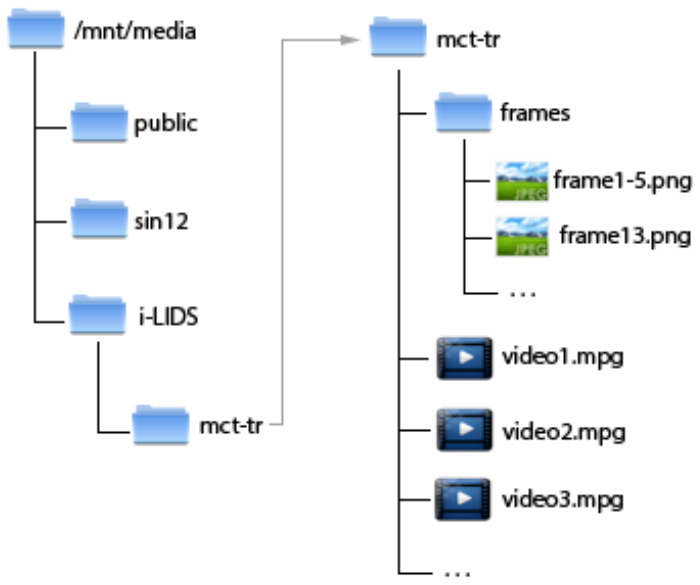
Typicky celá množina dat související s jedním projektem. Datasety jsou obvykle navzájem disjunktní, nicméně jeden může být modelován na základě předchozího.

Zdrojová data

Na úrovni diskových souborů se jedná o adresář obsahující soubory zpracovávaných videí a/nebo adresáře s obrázky. Tento adresář je uložen v základním umístění (*base location*), které je konfigurovatelné před použitím API (více v kapitole 3.4).

Obrázek 3.4 ilustruje příklad reprezentace třech datasetů na disku. Struktura adresářů je následující:

- Základním umístěním je adresář */mnt/media*.
- Datasety *public* a *sin12* nejsou zobrazeny podrobně
- Třetí dataset má komplikovanější cestu k datům *i-LIDS/mct-tr*
- Obsahuje jednu složku s obrázky *frames* a množství zpracovávaných videí



Obr. 3.4: Ukázka adresářové struktury datasetů

Databázový objekt

Seznam všech dostupných datasetů je uložen v tabulce **public.datasets.** Může vypadat následovně:

	dsname [PK] name	dslocation character varying	userid name	groupid name	created timestamp with	changed timestamp	notes text
1	public	public/	vfroml	vidte	2012-02-03		test dataset
2	sin12	sin12/	vfroml	vidte	2012-09-23		
3	sunar	i-LIDS/mct_tr/	vfroml	vidte	2012-02-03		TrecVID

Obr. 3.5: Ukázka seznamu datasetů v tabulce public.datasets

Takto definovaná struktura sad dat popisuje **databázi**, která **obsahuje**:

- tři definovaná schémata (*dsname* – názvy public, sin12, sunar)
- metainformaci o zdrojových datech těchto sad - jsou uloženy v adresářích „*base location*/*dslocation*“
- další doplňující informace o sadách dat (vlastník, skupina, datum vytvoření, poznámky)

Každé **schéma** odpovídající sadě dat typicky **obsahuje** následující objekty:

- tabulky *sequences*, *intervals*, *methods* a *processes* (viz datový model obr. 2)
- vlastní tabulky / sloupce tabulek **výstupních dat,** specifické pro vyvíjenou aplikaci
- uživatelem definované **datové typy**, lze rozdělit do dvou kategorií:
    - **povinné**: zejména výčtové typy, nutné v každém datasetu
    - **nepovinné**: zejména geometrické datové typy, příp. typy pro OpenCV, specifické pro každý projekt zvlášť
- další specifické databázové objekty (sekvence, triggery, pohledy atd.)

Nový dataset (schéma databáze) se z bezpečnostních důvodů **vytváří manuálně** administrátorem databáze.

Existující dataset lze v aplikaci získat k manipulaci jednoduše:

```
// vstupní třída API
// nastavení konfiguračním souborem "./vtapi.conf" (viz kap. 3.6)
VTApi* vtapi = new VTApi("./vtapi.conf");
// Vytvoření objektu reprezentujícího dataset sunar
Dataset* ds_sunar = vtapi->newDataset("sunar");

// ... práce

// Destrukce objektů
delete (ds_sunar);
delete (vtapi);
```

Třída **Dataset** dědí metody a atributy třídy *KeyValues* (viz kapitola 3.3), poskytuje následující vlastní funkce:

`string getName ()`	
▪ získá název datasetu	**parametry** • žádné **návratová hodnota** • *string*: název datasetu
`string getLocation ()`	
▪ získá umístění datasetu	**parametry** • žádné **návratová hodnota** • *string*: umístění datasetu
`Sequence* newSequence (const string& name)`	
▪ vytvoří objekt pro reprezentaci sekvencí (videa, složky obrázků) pro aktuální dataset	**parametry** • *name*: nepovinný, lze jím vybrat pouze jednu sekvenci datasetu **návratová hodnota** • *Sequence**: ukazatel na objekt reprezentující sekvence
`Video* newVideo (const string& name)`	
▪ vytvoří objekt pouze pro reprezentaci videa pro aktuální dataset	**parametry** • *name*: nepovinný, lze jím vybrat pouze jedno video datasetu **návratová hodnota** • *Video**: ukazatel na objekt reprezentující video (videa)

Tab. 3.1: Specifické funkce třídy *Dataset*

3.2.2 Sequence

Základní jednotka datasetu, reprezentuje jednu část dat, která je vnitřně časově uspořádána.

Zdrojová data

Na úrovni diskových souborů se jedná o samostatná videa či adresáře s obrázky. Tato data jsou uložena v umístění datasetu, do kterého přísluší. V **obrázku 3.4** (výše) existují v datasetu *sunar* čtyři sekvence: *frames* (složka obrázků), *video1.mpg*, *video2.mpg* a *video3.mpg* (videa).

Databázový objekt

Seznam všech sekvencí příslušného datasetu je uložen v tabulce **[jméno_datasetu].sequences.** Záznam o sekvenci se do ní uloží pomocí funkce *Sequence->add()* (viz. tab. 3.2). Tabulka může vypadat následovně:

	seqname [PK] name	seqnum integer	seqlocation character varying	seqtyp seqtype	userid name	groupid name	created timestamp witl	changed timestamp	notes text
1	frames	1001	frames/	images	vfroml	vidte	2012-02-03		
2	video1	1002	video1.mpg	video	vfroml	vidte	2012-02-03		
3	video2	1003	video2.mpg	video	vfroml	vidte	2012-02-03		
4	video3	1004	video3.mpg	video	vfroml	vidte	2012-02-03		

Obr. 3.6: Ukázka seznamu sekvencí v tabulce sunar.sequences

Objekt reprezentující sekvence lze získat pomocí třídy **Sequence** následovně:

```
// Objekt reprezentující dataset sunar
Dataset* ds_sunar = vtapi->new Dataset("sunar");
// Objekt reprezentující všechny sekvence datasetu sunar
Sequence* all_sunar = ds_sunar->newSequence();
// Objekt reprezentující pouze sekvenci video2
Sequence* video2_sunar = ds_sunar->newSequence("video2");

// .. práce

// Destrukce objektů
delete (all_sunar); delete (video2_sunar);
delete (ds_sunar);
```

V některých situacích může být vhodné pracovat pouze s jednou vybranou sekvencí, nikoli se všemi dostupnými. Z ukázky je zřejmé, že tohoto omezení lze dosáhnout omezením názvu sekvence ve funkci *newSequence*.

Jako alternativu k třídě *Sequence* lze použít intuitivnější třídu **Video**, která poskytuje stejnou funkčnost a navíc zajišťuje kontrolu konzistence a správný formát videa. Je tedy závislá na knihovně OpenCV.

```
// Objekt reprezentující video2
Video* video2_sunar = ds_sunar->newVideo("video2");
```

Třída **Sequence** (Video) opět dědí většinu svých metod a atributů ze třídy *KeyValues* (viz kapitola 3.3). Poskytuje tyto vlastní funkce:

<table>
<tr><td colspan="2"><code>string getName ()</code></td></tr>
<tr><td>▪ získá název sekvence</td><td>parametry
• žádné
návratová hodnota
• string: název sekvence</td></tr>
<tr><td colspan="2"><code>string getLocation ()</code></td></tr>
<tr><td>▪ získá umístění sekvence</td><td>parametry
• žádné
návratová hodnota
• string: umístění sekvence</td></tr>
<tr><td colspan="2"><code>Interval* newInterval (const string& name)</code></td></tr>
<tr><td>▪ vytvoří objekt pro reprezentaci intervalů (obrázků, snímků videa) pro aktuální sekvenci</td><td>parametry
• name: nepovinný, lze jím vybrat pouze jeden interval sekvence
návratová hodnota
• Interval*: ukazatel na objekt reprezentující interval (intervaly)</td></tr>
<tr><td colspan="2"><code>Image* newImage (const string& name)</code></td></tr>
<tr><td>▪ vytvoří objekt pouze pro reprezentaci obrázku pro aktuální sekvenci</td><td>parametry
• name: nepovinný, lze jím vybrat pouze jeden obrázek sekvence
návratová hodnota
• Image*: ukazatel na objekt reprezentující obrázek (obrázky)</td></tr>
<tr><td colspan="2"><code>bool add (string name, string location, string type)</code></td></tr>
<tr><td>▪ přidá do databázové tabulky [jméno_datasetu] .sequences záznam o nové sekvenci</td><td>parametry
• name: název sekvence
• location: umístění sekvence (název videa či složky obrázků)
• type: jeden z typů sekvence (images – obrázky, video)
návratová hodnota
• bool: ukazatel úspěchu</td></tr>
</table>

Tab. 3.2: Specifické funkce třídy *Sequence*

Následující příklad ilustruje přidání sekvence (existující video) do datasetu *sunar*:

```
// Objekt reprezentující dataset sunar
Dataset* ds_sunar = vtapi->new Dataset("sunar");
// Objekt reprezentující všechny sekvence datasetu sunar
Sequence* all_sunar = ds_sunar->newSequence();
// Přidání nového videa do datasetu pod názvem video8
all_sunar->add("video8", "video8.mpg", "video");
// Provedení databázového dotazu
all_sunar->addExecute();
```

3.2.3 Interval

Podmnožina sekvence zvolená tak, aby jí bylo možné přiřadit jednotný příznak či metadata (např. výskyt nějakého objektu v obraze). Jedná se tedy o jeden či více obrázků (snímků videa), u kterých se předpokládá časová spojitost.

Zdrojová data

Na úrovni diskových souborů se jedná o jednotlivé obrázky ve složce sekvence. Časové intervaly v konkrétním videu jsou vyjádřeny pouze metadaty v databázi.

Databázový objekt

Seznam všech intervalů příslušné sekvence je uložen v tabulce **[jméno_datasetu].intervals.** Záznam o intervalu se do ní uloží pomocí funkce *Interval->add()* (viz. tab. 3.3). Tabulka může vypadat následovně:

	seqname [PK] character	t1 [PK] intege	t2 [PK] intege	imglocation character varying	tags integer[]	svm real[]	userid name	created timestamp	notes text
1	frames	1	1	frame1-5.png	{2,3,5,8}		vfroml	2011-10-	
2	frames	2	2	frame13.png	{0,2,3,9}		vfroml	2011-10-	
3	video1	1	3		{1,9,10}		vfroml	2011-12-	
4	video1	7	12		{3,12,8,9,6}		vfroml	2013-01-	
5	video3	5	69		{0,1,7,8}		vfroml	2013-01-	

Obr. 3.7: Ukázka seznamu intervalů v tabulce sunar.intervals

Objekt reprezentující interval lze získat pomocí třídy **Interval** následovně:

```
// Objekt reprezentující dataset sunar
Dataset* ds_sunar = vtapi->new Dataset("sunar");
// Objekt reprezentující pouze sekvenci video2
Sequence* video3_sunar = ds_sunar->newSequence("video3");
// Objekt reprezentující všechny intervaly videa video3
Interval* intall_video3 = video3_sunar->newInterval();
// Objekt reprezentující pouze intervaly videa video3 <5,69>
Interval* int1_video3 = video3_sunar->newInterval(5, 69);

// .. práce

// Destrukce objektů
delete (intall_video3); delete (int1_video3);
delete (video3_sunar); delete (ds_sunar);
```

V některých situacích může být opět vhodné pracovat pouze s jedním vybraným intervalem, nikoli se všemi dostupnými. Podobně jako u třídy *Sequence* toho lze dosáhnout omezením názvu intervalu ve funkci *newInterval*.

Jako alternativu k třídě *Interval* pro manipulaci s obrázky lze použít třídu **Image**:

```
// Objekt reprezentující pouze složku obrázků frames
Sequence* frames_sunar = ds_sunar->newSequence("frames");
// Objekt reprezentující obrázek frame13.png
Image* frame13_sunar = frames_sunar->newImage("frame13.png");
```

Třída **Interval** (Image) opět dědí většinu svých metod a atributů ze třídy *KeyValues* (viz kapitola 3.3).

Poskytuje tyto vlastní funkce:

<table>
<tr><td colspan="2"><code>string getSequenceName ()</code></td></tr>
<tr><td>▪ získá název sekvence, do které tento interval přísluší</td><td>parametry
• žádné
návratová hodnota
• string: název sekvence</td></tr>
<tr><td colspan="2"><code>Sequence* getSequence ()</code></td></tr>
<tr><td>▪ získá objekt reprezentující sekvenci, do které tento interval přísluší</td><td>parametry
• žádné
návratová hodnota
• Sequence*: ukazatel na objekt sekvence</td></tr>
<tr><td colspan="2"><code>int getStartTime ()</code></td></tr>
<tr><td>▪ získá počáteční čas intervalu</td><td>parametry
• žádné
návratová hodnota
• int: počáteční čas intervalu</td></tr>
<tr><td colspan="2"><code>int getEndTime ()</code></td></tr>
<tr><td>▪ získá koncový čas intervalu</td><td>parametry
• žádné
návratová hodnota
• int: koncový čas intervalu</td></tr>
<tr><td colspan="2"><code>bool add (const string& sequence, const int t1, const int t2, const string& location)</code></td></tr>
<tr><td>▪ přidá do databázové tabulky [jméno_datasetu] .intervals záznam o novém intervalu</td><td>parametry
• sequence: název sekvence, do které bude nový interval příslušet
• t1: počáteční čas intervalu
• t2: koncový čas intervalu
• location: umístění obrázku
návratová hodnota
• bool: ukazatel úspěchu</td></tr>
</table>

Tab. 3.3: Specifické funkce třídy *Interval*

Následující příklad ilustruje přidání intervalu do videa *video3* v datasetu *sunar*:

```
// Objekt reprezentující dataset sunar
Dataset* ds_sunar = vtapi->new Dataset("sunar");
// Objekt reprezentující pouze sekvenci video3
Sequence* video3_sunar = ds_sunar->newSequence("video3");
// Objekt reprezentující všechny intervaly sekvence video3
Interval* intall_video3 = video3_sunar->newInterval();
// Přidání nového intervalu do videa video3
intall_video3->add("video3", 5, 8);
// Provedení databázového dotazu
intall_video3->addExecute();
```

3.3 Správa a dotazování dat

Všechny výše popsané třídy reprezentující datové objekty dědí atributy a metody třídy **KeyValues**. Tato třída je pojmenována podle základního principu přístupu klíč-hodnota k uloženým popisným metadatům obrazových dat. Jedná se tedy o nejpodstatnější část celého konceptu VTApi.
Třída **KeyValues** zajišťuje následující funkčnost:

- **procházení obrazových dat:** jak bylo uvedeno v kapitole 3.2, objekty *Dataset*, *Sequence* a *Interval* mohou reprezentovat více objektů zároveň. Jejich procházení zajišťuje nejdůležitější a zřejmě nejpoužívanější **funkce next()** (viz kapitola 3.3.1).
- **získávání uložených metadat**, pomocí sady tzv. **funkcí getX** (viz kapitola 3.3.2)
- **provádění databázových dotazů** a příkazů pomocí vnořených tříd **Select**, **Insert** a **Update** (viz kapitoly 3.3.3 až 3.3.5)
- **uživatelské nastavení:** připojení k databázi, logování a další parametry (viz kapitola 3.4). Toto zajišťuje vnořená třída **commons**, která je koncovému uživateli skryta.
- **tisk** výsledků dotazů (funkce *print()* a *printAll()*)

3.3.1 Funkce next()

Všechny instance tříd reprezentující obrazová data v sobě obsahují seznam příslušných objektů, které mohou potenciálně reprezentovat. Tedy např. objekt třídy *Dataset* obsahují seznam všech datasetů (pokud jsme jej neomezil pouze na jeden specifický dataset, jak bylo popsáno výše). Podobně objekty *Sequence*, *Interval* apod.

