Profiling - Cabinet veterinar

Ghiurțu Andrei-Ștefan - 10LF312
Olaru Karina-Elena - 10LF313

1. Prezentarea scenariului

Un cabinet veterinar cu vechime, care pentru mult timp a folosit o aplicație rămasă în urmă din punct de vedere tehnologic, a cărui cod sursa nu poate fi recuperat și modificat, ba chiar acest lucru ar fi o decizie proastă pentru afacere datorită costurilor ridicate, decide sa pună la punct o nouă aplicație dedicată doar acestui cabinet. Ei vor totodată sa se extindă și au nevoie sa afle statistici despre animalele care le trec pragul pentru a ști exact în ce arie trebuie sa investească mai mulți bani pentru a putea crește profitabilitatea cabinetului.

Fiecărui animal care a fost introdus în baza de date i s-a atribuit un cod de identificare personalizat din zece cifre, care a fost compus astfel: prima cifră reprezintă tipul de animal, următoarea este dată de sexul acestuia (0 pentru mascul, 1 pentru femelă), iar ultimele opt sunt data de naștere a animalului (an, lună, zi), dacă aceasta se cunoaște, sau opt cifre de 0 dacă nu se cunoaște. Animalele sunt împărțite în cinci categorii: animale de companie ținute în casă (cățel, pisică, hamster etc.), animale domestice, de curte (vacă, porc, cal etc.), animale exotice (șopârla, șarpe, iguana, etc.), păsări (rață, găină, porumbel, etc.) și pești de acvariu.

2. Unelte externe

- Visual Studio 2019 + 2022 (cu build tools pentru 2019) - IDE
- vcpkg - package manager
- sqlite - baza de date - instalată prin vcpkg
- Google Chrome Tracing - profiling vizual bazat pe un fișier json generat în timpul rulării programului

Deși există multe unelte specializate pentru partea de profiling, acestea necesită o adaptare la mediu, sunt destul de greu de utilizat în primele dăți și

deși pentru un utilizator experimentat complexitatea ridicată poate fi un factor în favoarea alegerii unui anumit profiler, pentru un începător profiler-ul pe care îl oferă Google Chrome (sau orice alt browser ce este bazat pe Chromium) este un loc prielnic pentru a putea extrage informații de bază despre codul lor. În plus, profiler-ul acesta poate fi folosit pretutindeni, trebuie să fie instalat un browser și să ai acces la fișierul de profiling (de tip JSON) pentru a putea să inspectezi ce se întâmplă în interiorul aplicației tale.

3. Arhitectură

Pentru a structura informațiile despre un animal am implementat o clasa în care am reținut un ID (cheia primară din baza de date), un cod de identificare personalizat (pe care l-am descris în detaliu mai sus), numele animalului și data în care acesta a fost înregistrat prima oară.

Metodele de validare și recunoaștere a tipului de animal au fost împărțite în metode ce folosesc expresii regulate și metode implementate fără ajutorul acestora. Seturile de metode ce țin de fiecare din tipurile de validare au fost incluse în namespace-uri separate și adăugate într-un namespace comun de validare.

În fișierul principal al proiectului sunt implementate și funcții care fac apeluri către baza de date (inserează *n* animale generate aleator, selectează și șterge animale în/din aceasta).

Pentru gruparea animalelor pe categorii în funcție de tipul acestora s-a folosit un *unordered_map* care are ca și cheie tipul, iar valoarea este un vector *STL* de animale. Scopul acestei grupări este de a structura mai bine datele, testele putând fiind efectuate atât pe toate animalele din baza de date, cât și pe anumite categorii/subcategorii.

