СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра програмування та математики

ПОЯСНЮВАЛЬНА ЗАПИСКА
до кваліфікаційної випускної роботи

освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)
спеціалізація „Інженерія програмного забезпечення”

на тему „Створення сховища даних мультимедіа з угрупованням даних за категоріями”

Виконав: студент групи ІПЗ-16д _____ М.С. Калашніков
(підпис) (ініціали і прізвище)

Керівник _____ В.Г. Іванов
(підпис) (ініціали і прізвище)

Завідувач кафедри _____ В.О. Лифар
(підпис) (ініціали і прізвище)

Рецензент_____

Сєвєродонецьк – 2020

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра програмування та математики

Освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)
спеціалізація „Інженерія програмного забезпечення”
(назва спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри ПМ,
д.т.н., доцент
_____ Лифар В.О.
“____” _____2020 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТУ

Калашніков Максим Сергійович
(прізвище, ім’я, по батькові)

1.Тема роботи Створення сховища даних мультимедіа з угрупованням даних за категоріями.

Керівник роботи доцент, к.т.н. Іванов Віталій Геннадійович,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджений наказом університету від “___”_____20__року №___

2. Строк подання студентом роботи 20 травня 2020 р.

3. Вихідні дані до роботи Об'єктом даної роботи є процес розробки сховища даних мультимедіа.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Аналітичний огляд, з висвітленням наступних питань: Сучасні файлові системи, вузькоспеціалізовані програми - каталоги відео і фотографій, організація пошуку. Основна частина, в якої висвітлити: робота з файлом, віртуальна файлова система Висновки. Перелік використаних джерел.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання__30 березеня 2020 року._____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	30.03.20	
2	Укладання і погодження з керівником плану і етапів виконання роботи	06.04.20	
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	13.04.20	
4	Аналіз шляхів виконання завдання. Вибір і погодження керівником оптимального шляху	20.04.20	
5	Укладання та тестування програмного продукту	27.04.20	
6	Укладання, оформлення та погодження пояснювальної записки з керівником	04.05.20	
7	Здача готової пояснювальної записки на кафедру	12.05.20	
8	Укладання доповіді і презентації	30.05.20	

Студент _____ М.С. Калашніков _____
(підпис) (ініціали і прізвище)

Керівник роботи _____ В.Г. Іванов _____
(підпис) (ініціали і прізвище)

ЛИСТ ПОГОДЖЕННЯ І ОЦІНЮВАННЯ

дипломної роботи студента гр. ІПЗ-16д Калашніков М.С.

Науковий керівник

Доцент, к.т.н. _____ Іванов В.Г.

Оцінка наукового керівника: _____

Рецензент _____

ПІБ, місто роботи, посада

Оцінка рецензента: _____

Кінцева оцінка за результатами захисту:

_____

Голова ЕК

_____ Лифар В.О.

підпис

РЕФЕРАТ

Робота містить: 40 сторінок основного тексту, 25 сторінок додатків, 7 рисунків, 15 використаних джерел.

Метою випускної кваліфікаційної роботи є проектування і розробка сховища мультимедіа, де нарівні з самими файлами повинна зберігатися і додаткова інформація про їх вміст у вигляді ключових слів.

У роботі було досліджено розробку алгоритму пошуку файлів по пошуковим запитам.

Система реалізована відповідно всім вимогам технічного завдання. Зроблено детальний опис процесу розробки системи.

ЗМІСТ

ВСТУП

Актуальність досліджень. В сучасних файлових системах людина працює з інформацією за допомогою таких абстракцій, як файл і папка [5]. Якщо розглянути сукупність всіх шляхів до всіх файлів файлової системи, що складаються з вкладених папок, то її можна представити у вигляді дерева. Тому, в цілому таку структуру організації файлів можна назвати деревовидної або ієрархічної. Вона є зручною для комп'ютера, оскільки є однаковою і добре масштабується. Однак людина мислить інакше, ніж комп'ютер, людина мислить асоціаціями. Один і той же об'єкт співвідноситься у свідомості людини з набором визначень, назв, схожих понять і категорій, до яких цей об'єкт можна віднести. Власне, людина і визначає сам об'єкт за допомогою цього безлічі асоціацій.

Таким чином, будучи побудовані за іншим принципом,

обмеження в описі даних незмінно тягне складності при їх пошуку: адже комп'ютер зберігає тільки малу частку інформації про фото, а людська пам'ять не безмежна. В даний час мова йде вже не про декілька десятків мегабайт інформації, що зберігаються у кожної людини.

Сучасні користувачі мають в своєму розпорядженні вже сотні гігабайт і тисячі унікальних файлів, як наприклад, відео, фотографії і документи. Люди, які рідко використовують деяку частину своєї інформації, швидко забувають, де і як вона зберігалася. При цьому, коли їм необхідно знайти її, вони мають у своєму розпорядженні лише неточним і уривчастих її описом. І такий опис становить певний образ вмісту необхідного файлу або необхідних файлів, за яким користувач може їх шукати.

Однак єдині легко задаються параметри, за якими ведеться пошук в сучасних файлових системах, - це назви всіх папок в дорозі до файлу і сама назва файлу. А такі параметри не завжди можуть повністю

відповідати вмісту файлу або, у всякому разі, не завжди зручні для повноцінного опису файлу. Отже, той образ потрібних користувачеві файлів, за яким користувач може їх шукати, може далеко не повністю відповідати параметрам, за якими можна здійснювати їх пошук в файлових системах. Внаслідок цього важливо збільшити ефективність організації інформації, щоб повніше описувати файли, і відповідно ефективність пошуку необхідних файлів. Саме існування більшого числа ключових слів, у зберігається файлу може сприяти тому, що людина швидше і легше знайде необхідну йому інформацію у великому масиві даних.

Таким чином, виникли ідеї створення систем, в яких файл буде визначатися не тільки його місцезнаходженням і назвою, а й описом його вмісту. Прикладами таких систем є Windows Media Player [12], Adobe Photoshop Lightroom [8], різні системи контролю та інші системи. Однак всі вони призначені лише для роботи з певними форматами даних, що виправдано для людей, які більшу частину часу працюють в конкретній предметній області. Тим не менш, не для всіх користувачів це прийнятно. Наприклад, при пошуку сукупності файлів абсолютно різних форматів доведеться використовувати кілька інструментів і відповідно витратити час на «ручне» зіставлення їх результатів.

Об’єкт досліджень: Створення сховища даних мультимедіа з угрупованням даних за категоріями.

Предмет досліджень: Проектування і розробка сховища мультимедіа, де нарівні з самими файлами повинна зберігатися і додаткова інформація про їх вміст у вигляді ключових слів.

Завданням даної дипломної роботи є проектування і розробка сховища мультимедіа, де нарівні з самими файлами повинна зберігатися і додаткова інформація про їх вміст у вигляді ключових слів. За допомогою цієї системи

має здійснюватися пошук файлів, як за назвою файлу і метаданих, так і за описом їх вмісту. Сама система повинна складатися з сховища і ядра, і, як додаток в майбутньому, віртуальної файлової системи. Найбільш відповідною основою для зберігання ключових слів є база даних [4], в якій без особливих труднощів можна реалізувати відношення «багато до багатьох», тобто багато файли можна описати багатьма відповідними ключовими словами. Таким чином, сховище повинне включати в себе певну частину традиційної файлової системи, відведену під зберігання самих файлів, і базу даних, в якій повинна зберігатися вся додаткова інформація.

Ядро, як основна компонента системи, має відповідати за управління і редагування файлів, а також за взаємодію з базою даних, збереження цілісності зв'язків і актуальності інформації. Віртуальна файлова система повинна емулювати традиційну файлову систему з деревовидної організацією файлів. Це необхідно для того, щоб програми, що працюють тільки з традиційними файловими системами, могли працювати з даними зі сховища.

При цьому ключові слова будуть відображатися в каталоги віртуальної файлової системи

Завдання дослідження: Проектування і розробка системи для зберігання файлів, що реалізує такі функції:

а) збереження файлів у сховищі і видалення файлів зі сховища;

б) додавання, зміна, видалення ключових слів, пов'язаних з файлом;

в) відкриття файлів для редагування у відповідних додатках;

г) пошук файлів по окремим параметрам;

е) пошук файлів по пошуковому запиту.

Методологічна та теоретична основа дослідження: Розробка алгоритму пошуку файлів по пошуковим запитам.

Методи дослідження: Огляд і аналіз існуючих рішень в області систем зі зберігання файлів з угрупованням даних за категоріями або схожих систем, що дозволяють зберігати опис даних, доданий користувачем.

Практичне значення отриманих результатів: Створення призначеного для користувача інтерфейсу для роботи з файловим сховищем.

РОЗДІЛ 1 АНАЛІТИЧНИЙ ОГЛЯД

Сховище даних - це база даних, яка зберігається окремо від оперативної бази даних організації. У сховище даних немає частих оновлень.

Він володіє консолідованими історичними даними, які допомагають організації аналізувати свій бізнес. Сховище даних допомагає керівникам організовувати, розуміти і використовувати свої дані для прийняття стратегічних рішень.

Системи сховища даних допомагають в інтеграції різноманітних прикладних систем. Система сховища даних допомагає в консолідованому аналізі історичних даних. Чому сховище даних відокремлено від оперативних баз даних

Сховища даних зберігаються окремо від операційних баз даних з наступних причин:

Оперативна база даних створюється для добре відомих завдань і робочих навантажень, таких як пошук певних записів, індексація і т. Д. У контрактах запити до сховища даних часто є складними і являють загальну форму даних.

Операційні бази даних підтримують одночасну обробку декількох транзакцій. Механізми контролю і відновлення паралелізму потрібні для діючих баз даних, щоб забезпечити надійність і узгодженість бази даних.

Оперативний запит до бази даних дозволяє читати і змінювати операції, тоді як для запиту OLAP потрібно тільки доступ тільки для читання до збережених даних.

Оперативна база даних підтримує поточні дані. З іншого боку, сховище даних підтримує історичні дані.