Funkce **next()** umožňuje procházení tohoto seznamu posouváním vnitřního ukazatele na právě aktivní prvek. Každým jejím zavoláním nad objektem bude tedy tento objekt reprezentovat následující dataset, sekvenci či interval. Toto umožňuje efektivní procházení velkého množství obrazových dat a vyhledávání v jejich příslušných metadatech.

Jednoduchý příklad užití ilustruje následující část kódu, ve kterém se vypíšou všechny sekvence datasetu *sunar*.

```
// Objekt reprezentující dataset sunar
Dataset* ds_sunar = vtapi->new Dataset("sunar");
// Objekt reprezentující všechny sekvence datasetu sunar
Sequence* seq_sunar = ds_sunar->newSequence();
// Cyklus procházení sekvencí
while (seq_sunar->next()) {
    string name = seq_sunar->getName();
    cout << name;
}
```

Objekt *seq_sunar* bude postupně reprezentovat všechny dostupné sekvence. Vrátí-li funkce *next()* hodnotu NULL, objekt se nastaví opět do původního stavu. Analogicky lze procházet datasety, videa, snímky videa, obrázky a procesy.

Další důležitou vlastností funkce *next()* je konzistentní reprezentace dat při jejich procházení, což zajišťuje **automatické provádění** nepotvrzených příkazů **add** a **update** před zavoláním funkce.

3.3.2 Sada funkcí getX

Funkce *next()* tedy umožňuje navigaci v databázi dat a popisných metadat. K získání požadovaných hodnot z databáze slouží sada funkcí **getX**, kde X je příslušný **datový typ**. Jako parametr funkce getX se zadává pořadí sloupce v tabulce či jeho název (klíč). VTApi implementuje následující funkce:

`getValue`	Získání hodnoty **jakéhokoliv** typu převedeného na textovou reprezentaci		
Číselné typy			
`getInt`	32bit číselná hodnota *int*	`getFloat`	64bit reálná hodnota *float*
`getIntA`	pole hodnot *int*	`getFloatA`	pole hodnot *float*
`getIntV`	standardní vektor hodnot *int*	`getFloatV`	standardní vektor hodnot *float*
`getIntVV`	libovolně rozměrný vektor hodnot *int*	`getFloatVV`	libovolně rozměrný vektor hodnot *float*
`getInt8`	64bit číselná hodnota *long*	`getFloat8`	64bit reálná hodnot *double*
Textové typy			
`getChar`	znak	`getString`	textový řetězec
`getCharA`	pole znaků		
Geometrické typy (PostGIS)			
`getPoint`	2D bod	`getPolygon`	mnohoúhelník
`getPointV`	pole 2D bodů	`getPath`	cesta
`getLineSegment`	část lomené čáry	`getCube`	n-dimenzionální kostka
`getBox`	ohraničující krabice	`getGeometry`	univerzální geometrie (GEOS)
`getCircle`	kružnice	`getLineString`	lomená čára (GEOS)
OpenCV typy			
`getCvmat`	OpenCV matice *cvMat*	`getCvMatND`	OpenCV matice *cvMatND*
Ostatní typy			
`getTimestamp`	časové razítko	`getIntOid`	databázový identifikátor *OID*

Tab. 3.4: Přehled podporovaných datových typů

Následující příklad ilustruje použití funkce **getX** (konkrétně *getFloatV*).

```
// Objekt reprezentující dataset sunar
Dataset* ds_sunar = vtapi->new Dataset("sunar");
// Objekt reprezentující video3 z datasetu sunar
Sequence* video3_sunar = ds_sunar->newSequence("video3");
// Objekt reprezentující intervaly v sekvenci video3
Interval* int_video3 = video3_sunar->newInterval();
// Cyklus procházení intervalů
while (int_video3->next()) {
    // Získání metadat ze sloupce tags
    vector<float> tags = int_video3->getFloatV("tags");
    if (tags.empty()) {
        // interval není otagován
        // .. práce
    }
    tags.clear();
}
```

3.3.3 Select

Každý objekt KeyValues obsahuje atribut instance třídy **Select.** Slouží ke konstrukci databázových dotazů, které se primárně volají po zavolání funkce *next()*. Lze je ale také konstruovat a provádět pomocí následujících funkcí:

`bool from (const string& table, const string& column)`	
▪ určuje tabulku, nad kterou se dotaz bude provádět (FROM) a příslušný sloupec	**parametry** • *table*: tabulka • *column*: sloupec tabulky **návratová hodnota** • *bool*: ukazatel úspěchu
`bool whereString (const string& key, const string& value, const string& oper, const string& table)`	
▪ konstruuje WHERE sekci dotazu	**parametry** • *key*: sloupec tabulky • *value*: hodnota pole • *oper*: operátor (implicitně =) • *table*: jiná tabulka (implicitně ‘’) **návratová hodnota** • *bool*: ukazatel úspěchu
`bool execute ()`	
▪ provede dotaz (commit)	**parametry** • žádné **návratová hodnota** • *bool*: ukazatel úspěchu

Tab. 3.5: Specifické funkce třídy Select

Následující příklad ukazuje práci s libovolnou tabulkou (např. *tags*) a třídou **Select**:

```
// Objekt datasetu
Dataset* dataset = vtapi->newDataset();
// Obecný objekt KeyValues
KeyValues* tags = new KeyValues (*dataset, "tags");
// Vytvoření instance třídy Select
tags->select = new Select(*tags);
// Specifikace podmínky real_name=NULL
tags->select->whereString("real_name", NULL);
// Procházení výsledků
while (tags->next()) {
    // .. prace
}
```

3.3.4 Insert

Třída **Insert** slouží ke konstrukci databázových příkazů vkládání do tabulky. Typicky není nutné ji používat, tuto funkčnost plně zaobalují metody *add()* třídy *KeyValues* (viz výše). Nicméně lze ji

(s určitou dávkou opatrnosti) využít i samostatně pro konstrukci vlastních dotazů za pomocí sady funkcí **`keyX`**, kde **X** je datový typ vkládané hodnoty.

`keyInt`	32bit číselná hodnota *int*	`keyFloat`	64bit reálná hodnota *float*
`keyIntA`	pole hodnot *int*	`keyFloatA`	pole hodnot *float*
`keyString`	textový řetězec	`keySeqtype`	výčtový typ typů sekvencí
`keyStringA`	pole textových řetězců	`keyInouttype`	výčtový typ typů vstup/výstup

Tab. 3.6: Přehled sady funkcí `keyX` třídy Insert

Všechny funkce **`keyX`** požadují jako první parametr název sloupce (klíč), do kterého hodnotu vkládat. Druhým parametrem je samotná hodnota. K potvrzení příkazu slouží funkce *execute()*.

```
// Objekt datasetu
Dataset* dataset = vtapi->newDataset();
// Objekt reprezentující všechny sekvence datasetu sunar
Sequence* seq_sunar = ds_sunar->newSequence();
// Vytvoření instance třídy Insert
seq_sunar->insert = new Select(*seq_sunar);
// Specifikace příkazu Insert
seq_sunar->insert->keyString("seqname", "video5");
seq_sunar->insert->keyString("seqlocation", "video5.mpg");
seq_sunar->insert->keySeqtype("seqtyp", "video");
// Potvrzení dotazu
seq_sunar->insert->execute();
```

3.3.5 Update

Třída **Update** slouží ke konstrukci databázových příkazů aktualizace záznamů tabulky. Pro její použití zaobaluje sada funkcí **`setX`**, kde **X** je datový typ aktualizované hodnoty.

`setInt`	32bit číselná hodnota *int*	`setFloat`	64bit reálná hodnota *float*
`setIntA`	pole hodnot *int*	`setFloatA`	pole hodnot *float*
`keyString`	textový řetězec		

Tab. 3.7: Přehled sady funkcí `setX`

Všechny funkce **`setX`** požadují jako první parametr název sloupce (klíč), do kterého hodnotu vkládat. Druhým parametrem je samotná hodnota. K potvrzení příkazu slouží funkce *execute()*. Předpokládejme, že máme naplněné pole tagů (*float* tags*) pro *video5* v datasetu *sunar*:

```
// Objekt datasetu
Dataset* dataset = vtapi->newDataset();
// Objekt reprezentující sekvenci video5
Sequence* video5_sunar = ds_sunar->newSequence("video5");
// Vytvoření instance třídy Update
video5_sunar->update = new Update(*video5_sunar);
// Aktualizace tagů sekvence video5
video5_sunar->setFloatA("tags", tags);
// Potvrzení dotazu
video5_sunar->update->execute();
```

3.4 Konfigurace

VTApi podporuje nastavení připojení k datovému úložišti a dalších možností, a to prostřednictvím **konfiguračního souboru** a možných **parametrů příkazové řádky** (ty mají v případě konfliktu přednost). Tato konfigurace se předává jako parametr vstupní třídě VTApi a následně se kopíruje do všech objektů reprezentujících data prostřednictvím skryté vnořené třídy **commons**.

Spuštění VTApi pouze s konfiguračním souborem:

```
VTApi* vtapi = new VTApi("/path/to/vtapi.conf");
// .. práce
delete(vtapi);
```

Spuštění VTApi s parametry příkazové řádky a konfiguračním souborem:

```
int main(int argc, char** argv) {
        VTApi* vtapi = new VTApi(argc, argv);
        // .. práce
        delete(vtapi);
    }
```

Přehled konfigurovatelných parametrů

Parametr	Zkratka	Implicitní hodnota	Popis
config	-	./vtapi.conf	Umístění konfiguračního souboru
log	-	./vtapi.log	Umístění souboru logu
user	u	-	Jméno uživatele
password	p	-	Heslo uživatele
location	l	-	Základní umístění souborů obrazových dat
connection	c	-	Specifikace připojení ve formátu: *host=... port=... dbname=... user=... password=....*
format	f	standard	Formát výstupu tisku (*standard / csv / html / binary / sparse*)
input	i	-	Specifikace vstupního streamu
output	o	-	Specifikace výstupního streamu
queryLimit	-	-	Omezení maximálního počtu řádků ve výsledku jednoho dotazu
arrayLimit	-	-	Omezení maximálního počtu zobrazených prvků tisknutého pole
Specifikace kontextu dat			
where	W	-	SQL řetězec definující klauzuli WHERE všech dotazů SELECT
dataset	D	-	Užití pouze určeného datasetu
sequence	S	-	Užití pouze určené sekvence
interval	I	-	Užití pouze určeného intervalu
method	M	-	Užití pouze určené metody
process	P	-	Užití pouze určeného procesu
selection	E	-	Užití pouze určené selekce

Tab. 3.8: Přehled konfigurovatelných parametrů

Ukázka jednoduchého konfiguračního souboru:

```
# VTApi configuration file

user="vfroml"
password="******"

location="/mnt/vidte/"
connection="host=vidte.cz port=1111 dbname=dbvidte user=vfroml password=*****"

verbose

log="./logs/vtapi.log"
format="csv"

querylimit=10000
arraylimit=10

dataset="sunar"
```

3.5 Případy užití

Následující případy užití jsou převzaty z technické zprávy projektu VTApi z roku 2012.

Vyhledávání statických obrázků

První příklad obsahuje implementaci jednoduchého podobnostního vyhledávání obrázků. Algoritmus využívá metodu *distance_square_int4* obsaženou v databázovém modulu *pgDistance* (je součástí kódu VTApi). Výsledkem bude nalezení obrázku nejpodobnějšího vzoru *Q.jpg* v datasetu *search*.

```
// VTApi entry point, using Dataset "search"
VTApi* vtapi = new VTApi(argc, argv);
Dataset* dataset = vtapi->newDataset("search");
// code of the ColorLayout Method, using Selection "image"
Image* image = new Image(&dataset, "image");
while (image->next()) {
    vector color = colorLayout(image->getDataLocation());
    image->setIntA("color", color);
}
// retrieve Image(s) according to their similarity to "Q.jpg"
Image* nearest = new Image(&dataset);
nearest->select->from("image", "distance_square_int4(color, "
    + toString(colorLayout("Q.jpg")) +")");
nearest->select->orderby("distance_square_int4");
nearest->next();   // is the most similar image
```

Sledování objektů

V následujícím příkladu je ukázáno, jak lze implementovat shlukování trajektorií s použitím VTApi a OpenCV aplikace algoritmu *Expectation-Maximization* (EM), který odhaduje parametry modelu

GMM (*Gaussian Mixture Model*). Vektory rysů se načtou z databáze a připraví se trénovací vzorky pro algoritmus EM. GMM se naučí pomocí algoritmu EM a příslušná označení shluků jsou nahrány do databáze. Předpokládejme, že trajektorie jsou ukládány jako intervaly *tracks*.

```
Mat samples; // cv::Mat of training feature vectors
VTApi* vtapi = new VTApi(argc, argv);
Dataset* dataset = vtapi->newDataset("train"); // training dataset
dataset->next();
Sequence* sequence = dataset->newSequence();
 while (sequence->next()) { // for each video
    Interval* track = new Interval(*sequence, "tracks");
    while (track->next()) { // for each trajectory
        Mat sample; // cv::Mat for feature vector of trajectory
        float feature = track->getFloat("feature");
        // ... read features and fill feature vector
        samples.push_back(sample);
    }
}
```

```
CvEM model, labels; // GMM-EM model and cluster labels
CvEMParams params; // EM parameters
// ... set EM parameters including number of clusters
model.train(samples, Mat(), params, labels);
// ... choose dataset and sequences according to sample code above
while (track->next()) {
    Mat sample;
    // ... read features and fill feature vector
    int cluster = (int) model.predict(sample); // get cluster label
    track->setInt("cluster", cluster); // store cluster label
}
```

3.6 VTCli

Nástroj **VTCli** je konzolová aplikace implementující jednoduché rozhraní pro spouštění nejpoužívanějších funkcí VTApi. Byla vyvinuta specificky pro účely automatizace některých objemově náročných úkonů (např. nahrání dat do databáze) a testování funkčnosti API.

Konfigurace aplikace se provádí pomocí parametrů příkazové řádky (viz kap. 3.4) a příslušného konfiguračního souboru.

Nástroj může pracovat ve dvou různých módech:

- **Interaktivní**
  - Aplikace bude čekat na vstupy uživatele a vypisovat na standardní výstup
- **Jednorázový**
  - Aplikace se spustí jednorázově, zadá-li uživatel jako parametry příkazové řádky za konfigurační parametry také **samotný příkaz** (jejich přehled viz níže)**.** Tento příkaz se pak jednorázově provede a aplikace se ukončí.

