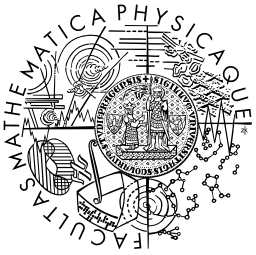


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Drahomír Hanák

Multifiltrový prohlížeč fotek

Katedra softwarového inženýrství

Vedoucí bakalářské práce:	RNDr. Filip Zavoral, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové a datové inženýrství

Praha 2019

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Děkuji svému vedoucímu RNDr. Filipu Zavoralovi, Ph.D. za cenné rady, ochotu a trpělivost, kterou mi během zpracování práce věnoval.

Název práce: Multifiltrový prohlížeč fotek

Autor: Drahomír Hanák

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Filip Zavoral, Ph.D., Katedra softwarového inženýrství

Abstrakt: Práce se zaměřuje na návrh a implementaci prohlížeče fotek, který narozdíl od existujících programů umí ukládat uživatelsky definovaná metadata přimo v souborech s fotkami. Metadata se tak neztratí při přesunu nebo přejmenování souborů. Dalším cílem práce je navrhnout dotazovací jazyk, který umožní hledat fotky na základě uživatelem definovaných metadat a existujících metadat souboru. Vytvořený program by měl mít přizpůsobivé uživatelské rozhraní pro zobrazení výsledků dotazu, prohlížení a editaci metadat souborů a editor dotazů s podporou integrované nápovědy (IntelliSense) pro vytvářený dotaz.

Klíčová slova: prohlížeč fotek, dotazovací jazyk, metadata

Title: Multifilter-based Image Viewer

Author: Drahomír Hanák

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: The aim of this work is to design and implement a photo viewer program that can save user-defined metadata directly in photo files. That way, metadata will not be lost in case files are renamed or moved. Other goal of this work is to design a query language which can be used to search photos based on both the user defined metadata and existing file metadata. Created program should have a customizable user interface for viewing query results, editing user defined metadata and query editor which has a code-completion (IntelliSense) support.

Keywords: photo viewer, query language, metadata

Obsah

1. Úvod

V současné době se zpracovává stále větší počet fotek a pro člověka je složité vyznat se v nich bez pomoci nějakého nástroje. Pro organizaci fotek proto existuje řada aplikací. Existující programy ale mají omezené možnosti hledání. Některé informace o fotkách neukládají přímo do souborů. Přesunutím souborů se tak tato metadata ztratí.

Jedním z hlavních cílů této práce je vytvořit systém pro ukládání tagů přímo v JPEG souborech. Vytvořený systém by měl podporovat uložení hodnoty tagu různého typu (celé číslo, reálné číslo, textový řetězec nebo datum a čas).

Dalším cílem práce je navrhnout dotazovací jazyk, pomocí kterého půjde hledat soubory podle různých kritérií, a implementovat systém pro vyhodnocení těchto dotazů. Vytvořený dotazovací jazyk by měl podporovat:

1. Hledání podle vzoru cesty k adresáři.
2. Hledání podle logické formule obsahující tagy zadané uživatelem a tagy odvozené z metadat souboru.
3. Seřazení výsledku dotazu podle hodnot tagů.
4. Seskupení výsledků dotazu.
5. Spojení dotazů pomocí množinových spojek sjednocení, průniku a množinového mínus.

Vytvořená aplikace by měla mít přizpůsobitelné uživatelské rozhraní. Toto rozhraní by mělo umět editovat a spouštět dotazy a zobrazit výsledky dotazů. Jednotlivé cíle jsou v následujícím seznamu.

1. Jednoduchá editace dotazů v programu.
   (a) Zvýraznění syntaxe.
   (b) Automatické napovídání možných pokračování dotazu (IntelliSense).
   (c) Čtení dotazů ze souboru, uložení dotazu do souboru.
2. Výsledky dotazu zobrazené v tabulce náhledů fotek.
   (a) Nastavitelná velikost náhledů.
   (b) Načítání náhledů na vyžádání.
   (c) Základní manipulace s vybranými soubory (kopírování, vyjmutí, přejmenování, smazání)
3. Editace tagů všech vybraných souborů najednou (změna jména, hodnoty, typu a smazání tagů).
4. Prezentace fotek z výsledku dotazu v okně aplikace nebo ve fullscreen režimu.

Cílem práce naopak není implementovat funkce pro editaci fotek. Pro tento účel existují jiné nástroje. Program by ale měl nabídnout uživateli možnost otevřít vybrané soubory v jiných aplikacích. Seznam těchto aplikací by měl být nastavitelný.

1.1 Struktura práce

Druhá kapitola popisuje formát JPEG souborů. Kapitola 3 zhodnotí výhody a nevýhody existujících programů pro hledání fotek a obsahuje rozbor hlavních problémů v této práci. Kapitola 4 popisuje, z jakých modulů se aplikace skládá, jak spolu moduly komunikují a implementaci hlavních funkcí programu. Pátá kapitola obsahuje uživatelskou příručku vytvořeného dotazovacího jazyka. Kapitola 6 popisuje uživatelské rozhraní vytvořené aplikace. Sedmá kapitola zhodnotí výkonnost programu. V osmé kapitole zhodnotíme vytvořený program a navrhneme možná vylepšení aplikace.

2. Formát souborů

Program pracuje se soubory v JPEG formátu. V této kapitole se seznámíme se základní strukturou JPEG souborů. Pro účely této práce se zaměříme hlavně na formát metadat a rozšiřitelnost souborového formátu JPEG.

JPEG je formát pro ukládání fotek. Pro komprimaci používá ztrátovou kompresní metodu JPEG. Zakódovaná jsou pouze obrazová data. V každém JPEG souboru jsou kromě obrazových dat také metainformace o souboru a uložené fotce. Tato data jsou typicky ve formátu JFIF (JPEG File Interchange Format) [1] nebo Exif (Exchangeable image file format). [2] Oba typy mají stejnou základní strukturu, se kterou se seznámíme v následujících sekcích. Typickými příponami jsou `.jpeg` nebo `.jpg`.

2.1 Struktura JPEG souboru

JPEG je binární formát. Soubor se skládá ze segmentů. Každý segment začíná bytem `0xFF`, za kterým následuje značka segmentu (1 byte). Značka segmentu může být libovolné číslo mezi `0x01` a `0xFE`. `0x00` a `0xFF` nejsou platné značky. Některé segmenty s proměnnou velikostí mají za hlavičkou délku dat uloženou jako dvou bytové bezznaménkové celé číslo ve formátu big-endian. Výjimkou je segment s komprimovanými daty, který má za hlavičkou přímo zakódovaná data.

JPEG soubor začíná segmentem se značkou `0xD8`. Tomuto segmentu se říká *Start of Image* a nemá žádná data (v binárním formátu jsou to pouze 2 byty `0xFF 0xD8`). Každý soubor obsahuje segment se značkou `0xDA` (komprimovaná data - *Start of Scan*). Na konci souboru musí být segment se značkou `0xD9` (konec JPEG souboru - *End of Image*). Všechna metadata se nacházejí v segmentech mezi *Start of Image* a *Start of Scan*. Pro přečtení metadat tak není nutné přečíst celý soubor.

Aplikace mohou ukládat data do JPEG souborů v takzvaných APPn segmentech. Tyto segmenty mají hlavičku `0xFF 0xEn` pro $n = 0, \ldots, 15$ zapsané hexadecimální číslicí. Za hlavičkou vždy následuje velikost dat jako 2 bytové bezznaménkové celé číslo s pořadím bytů big-endian. Maximální velikost každého segmentu je $2^{16} - 2$ bytů, protože pro uložení velikosti segmentu jsou dostupné pouze 2 byty a tyto byty jsou součástí obsahu segmentu. Vzhledem k rozšiřitelnosti formátu by se mohlo stát, že dvě aplikace použijí stejnou značku segmentu. Každý app segment proto začíná identifikačním ASCII řetězcem ukončeným nulovým bytem. Pro Exif segment je to např. `Exif` ukončený 2 nulovými byty. [2]

2.1.1 JFIF

JFIF [1] je minimální formát, který obsahuje jen několik málo informací o uložené fotce. Soubor začíná stejně jako jiné JPEG formáty značkou *Start of Image*. Za tímto segmentem musí následovat APP0 segment identifikovaný ASCII řetězcem `JFIF`.

2.1.2 Exif

Exif [2] je rozsáhlejší formát metadat. Informace jsou uloženy v APP1 segmentu s identifikačním řetězcem `Exif`. S formátem JFIF je nekompatibilní, protože vyžaduje, aby byl Exif segment prvním segmentem v souboru.

Většina Exif tagů je definovaná standardem a je nepovinná. Základními typy jsou celá čísla, ASCII řetězec a racionální číslo. Datum a čas je reprezentovaný jako řetězec ve formátu `"YYYY:MM:DD HH:MM:SS"`. Pokud není hodnota data známá, všechny znaky kromě `:` mohou být prázdné. Kromě základních metadat, jako je čas vytvoření, obsahuje Exif typicky také náhled fotky.

Tagy jsou v Exif segmentu zakódované ve formátu TIFF. [3] Jeho struktura je složitější než JFIF. Existuje ale mnoho knihoven, které umí číst Exif tagy.

3. Analýza řešení

3.1 Existující programy

V této části rozebereme výhody a nevýhody některých existujících programů.

3.1.1 Adobe Bridge

Adobe Bridge [4] je program pro organizaci fotek. Souborům je možné přiřazovat tagy a filtrovat podle nich zobrazené soubory. Tagy se ukládají přímo do JPEG souborů. Program má přizpůsobitelné uživatelské rozhraní a je zdarma. Nevýhodou Adobe Bridge jsou omezené možnosti hledání souborů (viz následující seznam).

- Podle tagů je možné filtrovat pouze zobrazené soubory. Aplikace dovoluje zobrazit soubory ze všech složek, ale indexování souborů je ale pomalé, protože program při indexování generuje i náhledy fotek.
- Uživatelem přidané tagy nemohou mít hodnotu.
- Při hledání podle metadat není možné zadat interval hodnot.
- Omezená podoba filtru (pouze seznam tagů, které mají být v souborech)
- Program neumí hledat soubory podle vzoru cesty.

3.1.2 digiKam

Aplikace digiKam [5] je open-source prohlížeč fotek, který umí přidávat tagy souborům. Tagy jsou uložené přímo v JPEG souborech. Program má pokročilé možnosti hledání souborů podle zadaných tagů.

Podmínky hledání jde specifikovat pouze pomocí formuláře. Vytvoření dotazu je tak pomalejší kvůli velkému množství formulářových polí. Soubory jde hledat pouze podle jména tagu. Tagy v programu nemají hodnotu. Další nevýhodou je, že program umí pracovat jen s indexovanými soubory. Při importu velkého množství fotek musí uživatel dlouho čekat, než program všechny fotky projde. Během této operace nejsou tyto fotky v programu dostupné.

3.2 Volba programovacího jazyka

Volba programovacího jazyka ovlivní výběr knihoven pro jednotlivé úkoly v této práci. V následující sekci porovnáme několik možností a vybereme jednu z nich.

3.2.1 C++

C++ je programovací jazyk, který se typicky překládá přímo do strojového kódu procesoru. Programátor má kontrolu nad tím, kde se objekty alokují a nad dealokací objektů. Nevýhodou je, že v C++ neexistuje standardní způsob pro vytváření uživatelského rozhraní a některé operace mají nedefinované chování.

3.2.2 C#

C# je programovací jazyk, který se kompiluje do Intermediate Language (IL) mezikódu. [6] Tento mezikód je při spuštění převeden Just-in-Time překladačem na strojový kód procesoru. Správu paměti v C# provádí Garbage Collector. V C# existují standardní knihovny pro tvorbu uživatelského rozhraní. Nástroje pro práci s těmito knihovnami jsou součástí vývojového prostředí Visual Studio.

Nevýhodou je režie spojená s překladem za běhu. Jakmile je ale metoda přeložená, použije se už vygenerovaný a optimalizovaný kód.

3.2.3 Java

Java se stejně jako C# překládá do mezikódu a spravuje paměť pomocí Garbage Collectoru. [7] Pro tvorbu uživatelského rozhraní v javě existují frameworky.

Jedním z cílů práce je vytvořit uživatelské rozhraní aplikace. Tvorba uživatelského rozhraní je jednodušší v Javě a C# ve srovnání s C++. Je tedy vhodnější implementovat program v těchto jazycích. Pro implementaci jsme zvolili jazyk C#, protože s ním máme větší zkušenosti.

3.3 Atributy

V této části popíšeme, jakým způsobem lze JPEG formát rozšířit a uložit v něm uživatelem zadané atributy.

Aplikace musí ukládat vlastní data do formátu JPEG v app segmentech. Tagy budou uloženy v APP1 segmentu (značka `0xE1`) s identifikátorem `Attr`. Segmenty mohou být uloženy v libovolném pořadí s výjimkou Exif a JFIF segmentů. Pro zajištění kompatibility s programy, které JPEG soubory čtou, program uloží segment s tagy až za všechna ostatní metadata.

V segmentu jsou data uložena v binárním formátu. Žádoucími vlastnostmi tohoto formátu je minimální velikost a jednoduché čtení a zápis. Na pořadí atributů nezáleží. Program vždy přečte všechny atributy ze souboru. Tagy jsou uloženy jako trojice: typ, jméno a hodnota. Formát ukládá více bytová čísla v pořadí little-endian. Pro toto pořadí bytů jsme se rozhodli kvůli zjednodušení implementace. Standardní třída `BinaryReader` umí číst data pouze v pořadí little-endian. Formát používá následující typy:

- `string` je UTF-8 řetězec zakončený nulovým bytem (např.: `0x00` je prázdný řetězec a `0x6A 0x6D 0xC3 0xA9 0x6E 0x6F 0x00` je řetězec `jméno`)
- `int32` je 32 bitové celé číslo se znaménkem uložené ve dvojkovém doplňku
- `uint16` je 16 bitové celé číslo bez znaménka

- `real` je 64 bitové číslo s plovoucí řádovou čárkou uložené ve formátu `binary64` ze standardu IEEE 754
- `DateTime` je datum a čas uložený jako `string` ve tvaru W3C DTF: `YYYY-MM-DDThh:mm:ss.sTZD`

Typ atributu se ukládá jako dvou bytové celé číslo (`uint16`):

1. `int32`
2. `real`
3. `string`
4. `DateTime`

Každý atribut je zapsán jako posloupnost bytů: typ atributu (`uint16`), jméno atributu (`string`), hodnota atributu (jeden z `int32`, `real`, `string`, `DateTime`). Seznam atributů je uložený jako posloupnost jednotlivých atributů. Formát si nemusí pamatovat délku posloupnosti, protože součástí segmentu je jeho přesná velikost. Čtení atributů skončí přečtením posledního bytu ze segmentu. Případnou nekompatibilní změnu formátu jde vyřešit zavedením nového identifikátoru app segmentu.

Velikost segmentu je omezená na $2^{16} - 2$ bytů. Ve většině případů jsou segmenty velmi malé. Musíme ale počítat i s možností, že se atribut nevejde do jednoho segmentu. V takovém případě se zakódované atributy rozdělí do více segmentů bez ohledu na hranice hodnot a hranice více bytových znaků. Nevýhodou tohoto řešení je, že k přečtení jednoho atributu musí program v nejhorším případě přečíst všechny segmenty. Na druhou stranu ale skoro vždy chceme přečíst všechny atributy a toto řešení neomezuje maximální velikost hodnoty ani jména atributu.

3.3.1 Čtení souborů

Program by měl umět číst tagy ze souborového systému, z Exif segmentu a ze segmentů ve formátu popsaném v 3.3.

Přístup k informacím o souboru ze souborového systému poskytuje standardní knihovna C# `System.IO`. Pro přístup k tagům z Exif segmentu jde použít standardní knihovnu `System.Drawing`. Nevýhodou je, že knihovna umí číst hodnoty tagů pouze jako bytová pole a jsou k dispozici jen některé tagy z Exif. Alternativou k `System.Drawing` je Metadata Extractor [8]. Tato knihovna umí číst tagy i z jiných segmentů než jen z Exif a umožňuje přístup k Exif tagům, které jsou specifické pro výrobce kamer. Knihovna je napsaná v C#. Třídy pro parsování formátů jde použít samostatně. Pro čtení metadat jsme proto zvolili knihovnu Metadata Extractor.

V JPEG formátu jsou všechna metadata na začátku souboru před obrazovými daty. Program ze souboru přečte všechny segmenty obsahující metadata. Pro získání tagů z Exif segmentu použije knihovnu Metadata Extractor. Segmenty ve formátu 3.3 přečte pomocí standardní třídy `BinaryReader`.

3.3.2 Zápis souborů

V této sekci navrhneme algoritmus pro zápis tagů do JPEG souboru. Hlavními vlastnostmi algoritmu jsou odolnost proti pádu programu a rychlost zápisu.

Nejjednodušší přístup je přepsat přímo původní soubor. Algoritmus přečte blok souboru. Pokud tento blok obsahuje tagy, přepíše je upravenými daty a upravený blok zapíše zpět na místo původního bloku. Tímto způsobem přepíše celý soubor. Program by také mohl segment s tagy ukládat ve větší velikosti, než je potřeba. Pokud se upravené tagy vejdou do předem alokované oblasti, algoritmus nemusí přepsat celý soubor. Problémem tohoto přístupu je odolnost proti pádu programu. Pokud program spadne během zápisu, nechá soubor v nekonzistentním stavu.

Protože jsou tagy relativně malé, můžeme předpokládat, že se vejdou do malého počtu bloků. Jiný algoritmus by si nejprve mohl uložit bloky, které upraví, do nového souboru. Potom začne přepisovat obsah JPEG souboru. Pokud program během zápisu spadne, je možné obnovit původní stav souboru. Problémem tohoto přístupu je, že pád programu poškodí soubor. Pro obnovení fotky musí uživatel program spustit znovu. Původní JPEG se mezitím nesmí přesunout ani přejmenovat a obsah souboru se nesmí změnit. Jinak by program nechal soubor nenávratně poškozený.