Оперативна база даних створюється для добре відомих завдань і робочих навантажень, таких як пошук певних записів, індексація і т. Д. У контрактах запити до сховища даних часто є складними і являють загальну форму даних.

Операційні бази даних підтримують одночасну обробку декількох транзакцій. Механізми контролю і відновлення паралелізму потрібні для діючих баз даних, щоб забезпечити надійність і узгодженість бази даних.

Оперативний запит до бази даних дозволяє читати і змінювати операції, тоді як для запиту OLAP потрібно тільки доступ тільки для читання до збережених даних.

Оперативна база даних підтримує поточні дані. З іншого боку, сховище даних підтримує історичні дані.

Ключові особливості сховища даних обговорюються нижче:

Предметно-орієнтований. Сховище даних є предметно-орієнтованим, оскільки надає інформацію по предмету, а не по поточних операціях організації. Такими суб'єктами можуть бути продукт, клієнти, постачальники, продажу, виручка і т. Д Сховище даних не орієнтоване на поточні операції, а зосереджено на моделюванні і аналізі даних для прийняття рішень.

Інтегрований. Сховище даних створюється шляхом інтеграції даних з різнорідних джерел, таких як реляційні бази даних, плоскі файли і т. Д Ця інтеграція підвищує ефективність аналізу даних.

Варіант часу - дані, зібрані в сховище даних, ідентифікуються з певним періодом часу. Дані в сховищі даних надають інформацію з історичної точки зору.

Незалежний - незалежний означає, що попередні дані не видаляються при додаванні нових даних. Сховище даних зберігається окремо від

оперативної бази даних, і тому часті зміни в оперативній базі даних не відображаються в сховище даних.

Додатки сховища даних

Як обговорювалося раніше, сховище даних допомагає керівникам підприємств організовувати, аналізувати і використовувати свої дані для прийняття рішень. Сховище даних служить єдиною частиною системи зворотного зв'язку «план-виконання-оцінка» для управління підприємством.

Типи сховищ даних

Обробка інформації, аналітична обробка та вилучення даних - це три типи додатків сховища даних, які обговорюються нижче:

Обробка інформації - сховище даних дозволяє обробляти зберігаються в ньому дані. Дані можуть бути оброблені за допомогою запитів, базового статистичного аналізу, звітності з використанням крос-таблиць, таблиць, діаграм або графіків.

Аналітична обробка - сховище даних підтримує аналітичну обробку інформації, що зберігається в ньому. Дані можна аналізувати за допомогою базових операцій OLAP, включаючи зрізи і деталі, деталізацію, деталізацію і поворот.

Інтелектуальний аналіз даних. Інтелектуальний аналіз даних підтримує виявлення знань шляхом пошуку прихованих закономірностей і асоціацій, побудови аналітичних моделей, виконання класифікації та прогнозування. Ці результати видобутку можуть бути представлені з використанням інструментів візуалізації.

1.1 Короткий історичний огляд

Ідея надати користувачеві більш дружню, ніж файлова система, середовище для зберігання даних має ті ж джерела що і ідея "канцелярських" метафор, таких як "робочий стіл", "папка", "корзина". Джерелом даних "канцелярських" метафор була компанія Xerox, яка застосувала їх в системах Xerox Alto і Xerox Star. Саме ці системи стали прототипами систем компанії Apple, Inc., про яку піде мова далі.

Закладаючи і зміцнюючи традиції метафори "офісного робочого простору", яких дотримуються і досі, система Apple Lisa відображала дані файлової системи у вигляді, звичних людині, канцелярських об'єктів. Каталоги показувалися у вигляді папок, вміст яких можна було переглянути у вікні. Файли, що зберігають документи різної природи (тексти, таблиці, зображення) відображалися за допомогою різних іконок. На тому ж робочому просторі перебували і метафори інструментів: текстових процесорів, календарів, годин.

Деякі метафори могли здатися за сучасними мірками дуже буквальними: так, наприклад, був спеціальний об'єкт "пачка паперу", з якого можна було витягти листи для написання нового документа. Оброблювані в даний момент документи поміщалися на робочий стіл. Для того, щоб зберегти і закрити документ, необхідно було перетягнути його іконку назад в папку, з якої документ був взятий або в іншу. [3], [7].

Одна з цікавих і важливих в контексті даної роботи особливостей метафори робочого простору - можливість завдання однакових імѐн для різних документів Apple Lisa [10]. Вже на початку 1980-х років творці системи усвідомлювали, що оператору комп'ютера ім'я документа, саме по собі, може говорити мало, і дозволили для відображення на робочому столі використовувати т.зв. віртуальні імена, що не були унікальні. Так, наприклад, створена копія документа називалася, так само як і оригінал, хоча при редагуванні еѐ і оригіналу властивості починали відрізнятися. Надалі

користувач розрізняв документи за їх властивостями (датою, розміром) і вмісту. При збереженні на диск дані документів записувалися в файли, фізичні імена яких були унікальні і відповідали накладається файлової системою обмеженням. Зрозуміло, користувач міг змінити і логічне ім'я документ, якщо вважав за потрібне. У сучасних ОС сімейств Windows і в UNIX-подібних логічні імена файлів не застосовуються.

Microsoft приділила належну увагу дружньому іменування документів в офісних і домашніх версіях своїх систем (розширивши можливості по іменування файлів файлової системи FAT за допомогою надбудови VFAT) більш ніж на 10 років пізніше, в 1995 році при створенні Windows 95 [6].

У Windows NT, орієнтованої спочатку на корпоративний сектор, можливості іменування файлів відразу були досить багатими. Частина з них файлової системою NTFS була успадкована від файлової системи HPFS ОС OS / 2.

Таким чином, вже в середині 1990-х рр. Microsoft пішла традиційним шляхом, надавши користувачу традиційну файлову систему з досить широкими можливостями в порівнянні з сімейством DOS, наприклад. Ця тенденція зберігалася ще більше 10 років аж до випуску Windows Vista, в якій була вперше прийнята деяка функціональність, ставилася до цього тільки до дослідницького проекту WinFS.

1.2 Сучасні файлові системи

Нагальна потреба в нових принципах систематизації даних привела до появи останнім часом розробок і досліджень по додаванню до файлів додаткової інформації про вміст файлу. Певні розробки на дану тему

з'явилися і в останніх версіях операційних систем, таких як Windows і Mac OS [11]. Що говорить про дійсну значимість даного питання.

У них були введені можливості для пошуку файлів не тільки по шляху, назвою, метаданих, а й за тегами (коментарів в Spotlight Mac OS [11]), що були додані користувачами до файлів.

Розглянемо докладніше Mac OS X. З появою в OS X нової функції Spotlight, пошук на Mac став як ніколи швидким і зручним. Spotlight спростив пошук файлів по тегам (або мітках), не вимагаючи від користувача знання точної назви файлів або їх вмісту.

Проте, прикріпити мітку до документа для швидкого пошуку не так-то просто. Ключові слова можна включати в назву документа або відкривати властивості файлу і прописувати теги в поле "Коментарі Spotlight". Для більш простого додавання міток в Mac, розроблені спеціальні програми, такі як Tags і Punakea.

У Windows ж починаючи з XP і до появи 7й версії, хоча й існувала можливість додавання міток, використовувати її було не тільки незручно, але й мало хто був в курсі даної можливості. Так як щоб дістатися до цього, потрібно було зайти в

«Властивості», а потім на вкладку «Зведення» і так вже виставити додаткову інформацію. Якщо уявити що дані маніпуляції потрібно проробити з кожним з тисячі файлів. Однак Microsoft усвідомило недоліки даної системи і необхідність і зручність для користувача додавати додаткову інформацію у вигляді ключових слів до файлів для майбутнього пошуку. Таким чином, вже в недавно вийшла Windows 7 з'явилися зручний інтерфейс для додавання ключових слів без заходу в властивості файлу, а також модифіковані можливості по каталогізації і категоризації файлів в бібліотеках за цими ключовими словами. Під бібліотекою в Windows 7

розуміється певний користувачем набір папок специфічного змісту, що представляє дані незалежно від ієрархії папок.

1.3 Вузькоспеціалізовані програми - каталоги відео і фотографій

На теми, аналогічні освітленій в роботі, багатьма невеликими компаніями були створені вузькоспеціалізовані програми - сховища, які є суто комерційними проектами і при цьому закритими для сторонніх очей. Тому складно розповісти, як вони вирішували технічні питання, пов'язані зі створенням цих додатків. Крім того, дані сховища орієнтовані на певні файлові системи і працюють тільки з певними типами файлів, такими як відео або зображення, іноді тексти. У них є можливість додавання тегів і пошуку по ним.

Наприклад Adobe Photoshop Lightroom використовує каталог для відстежування місцеположення файлів і запам'ятовує про них інформацію. Каталог, як база даних, містить записи фотографій користувача. Ця запис зберігається в каталозі і містить такі дані, як посилання, що вказують, де знаходяться фотографії на комп'ютері, метадані, що описують фотографії При оцінюванні фотографії можна додати метаданих і ключові слова, організовувати фотографії в колекції або видалити фотографії з каталогу, навіть якщо вихідні файли в автономному режимі фото налаштування зберігаються в каталозі.

Крім роботи з музикою, програвач Windows Media Player 11 надає нові величезні можливості зберігання і використання цифрового мультимедіа. Він значно спрощує доступ до всіх відеозаписів, зображень і записаним телепередач на комп'ютері. Не тільки музика, але інші види мультимедіа (відео, зображення і записані телепередачі) виділені тепер в окрему категорію на панелі «Бібліотека». У бібліотеці можна впорядкувати музику за ключовими словами.

1.4 WinFS - система зберігання і управління даними на основі реляційної бази даних

Єдиним далеко просунулися проектом по розробці більш універсальною файлової системи на основі категорій, до якої також було докладено багато сил, є проект Microsoft [14]. Він полягав у створенні нової файлової системи, скорочено званої WinFS. Передбачалося, що вона буде являти собою файлову систему майбутньої операційної системи. На жаль, даний проект був закритий, через ускладненість завдання, бажання охопити всі можливі її аспекти і якісно їх розробити. У ньому передбачалися не тільки складна функціональність, а й уніфікація всіх файлів під єдину структуру [13]. Однак все ж напрацювання з цього проекту все ж увійдуть в наступні операційні системи, а також в Microsoft SQL Server і ADO.NET [14].