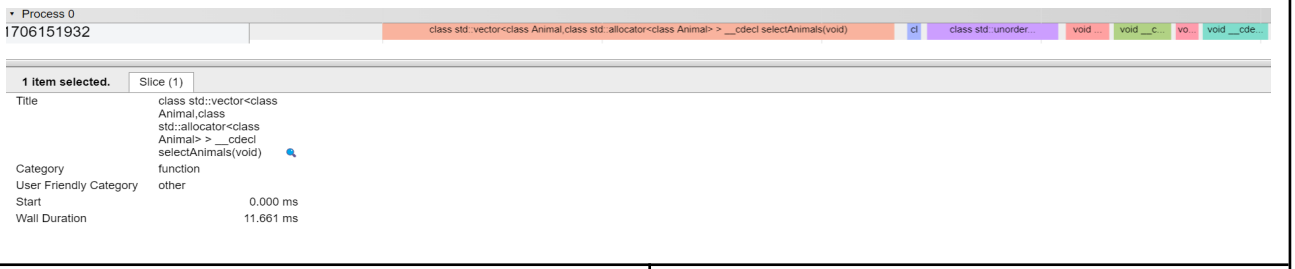
Pentru a genera fișierul de profiling s-au folosit două clase: **Intrumentor** și **IntrumentationTimer** [1]**. IntrumentationTimer** setează la inițializare timpul de start, iar la distrugere calculează diferența de timp dintre momentul actual (la care se întâmplă distrugerea) și timpul de start, setează thread-ul pe care s-au efectuat operațiile și cheamă funcția de scriere în fișier din **Instrumentor**.

Instrumentor este o clasă *singleton* care se ocupă de scrierea în fișier și de gestiunea sesiunii pentru care se face profiling. Profilingul se poate salva în fișiere diferite, la nevoie, și fiecare va reține informații despre numele funcției (pentru cazul nostru), durata de execuție și threadul pe care a fost executată funcția.

Pentru folosirea lor au fost adăugate două macrocomenzi ce construiesc obiectul de tip **InstrumentationTimer** și una ce o cheamă pe precedenta cu numele dat drept numele funcției din care a fost apelată macrocomanda [2].

4. Studierea profilului și a performanței

Tabelul următor prezintă comparația de performanță între validări de tip regex și validări personalizate pentru 10 elemente, folosind un *for* cu iterator.



Title	groupAnimalsCustom
Category	function
User Friendly Category	other
Start	12.307 ms
Wall Duration	0.208 ms

Title	groupAnimalsRegex
Category	function
User Friendly Category	other
Start	12.604 ms
Wall Duration	2.744 ms

Title	analyzeAnimalBirthdayCustom
Category	function
User Friendly Category	other
Start	15.508 ms
Wall Duration	0.782 ms

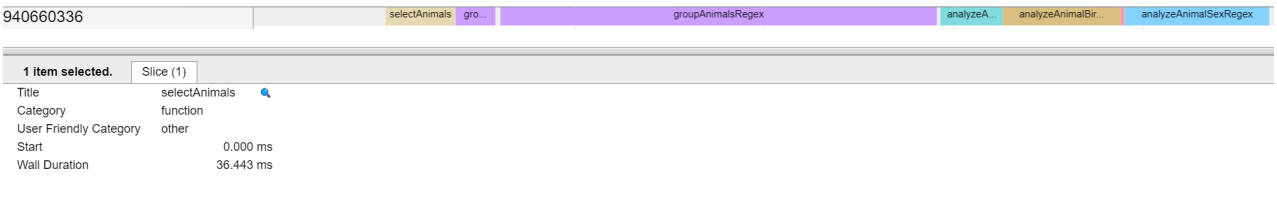
Title	analyzeAnimalBirthdayRegex
Category	function
User Friendly Category	other
Start	16.377 ms
Wall Duration	1.413 ms

Title	analyzeAnimalSexCustom
Category	function
User Friendly Category	other
Start	17.898 ms
Wall Duration	0.632 ms

Title	analyzeAnimalSexRegex
Category	function
User Friendly Category	other
Start	18.589 ms
Wall Duration	1.399 ms

Tabel 1.1

Al doilea tabel prezintă comparația de performanță între validări de tip regex și validări personalizate pentru 1000 elemente, folosind un *for* cu iterator.