3.6.1 Přehled příkazů

Následující tabulka popisuje příkazy nástroje VTCli:

Příkaz	Následuje	Popis
query	SQL řetězec dotazu	Provedení vlastního databázového dotazu
select	dataset | sequence | interval | process **[ARGS]**	Provedení dotazu
insert	sequence | interval | process **[ARGS]**	Provedení příkazu vložení
test	-	Provedení testovací funkce
help	-	Zobrazení nápovědy
exit	-	Ukončení nástroje

Tab. 3.9: Přehled příkazů nástroje VTCli

Formát parametrů **[ARGS]** je *vlastnost=hodnota*, příp. *vlastnost=hodnota1,hodnota2,…*

Parametry select			
dataset			
`name`	název datasetu	`location`	umístění datových souborů datasetu
sequence			
`name`	název sekvence	`location`	umístění datových souborů sekvence
`num`	unikátní číslo sekvence	`type`	typ sekvence (*images / video*)
interval			
`seqname`	název sekvence, do které tento interval patří	`t1`	počáteční čas intervalu
`location`	umístění obrázku	`t2`	koncový čas intervalu
method			
`name`	název metody		
process			
`name`	název procesu	`method`	název metody, jejíž instancí je tento proces
`inputs`	datový typ vstupů (tabulka)	`outputs`	datový typ výstupů (tabulka)
Parametry insert			
sequence			
`name`	název sekvence (povinné)	`location`	umístění datových souborů sekvence
`num`	unikátní číslo sekvence	`type`	typ sekvence (*images / video*)
interval			
`seqname`	název sekvence, do které tento interval patří	`t1`	počáteční čas intervalu (povinné)
`location`	umístění obrázku	`t2`	koncový čas intervalu (povinné)
process			
`name`	název procesu (povinné)	`method`	název metody, jejíž instancí je tento proces
`inputs`	datový typ vstupů (tabulka)	`outputs`	datový typ výstupů (tabulka)

Tab. 3.10: Přehled parametrů příkazů nástroje VTCli

3.6.2 Ukázky použití

Interaktivní mód

Spuštění s parametrem umístění konfiguračního souboru:

```
./vtcli --config="./vtapi.conf"
```

Dotaz na dostupné datasety:

```
> select dataset
dsname|dslocation    |userid|groupid|created            |changed  |notes
name  |varchar       |name  |name   |timestamp          |timestamp|text
------+--------------+------+-------+-------------------+---------+------------
sin12 |sin12/        |vfroml|vidte  |2012-08-23 04:41:22|         |
sunar |i-LIDS/mct_tr/|vfroml|vidte  |2012-01-03 20:41:41|         |TrecVID
public|public/       |vfroml|vidte  |2012-01-03 20:41:41|         |test dataset
(3 rows)
```

Dotaz na sekvence datasetu:

```
> select sequence
seqname|seqnum|seqlocation|seqtyp |userid|groupid|created            |changed  |notes
name   |int4  |varchar    |seqtype|name  |name   |timestamp          |timestamp|text
-------+------+-----------+-------+------+-------+-------------------+---------+-----
video1 |1002  |video1.mpg |video  |vfroml|vidte  |2012-01-03 20:41:41|         |
video2 |1003  |video2.mpg |video  |vfroml|vidte  |2012-01-03 20:41:41|         |
video3 |1004  |video3.mpg |video  |vfroml|vidte  |2012-01-03 20:41:41|         |
frames |1001  |frames/    |images |vfroml|vidte  |2012-01-03 20:41:41|         |
(4 rows)
```

Dotaz na intervaly sekvence *video1*:

```
> select interval seqname=video1
seqname|t1  |t2  |imglocation|tags      |svm    |userid|created            |notes
varchar|int4|int4|varchar    |_int4     |_float4|name  |timestamp          |text
-------+----+----+-----------+----------+-------+------+-------------------+-----
video1 |1   |3   |           |1,9,10    |       |vfroml|2011-11-08 20:01:55|
video1 |7   |12  |           |3,12,8,9,6|       |vfroml|2013-00-09 21:51:25|
video1 |2   |6   |           |2,5       |       |vfroml|2013-00-10 18:04:06|
(3 rows)
```

Vložení nového intervalu do sekvence *video1*:

```
> insert interval seqname=video1 t1=10 t2=16 tags=2,6,3
seqname|t1  |t2  |imglocation|tags      |svm    |userid|created            |notes
varchar|int4|int4|varchar    |_int4     |_float4|name  |timestamp          |text
-------+----+----+-----------+----------+-------+------+-------------------+-----
video1 |1   |3   |           |1,9,10    |       |vfroml|2011-11-08 20:01:55|
video1 |7   |12  |           |3,12,8,9,6|       |vfroml|2013-00-09 21:51:25|
video1 |2   |6   |           |2,5       |       |vfroml|2013-00-10 18:04:06|
video1 |10  |16  |           |2,6,3     |       |vfroml|2013-00-10 18:30:23|
(4 rows)
```

Jednorázový mód

Spuštění sady příkazů vložení intervalů do datasetu *sunar*:

```
$ ./vtcli --dataset="sunar" insert interval seqname=video2 t1=1 t2=6 tags=1,5,6,9
$ ./vtcli --dataset="sunar" insert interval seqname=video2 t1=3 t2=9 tags=2,6,9,11,12
$ ./vtcli --dataset="sunar" insert interval seqname=video3 t1=2 t2=3 tags=2,6,9,11,12
```

4 Návrh VTApi pro různá úložiště dat

Cílem projektu VTApi je ulehčení a sjednocení vývoje analytických aplikací založených na počítačovém vidění. Důležitý faktor při návrhu koncepce tohoto rozhraní tedy představuje použití co nejuniverzálnějších a nejdostupnějších nástrojů, prostředí a metodologie. Systém musí podporovat širokou škálu typů obrazových dat, popisných metadat a umožňovat také škálovatelnost náročnosti analytických algoritmů, které jej využívají. Lze identifikovat několik faktorů, které určují míru univerzálnosti konceptu VTApi:

- **nezávislost na typech obrazových dat**
  - API nepředpokládá žádný formát obrazových dat, použití metod zajišťujících validitu použitých dat (např. třídy *Video* místo obecné *Sequence*) je zcela volitelné
- **podpora velkého množství datových typů pro reprezentaci metadat**
  - API implicitně podporuje různorodé matematické (např. matice, vektory) a geometrické (např. body, čáry, plochy) datové typy. V současné verzi se jedná především o typy databázového rozšíření PostGIS, jejichž výčet je k naleznutí v kapitole 3.3.2
  - Součástí programové dokumentace je i návod pro nainstalování a registraci vlastních datových typů
- **dostupnost API pro různé programovací jazyky**
  - rozhraní je vyvíjeno primárně pro použití jako knihovna v jazyce C++, je převáděno (*binding*) pro použití v jazyce Python
- **nezávislost na typu datového úložiště**
  - v současné verzi knihovna VTApi podporuje pro ukládání metadat explicitně pouze databázi PostgreSQL

Z pohledu návrhu všestranného systému se tedy za nejzávažnější nedostatek dá považovat povinná závislost knihovny (a tedy analytické aplikace) na SŘBD PostgreSQL. Z pohledu klientské aplikace si toto vynucuje implementaci celého systému jako architekturu klient-server. Nutnost databáze s sebou dále přináší povinnost její instalace, spravování a složité přenositelnosti a replikovatelnosti, což může ve specifických podmínkách představovat výraznou překážku ve vývoji.

Tato kapitola představuje modifikaci konceptu VTApi, která by umožňovala implementaci podpory pro různá datová úložiště. Jedná se tedy o vytvoření určité úrovně abstrakce v současném návrhu, která bude reprezentovat rozhraní pro jednotlivé moduly obsahující metody přístupu k těmto datovým úložištím.

4.1 Požadavky na návrh

Vzhledem k tomu, že navrhovaná modifikace je plánována jako součást nové verze systému VTApi, je nutno počítat se z toho vyplývajícími požadavky a zahrnout je do návrhu rozšíření. Mezi ně patří především zajištění zpětné kompatibility se staršími používanými verzemi VTApi, umožnění přenositelnosti kódu aplikace mezi používáním různých datových úložišť a zachování dosavadní funkcionality.

Zpětná kompatibilita

Za základní požadavek na novou verzi lze považovat neprovedení změny konceptu API z pohledu jeho koncového uživatele. Toto znamená zejména zachování logické hierarchie reprezentace obrazových dat (*Dataset* - *Sequence* - *Interval*), dále principu konfigurace a správy připojení k databázi a v neposlední řadě funkcí umožňujících dotazování, vkládání a aktualizaci záznamů v datovém úložišti.

Přenositelnost kódu aplikace

Dalším požadavkem je kompatibilita celého systému mezi různými úložišti. Z pohledu návrhu se jedná zejména o zajištění jednotné definice poskytovaných funkcí, které využívají připojení k datovému úložišti. Podrobnou specifikaci těchto funkcí je nutno zařadit do návrhu jakéhokoliv objektu implementujícího jejich abstrakci.

Z tohoto požadavku nicméně vyplývá zásadní problém. V současném konceptu a implementaci VTApi (jak je popsán v kapitole 3) existuje množina funkcí, explicitně užívající geometrické datové typy specifické pro PostGIS (rozšíření SŘBD PostgreSQL). Aplikace tedy musí tyto typy podporovat a používat knihovnu pro manipulaci s nimi, což je v konfliktu s požadavkem na databázovou nezávislost.

Jako řešení se nabízí přechod na univerzálnější notaci geometrických typů, např. GML (*Geography Markup Language*). Toto řešení by kladlo na funkce manipulující s hodnotami těchto typů požadavek na provádění jejich konverze z typů podporovaných datovým úložištěm na GML (a naopak). Takto navržené rozhraní by tak zbavilo aplikace nutnosti závislosti na knihovně PostGIS.

Tato varianta řešení by ale zároveň vyžadovala změnu definicí funkcí a tím tedy i změnu návrhu konceptu VTApi, čímž by se dostala do konfliktu s požadavkem zpětné kompatibility. Z tohoto důvodu bylo rozhodnuto o zachování závislosti na datových typech PostGIS jakožto základní varianty datových typů pro manipulaci s metadaty, nicméně pouze pro aplikace, které tyto typy využívají.

Jednotné typování hodnot

Jednou z významných funkcí VTApi je korektní konverze databázových datových typů na podporované typy přístupné uživateli (viz tab. 3.4). Vzhledem k tomu, že není možné měnit definici rozhraní, se s touto konverzí musí vypořádat implementace pro jednotlivé typy datových úložišť, např. pomocí vlastních serializačních a deserializačních metod. Je potřeba, aby v návrhu byla tato potřeba zohledněna.

Zachování funkcionality

Nová verze VTApi by pochopitelně měla poskytovat veškerou funkcionalitu verze současné (popsána v kapitole 3). Tento požadavek se projeví zejména ve výběru samotných datových úložišť, ke kterým se budou implementovat konektory. Je nutné, aby obsahovaly podporu pro indexaci a vyhledávání v souladu s konceptem VTApi a pro kompletní škálu používaných datových typů.

Univerzálnost

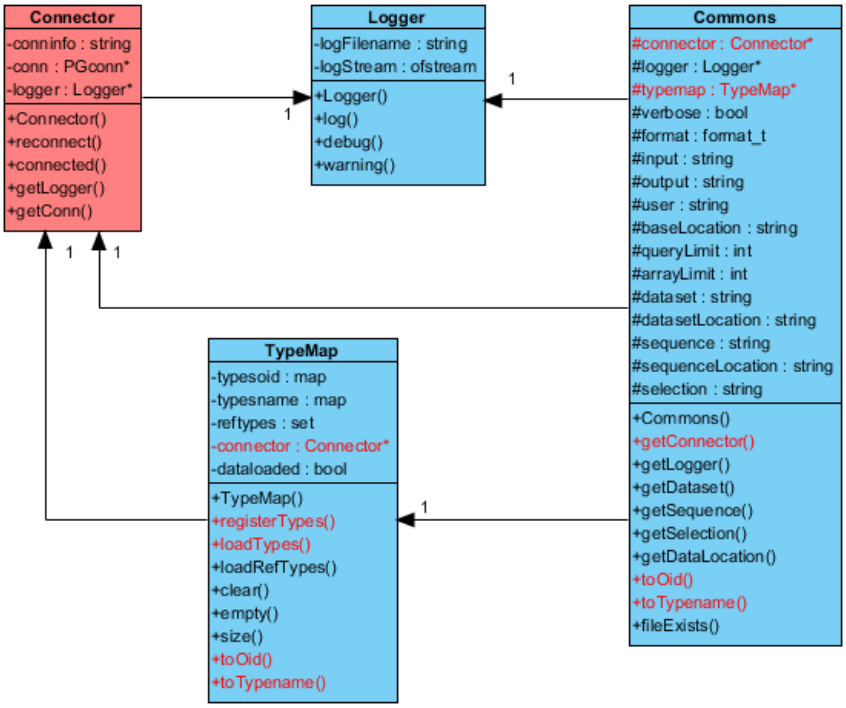
Navrhované rozšíření by mělo umožňovat implementovat správu připojení obecně pro všechna vhodná datová úložiště. Z toho plyne i flexibilita v návrhu celého systému používajícího knihovnu VTApi, např. zbavení se závislosti na architektuře klient-server.

4.2 Abstrakce tříd VTApi

Pro návrh rozšíření splňujícího návrhové požadavky je zapotřebí nejdříve identifikovat objekty a metody poskytované rozhraním, které využívají přístup k datovému úložišti. Na základě analýzy účelu těchto objektů a specifických vlastností modelu API se pak navrhne nová vrstva abstrakce v konceptu systému. Tato vrstva bude reprezentována jedním či více abstraktními třídami (rozhraními) sdružujícími související operace a atributy. Její ideální návrh by tak měl představovat množinu typických návrhových vzorů umožňujících abstrakci vhodně začleněných do celkového návrhu.

Metody a objekty k abstrakci

Následující obrázky (4.1 a 4.2) ukazují velmi zjednodušený diagram tříd VTApi obsahující všechny operace a atributy (červeně), které jsou v současném návrhu závislé na typu datového úložiště. Tabulky 4.1 až 4.5 pak tyto atributy a operace popisují.



Obr. 4.1: Část diagramu tříd VTApi s vyznačenými objekty k abstrakci

Commons

- Třída obsahující konfigurační parametry a instance objektů spravujících připojení k databázi (**Connector**), datové typy (**TypeMap**) a log (**Logger**)

`connector`	atribut instance třídy **Connector** spravující připojení k databázi
`getConnector ()`	getter atributu *connector*
`toOid ()`	převede název datového typu na jeho identifikátor OID pomocí
`toTypename ()`	převede identifikátor OID na název datového typu pomocí

Tab. 4.1: Atributy a operace třídy *Commons* k abstrakci

Connector

- Třída spravující připojení k databázi. V návrhu nové verze se vyskytovat nebude.