1.5 Використовувані технології

Microsoft .NET

В якості основної платформи в даній роботі використовується платформа Microsoft .NET. Оскільки існує необхідність тісної взаємодії програмної частини і бази даних, потрібно було вибрати найбільш зручні технології зі створення бази і по створенню програмного забезпечення працює з цією базою.

Мовою програмування для реалізації сховища мультимедіа був обраний C #. Таким чином, код реалізації можна буде скомпілювати за допомогою Mono, а Mono може виконувати модулі на багатьох інших операційних системах крім Windows, наприклад, Lunix. Тим самим забезпечується можливість в майбутньому використання розробленого

прототипу сховища мультимедіа кроссплатформенно. А для розробки бази даних використовується Microsoft SQL Server, що включає в себе .NET.

Проект Irony

Проект Irony є засіб розробки і реалізації мов на .NET. Він містить платформу для реалізації мов, LALR і NLALR парсер. За допомогою Irony можна писати свої власні граматики, по засобом яких можна розпарсити рядок введення в дерево за допомогоюпарсеру, який також описаний всередині окремої бібліотеки. Крім того даний проект написаний повністю на C #. Він використовує гнучкість і міць мови C # і. NET Framework 3.5 для здійснення абсолютно нової і модернізованої технології побудови компіляторів.

GNU libextractor

GNU libextractor - це бібліотека, використовується в даному проекті для вилучення метаданих з файлів довільного типу. Вона створена з використанням допоміжних бібліотек, що виробляють вилучення даних, і легко розширюється зовнішніми модулями, модулями, що підтримують інші типи файлів. libextractor це пакет GNU. GNU відкрита Unix подібна операційна система.

Вона має певні недоліки, такі як проблема з кодуваннями не підтримка UTF-8. Крім того в версія для Windows старіша, ніж поточні та не містять певних виправлень багів системи. Однак дані недоліки можна обійти і в більшості випадків. Використання цієї бібліотеки дозволяє зручно отримати метадані по файлах абсолютно різних форматів, таких як HTML, PDF, PS, OLE2 (DOC, XLS, PPT), OpenOffice (sxw), StarOffice (sdw), DVI, MAN, FLAC, MP3 (ID3v1 and ID3v2), NSF (E) (NES music), SID (C64 music), OGG, WAV, EXIV2, JPEG, GIF, PNG, TIFF, DEB, RPM, TAR (.GZ), ZIP, ELF, S3M (Scream Tracker 3), XM (eXtended Module), IT (Impulse Tracker), FLV, REAL, RIFF (AVI), MPEG, QT і ASF.

1.6 Компоненти структура системи

Головною частиною системи є ядро або, так звана, керуюча система, яка буде відповідати як за введення даних в сховищі, так і за обробку запитів пошуку до сховища, тобто відповідати за всю роботу з даними.

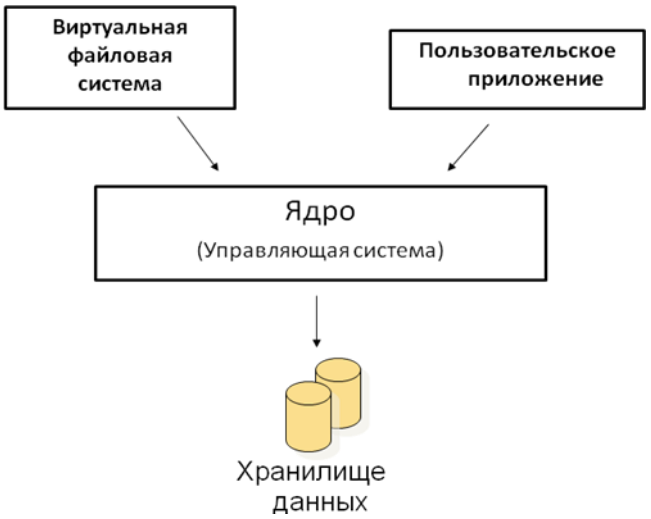

Рис.1.1 - Загальна структура компонент системи.

Визначено саме сховище, що дозволяє зберігати не тільки самі дані, але і всю додаткову інформацію, додану користувачем, при цьому має адекватну НЕ деревоподібну структуру. Структура повинна підтримувати відносини багато до багатьох. Так як кожен файл може мати багато тегів, а тег може бути у багатьох файлів.

Між ядром і користувачем або додатками, так би мовити, зовнішнім світом, створюються спеціальні бібліотеки. Ці бібліотеки є інтерфейсами, що приховують і структуровану функціональність ядра.

Вся система повинна бути зручною і простою. Зручність, тому що, не має сенсу створювати нову файлову систему, більш незручною, ніж існуюча файлова система. Простота, так як будь-який надмірне ускладнення в постановці завдання веде до ускладнення в її реалізації і не гарантує виграш в ефективності.

Узагальнену модель зберігання файлів в файлову систему на основі категорій можна представити у вигляді простої моделі Сутність-Зв'язок [2]. Сутностями є файли, інформація про яких повинна зберігатися в сховищі, а також теги. Між файлами і відповідними тегами створюється зв'язку, причому ці зв'язки типу багато до багатьох, внаслідок того, що файл може бути пов'язаний з багатьма тегами, а тег зі багатьма файлами. Атрибутами в даному випадку можуть бути, наприклад, назва файлу, або деякі метадані, такі як дата створення і дата модифікації файлу. Пошук ведеться як по атрибутах, так і по зв'язаних тегами. Тому основний ознака цієї моделі полягає в тому, чим більше критеріїв ми вводимо в пошук, тим вже вибірку з відповідних результатів ми отримуємо.

За умови розробки абстрактного ядра, сховище даних може бути по суті будь-яким. Однак була обрана реляційна база даних, як найбільш проста, зручна модель зберігання даних і подання їх, як для людини, так і для машини. За допомогою табличнійструктури можна показати всю багатогранність визначень сутності файлу. А база даних найбільш ефективна для зберігання будь-якої інформації. При цьому існує можливість варіювати структуру в залежності від необхідності.

Крім того, сучасні технології добре розроблені в рамках створення самих РСУБД [3]. Звичайно, не виключено, що спеціалізоване під нашу модель сховища даних може виявитися краще. Але на даному етапі його розробка не передбачена.

Необхідна розробка фронт-ендів для забезпечення можливості для існуючих додатків працювати з нової файлової системи. При цьому для програми не повинні турбуватися істотні зміни. Так як ніхто не стане робити нові версії додатків під яку-небудь нову файлову систему. Що в повніше логічно.

Для додатків, які будуть підтримувати нову файлову систему, будуть розроблені бібліотеки. Ці бібліотеки дозволятимуть працювати з додатками з моїм сховищем через ядро замість звичайної файлової системи. Також для існуючих програм, які підтримують плагіни, можна створити плагін для роботи з бібліотекою.

Не всі програми зможуть читати дані з нової файлової системи за допомогою бібліотек. Так, наприклад, легаси додатки. Вони можуть взаємодіяти тільки з файлової системою деревовидної структури. Для таких додатків потрібно розробити віртуальну файлову систему по нашій файлової системи, яка являє собою деревоподібну структуру, через яку легаси додатки зможуть мати доступ до даних.

Для обох фронт-ендів можна розглядати шлях до файлу як запит до бази. Так як обидва фронт-енду працюють з ядром від нашої системи, а вона знаходить файл за запитом до бази.

Представлену вище модель можна розширити, за допомогою імплікації міток, а також додавання синонімів, і предикатів - відносин між мітками або даними, наприклад відносини прямування. Це звичайно ускладнить функціональність, але може дати нові можливості.

У сховищі можна зберігати не тільки самі файли і мітки, пов'язані з файлами, але також і їх метадані. До речі це може бути досить корисно для користувачів. Так як їм цікаво не тільки, чи є в сховищі такий об'єкт з таким ім'ям і мітками, а й які у даного об'єкта є параметри, характеристики, як би його "фізичні" дані. Для цього необхідно витягувати метадані і додавати їх в базу.

Вся система складається з трьох частин:

1. База даних містить схему бази даних, збережених функції, написані на T- SQL, на MSServer.

2. Додатки, реалізовані на мові C #, що складаються з:

a. Class Library - Ядро

b. Віконне додаток - GUI, що припускає використання Ядра через API

3. Модуль синтаксичного аналізу для рядка запиту клієнта (див. Розділ Граматика)

a. Окрема бібліотека Irony [5].

b. Бібліотека, яка містить граматику.

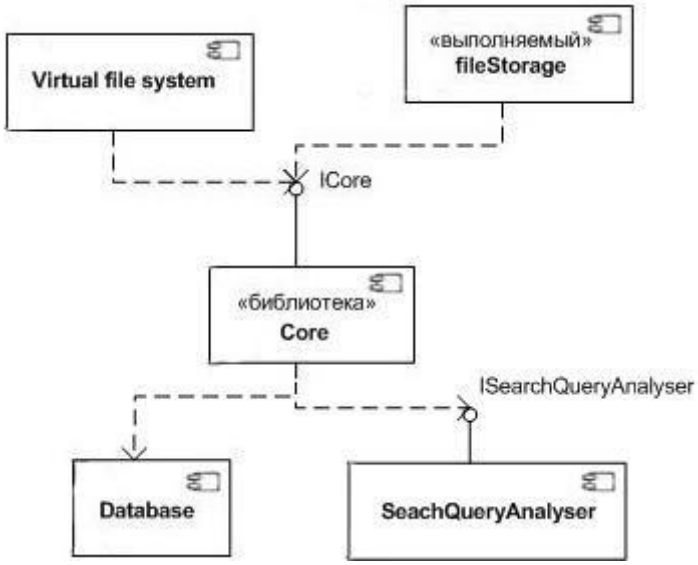

Рис. 1.2 - Діаграма компонент і їх взаємозв'язок.