Title	groupAnimalsCustom
Category	function
User Friendly Category	other
Start	36.765 ms
Wall Duration	20.959 ms

Title	groupAnimalsRegex
Category	function
User Friendly Category	other
Start	60.481 ms
Wall Duration	230.512 ms

Title	analyzeAnimalBirthdayCustom
Category	function
User Friendly Category	other
Start	292.929 ms
Wall Duration	32.925 ms

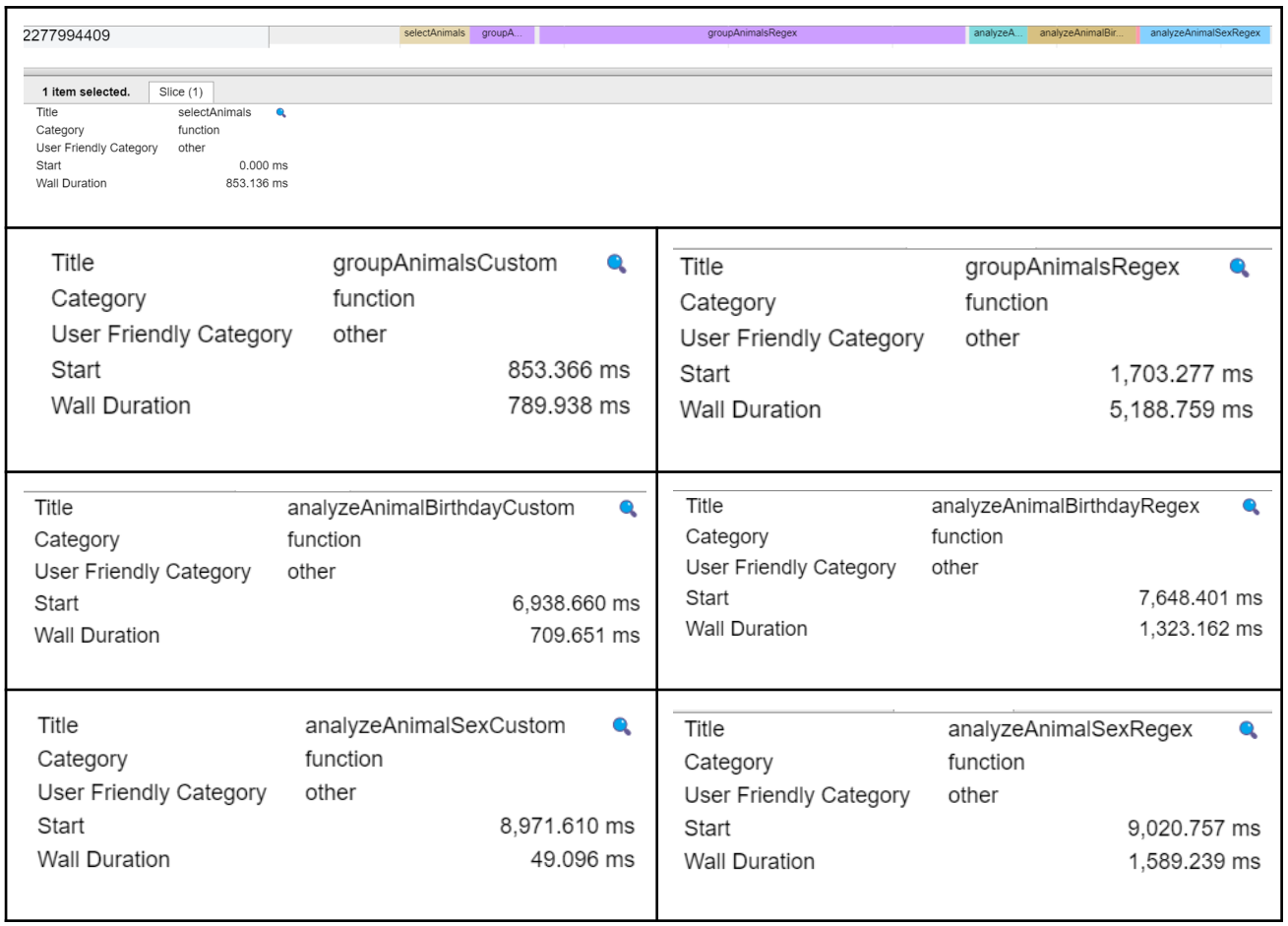
Title	analyzeAnimalBirthdayRegex
Category	function
User Friendly Category	other
Start	325.930 ms
Wall Duration	61.975 ms