Lepší přístup vzhledem k odolnosti vůči pádu programu je vytvořit nový upravený soubor na stejném disku a následně původní soubor nahradit upraveným souborem. Nevýhodou je pomalejší zápis tagů. Zápis ale může program provádět na pozadí. Protože je pro nás důležitější nepoškodit uživatelská data, rozhodli jsme se použít tento přístup.

Jakákoli chyba při zápisu do dočasného souboru zápis přeruší a nedojde ke změně v původním souboru. To je klíčové, protože kdyby se provedla jen částečná změna, mohlo by zrovna v ten moment dojít k výpadku proudu a soubor by pak zůstal v nekonzistentním stavu. Program také musí mít po celou dobu zápisu otevřený původní soubor, aby žádný jiný proces nemohl na místo původního souboru přesunout něco jiného. Nahrazení souboru upraveným souborem musí být atomická operace alespoň pro případ, kdy jsou oba soubory na stejném disku.

3.3.3 Indexování dat

Díky výše popsanému způsobu čtení tagů stačí pokaždé přečíst jen několik prvních JPEG segmentů ze souboru. Problémem je, že program bude číst tagy z více souborů. Bloky s metadaty několika souboru pravděpodobně nebudou na disku sekvenčně za sebou. Jakmile program přečte metadata jednoho souboru, musí najít začátek dalšího souboru a to může trvat poměrně dlouho.

Možným řešením tohoto problému je indexovat tagy často používaných souborů pomocí nějaké struktury, která minimalizuje počet diskových operací. Jednou takovou strukturou je B strom. B strom je vyvážený strom, jehož každý uzel má několik potomků. [9, s. 484] Tím se sníží výška stromu oproti vyváženému binárnímu stromu a zvětší se velikost uzlů. Při hledání záznamu pak stačí přečíst jen tolik uzlů, jaká je výška B stromu. Navíc prvních pár úrovní stromu může být přímo v hlavní paměti programu. Další výhodou indexového souboru je možnost soubor uložit na rychlý disk. Indexový soubor totiž obsahuje jen metadata, a tak je menší než celé fotky.

Na platformě .NET existuje několik knihoven implementující B strom. Kromě samotné datové struktury bychom ale chtěli, aby knihovna uměla ukládat strukturovaná data (tedy ne pouze klíč a hodnotu) a zajišťovala bezpečný zápis, příp. obnovení dat, pokud dojde k pádu programu. Zároveň nechceme použít klasický databázový systém, jako je například MSSQL. Takové systémy totiž běží na počítači nezávisle na programu v jiném procesu, jsou pro účely tohoto problému moc rozsáhlé a je složitější je distribuovat s programem.

Vhodným řešením by mohly být embedded databázové systémy, protože jsou malé, běží ve stejném procesu jako program a mají řadu funkcí klasických databázových systémů.

SQLite

SQLite [10] je příkladem takové embedded databáze. Výhody shrnuje následující seznam.

- SQLite používá velké množství programů (mimo jiné je na všech Android a iOS mobilních zařízeních) [11]. Je tedy pravděpodobné, že ji budou autoři SQLite udržovat.
- Obsahuje řadu funkcí klasických databázových systémů jako například transakce, podpora SQL jazyka a vytváření indexů i přes více sloupců.
- Existuje dokumentovaná knihovna v C#, kterou spravují přímo autoři SQLite, a různé nástroje pro editaci a zobrazení obsahu souborů.
- Malá velikost.
- Volná licence.

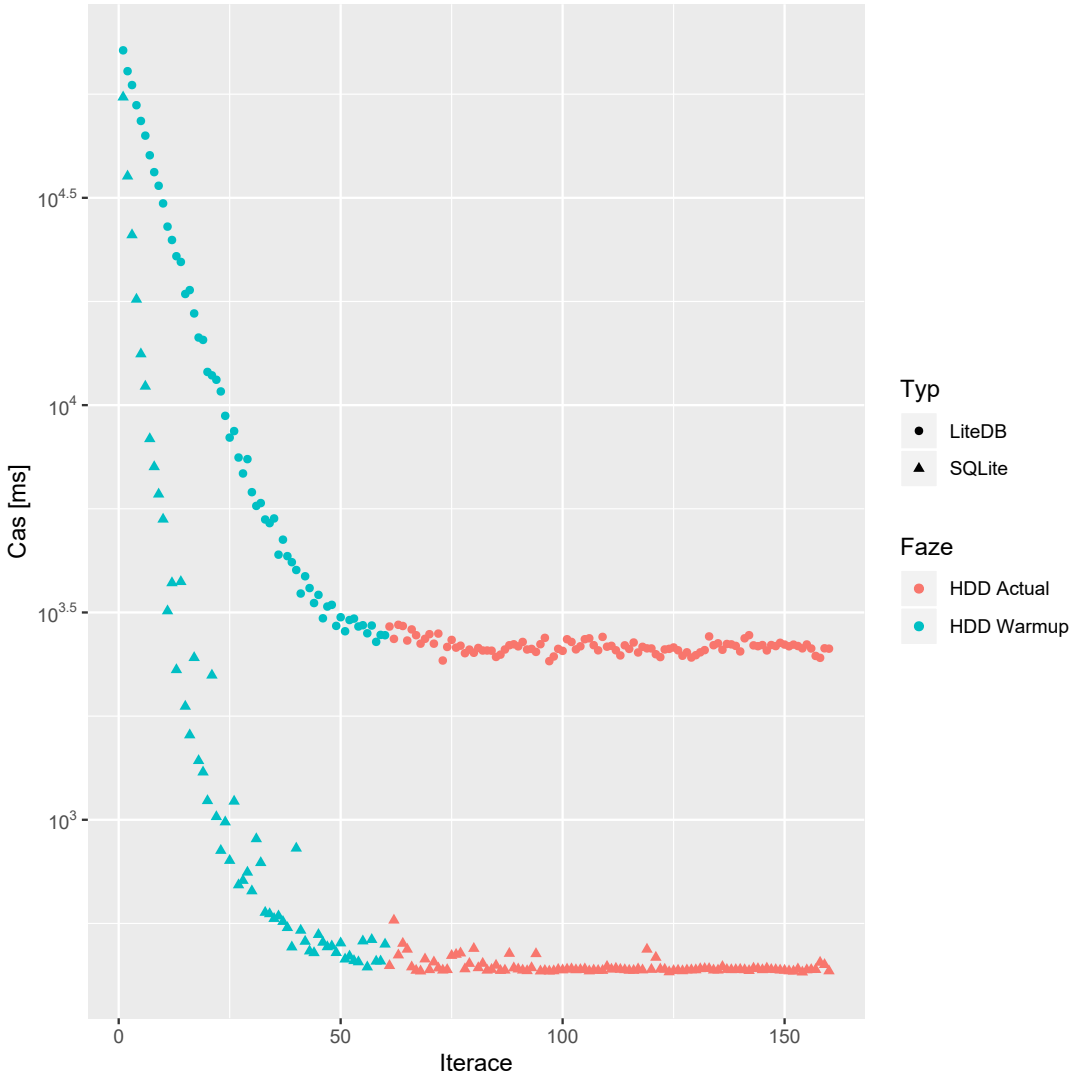
Nevýhodou SQLite je, že jde o nativní knihovnu. Se C# programem je tedy nutné distribuovat zkompilovaný kód pro danou platformu.

LiteDB

Alternativou ke SQLite je například databáze LiteDB. [12] Ta stejně jako SQLite podporuje transakce, je jedno souborová a umožňuje ukládat strukturovaná data. Výhodou LiteDB je jednoduché objektové rozhraní, které pracuje přímo se C# objekty. LiteDB je také celá naprogramovaná v C#. Knihovna nemá závislost na externí nativní knihovně a je tak snadno přenositelná. Porovnání knihovny (LiteDB v3) se SQLite [13] naznačuje, že jsou některé operace rychlejší než ve SQLite. Pokud se ovšem na metodu testování podíváme podrobněji, zjistíme, že jsou některé testy přinejmenším zavádějící.

Velikost testovaných dat je velmi malá. Výsledný soubor je menší než 8 KiB. Při velikosti diskového sektoru 4 KiB se tak vejde do dvou bloků, a tedy mohou být všechna data načtená v hlavní paměti. Jednotlivé testy se navíc spouští najedou. Jakmile benchmark dojde k testu `Query`, bude mít všechna data načtená v hlavní paměti. Samotný test `Query` prochází tabulku sekvenčně. Z výsledků testů tak není vidět výkon indexu. Testy `Insert` a `Update` také testují velmi umělý případ použití. Přidávání a upravování záznamů dělají mimo transakci, což je operace, která bude pomalá v obou knihovnách. Režie spojená s každou operací je totiž v takovém případě větší než délka samotné operace.

Porovnání knihoven



Obrázek 3.1: Porovnání knihoven SQLite a LiteDB. Osa y je logaritmická.

Výkon obou knihoven pro náš konkrétní případ použití můžeme vyzkoušet. Test v každé databázi vytvoří tabulku s celočíselným primárním klíčem, textovou hodnotou (cesta k souboru), časem přidání záznamu a bytovým polem (BLOB) velikosti 4 až 16 KiB. Tato tabulka modeluje případ, kde je vedle samotných souborů uložený i vygenerovaný náhled fotky pro rychlejší načítání.

Velikost souboru SQLite je 1,03 GiB, velikost LiteDB souboru je 1,17 GiB. Každý bod v grafu 3.1 představuje hledání 10000 rovnoměrně rozdělených náhodných záznamů z databáze se 100000 záznamy. V obou databázích jsme nastavili velikost cache na 5000 stránek při velikosti stránky 4 KiB. Před každým testem byla vymazána cache operačního systému nástrojem RAMMap. Prvních 60 měření (modře označené body) ukazuje, jak se mění výkon s rostoucím počtem cachovaných stránek. Ve zbytku měření (oranžově označené body) se výkon s časem moc neměnil, protože většina dat už byla načtena v cache. SQLite pro porovnání řetězců používá C# komparátor, který podporuje porovnání unicode řetězců

(konkrétně `StringComparer.InvariantCulture`), a je spuštěný ve WAL módu. Test byl spuštěn na počítači s procesorem Intel(R) Core(TM) i7-4790 CPU, 8 GB RAM a HDD diskem WDC WD7500BPKT-22pk4T0.

SQLite zvládá náhodné hledání podle indexované řetězcové hodnoty podstatně lépe než LiteDB (viz 3.1). Je tedy pro naši aplikaci vhodnější.

3.3.4 Konzistence záznamů

Informace o souborech uložené v indexu se nesmí lišit od dat v JPEG souborech. Jedním způsobem jak zajistit konzistenci by mohlo být sledovat změny v souborovém systému a následně upravit index. To je ale velmi náročné, protože by program musel neustále monitorovat všechny indexované soubory.

Lepší řešení je nechat data v indexovém souboru a přidat nějaký mechanizmus, který by dokázal rozeznat neplatné záznamy. To se dá vyřešit například přidáním času posledního zápisu do souboru. Pokud se tento čas neshoduje s časem posledního zápisu v souborovém systému, prohlásí se záznam za neplatný. Databáze tak může obsahovat velké množství neplatných záznamů nebo záznamů, které se dlouho nepoužily. Řešením je uložit s každým záznamem ještě čas posledního přístupu. Soubory, které uživatel dlouho nepoužil, buď na disku na dané lokaci už neexistují, nebo pro uživatele aktuálně nejsou moc zajímavé. Dá se tedy předpokládat, že je v dalším dotazu nebudeme potřebovat, a tak je můžeme smazat.

3.4 Parsování dotazu

Program musí uživateli umožnit hledat soubory podle uložených tagů. V této části navrhneme dotazovací jazyk pro přístup k souborům a vybereme knihovnu, která dotaz analyzuje.

Dotazy musí umět hledat soubory podle vzoru cesty a podle logické formule. Další funkcí dotazovacího jazyka je definovat pořadí a seskupení vrácených výsledků. Dotazy by ideálně mělo jít spojit pomocí množinových spojek sjednocení, průniku a množinového mínus. Chceme navrhnout jazyk, který obsahuje pouze kritéria hledání, aby bylo jednoduché ho použít. Vyhodnocení dotazu provede program.

Hlavním problémem je, jak dotaz popsat, analyzovat a následně vykonat. Překlad programovacího jazyka typicky probíhá ve fázích. První reprezentací dotazu je textový řetězec, který zadal uživatel. Lexikální analýza seskupí jednotilé znaky do větších celků, kterým říkáme tokeny. Podoba tokenů je daná regulárními výrazy. Druhou fází je parsování dotazu. Parser zjistí, jak jde zadaný dotaz vygenerovat z bezkontextové gramatiky, která popisuje strukturu jazyka všech dotazů. Výstupem je derivační strom. Vnitřní vrcholy v tomto stromě jsou neterminály, listy jsou pak terminály (tokeny). [14, s. 4–5]

3.4.1 Vlastní parser

První možnost, jak dotaz analyzovat, je vytvořit vlastní parser. To je ale časově náročné. Existuje proto mnoho knihoven, které umí automaticky vygenerovat parser z gramatiky.

3.4.2 Generátory parseru a lexeru

Generátory lexerů typicky na vstupu dostanou seznam regulárních výrazů. Tyto výrazy se pak převedou na deterministický konečný automat, který umí rozpoznávat jednotlivé tokeny v textu. Knihovny pro generování lexerů se liší syntaxí, ale základní myšlenka je velmi podobná.

Vstupem pro generátor parseru je bezkontextová gramatika. Knihovny pro generování parserů se naopak liší dost zásadně. Společným problémem je ale jednoznačnost gramatiky. V obecném případě může existovat více derivačních stromů zadanáho programu, které mohou představovat jiné programy. Například $E \rightarrow E + E, E \rightarrow id$ je nejednoznačná gramatika. Pro slovo $a + b + c$ není zřejmé, jestli se má provést nejprve $a + b$, a teprve pak přičíst c, nebo naopak. Víceznačným gramatikám se tedy budeme snažit vyhnout.

Dalším problémem je, že generátory typicky nepodporují libovolnou jednoznačnou bezkontextovou gramatiku. U nástrojů nás ještě bude zajímat, jestli používají přístup shora dolů, nebo přístup zdola nahoru. Přístup shora dolů začne s počátečním neterminálem gramatiky a snaží se odvodit program. Přístup zdola nahoru začne s terminály v listech stromu a postupně se snaží postavit celý derivační strom programu. Tyto přístupy se často liší třídou jazyků, které umí rozpoznávat.

Problémem analýzy shora dolů je, že přímo nepodporuje levou rekurzi. Ta je ale v jazycích velmi častá, protože se často používají levě asociativní operátory. Například pravidla $E \rightarrow E + F|F$, která generují sčítání identifikátorů, používají levou rekurzi na neterminálu E. Generátory parserů shora dolů často podporují alternativní způsob zápisu pravidel: $E \rightarrow F(+F)*$. Symbol $*$ zde značí obecnou iteraci. Jde o podobnou operaci, jako v regulárních výrazech. Výraz v závorce $(+F)$ se může libovolně krát opakovat. Při výběru knihovny se tak nemusíme omezit jen na generátory, které používají přístup zdola nahoru.

ANTLR4

Knihovna ANTLR4 [15] je nástroj, který umí vygenerovat parser v několika programovacích jazycích včetně C#. Lexer podporuje lexikální módy. Ty umožňují za běhu přepínat, která pravidla se smí použít.

Parser provádí analýzu shora dolů. Používá přitom algoritmus $ALL(*)$. [15] Ten umí vygenerovat parser pro jakoukoliv bezkontextovou gramatiku bez nepřímé levé rekurze (tj. pravidel typu $A \rightarrow B, B \rightarrow A$ apod.). Přímou levou rekurzi dokáže automaticky přepsat. Pokud má na výběr z několika možností, dynamicky si na základě následujících tokenů na vstupu vytváří deterministický konečný automat. Tento automat rozhodne podle následujících tokenů, jaké pravidlo se v daném případě použije.

Výhodami ANLTR4 je, že umí vygenerovat parser pro libovolnou bezkontextovou gramatiku bez levě rekurzivních pravidel, zvládá čitelné hlášení chyb, má dobrou dokumentaci, jde o velmi používaný a udržovaný nástroj. Zdrojový kód ANTLR4 a běhových knihoven pro jednotlivé jazyky je navíc otevřený a distribuovaný pod volnou licencí.

Coco/R

Dalším nástrojem pro generování parseru je Coco/R. [16] Stejně jako ANTLR4 použítá přístup shora dolů, umí vygenerovat parser v C# a má dobrou dokumentaci. Nevýhodou je, že rozpoznává menší třídu jazyků než ANTLR4. Náš jazyk je ale velmi jednoduchý, takže to tolik nevadí.

Hime

Hime [17] je příklad generátoru, který pouívá přístup zdola nahoru. Konkrétně implementuje algoritmus RNGLR. Má knihovnu pro C# a několik dalších jazyků.

Jazyk dotazů je relativně jednoduchý a dotazy často budou krátké. Z hlediska rozšiřitelnosti programu je ale žádoucí použít nějaký generátor, který v případě potřeby dokáže vygenerovat parser i složitějšího jazyka. Vybrali jsme proto nástroj ANTLR4.