1.7 База даних як основа сховища системи

Файл та база даних

Самі файли зберігаються не в базі, а в файлової системі. Це пояснюється недоцільністю реалізації фактичного зберігання файлу в базі, через не універсальності типів файлів, так як в базі даних передбачається зберігання однотипних елементів в шпальтах. Через це довелося б не тільки

створювати універсальний тип для всіх видів файлів, що само по собі велика задача, але і модифікувати базу даних під ці потреби.

При внесенні інформації про фото, необхідно враховувати, що слід вносити нову інформацію в інші таблиці, наприклад, якщо якогось розширення в базі немає, необхідно додати.

При видаленні файлу треба враховувати, що в базі можуть залишитися відповідні мітки або розширення, які більше ні з чим не пов'язані, в такому випадку також варто їх видалити.

Пошук по базі даних

Є набір даних, які можуть бути отримані від користувача для пошуку:

a. ім'я

b. розширення

c. Категорія

d. метадані

e. Мітки

У цьому списку вони впорядковані за значимістю, тобто якщо у нас є ім'я то, швидше за все спочатку шукати треба по ньому і потім шукати з наступними даними щодо отриманої вибірки.

1. Поки пошук по автору не враховується. Рішення як це можна буде зробити см. Пункт 4 списку аналогічно, тільки для Таблиці Автор.

2. Якщо нам дано розширення по категорії не шукаємо

3. Розширення файлу (як і категорія) в пошукових даних може бути в однині, тобто конкатенація результатів не передбачена і два варіанти розширення

можна вважати взаємовиключними (результат = 0) або просто ігнорувати все крім першого.

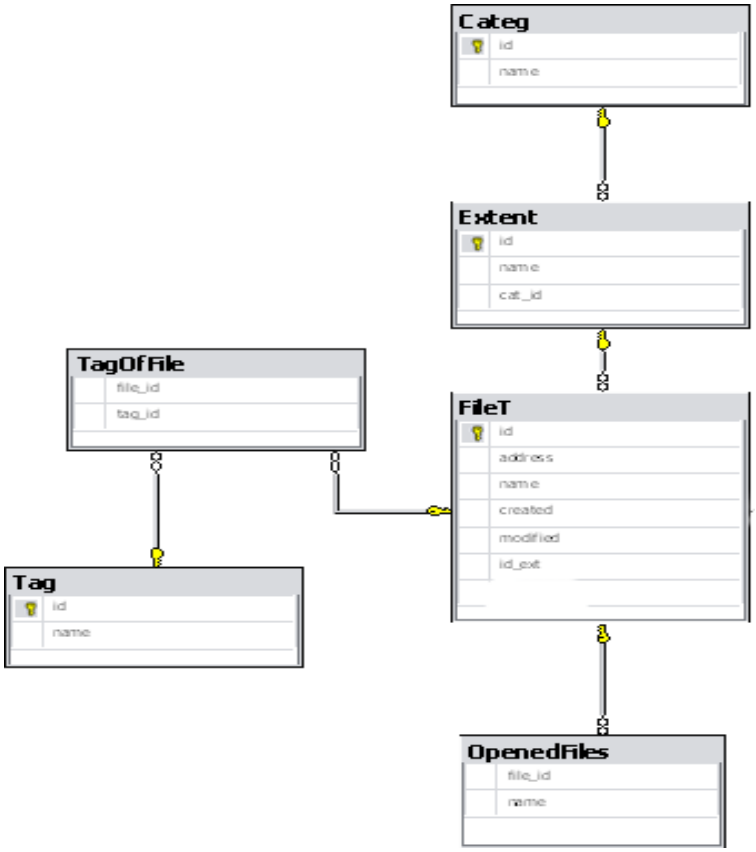

Рис. 1.3 - База даних ля зберігання даних про фото

1.8 Організація пошуку

Питання полягає в тому за допомогою, чого краще шукати.

Існує наступні варіанти способу реалізації функціональності бази:

1. Збережені процедури з динамічним

2. Статичні збережені процедури

3. Код на .NET в ядрі, т. Е. Реалізації поза самої бази

4. CLR функції

У пошукових запитах нам важлива швидкість роботи, як відомо методи, написані на CLR1, зручні для використання більш складної функціональності, яку не можна або майже не можна написати на T-SQL. Але вони працюють повільніше, ніж T- SQL, крім того, їх використання дещо псує кроссплатформенность.

У плані швидкості гарні збережені процедури і функції, без використання динамічного SQL. Однак якщо писати все статично, то потрібно близько 20 окремих функцій, з урахуванням значущості даних і для всіх варіантів їх поєднання. Хочу зауважити, що написання декількох збережених функцій, без залежності один від одного і потім об'єднання (перетин) результатів, спочатку вважається не оптимальним. Оскільки краще вести пошук по меншому безлічі, якщо його зменшення можливо за іншими більш конкретних даних.

Існує ще варіант використання динамічного SQL в процедурах, що зменшить загальне число методів. З процедурами на T-SQL є ще одна складність необхідність використання локальних таблиць, для передачі їй результатів роботи попередньої процедури (наприклад, результат роботи пошуку по розширенню повинен бути переданий в запит процедури пошуку по мітках). Тільки функції можна безпосередньо вставляти в запит без необхідності передачі її результату в локальну змінну.

Нарешті, в принципі, можна конструювати рядок запиту в ядрі, потім виконати цей запит в базі, і повернути результат в Ядро.

Отримуємо чотири варіанти написання коду

а. Написати частина у вигляді процедур, де динамічно підставляються дані в запит

b. Написати окремі статичні функції (так звані Inline stored functions) для кожного випадку на T-SQL

c. Написати функції на CLR, що, швидше за все, буде повільніше працювати, ніж варіант з T-SQL.

d. Зробити рядок запиту в ядрі і виконати її з нього ж.

Поки в якості основи був обраний варіант використання всього, крім CLR (він правда теж потрібен, якщо припустимо, буде потрібно зробити агрегує функцію для конкатенації рядків, наприклад). З урахуванням того що швидкість важливіше, найкраще використовувати Збережені процедури. Однак захоплюватися не варто. Вважаю, досить буде використовувати їх для випадків, коли є тільки дані одного роду або з двох типів - ім'я і ще що-небудь.

Крім того такі дані, як метадані, в запиті беруть участь швидше за все в нерівностях (наприклад "data_created <04.10.09") або рівності; важко передбачити яким знак буде використовуватися, тому оптимально для пошуку з метаданими передавати метадані як готову рядок, вставляти в запит, а краще як виклик функції з параметрами. Тобто для пошуку з метаданими можна використовувати наступні варіанти: збережені процедури з динамічним SQL або код на .NET в ядрі.

Решта не складно написати, збираючи рядок запиту в ядрі. отримуємо: 4 збережені функції від одного параметра даних, а також ще 3 функції від динамічної процедура для пошуку з метаданими, якщо нам не дано мітки. Щоб не було необхідності в локальній таблиці для вилучення результату з цієї процедури і використання його при пошуку по мітках. 4 функції повертають рядки (швидше за все лежать в ядрі) для збирання рядки запиту в ядрі.

1.9 Функціональність

Функції та процедури, які написані на T-SQL і знаходяться в базі.

Збережені функції з бази: Повертають

1. table

a. SearchByName (@name nvarchar (50))

b. SearchByExt (@ext nvarchar (4))

c. SearchByCat (@cat nvarchar (20)) - категорія

d. SearchByTags (@tags nvarchar (50))

e. SearchByExtwName (@ext nvarchar (4), @ name nvarchar (50))

f. SearchByCatwName (@catnvarchar (20), @ namenvarchar (50))-категорія

g. SearchByTagswName (@tags nvarchar (50), @ name nvarchar (50))

h. GetTagsOfFile (@fid uniqueidentifier)

i. GetCategs ()

j. GetExtOfCat (@cat nvarchar (20))

k. GetExt ()

l. GetTagsIdOfFile (@fid uniqueidentifier)

2. uniqueidentifier

a. GetIdOfFile (@fadr nvarchar (50))

b. GetExtId (@ext nvarchar (4))

c. CheckExExt (@ext nvarchar (4))

d. CheckExName (@ext nvarchar (4), @ name nvarchar (50))

e. CheckExTags (@list ntext, fid uniqueidentifier)

1. SearchByMeta (@metadata nvarchar (50), @ X nvarchar (50)) - Процедура, в якій запит виконується по динамічно з складеної рядку, де параметр X - рядок, що містить таблицю, з якої беруться дані. Тобто безліч, звідки ми вибираємо дані, може бути вже більш вузьким, в залежності від наявності інших даних.

2. UpdF (@fadrvarchar (50), @ namevarchar (50), @idext uniqueidentifier)

3. checkExistExt (@extern nvarchar (4), @res uniqueidentifier OUTPUT)

4. checkExistTags (@list ntext, @ fileid uniqueidentifier)

5. InsrtF (@fadr nvarchar (50), @name nvarchar (50), @ext nvarchar (4), @fileid uniqueidentifier OUTPUT)

6. deleteRedundantExt (@extid uniqueidentifier)

7. deleteRedundantTags (@fileid uniqueidentifier)

8. DeleteF (@fileid uniqueidentifier)

9. UpdTags (@fileid uniqueidentifier, @list ntext)

Ядро - основна частина системи контролює обробку файлів, посилає запити до бази для додавання, модифікації даних про файлах.

РОЗДІЛ 2. АЛГОРИТМ РАБОТИ

Загальний алгоритм роботи ядра представлений на схемі. Залежно від дії, яке потрібно зробити, або йде пошук по базі даних з висновком результату пошуку, або здійснюється управління файлами.

Управління файлами складається з двох незалежних етапів, що становлять одну транзакцію. Вони проводяться послідовно, так щоб в разі виникнення помилки, система раніше скасувала виконані зміни. Фізично файли обробляються на другому етапі, оскільки тоді складніше скасувати дії, вироблені в файлової системі, і так як при взаємодії з базою даних імовірніше виникнення неполадок.