`conninfo`	atribut obsahující řetězec připojení k databázi
`conn`	atribut instance objektu připojení k databázi
`logger`	atribut instance objektu logu
`reconnect ()`	připojí se k databázi
`connected ()`	otestuje stav připojení k databázi
`getConn ()`	getter atributu *conn*

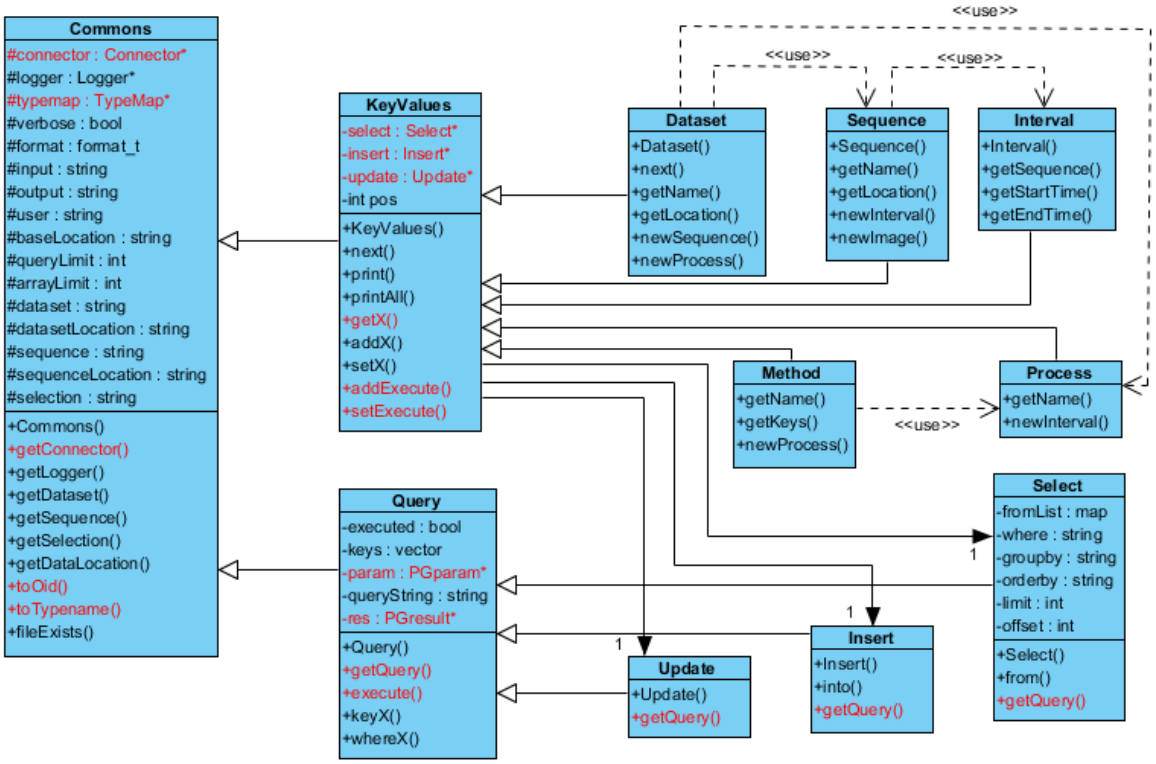
Tab. 4.2: Atributy a operace třídy *Connector* k abstrakci

TypeMap

- Třída spravující datové typy

`connector`	atribut instance třídy **Connector** spravující připojení k databázi
`registerTypes ()`	registruje z databáze uživatelské typy
`loadTypes ()`	nahrává seznam databází podporovaných datových typů
`toOid ()`	převede název datového typu na jeho identifikátor OID
`toTypeName ()`	převede identifikátor OID na název datového typu

Tab. 4.3: Atributy a operace třídy *TypeMap* k abstrakci



Obr. 4.2: Část diagramu tříd VTApi s vyznačenými objekty k abstrakci

KeyValues

- Třída poskytující základní funkčnost dotazování, vkládání a aktualizace záznamů v databázi. Dědí z ní všechny třídy reprezentující data (**Dataset, Sequence, Interval**) a algoritmy (**Method, Process**).

`select`	atribut instance třídy **Select** umožňující dotazování databáze
`insert`	atribut instance třídy **Insert** umožňující vkládání záznamů do databáze
`update`	atribut instance třídy **Update** umožňující aktualizaci záznamů v databázi
`getX()`	sada funkcí **getX**, dotazující podle klíče hodnoty polí tabulky
`addExecute()`	potvrdí provedení vložení záznamu (*insert*)
`setExecute()`	potvrdí provedení aktualizace záznamu (*update*)

Tab. 4.4: Atributy a operace třídy *KeyValues* k abstrakci

Query

- Třída, ze které dědí třídy **Select, Insert** a **Update** společnou funkčnost, zejména týkající se vkládání parametrů do SQL dotazů (sady funkcí `keyX`, `whereX`)

`param`	pomocný atribut pro tvorbu SQL řetězců, zejména při použití parametrizovaných dotazů
`res`	atribut objektu obsahující výsledek dotazu
`getQuery()`	konstruuje text SQL dotazu / příkazu
`execute()`	potvrdí dotaz/příkaz databázi (*commit*)

Tab. 4.5: Atributy a operace třídy *Query* k abstrakci

Nahrávání klientských knihoven

Pro odstranění závislosti na klientských knihovnách těch úložišť, které nebude aplikace využívat, je nutno implementovat platformně nezávislé nahrávání těchto knihoven za běhu aplikace.

Použití návrhových vzorů

K implementaci podpory jednotlivých datových úložišť se jako vhodné jeví použití návrhového vzoru **Factory**. Tento vzor poskytuje způsob zaobalení vytváření konkrétních instancí jednotlivých tříd (pro různá úložiště) do jednotného rozhraní bez nutnosti specifikace konkrétních tříd. Tyto instance implementují požadované operace definované novým rozhraním. Výběr tříd, jejichž instance bude *factory* vytvářet bude prováděn při inicializaci vstupní třídy VTApi. Z pohledu klientské aplikace lze tedy konfigurovat používané úložiště bez znalosti vnitřního konceptu API.

Sdružením požadovaných operací dle jejich účelu je lze kategorizovat do pěti skupin (viz tab. 4.6). Tyto skupiny budou odpovídat abstraktním třídám (rozhraním) realizovanými konkrétními instancemi vytvářenými objektem *factory*.

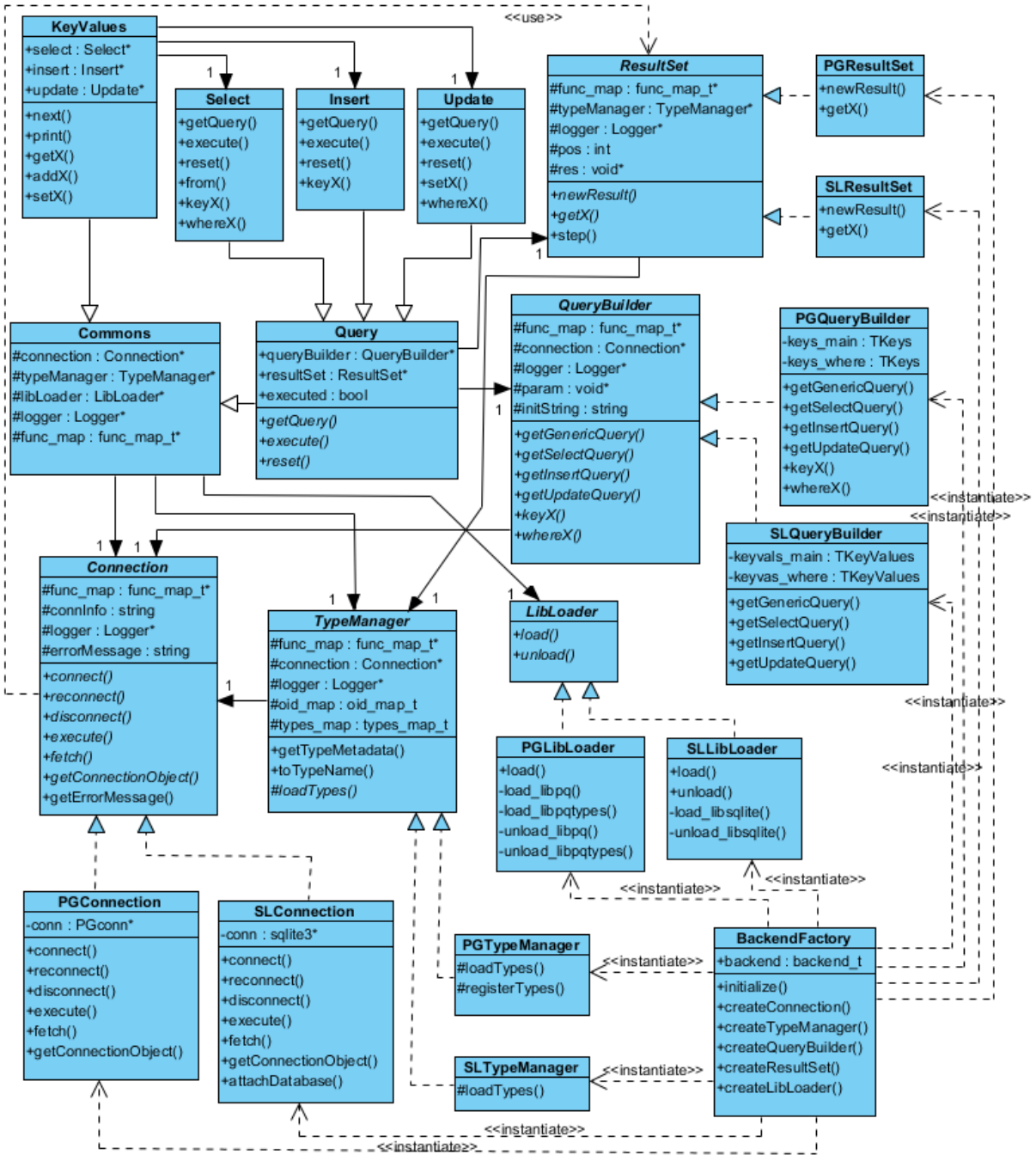
`Connection`	připojení k databázi (viz třída *Connector*) a commit příkazů (třída *Query*)
`TypeManager`	nahrávání a registrace datových typů (viz třída *TypeMap*)
`QueryBuilder`	konstrukce SQL dotazů / příkazů
`ResultSet`	získávání výsledků dotazů (sada funkcí *getX*)
`LibLoader`	nahrávání klientských knihoven pro datová úložiště

Tab. 4.6: Abstraktní třídy v návrhu nové verze VTApi

4.3 Koncept nové verze VTApi

Návrh pro konektory dvou různých datových úložišť ilustruje obr. 4.3. Pro přehlednost je množství z pohledu návrhu nové verze nepodstatných částí odstraněno, zejména třídy dědící funkčnost *KeyValues* (tedy *Dataset*, *Sequence*, *Interval*, *Method* a *Process*). Nově navržené třídy mají následující názvy:

- *factory* třída je pojmenována **BackendFactory**
- pojmenování abstraktních tříd (rozhraní) viz tab. 4.6
- názvy realizujících tříd se skládají z identifikátoru úložiště (zde **PG** nebo **SL**) a názvu realizované abstraktní třídy (rozhraní), např. **PGResultSet**



Obr. 4.3: Koncept nové verze VTApi

5 Implementace

Knihovna VTApi je vyvíjena v programovacím jazyce C++ s konfigurovatelným využitím externích knihoven pro rozšířenou funkčnost. Jedná se o klientské knihovny datových úložišť, tedy *libpq* (SŘBD PostgreSQL v9.1 [25]), včetně podpůrné knihovny *libpqtypes* v1.7 [26] pro manipulaci s datovými typy, a nově také *libsqlite3* (databázový engine SQLite v3.17 [27]). Pro manipulaci s geometrickými datovými typy slouží integrace podpory pro knihovnu GEOS v3.2.3 [28] a databázové rozšíření PostGIS [29] (bez závislosti na klientské knihovně). Dále VTApi poskytuje podporu pro analytické procesy využívající knihovnu OpenCV v2.4 [24]. K testování výkonnosti VTApi na Unixových operačních systémech lze využít funkcionalitu z knihovny *libproc*. Generování programové dokumentace zajišťuje nástroj *doxygen*.

Vývoj VTApi probíhá ve vývojovém prostředí NetBeans 6.9.1 s využitím otevřeného úložiště verzovacího systému Git s webovým rozhraním na adrese http://gitorious.org/vtapi. K projektu je vedena webová stránka http://vidte.fit.vutbr.cz/vtapi.html se základním rozcestníkem na informace o konceptu, instalaci a použití VTApi a dokumentaci s referenčním manuálem.

5.1 Implementace konceptu nové verze

Při práci na nové verzi jsem vycházel ze zdrojového kódu VTApi verze 1.5 vydané v únoru 2013, na které jsem se z velké části podílel v rámci projektu *VideoTeror* [1]. V rámci vývoje programové části této diplomové práce jsem provedl poměrně výrazné změny v návrhu interní části knihovny, které se však z pohledu uživatele API projeví minimálně. Zachoval jsem tak základní principy, se kterými bylo VTApi doposud vyvíjeno:

Jednoduchá reprezentace a správa obrazových dat

Tato je nadále zajištěna pomocí hierarchické struktury instancí objektů *Dataset* – *Sequence* – *Interval* (viz obr. 4.2). Jejich implementaci i rozhraní jsem ponechal beze změny.

Důraz na optimalizaci procházení velkého množství vícedimenzionálních dat

Princip funkce *next()* zajišťující procházení dat jednoduchým posouváním indexu na načtená data (*result set*) jsem zachoval. Rozhraní poskytující přístup k těmto datům (sada funkcí *getX*) jsem ale duplikoval do abstraktní třídy **ResultSet**, jeho implementace je totiž závislá na typu datového úložiště. Dosavadní rozhraní (ve třídě *KeyValues*, tedy základní datové třídy – viz obr. 4.2) je zachováno jako pouhá obálka pro přístup k funkcím interní instance třídy *ResultSet*. Toto řešení při optimalizované kompilaci nevyústí ke zvýšení výpočetní náročnosti datové iterace. Abstraktní třídu *ResultSet* je zapotřebí realizovat pro různé typy datových úložišť.

Intuitivní rozhraní zjednodušující složité databázové operace

Programovací rozhraní jsem zachoval v dosavadní podobě. Z uživatelského pohledu je nadále nutno znát základní sadu funkcí:

- konstruktory tříd reprezentujících data
- sady funkcí `getX` (získání hodnot), `addX` (vkládání) a `setX` (aktualizace)
- funkci `next()` pro iteraci přes záznamy

Framework VTApi zajistí konzistenci datového úložiště a komunikaci s ním (třída **Connection**), typování hodnot (třída **TypeManager**), konstrukci databázových dotazů (třída **QueryBuilder**) a provádění dalších nutných dílčích úkonů.

Snadná konfigurovatelnost

Konfigurační soubor (viz kap. 3.4) jsem rozšířil o možnosti související s volbou typu datového úložiště. Obsahuje nově následující dva parametry z tab. 5.1 týkající se nastavení SQLite:

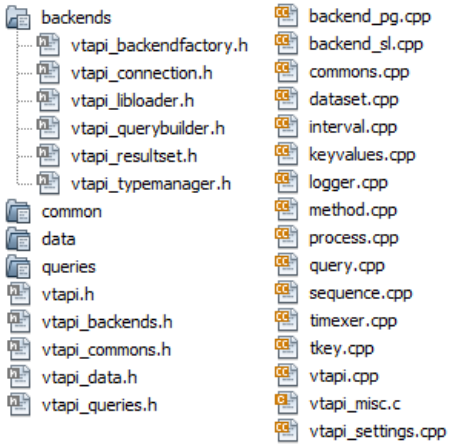
Parametr	Zkratka	Implicitní hodnota	Popis
backend	b	-	Typ datového úložiště [postgres|sqlite]
dbfolder	d	./sqlite	Umístění adresáře s databázovými soubory SQLite

Tab. 5.1: Objekty vytvářené pomocí *factory* datových úložišť

V případě volby SQLite pozbývá významu možnost *connection*, definice umístění databáze se určí pomocí *dbfolder*. Do ostatních původních možností konfigurace jsem nezasahoval.