Jazyk dotazů

Jazyk dotazů (viz gramatika 3.1) má syntaxi podobnou jazyku SQL. Pro tuto syntaxi jsme se rozhodli, protože umožňuje zapsat dotazy v podobě, která je čitelá i pro uživatele bez znalosti dotazovacího jazyka. Zvolili jsme stejnou prioritu operátorů, kterou používají databázové systémy jako je MSSQL a Oracle SQL [18] [19]. Operátor `not` (negace) tak má jinou prioritu než obdobný operátor (`!`) v jazycích C++ a C#, protože předpokládáme, že se negace bude častěji aplikovat na výsledky výrazů než na jednotlivé podvýrazy.

```
queryExpression → intersection ((UNION | EXCEPT) intersection)*
   intersection → queryFactor (INTERSECT queryFactor)*
    queryFactor → query | LPAREN queryExpression RPAREN
          query → SELECT source optWhere optOrderBy optGroupBy
         source → ID | STRING | LPAREN queryExpression RPAREN
       optWhere → WHERE predicate | λ
      optOrderBy → ORDER BY orderByList | λ
     orderByList → (predicate direction)+
       direction → DESC | ASC | λ
      optGroupBy → GROUP BY predicate | λ
       predicate → conjunction (OR conjunction)*
     conjunction → literal (AND literal)*
         literal → comparison | NOT comparison
      comparison → expression (REL_OP expression)?
      expression → multiplication (PLUS_MINUS multiplication)*
 multiplication → factor (MULT_DIV factor)*
          factor → INTEGER | REAL | STRING | ID |
                   PLUS_MINUS factor | ( predicate ) |
                   ID LPAREN arguments RPAREN
       arguments → argumentList | λ
    argumentList → predicate (, predicate)*
```

Listing 3.1: Gramatika jazyka dotazů

Gramatika 3.1 používá operátory `*` (výraz se může opakovat 0 a vícekrát) a `?` (výraz je volitelný). λ značí prázdný řetězec. Terminály jsou zvýrazněny modře. Neterminály jsou řetězce, které začínají malým písmenem.

3.5 Vyhodnocení dotazu

3.5.1 Hledání podle vzoru cesty

Fotky jsou jako každé jiné soubory uloženy v adresářové struktuře. Pokud nebudeme uvažovat nepřímé odkazy, je touto strukturou strom. Často se tak hodí hledat soubory podle zadaného jména, případně podle jména některého rodičovského adresáře. Pro tento účel se používají vzory cest - tzv. glob výrazy. Vzor cesty obsahuje jména adresářů oddělená oddělovači složek `/` jako normální cesty. Navíc může obsahovat speciální znaky, které nejsou v normálních cestách povolené. Tyto speciální znaky je možné nahradit nějakým řetězcem znaků. Jejich význam shrnuje následující seznam.

1. `?` jde nahradit právě jedním libovolným znakem kromě oddělovače adresářů.
2. `*` jde nahradit 0 a více libovolnými znaky kromě oddělovačů adresářů. Pokud je ale `*` jediný znak ve jméně adresáře, musí být nahrazený alespoň jedním znakem.
3. `**` jde nahradit 0 a více znaky včetně oddělovačů adresářů.

Například `a*` odpovídá cestám `a`, `ax`. `a/*/b` odpovídá `a/x/b`, ale neodpovídá `a/b`, protože v tomto případě se musí `*` nahradit alespoň 1 znakem.

Díky popularitě těchto výrazů existují v C# knihovny pro jejich parsování. Jednou z nich je `DotNet.Glob` [20]. Ta obsahuje pouze algoritmus, který rozhodne, jestli zadaná cesta odpovídá vzoru. Pro hledání souborů se tak moc nehodí. Dalšími možnostmi jsou `Glob.cs` [21] a `glob`. [22]

Algoritmy pro hledání souborů podle zadaného vzoru v těchto knihovnách prochází adresářový strom v DFS pořadí. Nabízí se tedy otázka, zda je tento způsob prohledávání výhodný. Zřejmým měřítkem výkonnosti algoritmu je celkový čas běhu. Pro uživatele je ale spíš důležitá latence, tedy čas od spuštení k prvnímu výsledku. Pokud je požadovaný soubor v prním adresáři, latence DFS přístupu bude velmi malá. Může se ale také stát, že je soubor v posledním adresáři a v tomto případě bude latence skoro tak dlouhá jako celkový čas hledání.

Tagy souborů se indexují. Díky tomu program zná rozdělení souborů v navštívených adresářích. Potom je možné spočítat přibližné rozdělení souborů v celém stromě. Hledání souborů může nejprve prohledat cesty, kde nejspíše budou požadované soubory. Problémem je, že obsah indexu nemusí odpovídat skutečnému rozdělení fotek v souborovém systému. Uživatel může celé podstromy přidávat, přesouvat a odebírat. V běžné situaci ale tento přístup zásadně vylepší latenci a moc nezhorší celkový čas hledání oproti DFS.

Nevýhodou zmíněných knihoven je, že v nich nejde jednoduše změnit pořadí prohledávání složek. V této části navrhneme přístup, jak bude program složky prohledávat. Jednoduché řešení je projít všechny adresáře v podstromě a následně zjistit, jestli daná cesta odpovídá vzoru. Problémem je, že algoritmus projde velké množství cest zbytečně, pokud vzor cesty neobsahuje `**`. Algoritmus 1 používá tento přístup, ale neprochází celý strom. Na začátku rozdělí vzor podle oddělovače adresářů na $p_0, \ldots, p_{n-1}$ a postupně se snaží najít cesty, které odpovídají vzoru. Ve frontě q jsou dvojice (c, i), kde c je cesta odpovídající vzoru $p_0, \ldots, p_{i-1}$.

Algorithm 1 Hledání souborů podle vzoru p

```
1: procedure SEARCH(p)
2:     (p_0, ..., p_{n-1}) = split(p)
3:     q = PriorityQueue()
4:     enqueue(q, (p_0, 1))                    ▷ Začneme v kořenovém adresáři
5:     while not empty(q) do
6:         (path, i) = dequeue(q)
7:         if i = n then
8:             oznam nový nalezený adresář path
9:         else if p_i je vzor ** then
10:            if path odpovídá vzoru p then
11:                oznam nový nalezený adresář path
12:            for child ∈ dirs(path) do
13:                enqueue(q, (child, i))
14:        else
15:            for child ∈ dirs(path, p_i) do
16:                enqueue(q, (child, i + 1))
```

Algoritmus tak prochází pouze cesty, které odpovídají nějakému prefixu vzoru. Celý podstrom daného adresáře projde jen tehdy, pokud je to nutné (vzor je `**`).

Funkce $dirs(c)$ vrátí všechny podadresáře složky c. Druhá varianta $dirs(c, p)$ vrátí všechny podadresáře složky c, které navíc odpovídají vzoru p. Funkce *split* na začátku algoritmu rozdělí cestu podle oddělovače adresářů na části.

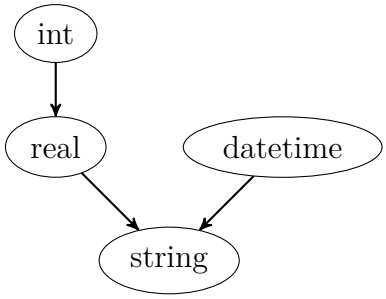
3.5.2 Hledání podle výrazu

Dotaz může obsahovat výrazy ve `where`, `order by` nebo `group by` části. Pro vyhodnocení dotazu je nutné vyhodnotit tyto výrazy.

Parser z dotazu vytvoří derivační strom. Pro výrazy se na tento strom můžeme dívat jako na uzávorkování. Vnitřní uzly tohoto stromu jsou jména funkcí nebo operátory. Přímí potomci uzlů jsou operandy. V listech pak může být buď konstantní hodnota, nebo jméno tagu. Samotné vyhodnocování probíhá rekurzivně. Nejprve se vyhodnotí všichni potomci daného uzlu, a pak se na vypočítané hodnoty aplikuje operátor nebo funkce.

Hodnoty tagů mohou mít různý typ (celé číslo, reálné číslo, textový řetězec nebo datum a čas). Funkci například nejde přímo předat hodnotu typu `real`, pokud na vstupu očekává hodnotu typu `int`. Dalším problémem jsou nepodporované varianty. Například pro volání `add(3.14159, 2)` s reálným a celočíselným parametrem nemusí existovat implementace funkce `add`. Pokud ale existuje funkce pro dva reálné operandy, program v tomto případě může automaticky provést konverzi a zavolat tu správnou variantu.

Ne vždy chceme implicitní konverzi provést. Pokud by se v `add(3.14159, 2)` převedl reálný parametr na celé číslo, ztratí se tím informace o desetinné části. Diagram 3.2 ukazuje povolené implicitní konverze. Implicitní konverze z typu T na typ U je možná právě tehdy, když v tomto grafu existuje cesta z T do U. Z typu `int` existuje více možných konverzí. Buď na typ `real` nebo na typ `string`. Při rozhodování jakou funkci zavolat tedy musíme ještě určit cenu konverze. Cenou tu



Obrázek 3.2: Povolené implicitní konverze

bude délka cesty v grafu (viz 3.2). Převedením `int` na `real` se neztratí informace o tom, že výraz pracuje s číslem. Tato konverze má proto nižší cenu než převod na `string`.

Funkce mají často více než jeden parametr. Výpočet ceny konverze musíme rozšířit přidáním agregační funkce. Pro každou variantu jde například spočítat cenu konverze každého parametru a tyto ceny sečíst. Sčítání celých čísel je komutativní operace. Cena konverze tak nezávisí na pořadí parametrů. Další vlastností je, že volání, která nevyžadují žádnou konverzi, budou mít nejnižší cenu 0. Pokud taková varianta neexistuje, vybere se alespoň funkce s celkově malou cenou konverze pro všechny parametry.

Operátory jsou v podstatě také funkce. Jediný rozdíl je syntaxe. Binární operátory se zapisují v infixní podobě. Pro vyhodnocení operátorů tak jde použít stejný mechanizmus jako pro vyhodnocení funkcí.

Dalším problémem při vyhodnocování výrazu je, že typ tagů ve výrazu nejde určit za překladu. Tag se stejným jménem může mít v několika souborech jinou hodnotu i typ. Hledání vhodné varianty funkce a konverze typů se tedy musí provést za běhu dotazu.

Konstantní podvýrazy

Jednou přímočarou optimalizaci, kterou bychom mohli udělat, je vyhodnocení konstantních podvýrazů už při překladu. Podvýrazy, které lze spočítat za překladu, pak není nutné vyhodnocovat pro každý soubor. Předpokládáme ale, že v dotazech často nebudou složité výrazy a funkce. U funkcí navíc nemůžeme předpokládat, že nemají vedlejší účinky. Vyhodnocení dotazu je v neposlední řadě operace, která primárně závisí na rychlosti disku. Celkový čas vykonání dotazu tak moc nezlepšíme.

3.5.3 Vyhodnocení množinových spojek

Dotazy jde spojit pomocí množinových spojek sjednocení, průniku a množinového mínus. V této části popíšeme, jakým způsobem se tyto spojky vyhodnocují.

Přímočaré řešení může dotazy vykonávat postupně. Nejprve najde množinu souborů z prvního dotazu, množinu souborů z druhého dotazu, a pak množiny sjednotí pomocí daného operátoru. Nevýhodou tohoto řešení je malá efektivita. Všechny operátory kromě sjednocení musí nejprve vyhodnotit oba poddotazy, než vrátí první prvek z výsledku. To může být mnoho souborů a uživatel by musel dlouho čekat.

Průnik poddotazů je možné vylepšit tak, že se budou poddotazy vykonávat postupně. V každé iteraci program najde další soubor z obou poddotazů. Nalezený soubor je v průniku, pokud už je ve výsledku druhého poddotazu. V nejhorším případě je stále nutné projít všechny soubory z obou dotazů pro nalezení prvního výsledku. Narozdíl od předchozího řešení ale jde najít první výsledek podstatně rychleji. Tento způsob nicméně neřeší množinové mínus. Pro nalezení prvního prvku z dotazu $A - B$ pro poddotazy A, B je stále nutné nalézt všechny výsledky dotazu B.

Pro zadaný soubor je možné zjistit, jestli patří do výsledku dotazu. Předpokládejme, že dotazy A ani B neobsahují množinové spojky. Později tento předpoklad odstraníme. V části 3.5.2 jsme popsali jak najít hodnotu výrazu ve `where` části dotazu pro zadaný seznam tagů. To samo o sobě nestačí. Cesta souboru také musí odpovídat vzoru cesty v dotazu. Platný vzor cesty ale jde převést na regulární výraz. Následující schéma ukazuje, jak takový převod provést. α, β jsou řetězce libovolných znaků a C je množina všech znaků, kromě oddělovače adresářů.

1. $\alpha/**/\beta$ nahradíme $\alpha(/C^+)^*/\beta$
2. $\alpha/**$ nahradíme $\alpha(/C^+)^*$ - stejné jako předchozí bod. Na konci není oddělovač adresářů.
3. $\alpha/*/\beta$ nahradíme $\alpha/C^+/\beta$ - pokud je $*$ jediný znak ve jméně adresáře, nesmí toto jméno být prázdné.
4. $\alpha*\beta$ nahradíme $\alpha C^*\beta$ - pokud $*$ není jediný znak ve jméně adresáře, může se nahradit prázdným řetězcem.
5. $\alpha?\beta$ nahradíme $\alpha C?\beta$ - ? se musí nahradit libovolným znakem kromě oddělovače adresářů.

Výhodou regulárních výrazů je, že jde relativně jednoduše zjistit, jestli zadané slovo odpovídá vzoru. Pro tento účel existuje třída ve standardní knihovně C#. Pro dotaz bez množinových spojek tak umíme rychle zjistit, jestli je zadaný soubor ve výsledku dotazu. Algoritmus 2 toto řešení rozšiřuje na dotazy s množinovými spojkami. Pomocná procedura $vyhodnot(q, f)$ vrátí hodnotu výrazu z `where` části dotazu q pro zadaný soubor f.

Algorithm 2 Je soubor f ve výsledku dotazu q?

```
1: procedure JeVeVysledku(f, q)
2:     if q je dotaz bez množinových spojek then
3:         return f odpovídá vzoru cesty q a vyhodnot(q, f) ≠ null
4:     else if q je A ∪ B pro poddotazy A, B then
5:         return JeVeVysledku(q, A) ∨ JeVeVysledku(q, B)
6:     else if q je A ∩ B pro poddotazy A, B then
7:         return JeVeVysledku(q, A) ∧ JeVeVysledku(q, B)
8:     else if q je A − B pro poddotazy A, B then
9:         return JeVeVysledku(q, A) ∧ ¬JeVeVysledku(q, B)
```

Snadno ověříme, že je algoritmus 2 korektní.

- Pokud je dotaz tvaru $A \cup B$, f je ve výsledku dotazu právě tehdy, když je ve výsledku alespoň jednoho z poddotazů A, B.
- Pokud je dotaz tvaru $A \cap B$, je f ve výsledku dotazu právě tehdy, když je ve výsledku obou poddotazů.
- Nakonec je f ve výsledku $A - B$, pokud je ve výsledku A a zároveň není ve výsledku B.

Pomocí algoritmu 2 jde vylepšit algoritmus vyhodnocení dotazu s množinovými spojkami. Vyhodnocení dotazu $A \cap B$, kde A, B jsou poddotazy, začne hledat soubory z dotazu A. Pro každý nalezený soubor z A program zjistí pomocí algoritmu 2, jestli nalezený soubor patří i do výsledku poddotazu B. Podstatnou výhodou tohoto přístupu je, že vyhodnocení nečaká na výsledky poddotazu B.

Tato optimalizace zásadně zrychlí vyhodnocení množinovéo mínus. Pro dotaz tvaru $A - B$ není vůbec nutné vyhodnotit druhý poddotaz B. Stačí najít všechny soubory z A a algoritmem 2 pro každý zjistit, jestli není i v poddotazu B.

Obnovení při změně v souborovém systému

Soubory ve výsledku dotazu se mohou měnit zásahem jiných programů. Pomocí algoritmu 2 je možné obnovit výsledek dotazu, aniž by bylo nutné dotaz znovu vyhodnotit. Program bude sledovat změny v adresářích dotazu a při každé změně zjistí pomocí algoritmu 2, jestli mají být změněné soubory ve výsledku dotazu.

Jeden způsob, jak sledovat změny souborů v souborovém systému, je procházet v pravidelných intervalech adresáře. Program musí zvolit vhodnou délku intervalu. Pokud budou intervaly moc malé, bude muset často procházet všechny soubory. Pokud bude interval moc dlouhý a dojde ke změně souborů, bude dlouho trvat, než program změnu zpracuje. Výhodou tohoto přístupu je, že se program vždy do určitého času dozví, že k nějaké změně došlo.

Operační systém poskytuje rozhraní pro sledování změn v souborovém systému. V C# pro tento mechanizmus existuje třída `FileSystemWatcher`. [23] Program může sledovat velké množství souborů. Výhodou tohoto řešení je, že nemusí pokaždé procházet všechny soubory. Nevýhodou ale je, že se změny ukládají do bufferu konstantní velikosti. Tento buffer může přetéct a program tak nezjistí, že k nějaké události došlo.

Oba způsoby je možné zkombinovat. Program bude primárně sledovat soubory pomocí událostí operačního systému a v pravidelných intervalech bude procházet všechny soubory ve sledovaných adresářích. Výhodou oproti předchozímu řešení je, že nemusí často procházet všechny soubory. Implementace sledování změn se ale zkomplikuje. Jednodušší řešení je použít některý z předchozích přístupů a dát uživateli možnost obnovit výsledek dotazu manuálně.

Další optimalizace

Vyhodnocení množinových spojek $Q_1 \cap Q_2$ a $Q_1 \cup Q_2$ pro poddotazy Q_1, Q_2 prohledá dvakrát soubory, které jsou ve výsledku Q_1 i Q_2. Jazyk vzorů cest je přitom regulární. Průnik a doplněk regulárních jazyků A, B je regulární [24], a tedy lze zjistit, jestli je $A \subseteq B$ nebo jestli jsou disjunktní. Tuto informaci může

program využít při vyhodnocování dotazu $Q_1 \cap Q_2$ nebo $Q_1 \cup Q_2$. Pokud jeden z poddotazů prohledává podmnožinu souborů druhého poddotazu, jde je spojit do jediného dotazu bez množinových spojek. Takový dotaz pak projde každý soubor právě jednou.