При виникненні помилки, робота припиняється, і система повертає код помилки. При цьому, якщо в сховище були проведені зміни, вони скасовуються.

Детальніше алгоритм пошук буде розглянуто в розділі «Пошук файлу».

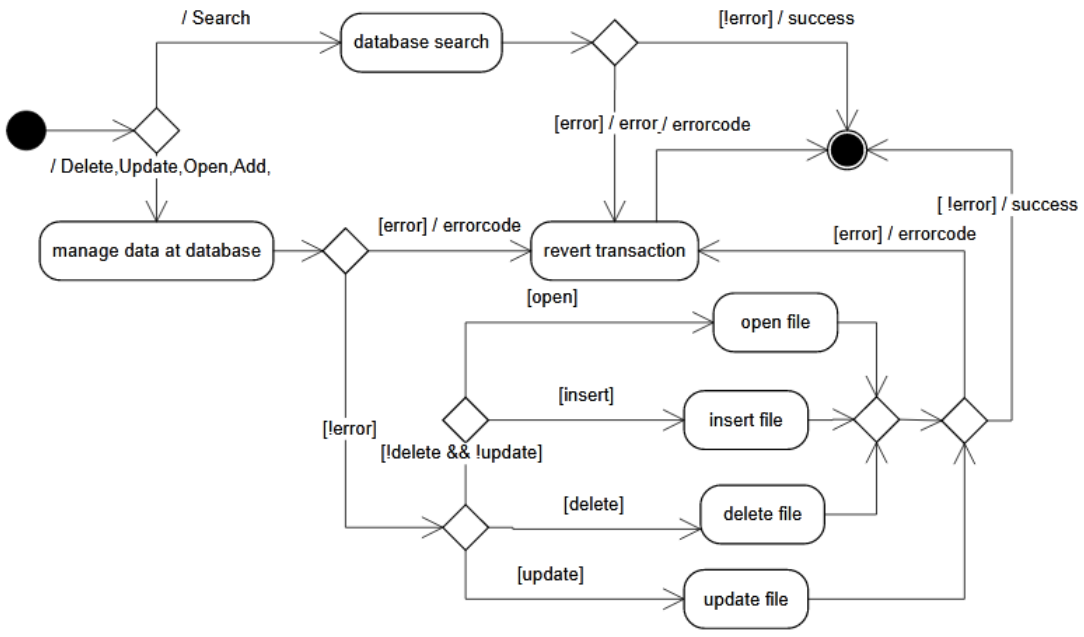

Рис. 2.1 - Алгоритм роботи ядра.

Управління файлами складається з двох частин: зміна відповідних даних в базі даних - manage data at database, управління файлом в сховище фізично - open, delete, update, insert file.

Таким чином, робота ядра ділиться за дією на:

- адміністрування файлів: збереження, видалення, модифікація, відкриття;
- пошук файлів. І по обробці даних.
- Обробка інформації про фото в базі даних.
- Обробка фізичного файлу.

Модель ядра

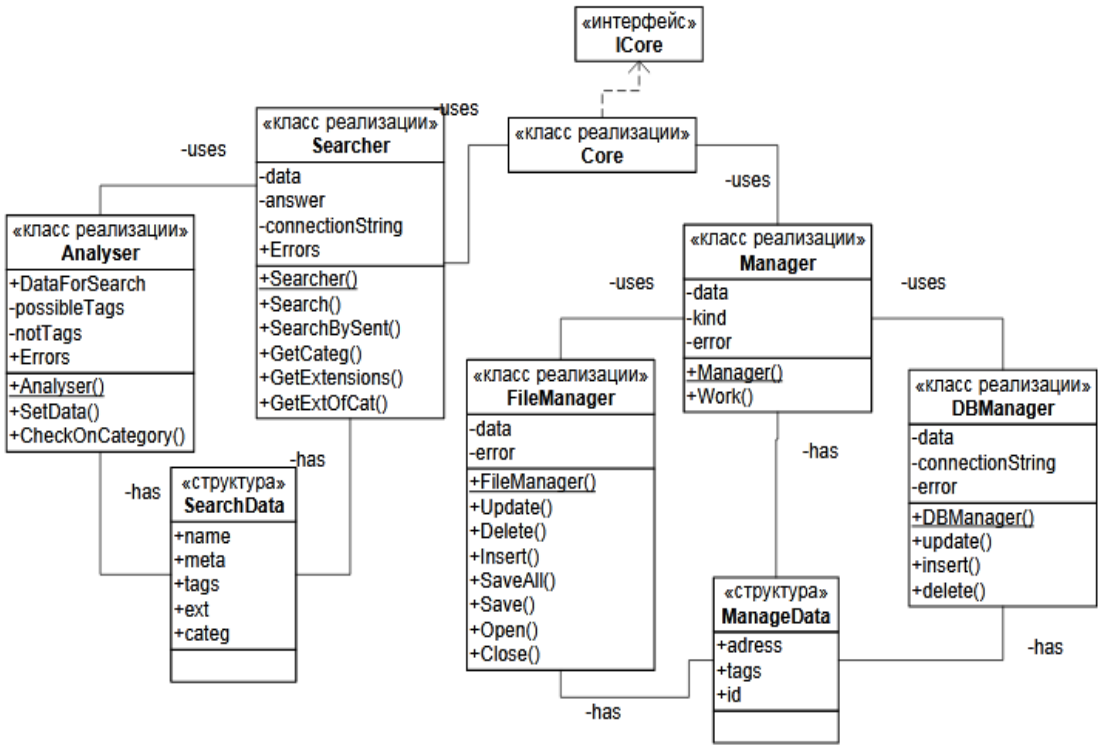

Рис. 2.2- Узагальнена діаграма класів ядра.

Ядро (Core) - керуюча система. Ядро складається з пошука (Searcher), що відповідає за пошук даних по базі даних, менеджера (Manager), що відповідає за управління взаємодією між менеджера файлів (FileManager) і менеджера бази даних DBManager. У свою чергу менеджер файлів працює з файлами зі сховища на рівні файлової системи, а менеджер бази даних - на рівні даних в базі даних. Для пошуку і для управління існують окремі

структури, в які зберігається інформація, за допомогою якої здійснюється необхідні дії, відповідно SearchData і ManageData. Також в ядрі міститься аналізатор (Analyser), який обробляє користувальницький запит для пошукача, так щоб його можна було зберегти як SearchData, за яким і буде складатися необхідні запити до бази даних.

2.1 API ядра

Для створення зручних інтерфейсів, в API передбачені основні методи, необхідні для роботи з сховищем. Їх реалізація прихована від користувачів.

API включає функції:

- пошук,
- виведення всіх файлів в системі, додавання файлу,
- видалення файлу, модернізація даних по файлу, збереження змін в файлах, відкриття / закриття файлу,
- видалити файл зі сховища (перенесення в звичайну файлову систему),
- перевірка існування файлу з таким же ім'ям (включаючи розширення файлу) в сховище,
- виведення всіх розширень файлів знаходяться в системі,
- виведення всіх розширень файлів, що відповідають конкретній категорії, виведення всіх категорій файлів системи.

2.2 Робота з файлом

Збереження, видалення, модифікація файлів складається з двох незалежних етапів: внесення даних в базу даних і обробка файлу на рівні файлової системи.

Самі файли зберігаються не в базі, вони знаходяться в окремій області файлової системи, спеціально виділеної під сховища. Так як немає необхідності зберігати файл всередині бази. Єдина проблема, яка може виникнути в даному випадку, видалення або переміщення і зміна файлу без синхронізації з базою даних. Однак щоб цього не сталося, файли будуть зберігатися в окремому прихованому місці, виділеному для зберігання, коли він додається в базу.

Причому, щоб уникнути проблем з іменами, було вирішено зберігати файли в вигляді файлів, імена яких відповідають їх ідентифікаторів в базі, а розширення є власним розширенням сховища.

Позитивні сторони даного підходу можна викласти в наступному. Ми, як було сказано вище, уникаємо проблеми з перейменуванням файлу при наявності іншого файлу з аналогічним ім'ям, але іншим змістом. При цьому доступ до нього повинен бути обмежений, оскільки файл не можна змінювати без повідомлення системи. При подібному підході ззовні не можна буде точно визначити "на око" який файл насправді знаходиться під тим чи іншим ідентифікатором. Також файл однозначно визначається, як зберігається в сховищі, що додає можливість проведення асоціацію між даними розширенням і додатком для роботи зі сховищем.

Потрібно зауважити, що при виникненні помилки на будь-якому етапі всі зміни в базі і в сховище, якщо вони були зроблені, повинні бути видалені.

2. 3 Додавання і видалення файлів

При збереженні дані про фото вводяться в базу даних.

Сам файл переноситися з поточного місця знаходження в сховище, окрема частина файлової системи. Перед цим здійснюючи перевірку на

існування файлів з таким же повним ім'ям, ім'я разом з розширенням. У разі існування аналогічних файлів, слід виводити їх у вигляді списку користувачеві, для того щоб він міг зорієнтуватися, чи був аналогічний файл вже внесений в сховище або в ньому міститься його стара версія, яку необхідно замінити.

Відповідно при видаленні аналогічно дані про нього видаляються з бази, а файл фізично видаляється зі сховища. Зауважимо, що в такому випадку в майбутньому необхідно передбачити відновлення, якщо треба, підвищуючи безпеку даних.

Крім повного видалення файлу, існує висновок файлу зі сховища, а саме збереження його по якому або адресою з його повним ім'ям і подальшим видаленням його зі сховища.

2.4 Відкриття файлу в сторонніх додатках

Для того щоб змінити сам файл, його потрібно відкрити через ядро системи, передавши йому, або адреса файлу, або ідентифікатор.

Процес відкриття файлу в сховище складається з двох етапів.

1. Створення копії файлу з його справжнім ім'ям в тимчасовій папці.

2. Відкриття копії у відповідній програмі.