5.1.1 Souborová struktura

V návrhu nové verze VTApi frameworku předpokládám možnou budoucí implementaci pro další typy datových úložišť. Z tohoto důvodu jsem přepracoval vnitřní strukturu hlavičkových a zdrojových souborů a jasně oddělil logiku frameworku od databázově závislé funkcionality. Souborovou strukturu zobrazuje obr. 5.1:



Obr. 5.1: Souborová struktura nové verze VTApi

Hlavičkové soubory jsem organizoval následovně:

`vtapi.h` - veřejný hlavičkový soubor

`backends/` - adresář s hlavičkami abstraktních a realizujících tříd pro různá datová úložiště

`common/` - adresář s interními hlavičkami sdílených základních tříd

`data/` - adresář s interními hlavičkami tříd reprezentujících data

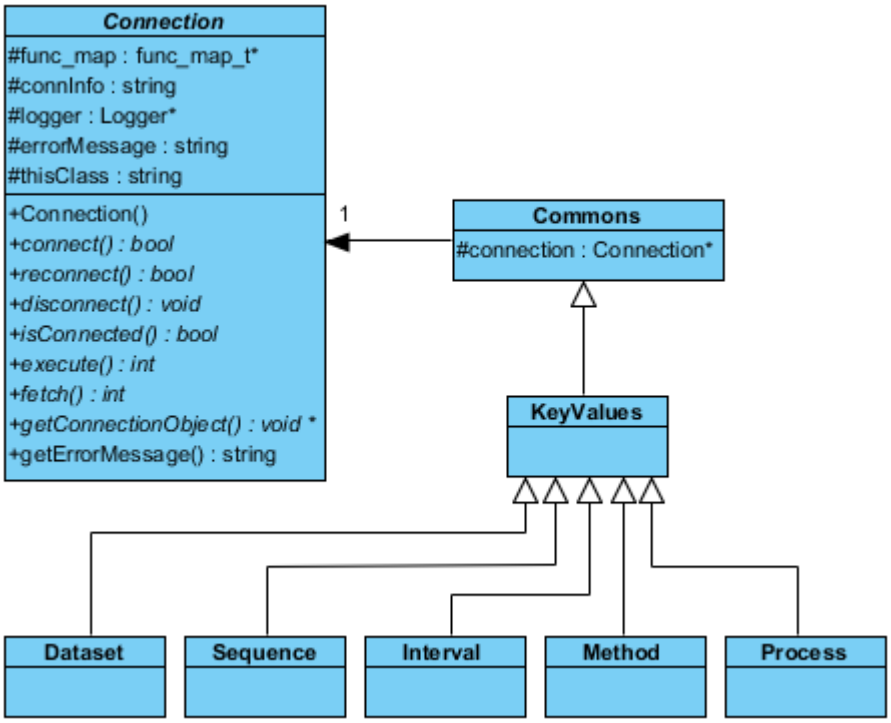
`queries/` - adresář s interními hlavičkami tříd reprezentující databázové dotazy / příkazy

Soubory `vtapi_backends.h`, `vtapi_commons.h`, `vtapi_data.h` a `vtapi_queries.h` pouze reprezentují obsahy jim odpovídajících adresářů.

Funkcionalitu závislou na typu datového úložiště tedy obsahují pouze hlavičkové soubory adresáře `backends`. Jim příslušně zdrojové kódy jsem umístil do souboru `backend_X.cpp`, kde X je zkratka pro datové úložiště.

5.1.2 Třída Connection

Správu připojení k databázi a provádění dotazů a příkazů nově zajišťuje třída **Connection**, která nahrazuje třídu *Connector* z původní verze. Stejně jako třída *TypeManager* má její instance v návrhu API globální úlohu jako objekt sdílený všemi datovými třídami prostřednictvím základní třídy *Commons*. Toto sdílení je naznačeno na zjednodušeném diagramu tříd na obr. 5.2 (podrobnější návrh zobrazených tříd ilustrují obr. 4.1 až 4.3).



Obr. 5.2: Umístění třídy *Connection* v nové verzi VTApi

Instance třídy *Connection* se inicializuje při vytvoření instance hlavní třídy *VTApi*. Požaduje inicializační parametry `func_map`, `connInfo` a `logger`, atributy třídy popisuje následující tabulka:

`func_map`	seznam načtených funkcí z klientské knihovny datového úložiště pomocí instance třídy *LibLoader*
`connInfo`	textový řetězec s informacemi potřebnými k připojení k databázi
`logger`	reference na instanci objektu třídy *Logger*, který zajišťuje textový výstup pro účely debugování, varování a chybových hlášení
`errorMessage`	textový řetězec poslední chyby při provádění příkazu / dotazu
`thisClass`	textová identifikace třídy pro účely debugování

Tab. 5.2: Atributy třídy *Connection*

Rozhraní, které definuje třída *Connection* popisuje tab. 5.3. Jeho implementace je vyžadována po odvozených třídách.

`bool connect (const string& connectionInfo)`	
▪ provede připojení k databázi na základě informace z parametru `connectionInfo`	**parametry** • *connectionInfo*: informace k připojení k databázi **návratová hodnota** • *bool*: ukazatel úspěchu
`bool reconnect (const string& connectionInfo)`	
▪ provede odpojení a připojení k databázi na základě informace z parametru `conectionInfo`	**parametry** • *connectionInfo*: informace k připojení k databázi **návratová hodnota** • *bool*: ukazatel úspěchu
`void disconnect ()`	
▪ provede odpojení od databáze	**parametry** • žádné **návratová hodnota** • žádná
`bool isConnected ()`	
▪ provede ověření, zda je připojení k databázi aktivní	**parametry** • žádné **návratová hodnota** • *bool*: ukazatel úspěchu

`int execute (const string& query, void *param)`	
▪ provede SQL příkaz bez očekávání souborů výsledků (*result setu*)	**parametry** • *query*: textový řetězec SQL příkazu • *param*: ukazatel na data parametrů příkazu **návratová hodnota** • *int*: počet ovlivněných řádků v případě úspěchu, při chybě < 0
`int fetch (const string& query, void *param, ResultSet *resultSet)`	
▪ provede SQL dotaz a získá soubor výsledků (*result set*)	**parametry** • *query*: textový řetězec SQL dotazu • *param*: ukazatel na data parametrů příkazu • *resultSet*: reference na objekt souboru výsledků, který se naplní v případě úspěšného provedení příkazu **návratová hodnota** • *int*: počet vrácených řádků v souboru výsledků, při chybě < 0
`void *getConnectionObject()`	
▪ získá referenci na objekt připojení k databázi	**parametry** • žádné **návratová hodnota** • *void **: reference na objekt připojení

Tab. 5.2: Rozhraní třídy *Connection*

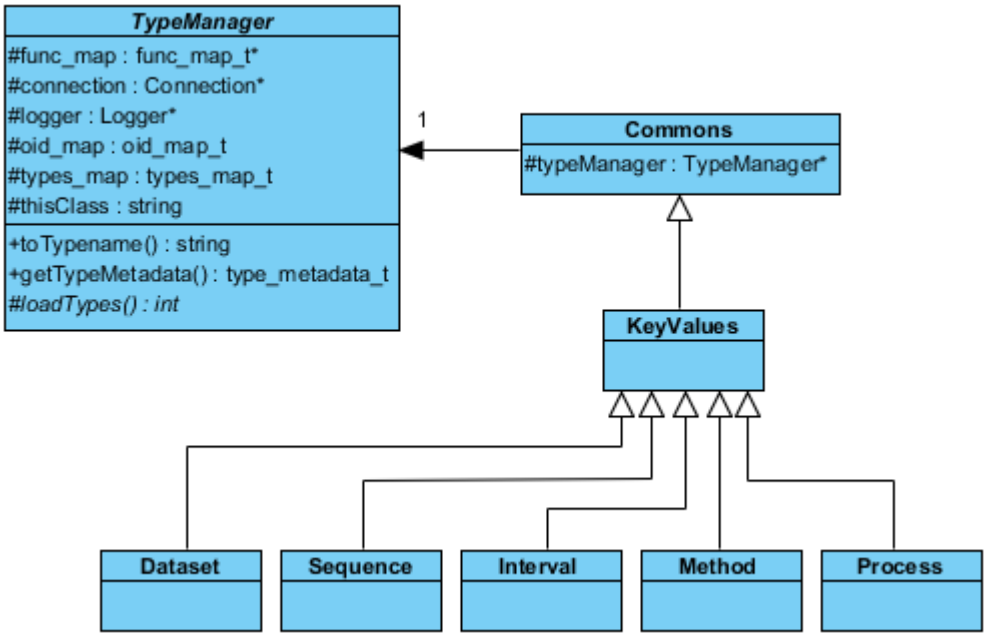
Funkce získání chybového hlášení je implementována v třídě *Connection*:

`string getErrorMessage ()`	
▪ vrátí chybové hlášení příslušející poslednímu provedenému příkazu / dotazu	**parametry** • žádné **návratová hodnota** • *string*: chybové hlášení

Tab. 5.3: Realizované metody třídy *Connection*

5.1.3 Třída TypeManager

Správu a manipulaci datových typů pro účely VTApi nově zaobaluje třída **TypeManager**, která nahrazuje třídu *TypeMap* z původní verze. Stejně jako třída *Connection* má její instance v návrhu API globální úlohu jako objekt sdílený všemi datovými třídami prostřednictvím základní třídy *Commons*. Toto sdílení je naznačeno na zjednodušeném diagramu tříd na obr. 5.3 (podrobnější návrh zobrazených tříd ilustrují obr. 4.1 až 4.3).



Obr. 5.3: Umístění třídy *TypeManager* v nové verzi VTApi

Instance třídy *TypeManager* se inicializuje při vytvoření instance hlavní třídy *VTApi*. Požaduje inicializační parametry `func_map`, `connection` a `logger`, atributy třídy popisuje tab. 5.4:

`func_map`	seznam načtených funkcí z klientské knihovny datového úložiště pomocí instance třídy *LibLoader*
`connection`	reference na instanci objektu realizovaného rozhraní *Connection*
`logger`	reference na instanci objektu třídy *Logger*, který zajišťuje textový výstup pro účely debugování, varování a chybových hlášení
`oid_map`	seznam pro účely mapování typových identifikátorů OID na názvy datových typů
`types_map`	seznam pro účely mapování názvů datových typů na jejich metadata (kategorie, délka, informace o prvcích pole)
`thisClass`	textová identifikace třídy pro účely debugování

Tab. 5.4: Atributy třídy *TypeManager*

Rozhraní, které definuje třída *TypeManager* popisuje tab. 5.5. Jedná se pouze o funkci zajišťující načtení typů.

`int loadTypes ()`	
▪ provede načtení databázových typů, případně připraví uživatelem definované typy k použití (u knihovny *libpqtypes* tzv. registrace)	**parametry** • žádné **návratová hodnota** • *int*: počet načtených typů, < 0 v případě chyby

Tab. 5.5: Rozhraní třídy *TypeManager*

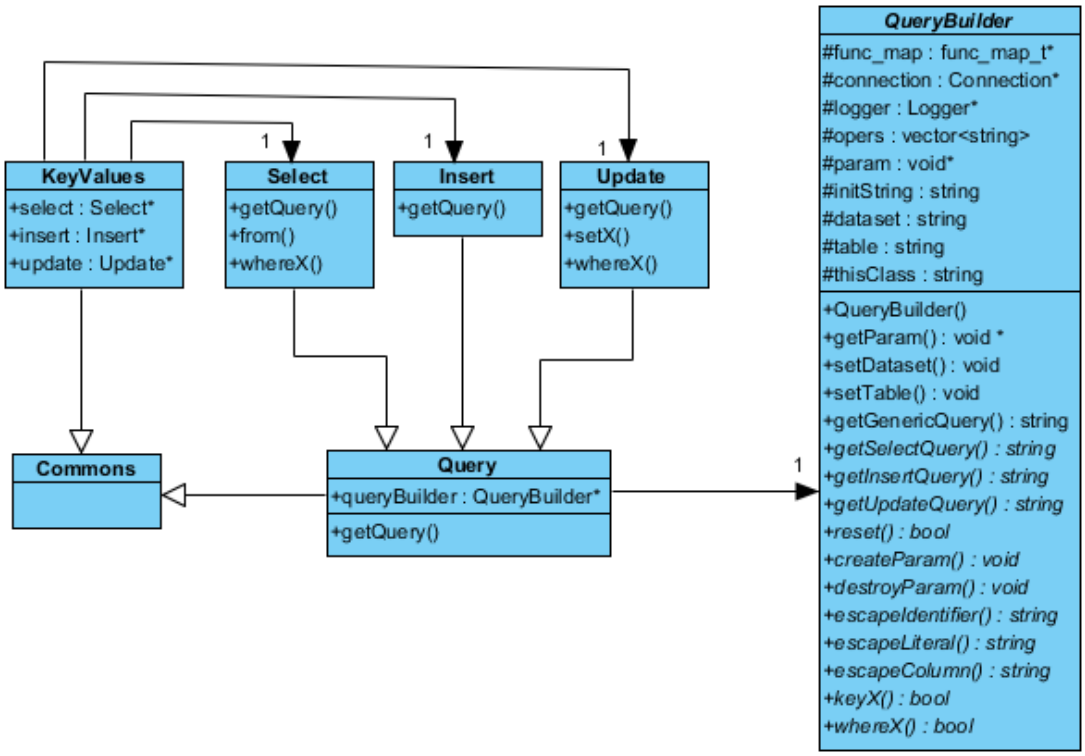
Následující metody jsem implementoval přímo v třídě *TypeManager*:

`string toTypename (int oid)`	
▪ převede databázový identifikátor OID na název datového typu	**parametry** • *oid*: OID datového typu **návratová hodnota** • *string*: název datového typu
`type_metadata_t getTypeMetadata (const string& name)`	
▪ vrátí metadata o datovém typu se jménem `name` (kategorie, délka, informace o prvcích pole)	**parametry** • *name*: název datového typu **návratová hodnota** • *type_metadata_t*: struktura metadat datového typu

Tab. 5.6: Realizované metody třídy *TypeManager*

5.1.4 Třída QueryBuilder

Konstrukci SQL dotazů a příkazů nově zaobaluje třída **QueryBuilder**. Navrhl a implementoval jsem ji s ohledem na možnou nejednotnou podporu syntaxe jazyka SQL různými datovými úložišti. Podporuje také možnost vytváření parametrizovaných dotazů. Stejně jako třída *ResultSet* přísluší třída *QueryBuilder* ke třídě *Query* reprezentující databázový příkaz nebo dotaz. Třída *Query* může být instanciována samostatně jako tzv. *generic query* (s uživatelsky vloženým SQL dotazem) či jako základní třída objektů *Select*, *Insert* nebo *Update*. Tuto asociaci znázorňuje ve zjednodušeném diagramu tříd obr. 5.4 (podrobnější návrh zobrazených tříd ilustrují obr. 4.1 až 4.3).



Obr. 5.4: Umístění třídy *QueryBuilder* v nové verzi VTApi

Princip tvorby SQL příkazů

Samotnou tvorbu SQL řetězců reprezentující databázové příkazy obstarávají funkce `getGenericQuery`, `getSelectQuery`, `getInsertQuery` a `getUpdateQuery`. Tyto metody při své invokaci využijí předchozích uživatelských (či automatických) vstupů z nějakých funkcí ze sady `keyX` (tab. 5.9) a `whereX` (tab. 5.10) následujícím způsobem:

- funkce `keyX` (např. `keyInt` pro celá čísla) ukládají záznamy klíč (sloupec)-hodnota pro hlavní část SQL dotazu (SELECT - FROM, INSERT INTO - VALUES, UPDATE - SET)
- funkce `whereX` (např. `whereString` pro řetězec) ukládají záznamy klíč (sloupec)-hodnota pro klauzuli WHERE dotazů SELECT a příkazů UPDATE
- je-li potřeba, v seznamu `opers` (viz tab. 5.7) jsou uloženy operátory vkládané mezi klíč a hodnotu

Dále se ke konstrukci může využít vlastním způsobem definovaná struktura `param` obalující dodatečné informace ke zkonstruovanému dotazu. To je výhodné zejména pro vytváření parametrizovaných dotazů.

Další klauzule dotazu SELECT jsou zkonstruovány dle parametrů metody `getSelectQuery` (viz tab 5.8).

Parametry `dataset` a `table` obsahují implicitní hodnoty užívaného datasetu (databázového schématu) a tabulky, které se užijí, nezadá-li tyto hodnoty uživatel explicitně.