Přestože pro práci s konečnými automaty v C# existují knihovny, vyhodnocení dotazů se touto optimalizací zkomplikuje. Navíc i když vyhodnocení dotazu prohledá některé soubory dvakrát, druhý přístup bude pravděpodobně výrazně rychlejší. Načtené soubory budou totiž v cache. V neposlední řadě předpokládáme, že se množinové operátory budou používat méně často. Z praktického hlediska se tedy implementace této optimalizace nevyplatí.

3.5.4 Vnořené dotazy

V `select` části dotazu je možné místo vzoru uvést poddotaz v kulatých závorkách nebo jméno pohledu. Pohled je dotaz uložený v souboru. Pohledy fungují stejně jako vnořené dotazy. Zkopírováním textu pohledu na místo identifikátoru dostaneme ekvivalenstní dotaz.

Dotaz s vnořenými poddotazy lze převést na ekvivalentní dotaz bez vnořených poddotazů. Označme q' dotaz, který má v `select` části vnořený poddotaz q a ve `where` části má podmínku p. Předpokládejme navíc, že q neobsahuje vnořené poddotazy. Překladač nahradí q' dotazem $Preved(q, p)$, kde $Preved$ je funkce z algoritmu 3.

Algorithm 3 Převod dotazu q na dotaz bez vnořeného poddotazu

```
1:  procedure PREVED(q, p)
2:      if q je A ∪ B pro poddotazy A, B then
3:          return Preved(A, p) ∪ Preved(B, p)
4:      else if q je A ∩ B pro poddotazy A, B then
5:          return Preved(A, p) ∩ Preved(B, p)
6:      else if q je A − B pro poddotazy A, B then
7:          return Preved(A, p) − Preved(B, p)
8:      else
9:          p' ← podmínka ve where části dotazu q
10:         nahraď podmínku v dotazu q podmínkou p ∧ p'
11:         return q
```

3.5.5 Rozdělení atributů

Algoritmus 1 prohledává soubory v pořadí daném prioritou. V této části navrhneme způsob, jak prioritu spočítat z podmínky ve `where` části dotazu.

Z databáze lze získat seznam cest k souborům, které obsahují atributy použité v dotazu. Program by následně mohl projít všechny podsložky vrácených cest a pro každou dvojici jméno atributu, složka by mohl uložit počet souborů, které atribut v daném podstromě obsahují.

Problémem popsaného způsobu je závislost jmen tagů. Například při zpracování podmínky `X and Y` není zřejmé, kolik existuje souborů, které obsahují atribut X a zároveň atribut Y. Místo rozdělení jednotlivých jmen tagů je možné

spočítat rozdělení jejich podmnožin. Počet podmnožin je 2^n pro n počet různých jmen tagů v podmínce. Algoritmus ale nemusí počítat rozdělení podmnožin atributů, které nejsou uloženy v žádném souboru. Navíc předpokládáme, že počet různých tagů ve `where` filtru bude malé číslo.

Pořadí prohledávání se nezmění, pokud bude velká část souborů obsahovat stejný atribut. Neuživatelské atributy, jako jsou například tagy z Exif, budou téměř v každém souboru. Popsaný mechanizmus se tedy hodí pouze pro tagy zadané uživatelem.

Následující schéma f rekurzivně najde podmnožinu jmen tagů, pro které by podmínka ve `where` části dotazu mohla být pravdivá. $A, B, A_1, \ldots, A_n$ jsou podvýrazy, f je samotné schéma aplikované rekurzivně na nějaký podvýraz.

- Pokud je A jméno atributu, vrať všechny podmnožiny obsahující A.
- Pokud je A konstanta, vrať všechny podmnožiny.
- A `and` B: vrať $f(A) \cap f(B)$
- A `or` B: vrať $f(A) \cup f(B)$
- `not` A: vrať $f(A)^c$ (doplněk množiny vzhledem ke všem podmnožinám)
- $funkce(A_1, \ldots, A_n)$: vrať $\bigcap_{i=1}^{n} f(A_i)$. Tedy předpokládáme, že funkce vrátí nenulovou hodnotu právě tehdy, když jsou všechny její parametry nenulové.
- Ostatní binární a unární operátory se vyhodnocují stejně jako funkce.

Prioritu prohledávání jde spočítat jako součet souborů, které v daném podstromě obsahují alespoň jednu podmnožinu vrácenou schématem f. Takto spočítaná priorita ale nezávisí na počtu souborů. Může se tak stát, že program nejprve prohledá velký podstrom, kde najde jen relativně malé množství výsledků. Je proto vhodné výsledné číslo vydělit počtem souborů v daném podstromě.

3.6 IntelliSense

Dotazovací jazyk je navržený tak, aby byl přehledný a srozumitelný. Některé dotazy tak mohou být poměrně dlouhé. V této části navrhneme systém automatického napovídání možných pokračování dotazu. Cílem je implemntovat systém, který zrychlí psaní dotazů a pomůže uživatelům se syntaxí dotazovacího jazyka. Nápovědy by měly obsahovat:

1. klíčová slova
2. jména pohledů v `select` části dotazu
3. jména adresářů ve vzoru cesty
4. jména nedávno použítých tagů a jejich hodnoty
5. jména funkcí

Nápovědy závisí na kontextu. Program by například neměl napovídat klíčové slovo `where`, pokud na dané místo dotazu nepatří. Pro nalezení vhodné nápovědy musí program zjistit seznam tokenů, které mohou být za kurzorem (aktuální pozicí v editoru), a cestu v derivačním stromě (jména použitých pravidel).

3.6.1 Získání stavu pod kurzorem

Vstupem algoritmu je dotaz a pozice kurzoru v dotazu. Problémem je, že dotaz nemusí být platný. Budeme ale předpokládat, že část dotazu od začátku do pozice kurzoru je prefix nějakého platného dotazu.

ANTLR4 reprezentuje gramatiku jako Augmented Recursive Transition Network (ATN). [15] Tato struktura je orientovaný graf. Hrany reprezentují přechody mezi stavy parseru a jsou označeny typem tokenu, který se má přečíst ze vstupu. Hrana také může reprezentovat ϵ přechod, který nečte token ze vstupu.

Algoritmus použije vygenerovaný lexer pro převod dotazu na tokeny. Do vstupu přidá token reprezentující kurzor a simuluje průchod ATN. Průchod skončí, pokud dojde k tokenu kurzoru. Program tak najde všechny možné cesty, které vedou ke kurzoru. Nevýhodou tohoto přístupu je, že se prohledávání grafu musí vracet. Gramatika dotazů je ale jednoduchá a předpokládáme, že dotazy budou krátké.

3.6.2 Výběr pokračování

Klíčová slova, jména funkcí a pohledů má program načtená v paměti. Pro získání jmen a hodnot tagů může použít vygenerovaný indexový soubor (viz 3.3.3). Nápovědy jmen adresářů ve vzoru cesty generuje algoritmus 1.

3.7 Uživatelské rozhraní

3.7.1 Výběr knihovny

Windows Forms

Windows Forms [25] je knihovna pro tvorbu uživatelských rozhraní. Výhodou Windows Forms je velký počet hotových komponent a dobrý editor formulářů, který je součástí Visual Studia. Editor také podporuje jednoduchou lokalizaci komponent.

Nevýhodou Windows Forms je, že podobu existujících komponent není jednoduché upravit.

WPF

WPF [26] vykresluje uživatelské prvky vektorově, pro vykreslení používá moderní hardware a podobu existujícíh komponent jde upravit. Uživatelské rozhraní se často vytváří pomocí jazyka XAML, pro který existuje vizuální i textový editor ve Visual Studiu. Nevýhodou WPF je komplikovanější definice uživatelského rozhraní.

V práci jsme se rozhodli použít Windows Forms, protože velký počet uživatelských prvků jde jednoduše implementovat pomocí existujících komponent. Při implementaci okna pro zobrazení výsledků dutazu se ale ukázalo, že jsou existující komponenty pro tento účel moc pomalé. V programu jsme proto museli vytvořit vlastní komponentu.

3.7.2 Náhledy fotek

Náhledy souborů jsou důležitým prvkem uživatelského rozhraní. Dovolují uživateli jednoduše najít hledaný soubor v eventuálně velkém množství souborů. V této části vybereme knihovnu pro načítání fotek.

Zdroje náhledů fotek

Exif segment může obsahovat vygenerovaný náhled fotky. Velikost náhledu je limitovaná specifikací JPEG formátu. Ten stanovuje maximální velikost JPEG segmentu. Náhled nemůže být větší než $2^{16} - 2$ bytů. Prakticky musí být mnohem menší, protože se do Exif segmentu musí vejít ještě další metadata. Přesná velikost náhledů není stanovená a soubor ani žádný náhled nemusí obsahovat. [2].

Pokud není vygenerovaný náhled dostatečně velký, program musí přečíst celý JPEG soubor a vygenerovat nový náhled. K tomu v C# existuje velké množství knihoven.

`System.Drawing` (GDI+)

Standardní knihovna `System.Drawing` je na Windows obálkou nad knihovnou GDI+. Protože jde o standardní součást .NET frameworku, existuje pro ni dobrá dokumentace. Nabízí také jednoduché rozhraní a snadno se používá. Nevýhodou je, že rozhraní pro kreslení a změnu velikosti fotek používá jeden zámek, i když použijeme izovolané objekty. Generování náhledů tak nejde dělat paralelně a navíc bude blokovat vykreslování uživatelského rozhraní.

ImageSharp

ImageSharp [27] je knihovna napsaná v C#, která umí načítat fotky v různých formátech. Implementuje také velké množství algoritmů pro manipulaci s obrázky. Mezi nimi jsou i algoritmy pro změnu velikosti. Díky tomu, že knihovna nezávisí na nativním kódu, je snadno přenositelná. Má také dobrou dokumentaci s velkým množstvím příkladů. Nevýhodou je nižší výkon generování náhledů v porovnání s ostatními knihovnami (viz 3.3).

Magick.NET

Magick.NET [28] je obálka pro populární knihovnu ImageMagick. Výhodou knihovny je podpora velkého množství formátů. Nevýhodou je závislost na nativní knihovně.

SkiaSharp

SkiaSharp [29] je binding pro knihovnu Skia. Skia je sice nativní knihovna, ale SkiaSharp podporuje řadu platforem a operačních systémů. Je součástí projektu mono, má dokumentované a jednoduché rozhraní a jde jednoduše použít společně se `System.Drawing`. V rychlosti načítání náhledů předčila všechny doposud zmíněné knihovny (viz 3.3). Nevýhodou je větší velikost zkompilované knihovny Skia.

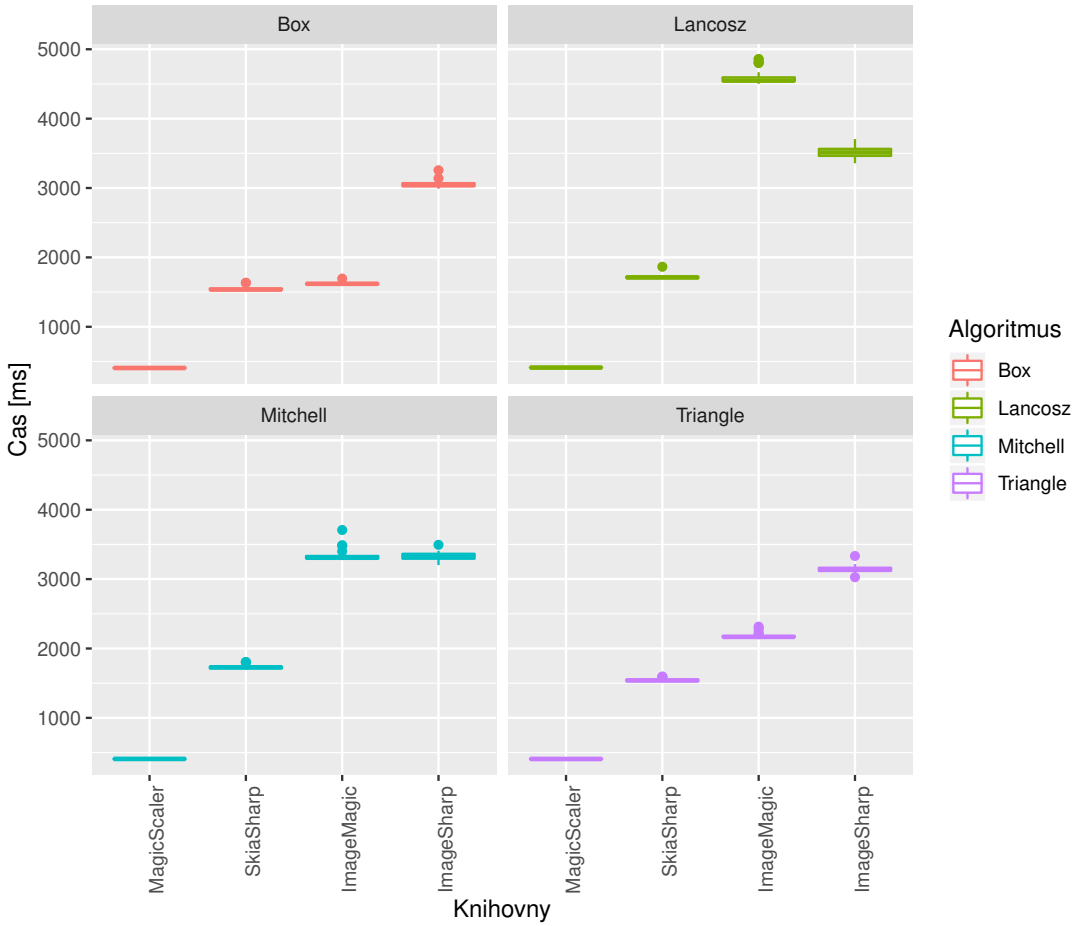
MagicScaler

MagicScaler [30] používá pro zpracování fotek Windows Imaging Component. [31] Načítání JPEG souborů zvládá podstatně lépe než ostatní knihovny (viz 3.3). Nevýhodou je omezené rozhraní a podpora pouze Windows 7 a novější.

Porovnání knihoven

Efektivitu knihoven pro generování náhledů fotek můžeme vyzkoušet. Test před začátkem měření načte 30 JPEG souborů průměrné velikosti 5,73 MiB do paměti. Každý test přečte zakódovaná data a vytvoří novou fotku velikosti 100×133 pixelů. Pro zpracování fotek používá 8 vláken. Test pro každou knihovnu a algoritmus provedl 70 měření. Graf 3.3 ukazuje výsledky posledních 50 měření pro čtyři různé algoritmy. Tyto algoritmy jsme vybrali, protože je implementují všechny testované knihovny. Test byl spuštěn na počítači s procesorem Intel Core i7-4790 @ 3.60 GHz s 8 GB RAM.

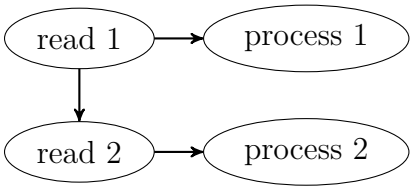
Náhledy nejrychleji vygenerovala knihovna MagicScaler (viz 3.3). Pro vygenerování náhledů jsme se proto rozhodli použít tuto knihovnu.



Obrázek 3.3: Dekódování a změna velikosti 30 JPEG souborů.

Generování náhldů

Generování náhledů fotek má dvě fáze: přečtení zakódovaných dat ze souboru a změna velikosti fotky. Jednovláknová implementace, která používá blokující čtení z disku, nemůže efektivně využít dostupný výkon systému.



Obrázek 3.4: Graf závislostí tasků pro zpracování náhledů. `read` je task, který čte data ze souboru. `process` je task, který generuje náhledy z už načtených fotek.

Program může skrýt latenci přístupu k disku použitím zřetězení. První fáze používá sdílený zdroj (disk) a je sériová. Druhá fáze dekóduje načtená data a generuje náhledy sobuorů. Pro úkoly ve druhé fázi je možné použít více vláken, protože jsou jednotlivé soubory nezávislé. Tento mechanizmus jde implementovat pomocí C# tasků (viz graf závislostí 3.4)

Načítání souborů může být rychlejší než zpracování fotek. V takovém případě popsaný postup využije dostupná jádra procesoru. Pokud je ale načítání dat z disku výrazně rychlejší než zpracování fotek, program alokuje velké množství paměti. Tato paměť bude dlouho čekat ve frontě než se nějaké vlákno dostane k jejímu zpracování. Načítání souborů je tak nutné omezit.

Postupné načítání

Ve výsledku dotazu mohou být tisíce souborů. Uživatele nemusí všechny soubory zajímat. V nejhorším případě se hned přesune do jiného adresáře. Kdyby program začal načítat všechny soubory najednou, musel by v takovém případě výpočet zastavit a všechny doposud zpracované náhledy zahodit. Fotky navíc zabírají poměrně hodně paměti.

Alternativní způsob je načítat náhledy až když jsou potřeba. Čtení souboru začne nejdříve tehdy, když se fotka dostane do viditelné části okna. Problémem je, že zpracování náhledů může být pomalejší než vykreslení uživatelského rozhraní. Program by měl zpracovat nejprve náhledy fotek, které jsou viditelné. Řešením je uložit si požadavky o načtení náhledů do nějaké datové struktury, která se sama přeuspořádá podle toho, jaká část okna je zrovna viditelná.

Nejjednoduší implementace takové struktury je pole. Soubory se zpracovávají od posledního prvku a program bude mít funkci, která položky přesune na konec pole. Tato funkce se může pravidelně volat pro soubory, které jsou zrovna viditelné. Vždy se tak vybere nedávno zobrazený soubor. Problémem je časová složitost funkce, která přesune soubor na konec pole. Ta je v nejhorším případě lineární s velikostí pole. Předpokládáme ale, že bude velikost struktury malá (desetitisíce fotek). Kromě toho jsou zobrazené fotky blízko ke konci pole, a tak funkce musí projít jen malou část dat.