Таким чином, задаються такі визначення: файл зі сховища є відкритим, якщо у нього існує тимчасова копія, зайнята в будь-якому процесі, модифікованим, якщо копія має більш пізню дату зміни файлу. Хочеться зауважити, що, по суті, модифіковані файли можна так само вважати відкритими, оскільки насправді зміни ще не були внесені в файл зі сховища.

В ядрі так само передбачені настройки, в яких зберігаються дані, про те які файли були відкриті і де їх тимчасова копія знаходиться. В такому випадку зручно замінювати стару версію файлу в сховище на тимчасовий файл, якщо в нього були внесені зміни і користувач бажає їх зберегти. Для цього можна послати ядру вказівку зберегти зміни (всі файли, відкриті до цього моменту, замінюються своїми тимчасовими копіями, потім копії видаляються) або зберегти зміни для конкретного файлу (конкретний файл замінюється на його тимчасову копію, копія видаляється). При цьому визначимо наступне, закрити файл в сховище - це наступна комбінація дій: прибрати файл зі списку відкритих, внісши або НЕ внісши зміни в сам файл, видалити копію. Поки мені не хочеться робити автоматичну заміну відкритого файлу при звільненні його копії процесами.

З легкістю закрити файл без внесення змін, якщо раптом були зроблені не бажані зміни. Така можливість теж передбачена в ядрі.

Ядро може повідомляти список відкритих файлів, для яких існують копії. Це полегшить моніторинг відкритих файлів.

2.5 Відстеження змін у файлі

Як було сказано вище, необхідно забезпечити спостереження за змінами в файлах. Це можна зробити за допомогою двома варіантами.

1. Запобігання зміни файлу без використання спеціального призначеного для користувача інтерфейсу.

2. Відстеження в реальному часі за зміною файлів.

Для відстеження в реальному часі необхідно, щоб система працювала весь час під час роботи комп'ютера. Крім того якщо система була відключена деякий час, то необхідно забезпечити сканування всіх файлів на предмет

невідповідність з даними з бази даних. В даному випадку нам немає необхідності відстеження в реальному часі, так як основне завдання сховища мультимедіа додавання ключових слів і реалізація пошуку по ним. Тому для реалізації прототипу системи був обраний перший варіант.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ

3.1 Пошук файлів

Ядро повертає набір даних містить дані з таблиць відповідають критеріям пошуку.

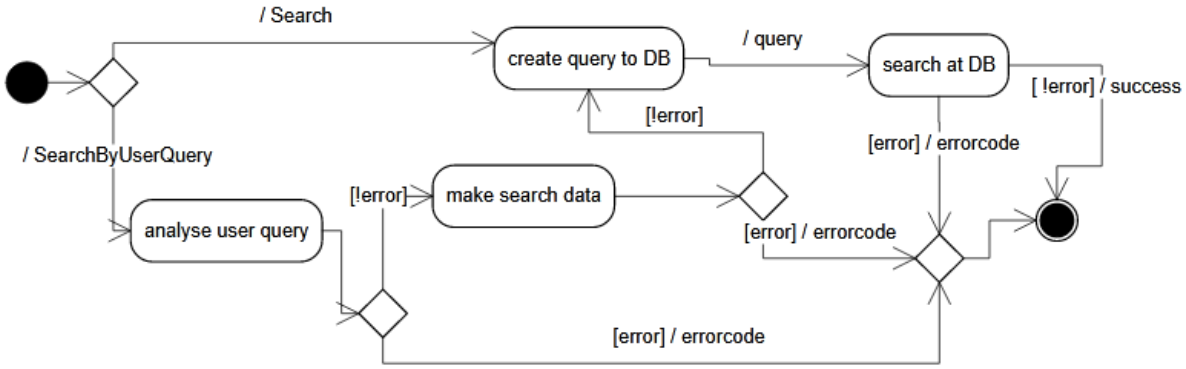

Рис.3.1 - Алгоритм роботи пошуку.

Загальна схема пошуку проста, ми отримуємо дані для пошуку, будуємо по ним запит до бази даних і посилаємо запит, якщо все в порядку повертаємо результат.

Було вирішено забезпечити користувачів можливістю шукати файли за допомогою так званих користувальницьких запитів, аналогічних інтернет пошуковим запитам до різних інтернет пошуковикам, таким як, наприклад, Google і Yahoo. Тобто користувальницький запит являє собою рядок, в якій перераховані всі критерії пошуку.

Якщо ж необхідно обробити користувальницький запит, то алгоритм ускладнюється. За допомогою аналізатора проводиться аналіз інформації з призначеного для користувача запиту, по ньому будуються дані для пошуку, а далі аналогічно звичайному пошуку.

3.2 Віртуальна файлова система

Віртуальна файлова система необхідна для того що б файли можна було відкривати без використання спеціального призначеного для користувача інтерфейсу над сховищем а прямо з сторонніх додатків. Тобто вона повинна емулювати традиційну файлову систему з деревовидної організацією файлів. При цьому ключові слова будуть відображатися в каталоги віртуальної файлової системи. А файли будуть відповідати файлам зі сховища.

Віртуальну файлову систему можна реалізувати різними варіантами. Однак створення віртуальної файлової системи з нуля і її реалізація в задачу даної роботи не входить. Тому, хочеться зауважити, яким чином її можна реалізувати і співвіднести зі сховищем.

По-перше, існує не мало розробок в сфері самих віртуальних файлових систем. Наприклад, Callback File System SDK дозволяє створювати віртуальні файлові системи і диски, які дозволяють керувати даними, так як якщо б вони були файли на локальному диску. Однак він не надається безкоштовно і не є відкритим, хоча вважається однією з кращих напрацювань у цій сфері.

По-друге, в якості віртуальної файлової системи можна використовувати локальний ftp сервер. Дане рішення значно простіше в реалізації, однак ускладнюється необхідністю налаштування можливості доступу до сервера, яка, наприклад, в Windows особливо жорстка для захисту самого користувача. Однак оскільки таке рішення простіше, воно і є оптимальним.

3.3 Пошук по запиту користувача

Під час реалізації виникло питання: як організувати пошук за запитом користувача? Ідея запиту користувача аналогічна по використанню користувачем пошуковим запитам до інтернет пошуковикам. Однак, так як сховище зберігає не тільки ASCII об'єкти, то будувати пошук аналогічно за алгоритмом не представляється можливим або скільки-небудь адекватним. Важливою властивістю використання призначеного для користувача запиту, яке треба реалізувати, і їх зручність, що можна отримати результат у вмісті абсолютно різного виду і поєднання критеріїв в ближчій для людини формі. Наприклад, «фотографії Каті з морем, на природі створені з 10.06 до 31.08»

3.4 Використання граматики

Пошукової запит містить в собі критерії пошуку, які треба розібрати, і максимально ефективно побудувати за цими критеріями запит до бази даних. Критерії пошуку поділяються на види: мітки, назва, категорія, розширення, дата створення або модифікації. Для розпізнавання інформації в рядку можна використовувати нейронні мережі. Однак розробка відповідної нейронної мережі і навчання її ж в більшості випадків складніше і тим самим невиправдано в рамках даного завдання. Завдання полягає в тому, щоб адекватно розібрати рядок на частини відносяться до одного виду. Її можна вирішити і за допомогою спеціально написаної граматики і парсера. Для реалізації цієї частини прототипу системи була обрана допоміжна відкрита бібліотека Irony.

За допомогою Irony була написана спеціальна граматика для

розбиття призначеного для користувача запиту в дерево. Правда, поки мова користувальницького запиту повинен бути англійською. Хотілося створити як би псевдо мову близький до людського. Але створити псевдо мову російською з урахуванням всіх відмін дуже складно, англійська ж не містить відмін слів.

Приклад конструкцій псевдо мови наступний: <something1> with

<Something2> або просто <something>, де <something> - список всіх даних про фото за якими вестиметься пошук, записані за правилами граматики. Крім with можна також використовувати where і about.

Така рядок введена користувачем передається ядру, де за допомогою, окремо написаної бібліотеки для роботи з одними запитами, SearchQueryAnalyser, яка працює з Irony і з граматикою, вона перетворюється в дерево, яке аналізується і дані з нього розбиваються на мітки, метадані, розширення і т.п. Так само теоретично можна вважати що перед «with» швидше за все ім'я або категорія. Так чином можна порівняти відразу дані перед with з набором категорій з бази (вибираються вони за допомогою збереженої функції), для спрощення пошуку. Тому граматика написана з урахуванням цієї особливості, так що в першій частині перед with не можна писати мітки.