Instance třídy *QueryBuilder* se inicializuje při vytvoření instance třídy *Query, Select, Insert* nebo *Update*. Požaduje inicializační parametry `func_map`, `connection`, `logger` a volitelně `initString`, atributy třídy popisuje tab. 5.7:

`func_map`	seznam načtených funkcí z klientské knihovny datového úložiště pomocí instance třídy *LibLoader*
`connection`	reference na instanci objektu realizovaného rozhraní *Connection*
`logger`	reference na instanci objektu třídy *Logger*, který zajišťuje textový výstup pro účely debugování, varování a chybových hlášení
`opers`	seznam operátorů pro účely konstrukce klauzule WHERE
`param`	obecná reference na data pro účely parametrizovaných dotazů
`initString`	inicializační řetězec zadaný uživatelem (může být celý SQL dotaz)
`dataset`	název datasetu, nad kterým se dotaz / příkaz bude provádět
`table`	název implicitní tabulky, bude použita, nebude-li zadána jinak
`thisClass`	textová identifikace třídy pro účely debugování

Tab. 5.7: Atributy třídy *QueryBuilder*

Rozhraní, které definuje třída *QueryBuilder* popisuje tab. 5.8. Jeho implementace je vyžadována po odvozených třídách.

`getSelectQuery (const string& groupby, const string orderby&, const int limit, const int offset)`	
▪ zkonstruuje dotaz SELECT	**parametry** • *groupby*: hodnota pro klauzuli GROUPBY • *orderby*: hodnota pro klauzuli ORDERBY • *limit*: hodnota pro klauzuli LIMIT • *offset*: hodnota pro klauzuli OFFSET **návratová hodnota** • *string*: textový řetězec SQL dotazu
`getInsertQuery ()`	
▪ zkonstruuje příkaz INSERT	**parametry** • žádné **návratová hodnota** • *string*: textový řetězec SQL dotazu
`getUpdateQuery ()`	
▪ zkonstruuje příkaz UPDATE	**parametry** • žádné **návratová hodnota** • *string*: textový řetězec SQL dotazu
`reset ()`	
▪ vymaže všechny záznamy klíč-hodnota vložené příkazy `keyX` a `whereX`	**parametry** • žádné **návratová hodnota** • *bool*: ukazatel úspěchu
`createParam ()`	
▪ vytvoří strukturu pro parametrizované dotazy	**parametry** • žádné **návratová hodnota** • žádná
`destroyParam ()`	
▪ zničí strukturu pro parametrizované dotazy	**parametry** • žádné **návratová hodnota** • žádná
`escapeIdentifier (const string& identifier)`	
▪ zabezpečí identifikátor pro použití v SQL dotazu	**parametry** • *identifier*: identifikátor (např. tabulky) **návratová hodnota** • *string*: zabezpečený identifikátor

`escapeLiteral (const string& literal)`	
▪ zabezpečí literál pro použití v SQL dotazu	**parametry** • *literal*: literál **návratová hodnota** • *string*: zabezpečený literál
`escapeColumn (const string& key, const string& table)`	
▪ zabezpečí identifikátor sloupce tabulky pro použití v SQL dotazu, včetně případných agregačních funkcí	**parametry** • *key*: sloupec tabulky • *table*: název tabulky **návratová hodnota** • *string*: zabezpečený identifikátor sloupce
`keyX (const string& key, X value, const int size, const string& from)`	
▪ sada funkcí pro vkládání klíčů (a hodnot) pro hlavní část SQL dotazu (SELECT FROM, INSERT INTO a UPDATE SET) – viz tab. 5.9	**parametry** • *key*: sloupec tabulky • *value*: hodnota pole tabulky • *size:* velikost pole (volitelné) • *from:* název odpovídající tabulky **návratová hodnota** • *bool*: ukazatel úspěchu
`whereX (const string& key, X value, const string& oper, const string& from)`	
▪ sada funkcí pro vkládání klíčů (a hodnot) pro klauzuli WHERE – viz tab. 5.10	**parametry** • *key*: sloupec tabulky • *value*: hodnota pole tabulky • *oper:* operátor mezi klíčem a hodnotou • *from:* název odpovídající tabulky **návratová hodnota** • *bool*: ukazatel úspěchu

Tab. 5.8: Rozhraní třídy *QueryBuilder*

`keyInt`	32bit číselná hodnota *int*	`keyFloat`	64bit reálná hodnota *float*
`keyIntA`	pole hodnot *int*	`keyFloatA`	pole hodnot *float*
`keyString`	textový řetězec	`keySeqtype`	výčtový typ typů sekvencí
`keyStringA`	pole textových řetězců	`keyInouttype`	výčtový typ typů vstup/výstup

Tab. 5.9: Sada funkcí `keyX` třídy *QueryBuilder*

`whereInt`	32bit číselná hodnota *int*	`whereFloat`	32bit reálná hodnota *float*
`whereString`	textový řetězec	`keySeqtype`	výčtový typ typů sekvencí
`whereInouttype`	výčtový typ vstup/výstup	`whereTimestamp`	časové razítko

Tab. 5.10: Sada funkcí `whereX` třídy *QueryBuilder*

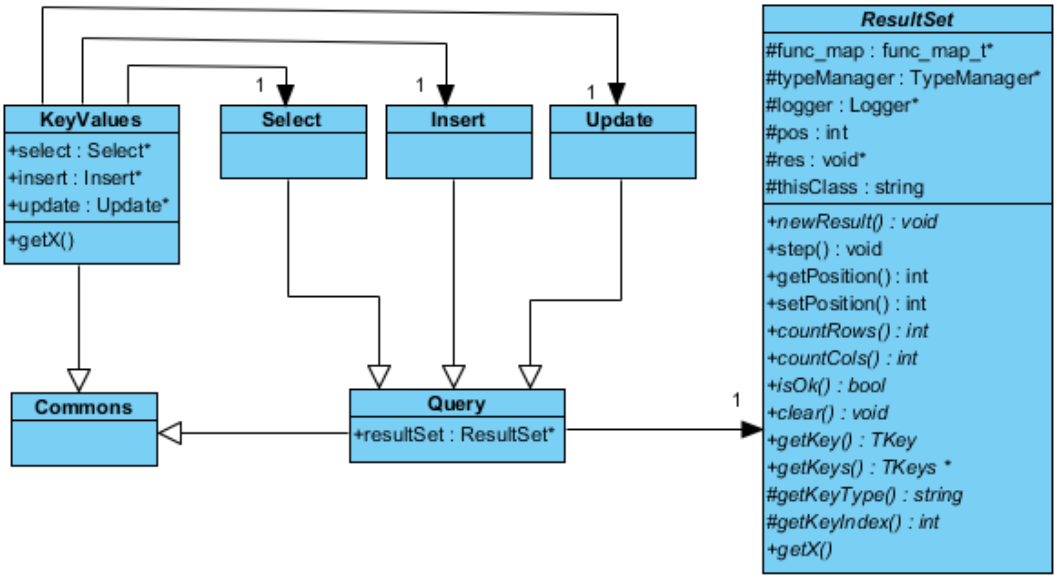
Následující metody z tab. 5.11 jsou implementovány v třídě *QueryBuilder*:

`getParam ()`	
▪ získá referenci na strukturu pro parametrizované dotazy	**parametry** • žádné **návratová hodnota** • *void **: obecný ukazatel na strukturu *param*
`getGenericQuery ()`	
▪ zkonstruuje obecný SQL příkaz z uživatelem zadaného textového řetězce	**parametry** • žádné **návratová hodnota** • *string*: textový řetězec SQL dotazu
`getUpdateQuery ()`	
▪ zkonstruuje příkaz UPDATE	**parametry** • žádné **návratová hodnota** • *string*: textový řetězec SQL dotazu
`reset ()`	
▪ vymaže všechny záznamy klíč-hodnota vložené příkazy `keyX` a `whereX`	**parametry** • žádné **návratová hodnota** • *bool*: ukazatel úspěchu

Tab. 5.11: Realizované metody třídy *QueryBuilder*

5.1.5 Třída ResultSet

Rozhraní pro přístup k načteným datům nově definuje třída *ResultSet*. Samotné provádění dotazů a načítání dat zajišťuje třída *Connection* (metoda `fetch()` – viz tab. 5.2). Stejně jako třída *QueryBuilder* přísluší třída *ResultSet* ke třídě *Query* reprezentující databázový příkaz nebo dotaz. Tuto asociaci znázorňuje ve zjednodušeném diagramu tříd obr. 5.5 (podrobnější návrh zobrazených tříd ilustrují obr. 4.1 až 4.3).



Obr. 5.5: Umístění třídy *ResultSet* v nové verzi VTApi

Instance třídy *ResultSet* se inicializuje při vytvoření instance třídy *Query* nebo *Select*. Požaduje inicializační parametry `func_map`, `typeManager` a `logger`, atributy třídy popisuje následující tabulka:

`func_map`	seznam načtených funkcí z klientské knihovny datového úložiště pomocí instance třídy *LibLoader*
`typeManager`	reference na instanci objektu realizovaného rozhraní *TypeManager*
`logger`	reference na instanci objektu třídy *Logger*, který zajišťuje textový výstup pro účely debugování, varování a chybových hlášení
`pos`	pozice aktuálního řádku v souboru výsledků
`res`	reference na objekt souboru výsledků
`thisClass`	textová identifikace třídy pro účely debugování

Tab. 5.12: Atributy třídy *ResultSet*

Rozhraní, které definuje třída *ResultSet* popisuje tab. 5.13. Jeho implementace je vyžadována po odvozených třídách.

`newResult (void *res)`	
▪ přiřadí nový objekt souboru výsledků	**parametry** • *res*: reference na nový objekt souboru výsledků **návratová hodnota** • žádná
`countRows ()`	
▪ vrátí počet řádků souboru výsledků	**parametry** • žádné **návratová hodnota** • *int*: počet řádků

`countCols ()`	
▪ vrátí počet sloupců souboru výsledků	**parametry** • žádné **návratová hodnota** • *int*: počet sloupců
`isOk ()`	
▪ ověří, zda existuje správně vytvořený objekt souboru výsledků	**parametry** • žádné **návratová hodnota** • *bool*: ukazatel úspěchu
`clear ()`	
▪ zruší objekt souboru výsledků	**parametry** • žádné **návratová hodnota** • žádná
`getKey (int col)`	
▪ získá metadata sloupce dle jeho indexu	**parametry** • *col*: index sloupce **návratová hodnota** • *TKey*: struktura s metadaty sloupce
`getKeys ()`	
▪ získá seznam metadat všech sloupců souboru výsledků	**parametry** • žádné **návratová hodnota** • *TKeys**: seznam metadat sloupců
`getKeyType (const int col)`	
▪ získá datový typ sloupce	**parametry** • *col*: index sloupce **návratová hodnota** • *string*: datový typ sloupce
`getKeyIndex (const string& key)`	
▪ získá index sloupce dle jeho názvu	**parametry** • *string*: název sloupce **návratová hodnota** • *int*: index sloupce
`getX (const int col) ; getX (const string& key)`	
▪ sada funkcí pro získání hodnot polí ze souboru výsledků – viz tab. 5.14	**parametry** • *col*: index sloupce tabulky • *key*: název sloupce tabulky **návratová hodnota** • *<TYP>*: hodnota v poli tabulky

Tab. 5.13: Rozhraní třídy *ResultSet*

getValue	Získání hodnoty **jakéhokoliv** typu převedeného na textovou reprezentaci		
Číselné typy			
getInt	32bit číselná hodnota *int*	getFloat	64bit reálná hodnota *float*
getIntA	pole hodnot *int*	getFloatA	pole hodnot *float*
getIntV	standardní vektor hodnot *int*	getFloatV	standardní vektor hodnot *float*
getIntVV	libovolně rozměrný vektor hodnot *int*	getFloatVV	libovolně rozměrný vektor hodnot *float*
getInt8	64bit číselná hodnota *long*	getFloat8	64bit reálná hodnot *double*
Textové typy			
getChar	znak	getString	textový řetězec
getCharA	pole znaků		
OpenCV typy			
getCvmat	OpenCV matice *cvMat*	getCvMatND	OpenCV matice *cvMatND*
Ostatní typy			
getTimestamp	časové razítko	getIntOid	databázový identifikátor *OID*

Tab. 5.14: Sada funkcí `getX` třídy *ResultSet* a jejich návratové hodnoty

Následující metody jsou implementovány v třídě *ResultSet*:

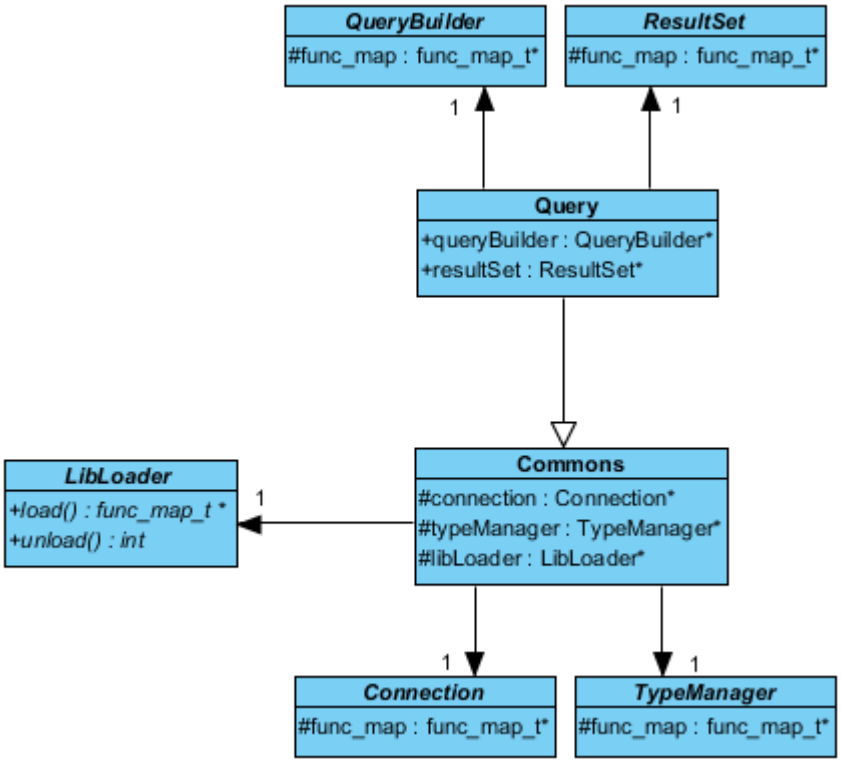
step ()	
▪ posune index aktuálního řádku v souboru výsledků	**parametry** • žádné **návratová hodnota** • žádná
getPosition ()	
▪ získá index aktuálního řádku v souboru výsledků	**parametry** • žádné **návratová hodnota** • *int*: index aktuálního řádku v souboru výsledků
setPosition (const int pos)	
▪ nastaví index aktuálního řádku v s ouboru výsledků	**parametry** • *pos*: index řádku **návratová hodnota** • žádná

Tab. 5.15: Realizované metody třídy *QueryBuilder*

5.1.6 Třída LibLoader

Dynamické nahrávání klientských knihoven datových úložišť (a dalších potřebných) zajišťuje realizace rozhraní *LibLoader*. Nahrání adres požadovaných funkcí z příslušné knihovny do interní struktury `func_map` proběhne při inicializaci základní třídy *Commons*. Struktura `func_map` je dále využívána všemi třídami závislými na typu datového úložiště. Umístění a užití třídy *LibLoader* je

zobrazeno ve zjednodušeném diagramu tříd na obr. 5.6 (podrobnější návrh zobrazených tříd ilustrují obr. 4.1 až 4.3).