3.7.3 Inkrementální obnovení výsledků dotazu

Vyhodnocení dotazu může být dlouhá operace, protože dotazy mohou procházet velkou část souborového systémů. Program by měl během vyhodnocení zobrazit částečné výsledky.

Problémem jsou dotazy s `order by` nebo `group by`. Pokud program přidá nové soubory do výsledku, musí je seřadit a seskupit. Dalším problémem je, že se soubory ve výsledku mohou dynamicky měnit ať už zásahem jiného programu nebo i změnou tagů vybraných fotek. Výsledek dotazu se tak mění i po dokončení vyhodnocení dotazu. Soubory musejí být uloženy v datové struktuře, která je seřadí a seskupí podle výrazů v dotazu.

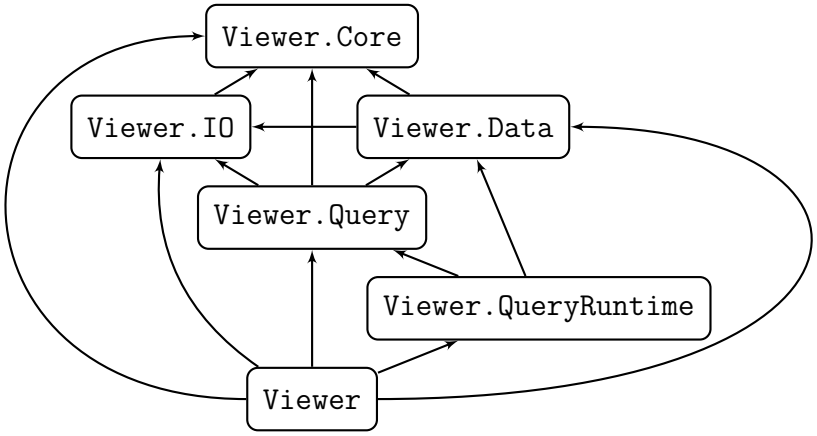
Pro seskupení a seřazení výsledků je možné použít vyvážený binární strom. Implementaci stromu poskytuje přímo knihovna C# ve třídě `SortedDictionary`. Klíč, podle kterého je do této datové struktury uložená hodnota, se ale nesmí změnit. Se samotným souborem musí být uložena ještě kopie hodnoty výrazu. Každý změněný soubor je nutné ve struktuře najít, odstranit ho a znovu ho přídat s upraveným klíčem. Hodnoty výrazů mohou mít různý typ. Porovnání hodnot je tedy složitější, než porovnání celých čísel nebo řetězců.

Lepší řešení je zpracovávat změny dávkově. Program místo stromu použije seřazené pole. Vlákno, které vyhodnocuje dotaz, uloží výsledky do fronty. Požadavky ve frontě může zpracovávat jiné vlákno, aby zpracování výsledků nezdržovalo vyhodnocení dotazu. Do fronty navíc mohou ukládat požadavaky i jiná vlákna například pokud dojde ke změně v souborovém systému. Všechny změny výsledku dotazu se tak zpracují stejným mechanismem.

4. Implementace

V této kapitole popíšeme z jakých komponent se aplikace skládá a jak spolu komponenty komunikují.

4.1 Architektura aplikace



Obrázek 4.1: Závislosti modulů aplikace

Aplikace je rozdělena do několika hlavních jmenných prostorů (viz obrázek 4.1). Každý modul implementuje jednu základní funkcionalitu a je v samostatné dynamické knihovně.

- `Viewer.Core` obsahuje funkce a kolekce, které používají všechny ostatní moduly.
- `Viewer.IO` obsahuje typy pro práci se soubory a hledání souborů podle vzoru.
- `Viewer.Data` obsahuje typy a funkce pro čtení a zápis atributů v souborech.
- `Viewer.Query` obsahuje funkce pro kompilaci, vyhodnocení dotazů a IntelliSense.
- `Viewer.QueryRuntime` obsahuje implementaci základních operátorů a funkcí, které je možné použít v dotazech.
- `Viewer` uživatelské rozhraní aplikace.

Aplikace používá standardní Managed Extensibility Framework (MEF) [32] dostupný jako součást .NET frameworku pro načtení jednotlivých typů. Třídy v modulech aplikace typicky implementují veřejné rozhraní, které je označeno `Export` tagem. MEF implicitně řeší vytváření objektů a předávání závislostí.

4.1.1 Závislosti

Všechny knihovny potřebné k běhu programu jsou stažené z balíčkovacího systému NuGet. S programem je distribuovaný generátor parseru a lexeru ANTLR4. Součástí kompilace ve Visual Studiu je spuštění tohoto nástroje. K tomu je potřeba mít nainstalovanou aktuální verzi Javy.

4.2 Data

Modul `Viewer.Data` obsahuje třídy pro čtení a zápis atributů. V tomto modulu jsou také definovány typy reprezentující tagy a soubory.

4.2.1 Reprezentace dat v programu

Hodnoty

Tagy souborů jsou reprezenotvány třídou `Attribute`. Ta obsahuje jméno tagu, zdroj (metadata nebo uživatelské atributy) a hodnotu. Hodnoty jsou třídy odvozené od `BaseValue`.

Soubory

Všechny soubory jsou v progrmau reprezentovány objektem typu `IEntity`. Entita je kolekce atributů (typ `Viewer.Data.Attribute`). Aplikace používá dva typy entit: `FileEntity`, který reprezentuje soubory, a `DirectoryEntity`, který reprezentuje složky na disku. Složky jsou pouze pro čtení. Každá entita má cestu k souboru, která jako oddělovač adresářů používá právě znak `/`. Program v cestách souborů nerozlišuje velikost písmen.

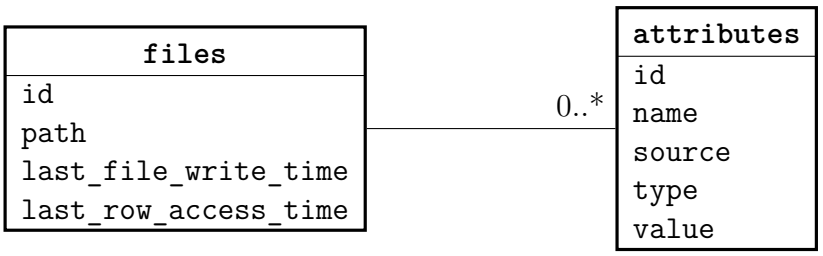
4.2.2 Uložiště tagů

Místo pro uložení tagů je reprezentované rozhranním `IAttributeStorage`. V programu jde uložit tagy do souborů (`FileSystemAttributeStorage`) nebo do SQLite databáze (`SqliteAttributeStorage`).

SQLite uložiště

SQLite uložiště tagů implementuje rozhraní `IDeferredAttributeStorage`. Všechny operace, které zapisují do databáze, nemusí být vykonány hned při zavolání metody. Místo toho si je implementace může uložit do fronty a aplikovat všechny změny najednou až při zavolání metody `ApplyChanges`. Operace zápisu jsou neblokující a jsou provedeny v jedné transakci najednou. Režie spojená se zápisem do databáze tak není obsažena v každé operaci zápisu.

Schéma databáze obsahuje dvě tabulky (viz 4.2). Obě tabulky mají jako primární klíč celočíselný sloupec `id`. V tabulce souborů je čas posledního zápisu do souboru v souborovém systému a čas posledního přístupu k záznamu v databázi. Pro tuto tabulku je vytvořený neklastrovaný index na sloupci `path` (cesta k souboru). Hodnoty ve slupci `path` se porovnávají komparátorem, který umí porovnat více bytové znaky a ignoruje velikost písmen.



Obrázek 4.2: Schéma databáze

Druhá tabulka `attributes` obsahuje hodnoty tagů souborů. Ukládají se zde metadata souboru a uživatelské tagy. Zdroj tagu rozlišuje sloupec `source`, jehož hodnota je celé číslo (viz výčtový typ `AttributeSource` v progrmau). Hodnota (sloupec `value`) je typu BLOB. Formát hodnoty je daný slupcem `type`, který obsahuje jednu z hodnot výčtového typu `TypeId` (viz následující seznam).

- `Integer`, `Real` a `String` jsou uložené jako textové řetězce v kódování databáze (UTF-8) bez nulového znaku na konci.
- `DateTime` je uložený jako textový řetězec v kódování databáze (UTF-8) ve formátu `yyyy-MM-ddTHH:mm:ss.szzz`. Formát data a času pro uložení do souboru je dostupný jako konstanta v kódu `DateTimeValue.Format`
- `Image` je obrázek (náhled fotky) uložený jako zakódovaný JPEG.

Pro porovnání jména se používá `StringComparer.InvariantCulture`. Nad tabulkou `attributes` je vytvořený neklustrovaný index se sloupci `source, name`. Tento index se používá pro hledání uživatelských tagů podle jména a pro napovídání jmen známých atributů v editoru dotazů.

Přístup k databázi

Pro přístupu k databázi je potřeba vytvořit databázové spojení. To vytváří exportovaná třída `SQLiteConnectionFactory`, jejíž metoda Create je thread-safe. Vytvoření prvního spojení přečte inicializační SQL skript `structure.sql`, rozdělí dotazy pomocí oddělovače `----` (4 pomlčky) a jednodlivé dotazy provede.

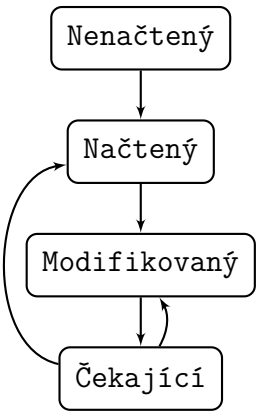
Verze programu se mohou lišit strukturou databáze. Ve SQLite existuje celočíselná pragma `user_version`, jejíž hodnotu SQLite nepoužívá. První spojení proto porovnává hodnotu `user_version` s aktuální verzí schématu. Pokud se verze liší, vytvoří se nový prázdný databázový soubor. Smazání databáze neovlivní správnost programu, protože se indexový soubor používá pouze jako cache dat ze souborového systému.

Kombinované uložiště

Speciální typ `CachedAttributeStorage` kombinuje dvě implementace uložiště tagů. Jednu používá jako primární a druhou jako cache. Při každém načtení souboru se nejdřív podívá, jestli je záznam v cache. Pokud záznam v cache nenajde, načte soubor z primárního uložiště a asynchronně uloží všechny tagy v cache. Aplikaci změn provádí v jiném vlákně, pokud interní fronta požadavků přesáhne určitou velikost nebo dojde čas.

Tento typ je exportovaný jako implementace `IAttributeStorage`.

Stav souborů



Obrázek 4.3: Stav souborů v programu

Načtené soubory jsou uloženy v implementaci typu `IEntityManager`. Soubory mohou být v několika stavech (viz 4.3). `Nenačtený` soubor ještě není v paměti programu. Program může mít pro každou entitu dvě různé verze. Jednu ve stavu `Načtený` nebo `Modifikovaný` a druhou ve stavu `Čekající`.

Soubor jde přesunout ze stavu `Načtený` do stavu `Modifikovaný` zavoláním metody `SetEntity`. Tato metoda vytvoří kopii objektu a vloží ji do seznamu upravených souborů. Další volání metody se stejným souborem nahradí kopii v seznamu upravených souborů. Pokud daný soubor ještě není v tomto seznamu, uloží se jeho další kopie jako původní stav objektu.

Zavoláním metody `GetModified` se přesunou všechny soubory ze seznamu upravených sobuorů (stav `Modifikovaný`) do stavu `Čekající`. Na vrácených objektech *musí* být zavolaná jedna z metod `Save`, `Revert` nebo `Return`.

- `Save` uloží změny v této kopii do souboru.
- `Revert` vrátí entitu ve stavu `Načtený` do původního stavu (stav objektu při prvním volání `SetEntity`).
- `Return` neuloží změny do souboru a přesune je do stavu `Modifikovaný`. Pokud už je ve stavu `Modifikovaný` jiná entita stejného souboru, neprovede se žádná změna. Aktuální verze ve stavu `Čekající` se zahodí.

4.3 Dotazy

Většina typů pro práci s dotazy je v modulu `Viewer.Query`. Velká část veřejného rozhraní celého modulu je tvořena pouze dvěma fasádami: `IQueryCompiler`, která zařizuje kompilaci dotazů, a `IQuerySuggestions` pro výpočet možných pokračování zadané části dotazu.

4.3.1 Kompilace dotazů

Kompilátor používá parser a lexer vygenerovaný nástrojem ANTLR4 z gramatik `QueryParser.g4`, resp. `QueryLexer.g4`. Dotaz se kompilací převede na immutable strom (třídy ve jmenném prostoru `Viewer.Query.Execution`). Vnitřní uzly

stromu jsou množinové operátory, listy jsou jednoduché dotazy (`select`, `where`, `order by`, `group by`). Výrazy v dotazech se kompilují na immutable strom výrazů typu `ValueExpression`. Typ výrazů se moc nemění a program nad nimi spouští různé operace (vyhodnocení, optimalizace apod.). Manipulace se stromem je proto implemnetovaná návrhovým vzorem visitor.

Výsledkem kompilace je objekt typu `IExecutableQuery` (kořen stromu). Složitější struktura dotazu a výrazů je před uživatelem skryta. Má k dispozici pouze zkompilované metody a komparátor pro seřazení výsledků.

Všechny dostupné pohledy jsou v objektu `IQueryViewRepository`. Pohledy je možné přidávat za běhu. Hledání funkcí, implicitní konverze argumentů a volání funkcí a operátorů zajišťuje implementace rozhraní `IRuntime`.

Detekce chyb

Pro detekci kompilačních a běhových chyb je nutné implementovat a exportovat rozhraní `IQueryErrorListener`. Alternativně lze implementaci tohoto rozhraní předat kompilátoru pro každý dotaz zvlášť. Dotazy nevyhazují výjimky při kompilaci ani vyhodnocení, protože se vyhodnocují na vyžádání až když jsou potřeba.

4.3.2 Přidávání funkcí

Vlastní funkci použitelnou ve `where`, `order by` nebo `group by` části dotazu jde přidat implemnetací rozhraní `IFunction` v C#. Jméno funkce je dané vlastností `Name`. Typy argumentů funkce jsou dané vlastností `Arguments`. Jméno funkce ani argumenty se nesmí měnit. Metoda `Call` dostane objekt `IExecutionContext`, který obsahuje argumenty funkce. Počet předaných argumentů bude vždy stejný jako velikost pole `Arguments`. Implementace by ale měla počítat s možností, že je hodnota některého parameteru `null`. Pokud se funkce dostane do chybového stavu, musí zavolat metodu `Error` na objektu typu `IExecutionContext` a vrátit návratovou hodnotu této metody (viz ukázka implementace funkce `my(string, int)` 4.1)

```
[Export(typeof(IFunction))]
public class MyFunction : IFunction
{
    public string Name => "my";
    public IReadOnlyList<TypeId> Arguments { get; }
        = new[]{ TypeId.String, TypeId.Integer };
    public BaseValue Call(IExecutionContext ctx)
    {
        if (ctx[0].IsNull || ctx[1].IsNull)
            return ctx.Error("Parameters must not be null");
        return new StringValue($"{ctx[0]} ({ctx[1]})");
    }
}
```

Listing 4.1: Ukázka implementace vlastní funkce `my`

4.3.3 Vyhodnocení dotazu

Objekt typu `IExecutableQuery` jde použít pro vyhodnocení dotazu. Metoda `Execute` vrací iterátor. Výsledky dotazu jsou nesetřízené a neseskupené, aby šlo vyhodnocování implementovat efektivně. Kompilátor zkompiluje výrazy v `order by` na C# komparátor (vlastnost `Comparer`) a výraz v `group by` na metodu `GetGroup`. Vyhodnocení probíhá v jednom vlákně a je blokující. Dotazu je možné předat `CancellationToken` určený k zastavení hledání a `IProgress` pro sledování průběhu operace.

Metoda `Match` implementuje algoritmus 2. Parametrem metody je objekt typu `IEntity` a metoda vrátí `true` právě tehdy, když je zadaný objekt ve výsledku dotazu.

4.3.4 IntelliSense

Automatické dokončování funguje ve dvou fázích. První najde kontext pro zadanou pozici kurzoru v dotazu. Kontext je cesta v derivačním stromě a množina typů tokenů, které mohou následovat (viz třída `SuggestionState`). K získání kontextu se používá rozhraní `IStateCollector`.

Druhá fáze podle kontextu vygeneruje možná pokračování dotazu na zadané pozici. Třídy, které generují nápovědy, musejí implementovat a exportovat rozhraní `ISuggestionProviderFactory`.

Nápověda je reprezentovaná rozhraním `IQuerySuggestion`. Jde pouze o popis operace, která se má vykonat nad předem daným dotazem. Změny se provedou až zavoláním metody `Apply`, která vrací nový immutable stav editoru (text a pozici kurzoru).

Vytvoření všech objektů je kvůli rozšiřitelnosti implementace složitější. Proto jde použít fasádu `IQuerySuggestions`, která vytvoří všechny potřebné objekty a prováže je mezi sebou. Metoda `Compute` na vstupu dostane text dotazu a pozici kurzoru a na výstupu vrátí seznam možných pokračování dotazu.

4.4 Uživatelské rozhraní

Modul `Viewer` obsahuje aplikaci, která vytváří uživatelské rozhraní pro třídy z ostatních modulů. Používá k tomu knihovnu `Windows.Forms` a `DockPanelSuite`. [33]

4.4.1 Komponenty

Celý modul je rozdělený do komponent. Každá komponenta typicky představuje jedno okno v aplikaci, ale mohou existovat i komponenty bez uživatelkského rozhraní. Komponentou je každá třída, která implementuje a exportuje rozhraní `IComponent`.