Граматики складається з наступних терміналів: Terminal WITH = Symbol ("With"); Terminal ABOUT = Symbol ("About"); Terminal WHERE = Symbol ("Where"); Terminal comma = Symbol (","); Terminal pc = Symbol (";"); Terminal dot = Symbol ("."); Terminal ABOUT = Symbol ("About"); Terminal greater = Symbol (">"); Terminal less = Symbol ("<"); Terminal equal = Symbol ("=");

Terminal BETWEEN = Symbol ("Between"); Terminal AT = Symbol ("At");

Правила:

F .-> FE | SWOTg + WE;

FE.Rule -> GT + FE | GT | AdvT | AdvT + FE; GT -> PTg | Extens;

WithT -> WITH | ABOUT | WHERE;

WE -> WithT + PTg | WithT + AdvT | WE + PTg | WE + AdvT; SWOTg -> GT + SWOTg | GT;

AdvT -> MTgs | ME; MTgs -> TBSep + TgS;

TgS -> LngTg + TBSep + TgS | LngTg; Sep -> pc | comma;

LngTg -> LngTg + PTg | Tag | Eof; TBSep -> Tag + Sep;

Ineq -> greater | less | equal;

ME -> Ineq + Date | BETWEEN + Date + Date | AT + Date; PTg -> identifier;

Extens -> ext;

Tag -> identifier;

Date -> da

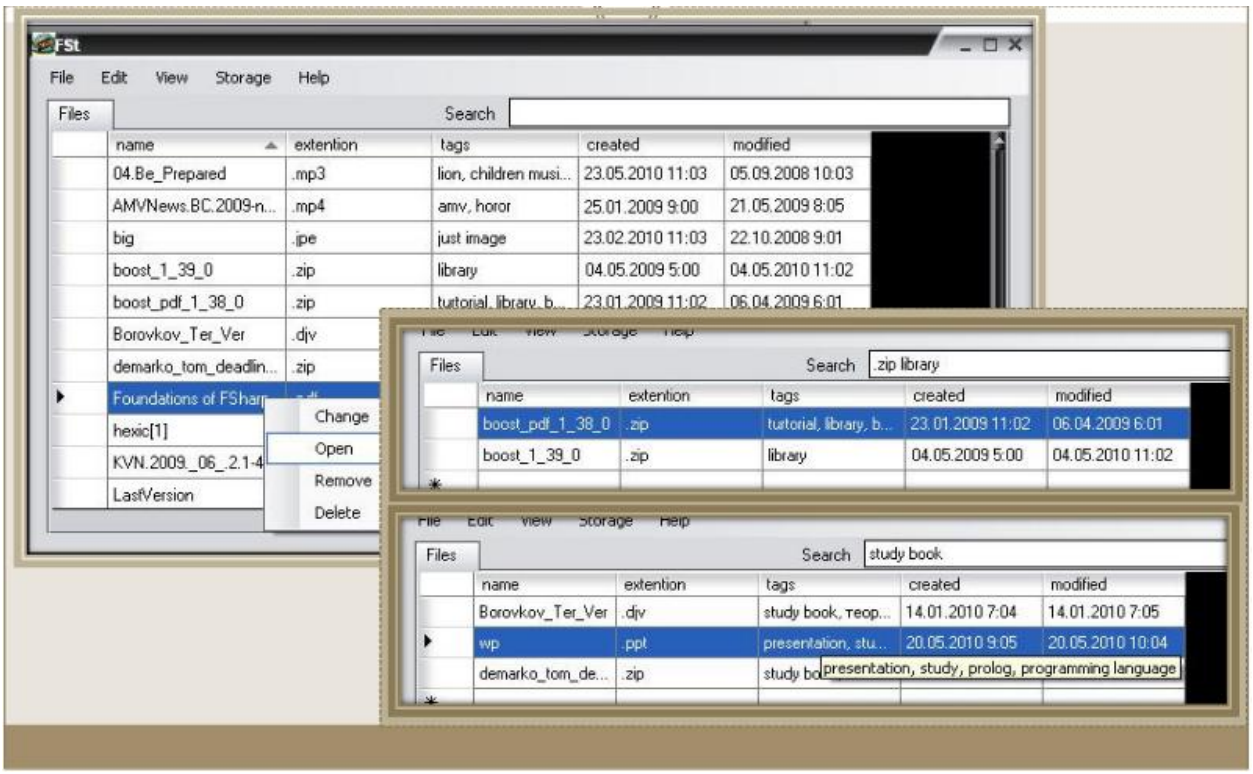

Рис. 3.2 – Інтерфейс користувача

ВИСНОВОК

При виконанні даної роботи були отримані наступні результати:

1. Проведено огляд і аналіз таких рішень:
    a) Windows Media Player;
    b) Adobe Photoshop Lightroom;
    c) сімейство операційних систем Windows NT: Windows XP, Windows Vista, Windows 7;
    d) Mac OS X;
    e) WinFS - система зберігання і управління даними на основі реляційної бази даних.
2. Спроектована архітектура прототипу системи і реалізовані наступні його компоненти:
    a) сховище з базою даних (Microsoft SQL Server);
    b) ядро (розроблено на мові C #);
    c) віртуальна файлова система.
3. Розроблено та реалізовано алгоритм пошуку файлів по пошуковим запитам в файловому сховищі.
4. Розроблено механізм підтримки і збереження цілісності та актуальності інформації в сховищі при модифікації файлів з сторонніх додатків.
5. Створено призначений для користувача інтерфейс для роботи з файловим сховищем.

СПИСОК ЛІТЕРАТУРИ

1. Новиков Б.А., Домбровська Г.Р. Налаштування додатків баз даних. Санкт-Петербург: БХВ-Петербург, 2006, 240с.
2. Chen P. The Entity-Relationship Model-Toward a Unified View of Data. // ACM Transactions on Database Systems (TODS), vol. 1, No. 1, 1976, ACM, New York, USA, p. 9-36.
3. Craig D. Lisa 1 Owner Guide, Apple Computer Inc. 1983. 418 p.
4. Date CJ An Introduction to Database Systems, 8th edition. Addison-Wesley, 2003. 1024 p.
5. Giampaolo DB Practical File System Design with the Be File System. San Francisco, California:Morgan Kaufmann Publishers, 1998. 256 p.
6. Stan M. Inside the Windows 95 file system. O'Reilly, 1997. 360 p.
7. Williams G. Journal Byte, BYTE Publications Inc, feb. 1983, p. 37-50.
8. Adobe. The library module. About Lightroom catalogs 2010. http://help.adobe.com/en_US/Lightroom/2.0/WS31C90D9B-2D4C-490f-B72F-EDD9D8DF60B6.html
9. Codeplex. Irony Project Specification, November 2009. http://irony.codeplex.com/
10. LisaFAQ. What is the Apple Lisa? 2006. http://lisafaq.sunder.net/single.html#lisafaq- hs_about_lisa
11. Macworld. Organize files with Spotlight comments, May, 2007. http://www.macworld.com/article/58012/2007/05/spotcomments.html

12. Microsoft. Що нового у Windows Media Player 11 для Windows Vista. Всі розваги в одному місці, 2010 року. http://www.microsoft.com/windows/windowsmedia/ru/player/windowsvista/features.aspx#AllYourEntertainmentinOnePlace
13. MSDN. A Developer's Perspective on WinFS: Part 1, March, 2004. http://msdn.microsoft.com/en-us/library/ms996622.aspx
14. MSDN. Blog of WinFS project, 2006. http://blogs.msdn.com/winfs/
15. MSDN. WinFS 101: Introducing the New Windows File System, March, 2004. http://msdn.microsoft.com/en-us/library/aa480687.aspx

ДОДАТОК А

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.IO;

namespace TveMA.SearchQueryAnalyser
{
    /// <summary>
    /// kind of data
    /// </summary>
    public enum DataKind
    {
        Ext,
        Meta,
        NotTag,
        PossibleTag,
        Tag
    }
```

```
public class TreeAnalyser
{
    private const string _xmlBuffer = "treeBuffer.xml";
    private XmlDocument tree;
    /// <summary>
    /// fills by path of visited and not left terms from xml
    /// </summary>
    private Stack<Term> stack;
    private bool noTags;
    /// <summary>
    /// possible current tag is continuation of last tag
    /// </summary>
    private bool possibleLongTag;
    /// <summary>
    /// how many tags could be one long tag
    /// </summary>
    private int possibleLongTagCount;
    /// <summary>
    /// should to join possible long tags
    /// </summary>
```

```
private bool addPrevToLastTg;
private Dictionary<DataKind, List<string>> _data;
/// <summary>
/// output data
/// </summary>
public Dictionary<DataKind, List<string>> Data
{
    private set { _data = value; }
    get { return _data; }
}

private TreeAnalyser() { }

public TreeAnalyser(XmlDocument inputTree)
{
    Data = new Dictionary<DataKind, List<string>>();
    foreach (DataKind dK in Enum.GetValues(typeof(DataKind)))
    {
        Data.Add(dK, new List<string>());
    }
    stack = new Stack<Term>();
```

```
    tree = inputTree;
}

/// <summary>
/// analyses input tree and fill output data
/// </summary>
/// <returns></returns>
public int AnalyseTree()
{
    using (FileStream str = File.Create(_xmlBuffer))
    {
        tree.Save(str);
    }
    try
    {
        stack.Push(Term.ParseTree);
        using (XmlTextReader reader = new XmlTextReader(_xmlBuffer))
        while (reader.Read())
        {
            switch (reader.NodeType)
            {
```

```
                case XmlNodeType.Element:
                    if (reader.AttributeCount == 3)
                    {
                        reader.MoveToFirstAttribute();
                        Term kind = (Term)Enum.Parse(typeof(Term), reader.Value);
                        //identifier
                        if (kind == Term.Ident)
                        {
                            AnalyseIdent(reader);
                        }
                        //extension
                        else if (kind == Term.Ext)
                        {
                            reader.MoveToAttribute(2);
                            Data[DataKind.Ext].Add(reader.Value);
                        }
                        else if (kind == Term.Day || kind == Term.Year || kind ==
Term.Month)
                        {
                            //to do
                        }
```

```
                }
                else
                {
                    if (reader.HasAttributes)
                        AnalyseTerm(reader);
                }
                break;
            case XmlNodeType.EndElement:
                stack.Pop();
                break;
            default:
                break;
        }
    }
    catch { return -1; }
    return 0;
}

/// <summary>
/// analyses situation where the term is current
```

```
/// </summary>
/// <param name="reader"></param>
private void AnalyseTerm(XmlTextReader reader)
{
    reader.MoveToFirstAttribute();
    var termKind = (Term) Enum.Parse(typeof(Term), reader.Value);
    stack.Push(termKind);
    switch (termKind)
    {
        case Term.ExprWithOutTag:
            noTags = true;
            break;
        case Term.WithExpr:
            noTags = false;
            break;
        case Term.TagBeforeSepr:
            if (possibleLongTag)
            {
                possibleLongTag = false;
                addPrevToLastTg = true;
            }
```

```
            break;
        case Term.LongTag:
            possibleLongTag = true;
            break;
        case Term.PossibleTag:
            if (possibleLongTag)
                possibleLongTagCount++;
            break;
    }
}

/// <summary>
/// analyses identifier where the term is current
/// </summary>
/// <param name="reader"></param>
private void AnalyseIdent(XmlTextReader reader)
{
    reader.MoveToAttribute(2);
    switch (stack.Peek())
    {
        case Term.Tag:
```