Obr. 5.6: Umístění třídy *LibLoader* v nové verzi VTApi

Rozhraní, které definuje třída *LibLoader* obsahuje pouze metody pro nahrání/zavření knihoven a načtení knihovních funkcí do struktury `func_map`. Jsou popsány v následující tabulce:

`load ()`	
▪ dynamicky nahraje knihovny, vytvoří strukturu `func_map` a naplní ji adresami na požadované metody	**parametry** • žádné **návratová hodnota** • *func_map_t* *: ukazatel na nově vytvořenou a naplněnou strukturu *func_map*
`unload ()`	
▪ zavře dynamicky nahrané knihovny	**parametry** • žádné **návratová hodnota** • *int*: 0 v případě úspěchu, < 0 při chybě

Tab. 5.16: Rozhraní třídy *LibLoader*

5.2 Implementace podpory pro datová úložiště

V rámci této diplomové práce jsem implementoval konektory dvou datových úložišť – SŘBD **PostgreSQL 9.1** a databázového engine **SQLite3**. VTApi bylo původně navrženo primárně pro použití s databázovým back-endem PostgreSQL, který byl zvolen zejména pro svou otevřenost, výkonnost a rozsáhlou funkcionalitu. Tuto základní charakteristiku jsem zachoval a z větší části pouze zapracoval její logiku do konceptu nového návrhu. Nově jsem implementoval konektor pro bez-serverové souborové datové úložiště SQLite, které je vhodné zejména pro testování, off-line vývoj a přenositelnost aplikací užívajících framework VTApi.

5.2.1 PostgreSQL

Databázový model pro PostgreSQL zůstal nezměněn z původního návrhu, ilustruje jej obr. 3.2. K reprezentaci dat příslušející jednomu datasetu je vyhrazeno databázové schéma s názvem odpovídajícím názvu datasetu. Sdílené informace (o samotných datasetech či metodách) obsahuje hlavní databázové schéma *public*. Pro využití plné funkcionality VTApi by databáze měla obsahovat rozšíření PostGIS pro správu složitějších geometrických typů, dále datový typ OpenCV matice `cvMat`, eventuálně datový typ `cube` reprezentující n-dimenzionální kostku.

K instalaci základní funkční kostry databázového modelu VTApi jsem poskytl inicializační SQL skripty v adresáři `script/`.

Databáze

Skript **`create_public_pg.sql`** vytvoří schéma `public` včetně výčtových datových typů, seznamu datasetů a tabulek pro budoucí využití v metodologii VTApi.

Výčtové datové typy `inouttype` a `seqtype`:

```
-- výčtové datové typy
CREATE TYPE inouttype AS ENUM ('in', 'inout', 'out');
CREATE TYPE seqtype   AS ENUM ('video', 'images', 'data');
```

Tabulka `datasets` obsahuje seznam a fyzické umístění datasetů:

```
-- tabulka se seznamem datasetů
CREATE TABLE datasets (
    dsname name NOT NULL,
    dslocation character varying,
    userid name,
    groupid name,
    created timestamp without time zone DEFAULT now(),
    changed timestamp without time zone,
    notes text,
    CONSTRAINT dataset_pk PRIMARY KEY (dsname)
);
```

Tabulka `methods` bude v metodologii obsahovat seznam dostupných metod:

```
-- tabulka s dostupnými metodami
CREATE TABLE methods (
    mtname name NOT NULL,
    userid name,
    created timestamp without time zone DEFAULT now(),
    notes text,
    CONSTRAINT methods_pk PRIMARY KEY (mtname)
);
```

Tabulka `methods_keys` bude v metodologii obsahovat seznam parametrů pro jednu metodu:

```
-- tabulka s parametry metod
CREATE TABLE methods_keys (
    mtname name NOT NULL,
    keyname name NOT NULL,
    typname regtype NOT NULL,
    "inout" inouttype NOT NULL,
    CONSTRAINT methods_keys_pk PRIMARY KEY (mtname, keyname),
    CONSTRAINT methods_keys_mtname_fkey FOREIGN KEY (mtname)
      REFERENCES methods(mtname)
);
```

Tabulka `processes` bude v metodologii obsahovat seznam dostupných procesů:

```
-- tabulka s definovanými procesy
CREATE TABLE processes (
    mtname name,
    prsname name NOT NULL,
    inputs regclass,
    outputs regclass,
    userid name,
    created timestamp without time zone,
    notes text,
    CONSTRAINT processes_pk PRIMARY KEY (prsname),
    CONSTRAINT method_fk FOREIGN KEY (mtname)
      REFERENCES methods(mtname) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Tabulka `selections` obsahuje seznam selekcí – podmnožin dat pro jednotlivé úlohy:

```
-- tabulka selekcí
CREATE TABLE selections (
    selname regclass NOT NULL,
    dataset character varying NOT NULL,
    userid name,
    created timestamp without time zone DEFAULT now(),
    notes text,
    CONSTRAINT selections_pk PRIMARY KEY (selname)
);
```

Skript **`create_dataset_pg.sql`** vytvoří schéma databáze odpovídající jednomu datasetu (zde s názvem *vidte*). Před jeho použitím je třeba modifikovat název vytvořeného datasetu, příp. manuálně definovat tabulky/sloupce tabulky `intervals` (ta může mít i jiný název) pro vlastní vstupní / výstupní hodnoty.

Vytvoření schématu s názvem datasetu:

```
--vytvoření schématu
CREATE schema vidte;
```

Tabulka `sequences` obsahuje seznam sekvencí jednoho datasetu:

```
-- tabulka sekvencí
CREATE TABLE sequences (
    seqname name NOT NULL,
    seqnum integer,
    seqlocation character varying,
    seqtyp public.seqtype DEFAULT 'data',
    userid name,
    groupid name,
    created timestamp without time zone,
    changed timestamp without time zone,
    notes text,
    CONSTRAINT sequences_pk PRIMARY KEY (seqname),
    CONSTRAINT sequences_unq UNIQUE (seqnum)
);
```

Tabulka `intervals` (volitelný název) obsahuje seznam intervalů jedné sekvence:

```
-- tabulka intervalů
CREATE TABLE intervals (
    seqname character varying NOT NULL,
    t1 integer NOT NULL,
    t2 integer NOT NULL,
    imglocation character varying,
    userid name,
    created timestamp without time zone DEFAULT now(),
    notes text,
    CONSTRAINT intervals_pk PRIMARY KEY (seqname, t1, t2),
    CONSTRAINT sequences_fk FOREIGN KEY (seqname)
      REFERENCES sequences(seqname) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Vložení nového datasetu do seznamu datasetů:

```
-- vložení do seznamu datasetů
INSERT INTO public.datasets(dsname, dslocation)
  VALUES ('vidte', 'data/vidte/');
```

Vložení tabulky `intervals` jako možné selection:

```
-- definice tabulky intervals jako selection
INSERT INTO public.selections(selname, dataset)
  VALUES ('vidte.intervals', 'vidte');
```

Podpora datových typů

Konektor pro PostgreSQL podporuje základní datové typy z tab. 5.15:

Název	Alias	Popis
`bigint`	`int8`	8-bytové celé číslo se znaménkem
`character varying`	`varchar`	textový řetězec s proměnnou délkou

character	char	znak
double precision	float8	8-bytové desetinné číslo se znaménkem
integer	int, int4	4-bytové celé číslo se znaménkem
real	float4	4-bytové desetinné číslo se znaménkem
smallint	int2	2-bytové celé číslo se znaménkem
text	-	textový řetězec
timestamp	-	časové razítko (bez časové zóny)

Tab. 5.15: Základní datové typy podporované konektorem PostgreSQL

Zpracování obrazových dat vyžaduje podporu pro vícedimenzionální pole hodnot. Implementoval jsem podporu pro základní 2-dimenzionální datové typy z tab. 5.16.

Název	Popis
int4[]	pole 4-bytových celých čísel
int8[]	pole 8-bytových celých čisel
float4[]	pole 4-bytových desetinných čísel
float8[]	pole 8-bytových celých čísel

Tab. 5.16: 2-dimenzionální datové typy podporované konektorem PostgreSQL

Pro podporu n-dimenzionálních dat lze využít podpory pro nestandardní datové typy z tab. 5.17:

Název	Popis
cube	n-dimenzionální kostka • obsažen v oficiálních rozšířeních PostgreSQL jako instalovatelný modul
cvmat	OpenCV matice • kompozitní datový typ

Tab. 5.17: n-dimenzionální datové typy podporované konektorem PostgreSQL

Typ `cvmat` lze v databázi vytvořit následujícím příkazem CREATE TYPE:

```
CREATE TYPE cvmat AS (
    type integer,
    dims integer,
    step_arr integer[],
    rows integer,
    cols integer,
    data_loc character varying
);
```

Načítání seznamu databázových datových typů pro účely automatické konverze zajišťuje realizace třídy *TypeManager* automaticky po inicializaci hlavní třídy VTApi a připojení k databázi. Stejně tak se v databázi nainstalované nestandardní databázové typy (např. `cube`, `cvmat`) na klientské straně registrují pro účely pomocné knihovny *libpqtypes* (jednoduché získávání hodnot a tvorba parametrizovaných dotazů / příkazů).

Parametrizované dotazy

Ke konstrukci SQL dotazů / příkazů jsem využil podpory PostgreSQL a knihovny *libpqtypes* pro parametrizované dotazy. Tímto odpadá nutnost manuálně serializovat a deserializovat vkládané parametry příkazu či dotazu, což je velmi výhodné z hlediska výkonnosti.

Ukládání informací o parametrech (sloupců tabulky, příp. hodnoty) zajišťují metody `keyX`, `setX` a `whereX`. (viz kap. 5.1.4 o třídě *QueryBuilder*). Parametrizovaný dotaz může mít následující podobu:

```
-- vložení sekvence do datasetu vidte
INSERT INTO vidte.sequences (name, seqlocation, seqtyp)
    VALUES ($1, $2, $3);
```

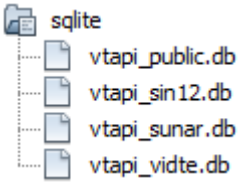
5.2.2 SQLite

SQLite je bez-serverová souborová light-weight databáze s minimem integrované funkcionality, na kterou se můžeme spolehnout v případě PostgreSQL. Jedná se z pohledu VTApi především o neexistující podporu uživatelem definovaných datových typů. V integraci podpory SQLite do VTApi jsem se tedy rozhodl převést složitější (vícedimenzionální, kompozitní) datové typy na textovou reprezentaci. K tomuto účelu jsem implementoval k datovým typům serializační a deserializační funkce. Další omezení SQLite představuje nemožnost zabezpečení databáze proti její manipulaci neoprávněným uživatelem. Řešení bezpečnosti spočívá čistě na uživateli. Databáze také podporuje pouze výrazně omezenou syntaxi jazyka SQL.

Databázové soubory

Na úrovni souborového systému odpovídá databáze jednomu souboru. Pro jednu databázi nelze vytvářet více schémat. Datové úložiště jsem tedy implementoval jako adresář obsahující následující databázové soubory s předdefinovaným způsobem jejich pojmenování (ukázka viz obr. 5.7):

- `vtapi_public.db` : odpovídá schématu public v původním návrhu
- `vtapi_X.db` : soubory odpovídající každý jednomu datasetu s názvem X



Obr. 5.7: Ukázka obsahu databázového adresáře SQLite

Umístění adresáře s databázovými soubory lze konfigurovat parametrem `dbfolder` (viz kap. 5.1). Při inicializaci hlavní třídy VTApi se otevře hlavní databázový soubor `vtapi_public.db`, za běhu aplikace se posléze příkazem ATTACH DATABASE připojí vyžadované databáze datasetu:

```
ATTACH DATABASE vtapi_vidte.db AS 'vidte';
```

Skript **create_public_sl.sql** vytvoří základní objekty databáze odpovídající schématu `public` - seznam datasetů a tabulky pro budoucí využití v metodologii VTApi.

Tabulka `datasets` obsahuje seznam a fyzické umístění datasetů:

```
-- tabulka se seznamem datasetů
CREATE TABLE [datasets] (
    [dsname] TEXT NOT NULL,
    [dslocation] TEXT,
    [userid] TEXT,
    [groupid] TEXT,
    [created] TIMESTAMP,
    [changed] TIMESTAMP,
    [notes] TEXT,
    CONSTRAINT [dataset_pk] PRIMARY KEY ([dsname])
);
```

Tabulka `methods` bude v metodologii obsahovat seznam dostupných metod:

```
-- tabulka s dostupnými metodami
CREATE TABLE [methods] (
    [mtname] TEXT NOT NULL,
    [userid] TEXT,
    [created] TEXT,
    [notes] TEXT,
    CONSTRAINT [methods_pk] PRIMARY KEY ([mtname])
);
```

Tabulka `methods_keys` bude v metodologii obsahovat seznam parametrů pro jednu metodu:

- parametr `typname` je omezen na název databázového typu
- parametr inout je omezen výčtem hodnot pro typ vstupu / výstupu

```
-- tabulka s parametry metod
CREATE TABLE [methods_keys] (
    [mtname] TEXT NOT NULL,
    [keyname] TEXT NOT NULL,
    [typname] TEXT NOT NULL,
    [inout] TEXT NOT NULL,
    CONSTRAINT [methods_keys_pk] PRIMARY KEY ([mtname], [keyname]),
    CONSTRAINT [methods_keys_mtname_fk] FOREIGN KEY ([mtname])
      REFERENCES [methods]([mtname])
);
```

Tabulka `selections` obsahuje seznam selekcí – podmnožin dat pro jednotlivé úlohy:

- parametr `selname` je omezen na název tabulky v databázi

```
-- tabulka selekcí
CREATE TABLE [selections] (
    [selname] TEXT NOT NULL,
    [dataset] TEXT NOT NULL,
    [userid] TEXT,
    [created] TIMESTAMP,
    [notes] TEXT,
    CONSTRAINT [selections_pk] PRIMARY KEY ([selname])
);
```

Tabulka `processes` bude v metodologii obsahovat seznam dostupných procesů:

- parametry `inputs` a `outputs` jsou omezeny na název tabulky v databázi

```
-- tabulka s definovanými procesy
CREATE TABLE [processes] (
    [mtname] TEXT,
    [prsname] TEXT NOT NULL,
    [inputs] TEXT,
    [outputs] TEXT,
    [userid] TIMESTAMP,
    [created] TIMESTAMP,
    [notes] TEXT,
    CONSTRAINT [processes_pk] PRIMARY KEY (prsname),
    CONSTRAINT [method_fk] FOREIGN KEY ([mtname])
      REFERENCES [methods]([mtname]) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Skript `create_dataset_sl.sql` vytvoří objekty databáze příslušející jednomu datasetu:

Tabulka `sequences` obsahuje seznam sekvencí jednoho datasetu:

- parametr seqtyp je omezen výčtem hodnot pro typ sekvence

```
-- tabulka sekvencí
CREATE TABLE [sequences] (
    [seqname] TEXT NOT NULL,
    [seqnum] INTEGER,
    [seqlocation] TEXT,
    [seqtyp] TEXT,
    [userid] TEXT,
    [groupid] TEXT,
    [created] TIMESTAMP,
    [changed] TIMESTAMP,
    [notes] TEXT,
    CONSTRAINT [sequences_pk] PRIMARY KEY ([seqname]),
    CONSTRAINT [sequences_unq] UNIQUE ([seqnum])
);
```

Tabulka `intervals` (volitelný název) obsahuje seznam intervalů jedné sekvence:

```
-- tabulka intervalů
CREATE TABLE [intervals] (
    [seqname] TEXT NOT NULL,
    [t1] INTEGER NOT NULL,
    [t2] INTEGER NOT NULL,
    [imglocation] TEXT,
    [userid] TEXT,
    [created] TIMESTAMP,
    [notes] TEXT,
    CONSTRAINT [intervals_pk] PRIMARY KEY ([seqname], [t1], [t2]),
    CONSTRAINT [sequences_fk] FOREIGN KEY ([seqname])
      REFERENCES [sequences]([seqname]) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Vložení nového datasetu do seznamu datasetů:

```
-- vložení do seznamu datasetů
INSERT INTO [public].[selections]([selname],[dataset])
    VALUES ('vidte.intervals','vidte');
```

Vložení tabulky `intervals` jako možné selection:

```
-- definice tabulky intervals jako selection
INSERT INTO [public].[datasets]([dsname],[dslocation])
    VALUES ('vidte','vidte/');
```

Podpora datových typů

SQLite nepodporuje definici vlastních datových typů, implementace třídy *TypeManager* (viz kap. 5.1.3) je tedy zbytečná. Nepodporuje však ani výčtové typy či správnost zadání číselných (a jiných) serializovaných hodnot. Ač lze tyto nedokonalosti řešit na úrovni databáze, nepřineslo by takové řešení žádný přínos z hlediska výkonnosti. Žádoucí je tedy optimalizované řešení na úrovni API, které jsem implementoval k odpovídajícím funkcím vkládání a aktualizace záznamů `keyX` a `setX`.