Komponenty s uživatelským rozhraním používají vzor Model View Presenter (MVP). View je Windows Forms komponenta implementující `IView`. Stav pohledu mění presenter (třída odvozená od `Presenter<TView>`) voláním metod pohledu v obsluze událostí. Presenter může automaticky zaregistrovat obsluhu všech událostí pohledu zavoláním metody `SubscribeTo(pohled, prefix)`. Pro

každou událost s názvem `udalost` objektu `pohled` se metoda podívá, jestli je v presentru metoda s názvem `prefix_udalost`. Pokud taková metoda existuje, zaregistruje ji jako obsluhu události pohledu. Odstranění obsluhy všech událostí pohledu se udělá automaticky v metodě `Dispose` presenteru.

Knihovna DockPanelSuite serializuje okna do XML dokumentu. Každé okno má metodu `GetPersistString`. Vrácený řetězec se uloží do stejného XML. Rozhraní `IViewerApplication` má metodu `AddLayoutDeserializeCallback`, která umožňuje komponentám přidat deserializační funkci. Parametrem funkce je uložený řetězec a vrací vytvořené okno. Komponenty by měly do řetězce ukládat své jméno. Deserializační funkce musí vrátit hodnotu `null`, pokud toto jméno v předaném řetězci není. Program při deserializaci pro každé okno spustí zaregistrované funkce a vrátí první hodnotu, která není `null`.

4.4.2 Zobrazení postupu dlouhé operace

Komponenta `Viewer.UI.Tasks` vytváří uživatelské rozhraní pro zobrazení postupu dlouhých operací. Uživatel může pomocí tohoto rozhraní operace zrušit. Pro vytvoření nové operace je nutné použít exporotovanou implementaci rozhraní `ITaskLoader`, která vytvoří objekt typu `IProgressController`. Tento objekt má vlastnost `TotalTaskCount` (celkový počet úkolů), který se může měnit v průběhu operace, a metodu `Report`, která dokončí jeden úkol a je thread-safe.

4.4.3 Strom adresářů

Komponenta `Viewer.UI.Explorer` vytváří okno se stromem adresářů. Pro dlouhé operace se souborovým systémem (kopírování a přesouvání souborů) implementuje a exportuje rozhraní `IExplorer`. Metody tohoto rozhraní provádí operace v jiném vlákně a zobrazují postup pomocí komponenty 4.4.2.

Serializační řetězec je `Viewer.UI.Explorer.DirectoryTreeView`.

4.4.4 Výsledek dotazu

Komponenta `Viewer.UI.Images` vykonává dotazy, řeší efektivní zobrazení výsledků dotazu, načítání náhledů fotek a operace se zobrazenými soubory. Serializační řetězec je `Viewer.UI.Images.ImagesView;<dotaz>`, kde `<dotaz>` je aktuální dotaz.

Vyhodnocení dotazu

Vyhodnocení dotazu provádí třída `QueryEvaluator`. Metoda `RunAsync` vyhodnotí dotaz v jiném vlákně. Nalezené soubory se uloží do fronty. Uživatelské rozhraní může obnovit výsledky dotazu zavoláním metody `Update`. Třída má dvě kolekce výsledků. První kolekci upravuje pouze metoda `Update`, a je tak bezpečné ji použít v uživatelském rozhraní. Druhá kolekce se používá pro zpracování změn výsledku dotazu v jiném vlákně. Všechny změny v druhé kolekci se zkopírují do první kolekce během volání `Update`.

Třída `QueryEvaluator` sleduje změny v souborovém systému a v aplikaci pomocí událostí `IEntityManager`. Změněné soubory ukládá do stejné fronty jako při vyhodnocení dotazu. Program je tak zpracuje stejným mechanismem.

Náhledy souborů

Náhled souboru je reprezentovaný rozhraním `ILazyThumbnail`. Náhledy vytváří implementace `ILazyThumbnailFactory`. V programu jsou dvě implementace `PhotoThumbnail` a `DirectoryThumbnail` pro načtení náhledu fotky, resp. adresáře. Tovární třída vybere implementaci podle typu předané entity.

Každé volání metody `GetCurrent` zkontroluje, jestli je načtený náhled v dostatečně velkém rozlišení. Pokud takový náhled není načtený, použije se exportovaná implementace `IThumbnailLoader` pro vygenerování nového náhledu. Generování náhledů se provádí asynchroně v jiném vlákně. Objekt si pamatuje poslední načtený náhled a ten vrací, dokud neskončí zpracování náhledu s lepším rozlišením.

4.4.5 Editor dotazů

Komponenta `Viewer.UI.QueryEditor` obsahuje editor dotazů. Po spuštění programu načte pohledy z disku a přidá je do kolekce `IQueryViewRepository`. Změny v adresáři pohledů monitoruje a mění kolekci pohledů. Všechna okna editoru spravuje exportované rozhraní `IEditor`. Aplikace používá knihovnu Scintilla.NET [34] pro komponentu editoru.

Serializační řetězec editoru má podobu:

```
Viewer.UI.QueryEditor.QueryEditorView;<dotaz>;<cesta>
```

`<dotaz>` je obsah editoru a `<cesta>` je cesta k souboru nebo prázdný řetězec, pokud otevřený dotaz není uložený v žádném souboru.

IntelliSense v editoru používá rozhraní `IQuerySuggestions`. Nápovědy se generují v jiném vlákně při každé změně obsahu. Editor si ukládá verzi aktuálního stavu (celé číslo). Vygenerované nápovědy si pamatují, pro jakou verzi byly vytvořeny. Komponenta zobrazuje pouze nápovědy pro aktuální verzi.

4.4.6 Editor atributů

Komponenta `Viewer.UI.Attributes` vytváří rozhraní pro editaci a uložení tagů. V aplikaci jsou dva druhy oken editoru atributů (metadata a uživatelské atributy). Serializační řetězce jsou v následujícím seznamu:

1. `Viewer.UI.Attributes.AttributeTableView;exif`: okno s metadaty souboru
2. `Viewer.UI.Attributes.AttributeTableView;attributes`: okno s uživatelskými atributy.

Editor atributů pro ukládání tagů poskytuje rozhraní `ISaveQueue`. Soubory se ukládají do fronty zavoláním metody `SaveAsync`. Postup operace se zobrazuje pomocí komponenty 4.4.2.

4.4.7 Prezentace

Komponenta `Viewer.UI.Presentation` umí zobrazit výsledky dotazu jako prezentaci jednotlivých fotek. Načítání fotek může být dlouhá operace. Komponenta proto implementuje třídu `ImageWindow`, která načte několik souborů kolem zobrazené fotky.

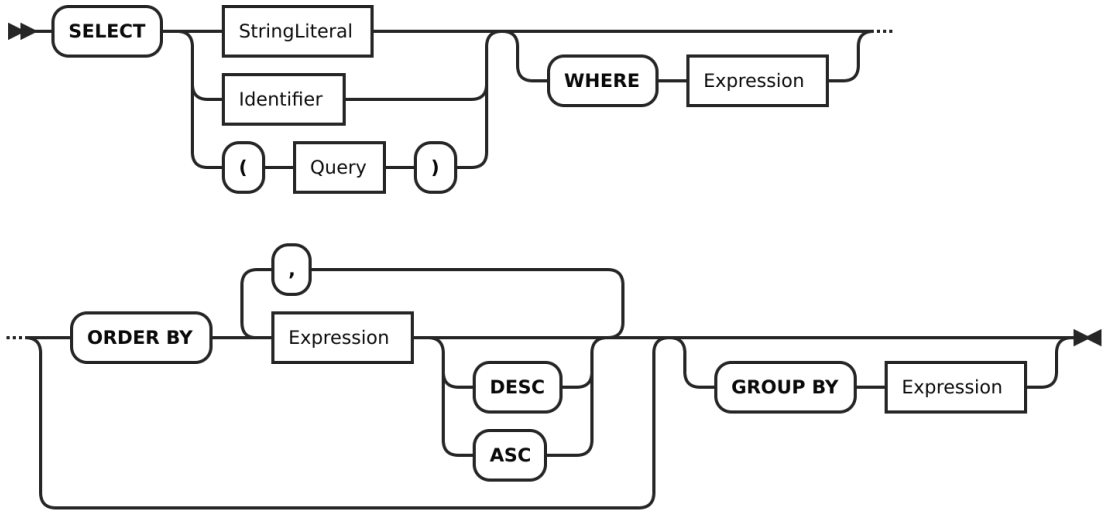
Prezentaci fotek jde otevřít pomocí exportovaného rozhraní `IPresentation`.

4.4.8 Sdílený stav

Komponenty mají přístup k historii dotazů (exportovaný typ `IQueryHistory`). Dotazy se v programu vyhodnocují právě zavoláním metody `ExecuteQuery`. Tato metoda přidá dotaz do historie a zavolá událost `QueryExecuted`. Komponenta 4.4.4 na tuto událost reaguje vyhodnocením dotazu. Komponenta 4.4.3 na událost reaguje otevřením složky ve stromě adresářů.

Všechny vybrané soubory a adresáře jsou ve sdílené kolekci `ISelection`. Událsot `Changed` je zavolaná vždy, když dojde ke změně výběru. Komponenta 4.4.6 na změnu reaguje zobrazením tagů vybraných souborů.

5. Dotazovací jazyk



Obrázek 5.1: Syntaxe jednoduchého dotazu `SimpleQuery`

Diagram 5.1 ukazuje strukturu jednoduchých dotazů bez množinových spojek. Terminály jsou v blocích se zakulacenými rohy. U klíčových slov nezáleží na velikosti písmen.

5.1 Select

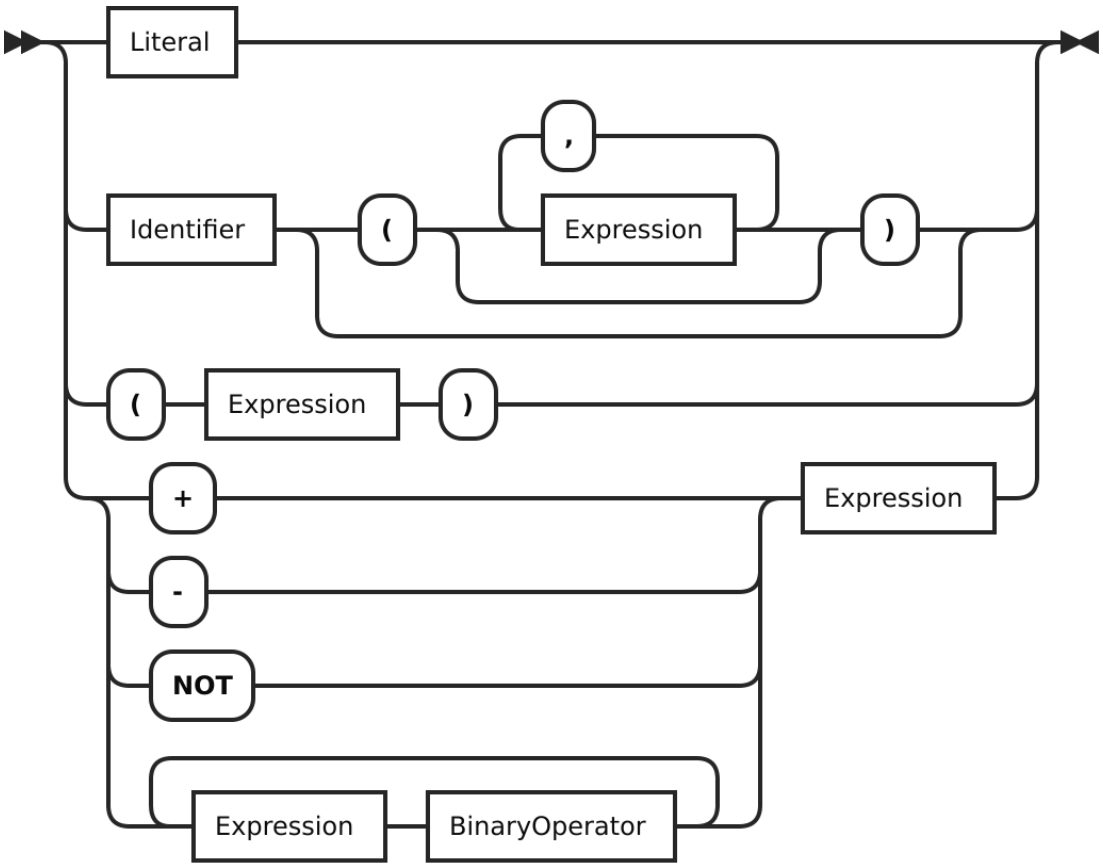
`select` část dotazu určuje, jaké soubory se mají prohledat.

1. `StringLiteral`: vzor cesty k adresáři jako textový řetězec mezi uvozovkami (např. `"adresář"` nebo `"D:/fotky/**/2019/**"`)
2. `Identifier`: identifikátor pohledu (např. `dovolená`). Pohled je dotaz uložený v souboru v adresáři `%userprofile%/Documents/Viewer/Views`. Identifikátor pohledu je jméno souboru *bez přípony* `.vql`. Pokud identifikátor obsahuje mezery nebo jiné bílé znaky, je nutné použít zpětné uvozovky ‘ (např. ‘`složitý pohled`‘). Nový dotaz bude prohledávat pouze soubory ze zadaného pohledu.

   Pohled může být libovolně složitý dotaz. Užitečným pohledem jsou často používané vzory cest.
3. `Query`: poddotaz uzavřený v kulatých závorkách (např. `select (select "D:/photos/**"where city)`). Vnořený dotaz se vyhodnocuje stejně jako pohled.

5.2 Where

Pro hledání souborů podle tagů se dá použít volitelná část `where`. Za klíčovým slovem následuje výraz, jehož strukturu ukazuje diagram 5.2. Program nepracuje



Obrázek 5.2: Syntaxe výrazu `Expression`

s datovým typem `bool`. Místo toho mohou všechny datové typy nabývat hodnoty `NULL`. Jméno tagu se vyhodnotí jako hodnota tagu v souboru nebo `NULL`, pokud soubor zadaný tag neobsahuje. Pokud se pro danou fotku výraz ve `where` částí vyhodnotí na `NULL`, nebude tento soubor zahrnut ve výsledku dotazu.

- `Literal`: Literálem se rozumí konstantní hodnota. To může být buď textový řetězec uzavřený do uvozovek, celé číslo, nebo číslo s desetinnou tečkou. Příklady těchto konstantních hodnot jsou `"hodnota"`, `12` nebo `3.14159`.
- `Identifier`: Identifikátor představuje buď jméno tagu v souboru, nebo jméno funkce, která se má zavolat. Pokud identifikátor obsahuje mezery nebo jiné bílé znaky, musí být uzavřený do zpětných uvozovek ‘ (např. ‘`dovolená 2019`‘). Identifikátory bez bílých znaků lze zapsat mimo uvozorvky (např. `hrad`). Pokud za identifikátorem následuje kulatá závorka, jedná se o volání funkce. Například `date(DateTaken)` zavolá funkci se jménem `date` a časem vytvoření dané fotky (hodnotou atributu jménem `DateTaken`).
- `BinaryOperator`: V dotazu lze použít řadu binárních operátorů. Jejich úplný výčet je v následujícím seznamu. Operátory jsou v seznamu zapsané s klesající prioritou. Pokud není uvedeno jinak, operátory na stejné řádce mají stejnou prioritu a jsou levě asociativní. Jako první se tedy vždy aplikuje nejlevější výskyt operátoru.

1. `*, /`: Násobení a dělení.
2. `+, -`: Sčítání a odčítání.
3. `<, <=, ==, =<, !=, <>, >=, >` Porovnávací operátory nejsou asociativní. Porovnat lze najednou právě 2 výrazy. Pro porovnání rovnosti, resp. nerovnosti, jdou použít oba časté způsoby zápisu: `==`, `=`, resp. `!=`, `<>`. Tyto Možnosti se nijak neliší.
4. `AND` logický operátor konjunkce (a zároveň)
5. `OR` logický operátor disjunkce (nebo)

Hodnoty použité jako parametry porovnávacích a aritmetických operací mohou být `NULL`. V takovém případě se celý operátor vyhodnotí jako hodnota typu `NULL`. Například výraz `a != 3` **není** pravdivý pro soubory bez atributu jménem `a`. Pokud takové soubory mají být ve výsledku dotazu, je nutné výraz upravit na `not a or a != 3`.

5.3 Seřazení výsledků

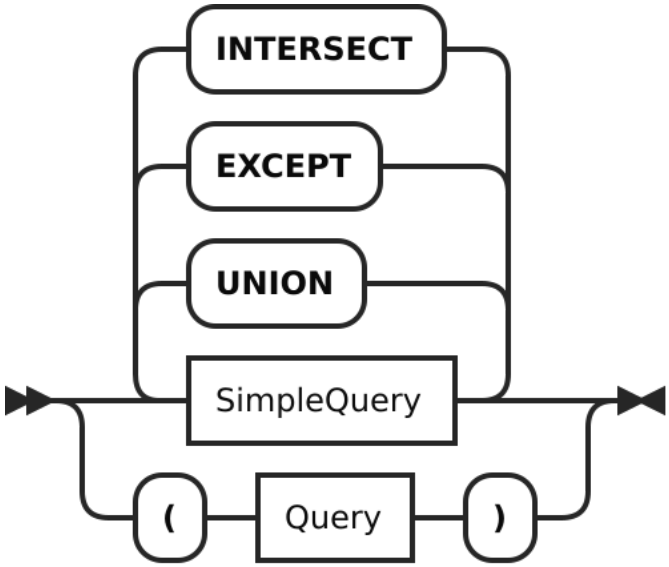
Pro seřazení výsledků dotazu jde použít volitelnou část `ORDER BY`. Pořadí je dané seznamem výrazů. Za každým výrazem je voliteně typ uspořádání (`asc` vzestupně - výchozí, `desc` sestupně). Soubory budou porovnané podle druhého a dalšího výrazu jen tehdy, pokdu je hodnota prvního výrazu pro oba soubory stejná.

- Dotaz `SELECT "D:/photos"ORDER BY DateTaken` seřadí soubory v adresáři `D:/photos` vzestupně podle času vytvoření.
- Dotaz `SELECT "D:/photos"ORDER BY FileSize DESC` seřadí soubory v adresáři `D:/photos` sestupně podle velikosti fotky.
- Dotaz `SELECT "D:/photos"ORDER BY category, DateTaken` nejprve seřadí fotky podle kategorie (uživatelem zadaný tag souboru). Všechny fotky ve stejné kategorii budou seřazeny podle času vytvoření.