```
                if (addPrevToLastTg)
                {
                    addPrevToLastTg = false;
                    for (int i = 0; i < possibleLongTagCount; i++)
                    {
                        int lastIndex1 = Data[DataKind.Tag].Count - 1;
                        int lastIndex2 = Data[DataKind.PossibleTag].Count - 1;
                        Data[DataKind.Tag][lastIndex1] =
Data[DataKind.Tag][lastIndex1]
                            + " " + Data[DataKind.PossibleTag][lastIndex2];
                        Data[DataKind.PossibleTag].RemoveAt(lastIndex2);
                    }
                    possibleLongTagCount = 0;
                }
                Data[DataKind.Tag].Add(reader.Value);
                break;
            case Term.PossibleTag:
                if (noTags)
                    Data[DataKind.NotTag].Add(reader.Value);
                else
                    Data[DataKind.PossibleTag].Add(reader.Value);
```

```
                break;


            }
        }
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Irony.Parsing;
using System.Diagnostics;
using System.Windows.Forms;
using System.Xml;


namespace TveMA.SearchQueryAnalyser
{
    /// <summary>
    /// search query parser
    /// </summary>
    public class SQParser
```

```
{
    Grammar grammar;
    LanguageData language;
    Parser parser;
    ParsingContext parsingContext;
    ParseTree parseTree;
    string _elapsedTime;
    string _source;
    private TreeView tvParseTree;
    private List<CompErrors> gridCompileErrors;

    public SQParser(string source)
    {
        tvParseTree = new TreeView();
        gridCompileErrors = new List<CompErrors>();

        grammar = getGrammar();
        _source = source;
        ConstructParser();
    }
```

```
/// <summary>
/// initializes a new instance of grammar
/// </summary>
/// <returns></returns>
private Grammar getGrammar()
{
    Grammar grammar = new SQGrammar();
    return grammar;
}

/// <summary>
/// constructs parser
/// </summary>
private void ConstructParser()
{
    parseTree = null;
    grammar.ParseMethod = ParseMethod.Lalr;
    Stopwatch sw = new Stopwatch();
    try
    {
        sw.Start();
```

```
        language = new LanguageData(grammar);
        parser = new Parser(language);
        parsingContext = new ParsingContext(parser);
        sw.Stop();
    }
    finally
    {
        _elapsedTime = sw.ElapsedMilliseconds.ToString();
    }
}

/// <summary>
/// parses source
/// </summary>
public void Parse()
{
    ParseSample();
}

/// <summary>
/// returns
```

```
/// </summary>
/// <returns></returns>
public TreeView GetTree()
{
    ShowParseTree();
    return tvParseTree;
}

public List<CompErrors> GetErrors()
{
    ShowCompilerErrors();
    return gridCompileErrors;
}
public XmlDocument GetXmlTree()
{
    return parseTree.ToXmlDocument();
}

private void ParseSample()
{
    if (parser == null || !parser.Language.CanParse()) return;
```

```
    parseTree = null;
    parsingContext = new ParsingContext(parser);
    parsingContext.SetOption(ParseOptions.TraceParser, true);
    try
    {
       if(String.IsNullOrEmpty(_source))
          throw new Exception("there is no any input");
       parser.Parse(parsingContext, _source, "<source>");
    }
    catch (Exception ex)
    {
       gridCompileErrors.Add(new CompErrors(null, ex.Message, null));
       throw;
    }
    finally
    {
       parseTree = parsingContext.CurrentParseTree;
       ShowCompilerErrors();
       ShowParseTree();
    }
}
```

```
    private void ShowCompilerErrors()
    {
        gridCompileErrors.Clear();
        if (parseTree == null || parseTree.Errors.Count == 0) return;
        foreach (var err in parseTree.Errors)
            gridCompileErrors.Add(new CompErrors(err.Location, err.Message,
err.ParserState));
    }

    private void ShowParseTree()
    {
        tvParseTree.Nodes.Clear();
        if (parseTree == null) return;
        AddParseNodeRec(null, parseTree.Root);
    }

    private void AddParseNodeRec(TreeNode parent, ParseTreeNode nodeInfo)
    {
        if (nodeInfo == null) return;
        string txt = nodeInfo.ToString();
```

```
            TreeNode newNode = (parent == null ?
              tvParseTree.Nodes.Add(txt) : parent.Nodes.Add(txt));
            newNode.Tag = nodeInfo;
            foreach (var child in nodeInfo.ChildNodes)
               AddParseNodeRec(newNode, child);
        }
    }

    public struct CompErrors
    {
        public CompErrors(Nullable<SourceLocation> loc, string err, ParserState
state)
        {
            this.loc = loc; this.err = err; this.state = state;
        }
        Nullable<SourceLocation> loc;
        string err;
        ParserState state;
    }
}
/// <summary>
```

```
    /// search query grammar
    /// </summary>
    [Language("SearchFileData", "3", "Search of File Data Grammar")]
    class SQGrammar : Grammar
    {
        private HashSet<string> _keyterm = new HashSet<string>(new String[]
        { "with","about","where","between","at",
            ">","<","=","and"});

        public HashSet<string> KeyTerm
        {
            get { return _keyterm; }
            private set { _keyterm = value; }
        }

        public SQGrammar()
        {
            // Terminals
            IdentifierTerminal identifier = new
IdentifierTerminal(GetNameOfTerm(Term.Ident), IdFlags.IsNotKeyword);
            identifier.AllFirstChars += Strings.DecimalDigits;
```

```
        IdentifierTerminal ext = new
IdentifierTerminal(GetNameOfTerm(Term.Ext), IdFlags.NameIncludesPrefix
            & IdFlags.IsNotKeyword);
        ext.AllChars = Strings.AllLatinLetters;
        ext.AddPrefix(".", IdFlags.NameIncludesPrefix & IdFlags.IsNotKeyword);
        ext.EscapeChar = ' '; ext.AllFirstChars = ".";

        NumberLiteral day = new NumberLiteral(31,
GetNameOfTerm(Term.Day), 2, 1);
        day.Flags = NumberFlags.IntOnly; //day.EscapeChar = ' ';
        NumberLiteral mnth = new NumberLiteral(12,
GetNameOfTerm(Term.Month), 1, 2);
        mnth.Flags = NumberFlags.IntOnly;
        NumberLiteral year = new NumberLiteral(GetNameOfTerm(Term.Year),
4, 2);
        year.Flags = NumberFlags.IntOnly;
        KeyTerm WITH = ToTerm("with");
        KeyTerm ABOUT = ToTerm("about");
        KeyTerm WHERE = ToTerm("where");
        KeyTerm comma = ToTerm(",");
        KeyTerm and = ToTerm("and");
```

```
KeyTerm pc = ToTerm(";");

KeyTerm dot = ToTerm(".");

KeyTerm greater = ToTerm(">");

KeyTerm less = ToTerm("<");

KeyTerm equal = ToTerm("=");

KeyTerm BETWEEN = ToTerm("between");

KeyTerm AT = ToTerm("at");


// Non-terminals

var FE = CreateNonTerminal(Term.FullExpr);

var Tag = CreateNonTerminal(Term.Tag);

var AdvT = CreateNonTerminal(Term.AdvancedTerm);

var MTgs = CreateNonTerminal(Term.ManyTags);

var TgS = CreateNonTerminal(Term.TgS);

var PTg = CreateNonTerminal(Term.PossibleTag);

var Ineq = CreateNonTerminal(Term.Inequality);

var Date = CreateNonTerminal(Term.Date);

var Sep = new NonTerminal(",");

Sep.SetOption(TermOptions.IsPunctuation);

var Extens = CreateNonTerminal(Term.Extension);

var WithT = CreateNonTerminal(Term.WithTerm);
```

```
var WE = CreateNonTerminal(Term.WithExpr);
var GT = CreateNonTerminal(Term.GeneralTerm);
var TBSep = CreateNonTerminal(Term.TagBeforeSepr);
var F = CreateNonTerminal(Term.Full);
var SWOTg = CreateNonTerminal(Term.ExprWithOutTag);
var ME = CreateNonTerminal(Term.MetaExpr);
var LngTg = CreateNonTerminal(Term.LongTag);

// BNF rules
this.Root = F;
F.Rule = FE
       | SWOTg + WE;
FE.Rule = GT + FE
      | GT
      | AdvT
      | AdvT + FE;
GT.Rule = PTg
          | Extens;

WithT.Rule = WITH
             | ABOUT
```

```
            | WHERE;
WE.Rule = WithT + PTg
            | WithT + AdvT
            | WE + PTg
            | WE + AdvT;
WE.SetOption(TermOptions.IsList);

SWOTg.Rule = GT + SWOTg
          | GT;

AdvT.Rule = MTgs
           | ME;
MTgs.Rule = TBSep + TgS;
TgS.Rule = LngTg + TBSep + TgS
         | LngTg;

Sep.Rule = pc
          | comma;
LngTg.Rule = LngTg + PTg
          | Tag
          | Eof;
```

```
    TBSep.Rule = Tag + Sep;
    Ineq.Rule = greater
          | less
          | equal;
    ME.Rule = Ineq + Date
                | BETWEEN + Date + Date
                | AT + Date;
    PTg.Rule = identifier;
    Extens.Rule = ext;
    Tag.Rule = identifier;
    Date.Rule = day + dot + mnth + dot + year
              | day + dot + mnth;

    // Operators precedence

    MarkTransient(GT, AdvT, LngTg, TgS, SWOTg, FE, Ineq);
}

/// <summary>
/// creates non-terminal by constant from enumeration
/// </summary>
```

```
        /// <param name="value"></param>
        /// <returns></returns>
        private NonTerminal CreateNonTerminal(Term value)
        {
            return new NonTerminal(GetNameOfTerm(value));
        }
        /// <summary>
        /// gets name of constant from enumeration
        /// </summary>
        /// <param name="value"></param>
        /// <returns></returns>
        private static string GetNameOfTerm(Term value)
        {
            return Enum.GetName(typeof(Term), value);
        }
    }
}
```