Parametrizované dotazy

Klientská knihovna SQLite podporuje tvorbu parametrizovaných dotazů, nicméně jejich rozhraní je oproti např. knihovně *libpq* poměrně nevyhovující a méně úsporné z hlediska výkonnosti. Vzhledem k malé podpoře datových typů a tedy nutné serializaci např. polí hodnot se použití parametrizovaných dotazů jeví jako zbytečné a tedy jsem jej neimplementoval.

6 Testy výkonnosti

V této kapitole porovnávám výkonnost databázových operací u aplikací užívajících VTApi na implementovaných datových úložištích PostgreSQL a SQLite. Testy jsem provedl v prostředích, které odpovídaly účelu těchto úložišť. **SQLite**, jakožto light-weight bez-serverová přenositelná databáze, byla testována na běžném notebooku s procesorem Intel Core i3(2 x 2.53GHz), 4GB RAM a HDD s 5400rpm na rozhraní SATA s rychlostí 3Gb/s. **PostgreSQL** jsem testoval na stejném notebooku s gigabitovým připojením k databázi na školním serveru vidte.fit.vutbr.cz.

Celkem jsem provedl 6 typů testů, a to zejména na vkládání hodnot, kde jsem předpokládal největší rozdíl ve výkonnosti. Tab. 6.1 shrnuje celkové výsledky.

Test #1 Vkládání sekvencí

Spočítáme počet sekvencí vložených za 10 sekund.

```
while(timex->getTime() < 10.0) {
    id = (rand()%100000);
    sequence->add("name"+toString(id), "vid.mpg","video","vfroml","group","");
    sequence->addExecute();
}
```

Test #2 Vkládání intervalů do sekvencí

Spočítáme počet intervalů vložených za 10 sekund. Každých 50 intervalů bude příslušet vlastní sekvenci.

```
while(timex->getTime() < 10.0) {
    id = (rand()%100000);
    sequence->add("name"+toString(id), "images/","images","vfroml", "group","");
    sequence->addExecute();
    for (int i = 0; i < 50; i++) {
        if (timex->getTime() >= 10.0) break;
        interval->add("name"+toString(id), i, i, "img.bmp");
        interval->addExecute();
    }
}
```

Test #3 Vkládání většího počtu intervalů

Vložíme větší objem dat - 100 sekvencí po 100 intervalech – a změříme uplynulý čas.

```
for (int i = 0; i < 100; i++) {
    sequence->add("name"+toString(i), "images/", "images","vfroml", "group","");
    sequence->addExecute();
    for (int j = 0; j < 100; j++) {
        interval->add("name"+toString(i), j, j, "img.bmp");
        interval->addExecute();
    }
}
```

Test #4 Vkládání rozměrného pole hodnot

Vložíme 10 sekvencí po 100 intervalech a změříme uplynulý čas. Každý interval bude obsahovat pole `svm` se 100 4-bytovými desetinnými čísly.

```
for (int i = 0; i < 10; i++) {
    sequence->add("name"+toString(i), "images/","images","vfroml", "group","");
    sequence->addExecute();
    for (int j = 0; j < 100; j++) {
        interval->add("name"+toString(i), j, j, "img.bmp");
        interval->insert->keyFloatA("svm", svm, 100);
        interval->addExecute();
    }
}
```

Test #5 Aktualizace hodnot

Projdeme všechny vložené intervaly a nastavíme hodnotu pole `svm` na nové pole hodnot, tentokrát rozšířené na 150 4-bytových desetinných čísel. Funkce `next()` zajistí potvrzení aktualizace. Změříme uplynulý čas.

```
while (interval->next()) {
    interval->setFloatA("svm", svm, 150);
}
```

Test #6 Dotazování hodnot

Dotážeme se na pole `svm` všech vložených intervalů a vytiskneme jej na standardní výstup. Změříme uplynulý čas.

```
while (interval->next()) {
    svm = interval->getFloatA("svm", size);
    cout << toString(svm) << endl;
    delete svm;
}
```

Naměřené hodnoty zobrazuje tab. 6.1. Každý test jsem provedl 5-10krát, v závislosti na jeho časové náročnosti, a výsledky zprůměroval. Všechny testy, které nepracují s vloženými daty, jsem provedl nad prázdnými tabulkami po provedení jejich údržby (příkaz VACUUM). Jinak jsem je nijak výrazně neoptimalizoval, nedoporučuji tedy jejich výsledky brát jako absolutní míru výkonnosti, slouží pouze ke srovnání. Z naměřených hodnot jednotlivých metrik jsem získal poměr výkonnosti nad datovým úložištěm PostgreSQL vůči SQLite, který byl vhodně zaokrouhlen.

Test	Metrika	PostgreSQL	SQLite	Poměr výkonnosti
#1	Počet vložených sekvencí	4824	51	**94:1**
#2	Počet vložených intervalů	4661	51	**91:1**
#3	Doba vložení intervalů	19,29s	1997,84s	**103:1**
#4	Doba vložení intervalů s daty	3,49s	240,35s	**68:1**
#5	Doba aktualizace záznamů	4,82s	246,76s	**51:1**
#6	Doba získání všech záznamů	0,28s	0,26s	**1:1**

Tab. 6.1: Výsledky testů výkonnosti VTApi pro různá datová úložiště

Z výsledků je naprosto zřejmé, že databáze SQLite velmi zaostává zejména při vkládání velkého množství záznamů o relativně malé velikosti. Při použití databáze PostgreSQL lze dosáhnout až 100x větší výkonnosti databázových operací. Tento poměr se zmenší při vkládání menší kvantity objemnějších záznamů či jejích aktualizaci, nicméně rozdíl ve výkonnosti je stále extrémní. Z praktického hlediska může použití SQLite při vývoji poskytovat výhodu díky své přenositelnosti a jednoduché architektuře, nicméně pro ostré nasazení nad větším objemem více-dimenzionálních dat je naprosto nevhodné.

7 Závěr

V této diplomové práci jsem se zabýval návrhem nové verze systému **VTApi**, což je databázové rozhraní a související metodologie pro ulehčení a urychlení vývoje aplikací založených na počítačovém vidění. Tento projekt vyvíjí na FIT VUT od roku 2010 projektový tým, jehož jsem členem, v rámci projektu MV ČR „*Metody a nástroje pro zpracování obrazu a videa pro boj s terorismem*“.

Nastudoval jsem základní techniky zpracování obrazu a videa, které mohou být využity klientskými aplikacemi knihovny VTApi. Jedná se především o různé metody extrakce rysů z obrázku či detekce pohybu ve videu. Dále jsem popsal princip podobnostního vyhledávání a základní postupy dolování z obrazových dat, jako jsou např. klasifikace, shlukování a regresní analýza.

Dále jsem se pokusil přiblížit základní koncept rozhraní VTApi. Vysvětlil jsem principy reprezentace obrazových dat a jejich popisných metadat v datovém úložišti. Podrobně jsem probral správu těchto dat a praktické využití funkcionality především z pohledu koncového uživatele API. Popsal jsem možnosti konfigurace a doplňkovou konzolovou aplikaci VTCli.

Dle zadání jsem navrhl novou verzi knihovny VTApi s cílem dosažení databázové nezávislosti na PostgreSQL a umožnění rozšíření systému o moduly pro jiná datová úložiště. Po analýze požadavků na novou verzi jsem navrhl úroveň abstrakce v systému s použitím klasických návrhových vzorů. Tento návrh jsem následně zakomponoval do celkového konceptu VTApi.

Po dohodě s vedoucím práce jsem implementoval podporu pro konektory dvou datových úložišť: SŘBD PostgreSQL a bez-serverové souborové databáze SQLite. Integrace PostgreSQL spočívala zejména v přepracování už existující funkcionality do nového konceptu VTApi. Relativně malé množství podporovaných funkcí a datových typů v případě SQLite si vyžádalo jejich naprogramování na úrovni API. Jednalo se zejména o vyřešení reprezentace a serializace hodnot vícedimenzionálních datových typů.

Implementace konektorů pro obě datová úložiště jsem porovnal z hlediska výkonnosti databázových operací. Podle předpokladů se ukázalo, že pro ostré nasazení nad velkým objemem více-dimenzionálních dat je nesmyslné používat databázi SQLite, která dosahuje až 100x menšího počtu provedených operací za jednotku času. Výhoda jejího použití spočívá zejména v přenositelnosti a jednoduché architektuře.

Programový produkt této diplomové práce představuje novou verzi knihovny VTApi. Projekt bude i nadále předmětem vývoje, zejména v oblasti optimalizace výkonnosti a implementace metodologie – podpory tvorby a řetězení metod zpracování obrazových dat. Nová verze také umožňuje relativně snadnou integraci konektorů dalších datových úložišť.

Literatura

[1] Webová stránka výzkumného projektu "*Nástroje a metody zpracování videa a obrazu pro boj s terorismem*" [online]. 2010 [cit. 2013-01-05]
Dostupné z: http://www.fit.vutbr.cz/research/grants/index.php?id=498

[2] Webová stránka projektu VTApi [online]. 2010 [cit. 2013-01-05]
Dostupné z: http://vidte.fit.vutbr.cz/vtapi.html

[3] *Presenting Effective Presentations with Visual Aids*. U.S. Department of Labor, OSHA Office of Training and Education, May 1996.

[4] Image and Video Systems Lab: *Image/Video Contents based Indexing & Retrieval* [online]. 2012 [cit. 2013-01-05]
Dostupné z: http://ivylab.kaist.ac.kr/htm/research/ivy_research/nara/image_video_contents_indexing_retrieval.htm

[5] Datta, R.; Joshi, D.; Li, J.; Wang, J.: *Image retrieval: Ideas, influences, and trends of the new age*. ACM Comput. Surv. 40, 2, Article 5 (April 2008) [online]. 2008 [cit. 2013-01-05]
Dostupné z: http://infolab.stanford.edu/~wangz/project/imsearch/review/JOUR/datta.pdf

[6] Chmelař, P.; Stryka, L.: *Multimediální databáze, opora předmětu PDB* [online]. 2007 [cit. 2013-01-05]
Dostupné z: http://www.fit.vutbr.cz/~chmelarp/pdb/MMDBs%200009b.pdf

[7] Zendulka, J.: *Materiály k přednáškám předmětu IDS* [online]. 2012 [cit. 2013-01-05]
Dostupné z: https://www.fit.vutbr.cz/study/courses/IDS/private/

[8] Nixon, M.; Aguado A.: *Feature Extraction and Image Processing*, Newnes, 2002. ISBN 0-7506-5078-8

[9] Kršek, P.: *Základy počítačové grafiky, přednášky* [online]. 2012 [cit. 2013-01-05]
Dostupné z: https://www.fit.vutbr.cz/study/courses/IZG/private/

[10] Lan, J.; Zhang, M.: *Fast and robust corner detector based on double-circle mask*, Opt. Eng [online]. 2010 [cit. 2013-01-05]
Dostupné z: http://dx.doi.org/10.1117/1.3526330

[11] Harris, C.; Stephens, M.: *A combined corner and edge detector*. Alvey vision conference, ročník 15, Manchester, UK, 1988, str. 50.

[12] Moravec, H.: *Obstacle avoidance and navigation in the real world by a seeing robot rover*. Dizertační práce, Department of Computer Science, Stanford University, září 1980.

[13] Duda, R.; Hart, P.: *Use of the Hough transformation to detect lines and curves in pictures*. Communications of the ACM, ročník 15, č. 1, 1972.

[14] Chlum, J.; Matas., O.: *Matching with PROSAC - progressive sample consensus*. Proc. of Conference on Computer Vision and Pattern Recognition [online]. 2005 [cit. 2013-01-05] Dostupné z: http://cmp.felk.cvut.cz/~matas/

[15] Jähne, B.; Haussecker, H.; Geissler, P.: *Handbook of Computer Vision and Applications*. Academic Press, 1999.
ISBN 0-12-379770-5.

[16] Chmelař, P.: *Bayesian Concepts for Human Tracking and Behavior Discovery*, Student EEICT 2006. VUT Brno [online]. 2006 [cit. 2013-01-05]
Dostupné z: http://www.feec.vutbr.cz/EEICT/2006/sbornik/03-Doktorske_projekty/07-Informacni_systemy/03-chmelarp.pdf

[17] Lloyd, S.: *Least squares quantization in PCM*. IEEE Transactions on Information Theory, ročník 28, č. 2, 1982.

[18] Zprávy projektu MV ČR Nástroje a metody zpracování videa a obrazu pro boj s terorismem. FIT VUT. 2010 - 2012.

[19] UK Visualisation Support Network: *Python GPU Programming Guide* [online] 2013 [cit. 2013-05-07]
Dostupné z: http://www.viznet.ac.uk/reports/gpu/10

[20] Lazebnik, S.: *Computer Vision lecture slides* [online] 2013 [cit. 2013-05-07]
Dostupné z: http://www.cs.illinois.edu/~slazebni/spring13/lec09_blob.pdf

[21] Nilanjan, R; Baidya Nath S..: *Edge Sensitive Variational Image Thresholding*[online]. 2008 [cit. 2013-05-07]
Dostupné z: http://webdocs.cs.ualberta.ca/~nray1/MyWebsite/AdaptiveThreshold/icip_final5.pdf

[22] Fawaz, A.: *Fast and Accurate Template Matching Algorithm Based on Image Pyramid and Sum of Absolute Difference Similarity Measure*. Research Journal of Information Technology, 4: 204-211. [online] 2012 [cit. 2013-05-07]
Dostupné z: http://scialert.net/fulltext/?doi=rjit.2012.204.211&org=10

[23] Nosek, F.: Podpora pro práci s multimediálními daty u serveru Oracle. [online] 2006 [cit. 2013-05-07]
Dostupné z: http://www.fit.vutbr.cz/study/DP/DP.php?id=4742

[24] OpenCV - Open Computer Vision library [online] 2013 [cit. 2013-05-07]
Dostupné z: http://opencv.org/

[25] PostgreSQL - open source database [online] 1996 [cit. 2013-05-07]
Dostupné z: http://www.postgresql.org/

[26] Libpqtypes - libpq extension [online]. 2008 [cit. 2013-05-07]
Dostupné z: http://libpqtypes.esilo.com/

[27] SQLite - SQL database engine [online]. 2013 [cit. 2013-05-07]
Dostupné z: http://www.sqlite.org/

[28] GEOS - Geometry Engine, Open Source [online]. 2013 [cit. 2013-05-07]
Dostupné z: http://trac.osgeo.org/geos/

[29] PostGIS - a spatial database extender for PostgreSQL [online]. 2013 [cit. 2013-05-07]
Dostupné z: http://postgis.net/

Seznam příloh

Příloha A

Obsah CD

`/doc`

programová dokumentace a referenční manuál (HTML, LaTeX)

`/script`

SQL skripty k vytvoření základní kostry databází PostgreSQL a SQLite

`/vtapi/vtapi.conf`

konfigurační soubor VTApi

`/vtapi/include`

hlavičkové soubory VTApi

`/vtapi/nbproject`

projekt NetBeans k nástroji VTCli

`/vtapi/src`

zdrojové soubory VTApi

`/vtapi/src/nbproject`

projekt NetBeans k VTApi

`/gitorious_diff.pdf`

commit na novou verzi VTApi

`/API_datoveho_uloziste_pro_praci_s_videem_a_obrazky.docx`

technická zpráva ve formátu Microsoft Word

`/API_datoveho_uloziste_pro_praci_s_videem_a_obrazky.pdf`

technická zpráva ve formátu PDF