5.4 Seskupení fotek

Fotky jde seskupit podle zadaného výrazu pomocí volitelného operátoru `GROUP BY`. Fotky se stejnou hodnotou výrazu budou zařazeny do stejné skupiny.

- Dotaz `SELECT "D:/photos"GROUP BY elapsed(DateTaken)` seskupí fotky podle času vytvoření.
- Dotaz `SELECT "D:/photos"GROUP BY FileSize / (1024 * 1024)` seskupí fotky podle velikosti souboru.
- Dotaz `SELECT "D:/photos/**"GROUP BY Directory` seskupí fotky podle jména adresáře.



Obrázek 5.3: Syntaxe složeného dotazu `Query`

5.5 Množinové operátory

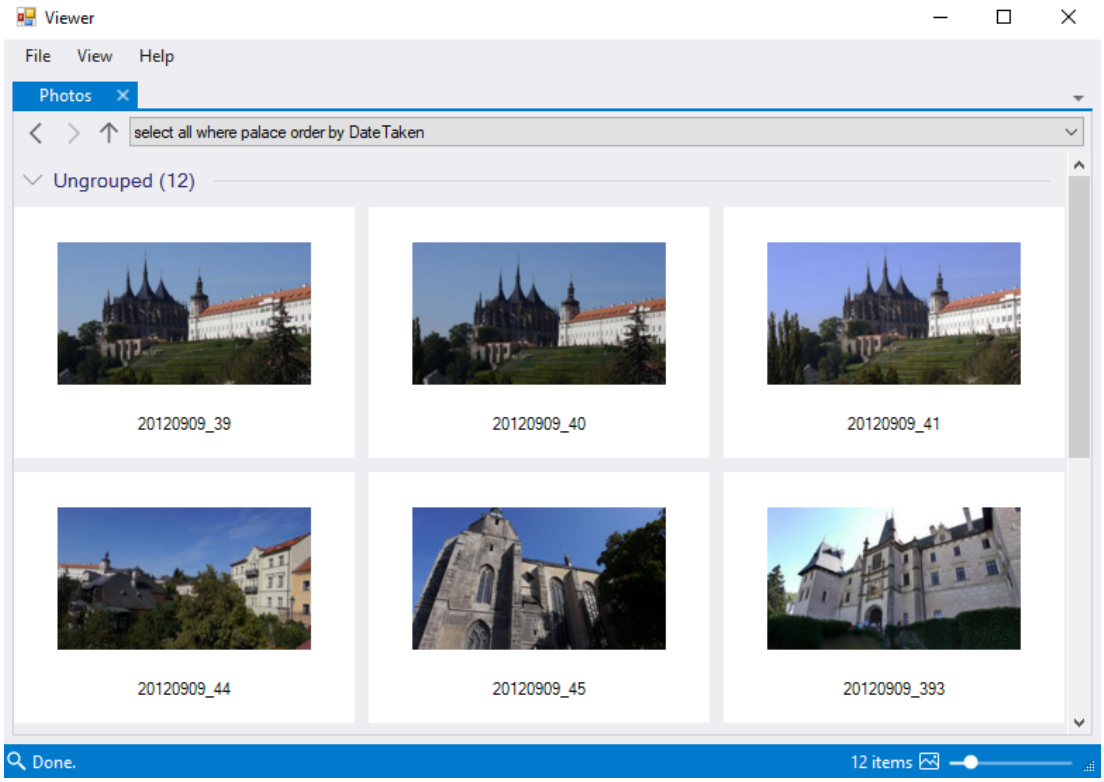
Výsledky celých dotazů je možné kombinovat množinovými operátory sjednocení, průniku a množinového rozdílu. Sytnaxi složených dotazů ukazuje diagram 5.3. Neterminál `SimpleQuery` se odkazuje na jednoduchý dotaz bez množinových operátorů, jehož strukturu jsme popsali na začátku kapitoly 5.

Všechny množinvé operátory jsou levě asociativní. Jako první se vždy aplikuje nejlevější výskyt operátoru.

6. Uživatelské rozhraní

Uživatelské rozhraní se skládá z přizpůsobitelných oken. Pozici všech oken jde změnit kliknutím na záhlaví okna a přetažením okna na požadovanou pozici. Pozice a velikost oken se automaticky ukládá při zavření programu.

6.1 Náhledy fotek



Obrázek 6.1: Výsledek dotazu

Okno se jménem `Photos` zobrazuje výsledky dotazu jako tabulku s náhledy fotek a s adresáři. Okno se automaticky otevře při spuštění dotazu z Editoru dotazů nebo dvojklikem myší na adresář v okně `Explorer`.

Ovládání je velmi podobné prohlížeči souborů. Dvojklikem na složku se zobrazí její obsah v tomto okně. Dvojklikem na fotku se otevře v plné velikosti v okně 6.2 Prezentace.

Fotky a složky lze vybrat myší nebo šipkami na klávesnici. Následující seznam poskytuje všechny akce, které je možné s výběrem souborů provést. Hodnota v závorce určuje klávesovou zkratku. Zkratky si není nutné pamatovat, protože celá nabídka akcí se zobrazí v kontextovém menu kliknutím pravým tlačítkem.

- Přejmenovat soubor (`F2`)
- Permanentně smazat všechny vybrané soubory (`Delete`)

- Zkopírovat soubory do schránky (`Ctrl + C`)
- Vyjmout soubory (`Ctrl + X`)
- Vložit soubory ze schránky (`Ctrl + V`)

Soubory se dají přesouvat a kopírovat i přesunutím myší a to i z jiné aplikace jako je například standardní prohlížeč souborů ve Windows. Výsledky v tomto okně jsou často soubory z různých adresářů. Pokud není jednoznačné, do kterého adresáře se mají soubory zkopírovat nebo přesunout, zobrazí se nabídka možných adresářů.

Posuvník v pravém dolním rohu 6.1 mění velikost náhledů fotek. Náhledy se generují postupně až když jsou potřeba. Změnou velikosti se může nejprve načíst náhled s nižší kvalitou. Pro všechny viditelné fotky se ale nakonec načte náhled v požadované velikosti.

6.1.1 Nastavení

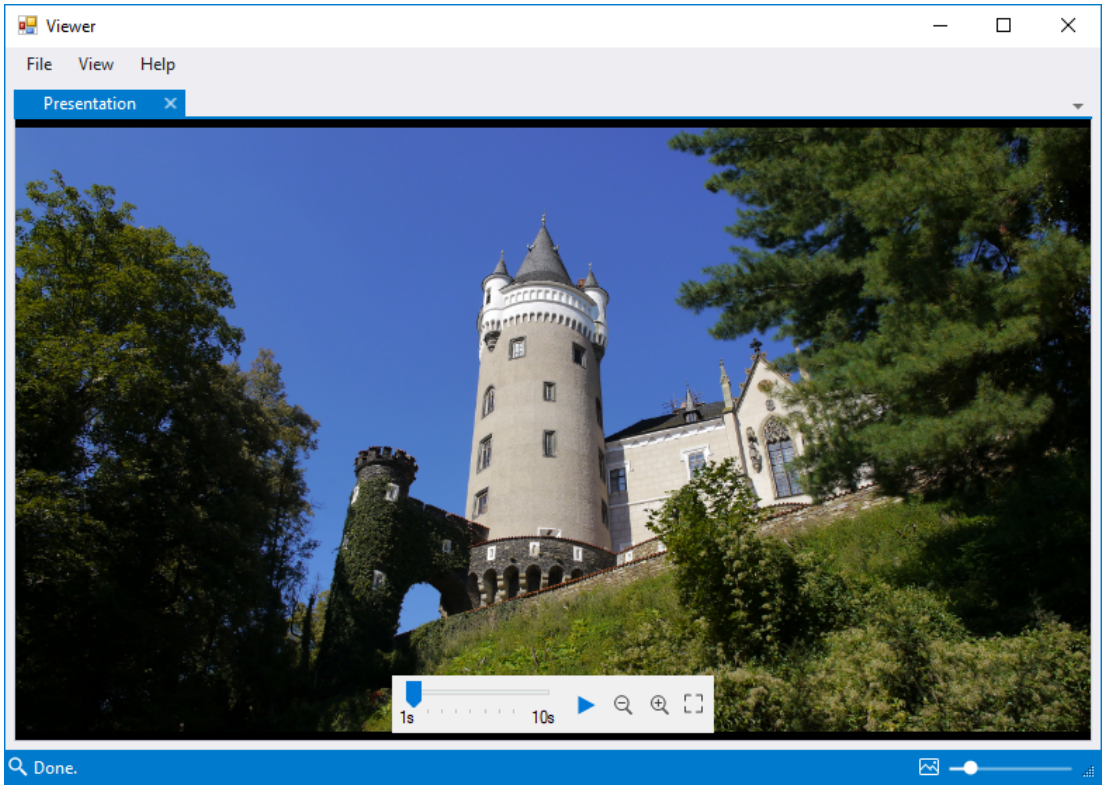
Výběr souborů je navíc možné předat libovolné aplikaci jako například editoru fotek. Úprava seznamu aplikací v kontextovém menu je možná z aplikačního menu `File` → `Settings`. V záložce `Programs` je tabulka všech programů. Editací posledního prázdného řádku se přidá nový program. Následující seznam popisuje sloupce tabulky.

- `Name` je jméno programu zobrazené v kontextovém menu
- `Command` je cesta k programu. Kliknutím na buňku v tomto sloupci se otevře dialog pro výběr cesty.
- `Arguments` jsou parametry předané programu před soubory.
- `Files` určuje, jestli jde program použít pro otevření souborů
- `Directories` určuje, jestli jde program použít pro otevření složek
- `Allow multiple paths` Pokud je tato možnost zaškrtnutá, programu se předají všechny vybrané soubory a nebo složky. Jinak se programu předá právě 1 soubor.

Nastavení se automaticky uloží změnou libovolné hodnoty. Neuloží se ale neplatné záznamy jako například program s chybějící cestou.

6.2 Prezentace

Okno prezentace se otevře dvojklikem na fotku v okně 6.1 Náhledy fotek. Prezentace postupně prochází soubory ve stejném pořadí jako jsou v tabulce náhledů. Fotky se zde zobrazují v cyklu. Po poslední fotce následuje první fotka. Posunutím kurzoru myši k dolnímu okraji okna se zobrazí panel s ovládacími prvky, kde jde nastavit rychlost prezentace jak je vidět v obrázku 6.2



Obrázek 6.2: Prezentace výsledků dotazu

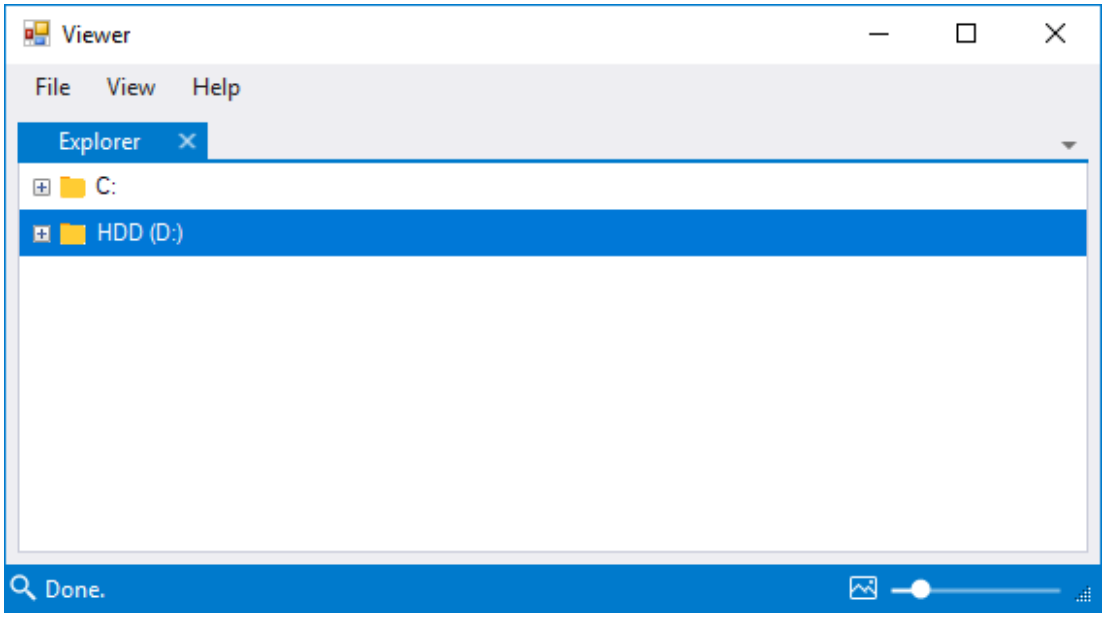
Posunutím kurzoru myši k levému, resp. prvému okraji okna se zobrazí tlačítko pro přechod k předchozí, resp. následující fotce. Většina operací má klávesovou zkratku, kterou shrnuje následující seznam.

- spuštění, zastavení prezentace (`mezerník`)
- další fotka (`pravá šipka` nebo `kolečko myši nahoru`)
- předchozí fotka (`levá šipka` nebo `kolečko myši dolů`)
- vypnutí/zapnutí fullscreen režimu (`F5` nebo `F` nebo `dvojklik myši`)
- přiblížení fotky (`+` nebo `cltr + kolečko myši nahoru`)
- oddálení fotky (`-` nebo `cltr + kolečko myši dolů`)
- výchozí přiblížení fotky (`0`)
- posun přiblížené fotky (`šipky` nebo `levé tlačítko myši`)

6.3 Strom adresářů

Okno `Explorer` zobrazuje strom adresářů pro všechny logické disky. Okno jde otevřít z aplikačního menu `View` → `Explorer`.

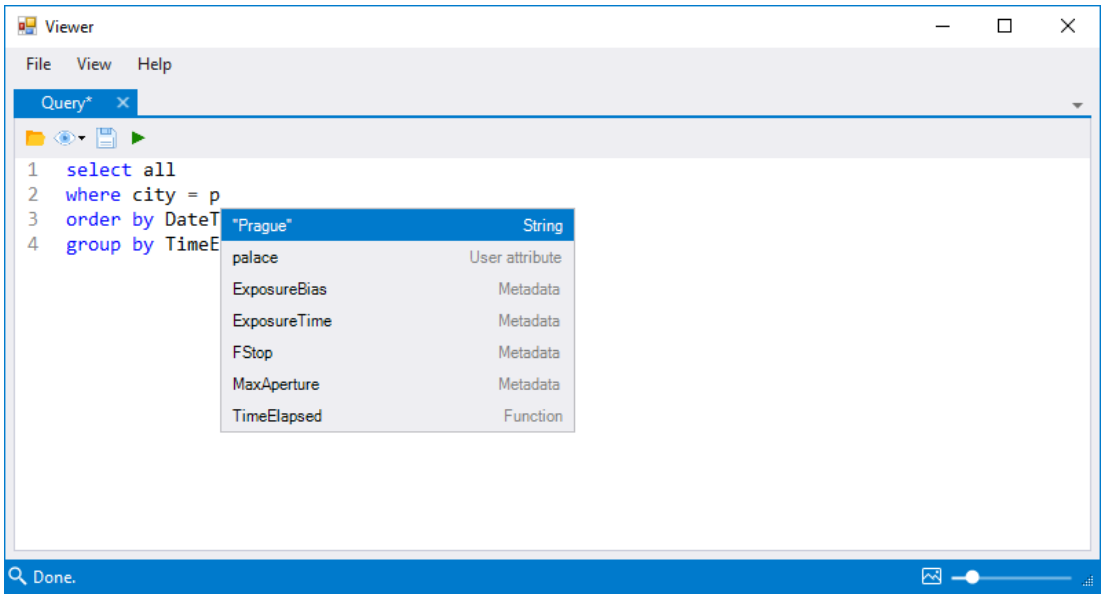
Kliknutí na adresář zobrazí všechny podsložky a spustí dotaz `SELECT "cesta k adresáři"`. Tím se zobrazí obsah složky v okně 6.1 Náhledy fotek. Adresáře



Obrázek 6.3: Strom adresářů

jde kopírovat (`Ctrl + C`), vložit soubory a složky ze schránky (`Ctrl + V`), přejmenovat (`F2`), odstranit (`Delete`) a vytvořit novou složku (`Ctrl + N`).

6.4 Editor dotazů



Obrázek 6.4: Editor dotazů

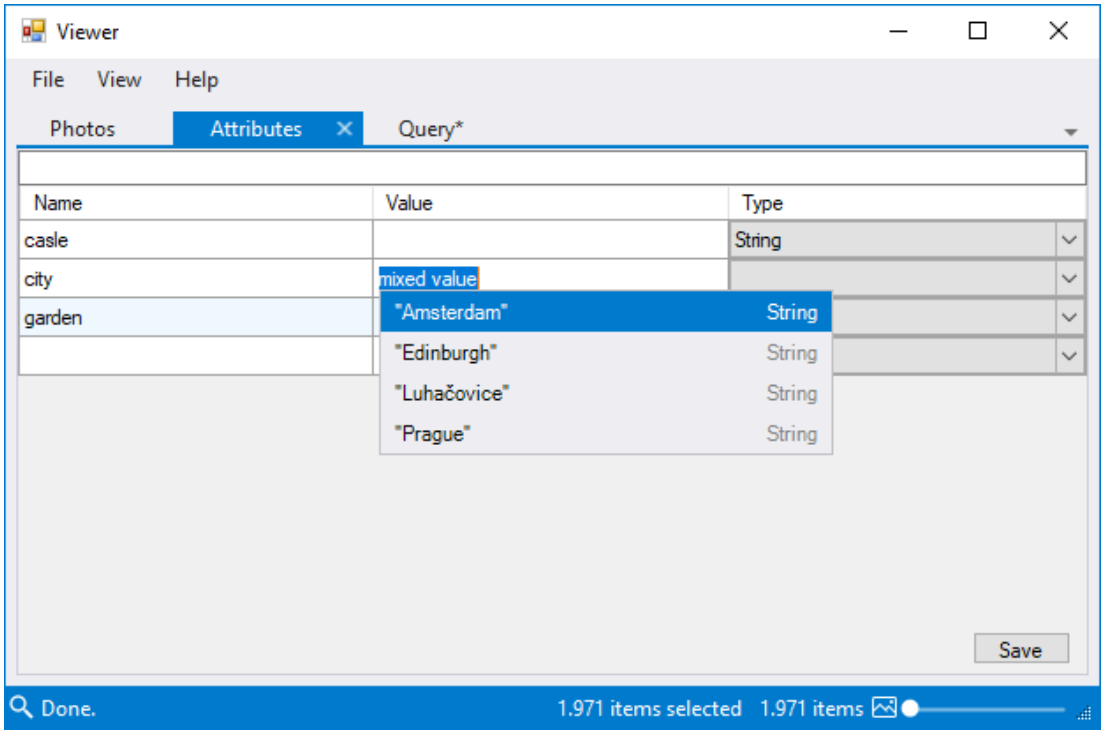
V okně editoru dotazů se dají vytvářet a upravovat dotazy. Okno se otevře z aplikačního menu `View` → `Query`.

Editor pracuje se soubory dotazů (přípona `.vql`). Klávesová zkratka `Ctrl + O` (ikona složky v levém horním rohu okna, viz 6.4) zobrazí dialog pro otevření souboru. Zkratkou `Ctrl + S` (ikona diskety v nástrojích okna) se uloží obsah editoru do souboru a zkratkou `F5` nebo `Ctrl + Enter` se dotaz spustí. Ikona oka otevře seznam dostupných pohledů. Pohledy jsou prosté dotazy uložené ve speciálním adresáři `%userprofile%/Documents/Viewer/Views`. Pro zjištění přesné cesty na konkrétním počítači jde tuto cestu zkopírovat do adresového řádku Windows File Exploreru.

Editor zvýrazňuje klíčová slova dotazu modře, číselné konstanty zeleně a textové konstanty červeně. Při psaní dotazu se snaží napovídat možná pokračování dotazu. Napovídá například klíčová slova, jména atributů, jména funkcí a hodnoty atributů. Napovídání závisí na pozici kurzoru v textu. V obrázku 6.4 je vidět, že `"Prague"` je jediná hodnota, kterou editor doporučuje jako možné pokračování. V tomto případě je totiž kurzor u písmene p, které je navíc v porovnání s atributem jménem `city`. Mezi hodnotami atributu city obsahující písmeno `p` program zatím viděl jen hodnotu `"Prague"`. Proto je to taky jediná konstantní hodnota, kterou editor nabízí. Nápovědy může uživatel procházet šipkami nahoru a dolů a přijmout je lze klávesou `Enter`. Klávesou `Escape` se okno s nápovědami zavře.

6.5 Editor atributů

Editor atributů upravuje atributy všech vybraných souborů. Okno se otevře z aplikačního menu `View` → `Attributes`. Kromě uživatelem zadaných atributů je taky možné zobrazit metadata o souboru v okně `View` → `Exif`.



Obrázek 6.5: Editor atributů

V tabulce atributů je seznam všech uživatelem zadaných atributů ve všech vybraných souborech. Světle modré řádky zvýrazňují atributy, které nejsou uložené ve všech vybraných souborech. Pokud mají alespoň 2 vybrané soubory jinou hodnotu pro ten samý atribut, bude ve sloupci `Value` šedá hodnota `mixed value`. Například v 6.5 mají všechny vybrané soubory atribut `castle`, jen některé soubory mají atribut `garden` a všechny soubory mají atribut `city`, ale jeho hodnota se mezi vybranými soubory liší. V tomto příkladu je také vidět, že editor podobně jako 6.4 Editor dotazů napovídá hodnoty a jména atributů.

Dvojklikem na buňku tabulky nebo klávesou `F2` jde hodnotu buňky upravit. Klávesou `Delete` se atribut odstraní. Pro přidání nového atributu stačí editovat poslední prázdný řádek tabulky. Změny se aplikují na všechny vybrané soubory. Po editaci je nutné změny uložit tlačitkem `Save`. Pokud se v editoru změny neuloží, program se při změně výběru souborů uživatele zeptá, co má udělat s neuloženými změnami. Možnost `Save` změny uloží, `Revert` vrátí soubory do původního stavu a `Cancel` nechá uživatele dokončit úpravu vybraných souborů.

7. Evaluace

V této kapitole provedeme několik experimentů, které porovnají efektivitu implementovaných funkcí. Ve všech testech se používá jeden HDD disk (WDC WD7500BPKT-22pk4T0) a jeden SSD disk (Samsung 850 Evo 250 GB).

7.1 Efekt indexování na vyhodnocení dotazu

V sekci 3.3.3 jsme navrhli způsob, jak indexovat data. První experiment vyzkouší, jaký má indexování tagů vliv na čas vyhodnocení dotazu.

První polovina testů ukládá indexový soubor na stejný HDD disk jako data. Druhá polovina testů ukládá indexový soubor na jiný SSD disk. Experiment vyzkouší funkci programu na sadě 23790 fotek organizovaných v adresářové struktuře s maximální hloubkou pět adresářů. Průměrná velikost fotky je 5, 12 MiB. Celková velikost všech fotek je 119 GiB, přičemž vytvořený indexový soubor má velikost 185 MiB. Před každým testem byla vymazána cache souborového systému nástrojem RAMMap.

Testovaný dotaz je `select "D:/photos/**" where city = "prague"`. Výsledek dotazu obsahuje celkem 160 fotek. Každý test zaznamenal čas od začátku vyhodnocení do nalezení prvního výsledku, čas nalezení všech 160 souborů z výsledku a celkový čas vykonování dotazu (načtení atributů ze všech 23790 fotek). Průměrný čas pěti vyhodnocení dotazu ukazuje 7.1. Indexový soubor je buď na stejném HDD disku jako data, nebo na SSD disku. Tabulka 7.2 obsahuje počet prohledaných souborů pro nalezení prvního výsledku, všech výsledků a celkový počet prohledaných souborů.

Program při vyhodnocení dotazu záměrně neomezuje své další funkce, které používají disk (načítání náhledů a zápis tagů do souboru). Tyto operace mohou ovlivnit rychlost vyhodnocení dotazu a nejsou zahrnuty v tomto testu.

Index	**Test**	**První výsledek**	**Poslední výsledek**	**Celkový čas**
HDD	Ze souborů	5498	384901	477685
HDD	Z indexu	2304	7007	30898
SSD	Ze souborů	7215	347260	427653
SSD	Z indexu	1139	3678	12287

Tabulka 7.1: Časy [ms] vyhodnocení dotazu.

Indexování dat zásadně ovlivňuje čas běhu dotazu a to i v případě, kdy je indexový soubor na stejném disku jako data (viz 7.1). Test bez indexu musel projít 79, 39% fotek, než nalezl všechny soubory z výsledku dotazu (viz 7.2). Vyhodnocení dotazu s indexovanými daty nalezlo všechny soubory již po přečtení 16, 95% všech souborů. V tomto konkrétním případě se tedy vyplatilo použít optimalizaci popsanou v sekci 3.5.5.

Test	První výsledek	Poslední výsledek	Celkem souborů
Bez indexu	396	18887	23790
S indexem	363	4033	23790

Tabulka 7.2: Počet prohledaných souborů.

7.2 Optimalizace množinového mínus

Tento experiment porovná čas vyhodnocení dotazu s operátorem `except`. Test *optimalizovaný* používá optimalizaci popsanou v části 3.5.3. Test *neoptimalizovaný* používá přímočarý přístup, který nejdříve vyhodnotí oba poddotazy, a pak výsledky spojí.

Experiment je provedený ve stejném prostředí jako 7.1. Součástí testu není čas vytvoření indexu. Před každým testem experiment vymazal cache operačního systému. Testovaným dotazem je:

```
select "D:/photos/**" where city
except
select "D:/photos/**/20??/**" where ireland
```

Tabulka 7.3 ukazuje průměrný čas pěti vyhodnocení dotazu. Indexový soubor je buď na stejném HDD disku jako data, nebo na SSD disku.

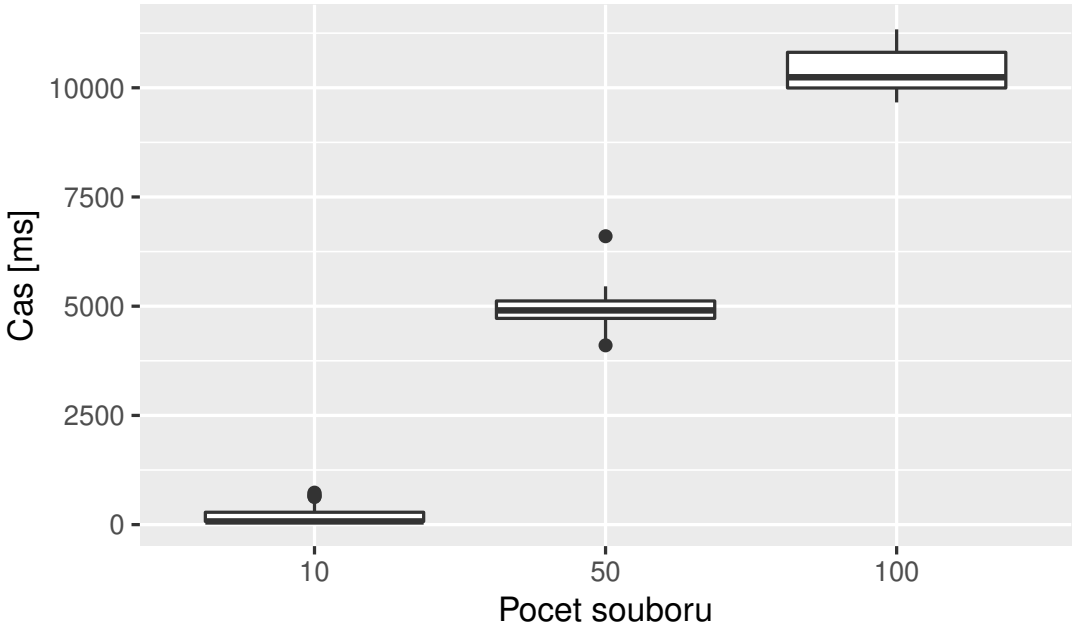
Index	Test	První výsledek	Poslední výsledek	Celkový čas
HDD	neoptimalizovaný	25229	31141	43753
HDD	optimalizovaný	1964	9514	33510
SSD	neoptimalizovaný	8151	10647	14458
SSD	optimalizovaný	692	4181	11554

Tabulka 7.3: Časy [ms] vyhodnocení dotazu.

Neoptimalizovaná verze musí projít všechny soubory dvakrát. Celkový čas vyhodnocení dotazu je tedy pomalejší (viz 7.3). Optimalizace v tomto případě výrazně zlepšuje čas do nalezení prvního výsledku a čas do nalezení všech výsledků dotazu.

7.3 Zápis do souboru

Poslední experiment vyzkouší zápis tagů do různého počtu souborů. Indexový soubor je uložený na SSD disku. Fotky jsou na HDD disku. Test provedl celkem 60 měření. Prvních 20 měření není ve výsledku započteno. Graf 7.1 ukazuje výsledky měření zápisu do 10, 50 a 100 souborů.



Obrázek 7.1: Zápis tagů do souborů

V této kapitole jsme ukázali výkon hlavních funkcí programu. Vyhodnocení dotazu bez indexu může při prvním spůštění trvat relativně dlouho. Vykonání dotazu se zrychlí, pokud už má program indexovanou většinu souborů. Dotazy je dále možné zrychlit uložením indexu na SSD disk. Díky poměrně malé velikosti indexového souboru lze indexovat velké množství fotek i v případě, kdy je na SSD disku málo volného prostoru.

Pro zápis tagů do souboru jsme zvolili pomalejší algoritmus, který je ale odolnější vůči pádu programu než ostatní algoritmy popsané v části 3.3.2. Vytvořený program je možné bez omezení používat i během zápisu do souborů.

8. Závěr

V této práci jsme navrhli a implementovali program, který umí hledat JPEG soubory podle tagů zadaných uživatelem. Vytvořený program se liší od ostatních programů pro organizaci fotek tím, že všechny tagy ukládá přímo do JPEG souborů. Soubory lze přesouvat i mezi různými počítači, aniž by se ztratily uložené tagy.

Povedlo se nám navrhnout dotazovací jazyk, který dovoluje jednoduše specifikovat podmínky hledání. Program umí hledat soubory podle vzoru cesty ke složce a podle filtru. Dotazovací jazyk dovoluje definovat uspořádání a seskupení vrácených výsledků. Dotazy je možné spojit množinovými spojkami sjednocení, průniku a množinového mínus. Povedlo se nám také implementovat několik optimalizací vyhodnocení dotazu.

Vytvořili jsme přizpůsobitelné uživatelské rozhraní. Součástí rozhraní je editor dotazů, který zvýrazňuje syntaxi a umí napovídat možná pokračování dotazu. Výsledky dotazu program zobrazuje v tabulce náhledů. Náhledy generuje, až když jsou potřeba, a tak šetří paměť a procesorový čas.

8.1 Možná vylepšení

8.1.1 Podpora dalších formátů

Vytvořený program umí uložit tagy pozue v JPEG souborech. Většina komponent v aplikaci s formátem souboru přímo nepracuje. Do programu tak jde přidat podporu pro jiný formát souborů úpravou jen několika tříd.

8.1.2 Náhledy složek

Některé prohlížeče fotek generují náhledy složek tak, že kromě samotné ikony zobrazí ještě miniaturu několika fotek v adresáři. Vytvořený program pro adresáře zobrazí jen ikonu složky. Program by šel rozšířit přidáním kódu pro načítání náhledů složek do třídy `DirectoryThumbnail`.

Literatura

[1] Eric Hamilton. JPEG File Interchange Format: Version 1.02. https://www.w3.org/Graphics/JPEG/jfif3.pdf, 1992.

[2] Exchangeable image file format for digital still cameras: Exif version 2.2. http://www.exif.org/Exif2-2.PDF, 2002.

[3] TIFF Revision 6.0. https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf, 1992.

[4] Adobe Bridge. https://www.adobe.com/products/bridge.html, 2019.

[5] digiKam. https://www.digikam.org/, 2019.

[6] Jeffrey Richter. *CLR via C#*. Microsoft Press, Redmond, Washington, 2012.

[7] Java Language Specification. https://docs.oracle.com/javase/specs/jls/se12/html/jls-1.html.

[8] Drew Noakes. Metadata Extractor. https://github.com/drewnoakes/metadata-extractor-dotnet, 2019.

[9] Thomas H. CORMEN, Charles E. LEISERSON, Ronald L. RIVEST, and Clifford STEIN. *Introduction to Algorithms, 3rd Edition.* MIT Press, Cambridge, Massachusetts, 2009.

[10] SQLite. https://www.sqlite.org/index.html, 2019.

[11] Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html.

[12] LiteDB. https://www.litedb.org/, 2019.

[13] LiteDB-Perf. https://github.com/mbdavid/LiteDB-Perf, 2016.

[14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools Second Edition.* Addison Wesley, 2006.

[15] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. https://www.antlr.org/papers/allstar-techreport.pdf, 2014.

[16] Coco/R. http://www.ssw.uni-linz.ac.at/coco/, 2019.

[17] Hime. https://bitbucket.org/cenotelie/hime/src/default/, 2019.

[18] Operator Precedence (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/language-elements/operator-precedence-transact-sql?view=sql-server-2017, 2017.

[19] About SQL Operators. https://docs.oracle.com/cd/B19306_01/server.102/b14200/operators001.htm, 2019.

[20] DotNet.Glob. https://github.com/dazinator/DotNet.Glob, 2019.

[21] Glob.cs. https://github.com/mganss/Glob.cs, 2019.

[22] Glob. https://github.com/kthompson/glob, 2019.

[23] FileSystemWatcher Class. https://docs.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?view=netframework-4.7.2, 2019.

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.

[25] Windows Forms overview. https://docs.microsoft.com/en-us/dotnet/framework/winforms/windows-forms-overview, 2017.

[26] WPF overview. https://docs.microsoft.com/en-us/visualstudio/designers/introduction-to-wpf, 2016.

[27] ImageSharp. https://github.com/SixLabors/ImageSharp, 2019.

[28] Magick.NET. https://github.com/dlemstra/Magick.NET, 2019.

[29] SkiaSharp. https://github.com/mono/SkiaSharp, 2019.

[30] MagicScaler. https://github.com/saucecontrol/PhotoSauce, 2019.

[31] John Kennedy and Michael Satran. Wimdows Imaging Component Overview. https://docs.microsoft.com/en-us/windows/desktop/wic/-wic-about-windows-imaging-codec, 2018.

[32] Managed Extensibility Framework (MEF). https://docs.microsoft.com/en-us/dotnet/framework/mef/, 2019.

[33] DockPanel Suite. http://dockpanelsuite.com/, 2019.

[34] ScintillaNET. https://github.com/jacobslusser/ScintillaNET, 2019.

Seznam obrázků

Seznam tabulek

A. Přílohy

A.1 Struktura zdrojových souborů

- `bin` obsahuje spustitelný soubor aplikace `Viewer.exe`.
- `src` obsahuje zdrojový kód modulů aplikace. Složky modulů jsou dále rozděleny podle jmenných prostorů.
- `src/ViewerInstaller` a `src/ViewerSetup` obsahují definici instalačního programu vytvořené aplikace, který se generuje nástrojem *WiX Toolset*.
- `docs` obsahuje dokumentaci vygenerovanou programem *DocFX*.
- `tests` obsahuje kód automatických testů vytvořené aplikace.