Title	analyzeAnimalSexCustom
Category	function
User Friendly Category	other
Start	387.982 ms
Wall Duration	2.111 ms

Title	analyzeAnimalSexRegex
Category	function
User Friendly Category	other
Start	390.162 ms
Wall Duration	76.622 ms

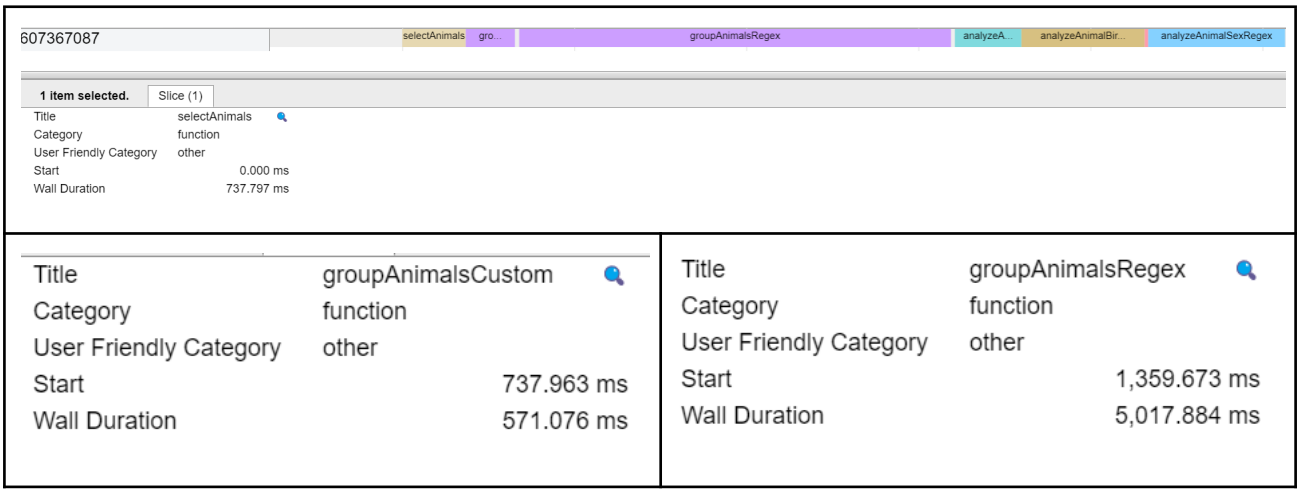
Tabel 1.2

Al treilea tabel prezintă comparația de performanță între validări de tip regex și validări personalizate pentru 40000 elemente, folosind un *for* cu iterator.



Tabel 1.3

Ultimul tabel prezintă comparația de performanță între validări de tip regex și validări personalizate pentru 40000 elemente, folosind un *for* cu indice.



Title	analyzeAnimalBirthdayCustom
Category	function
User Friendly Category	other
Start	6,423.162 ms
Wall Duration	773.215 ms

Title	analyzeAnimalBirthdayRegex
Category	function
User Friendly Category	other
Start	7,196.430 ms
Wall Duration	1,433.500 ms

Title	analyzeAnimalSexCustom
Category	function
User Friendly Category	other
Start	8,629.982 ms
Wall Duration	40.506 ms

Title	analyzeAnimalSexRegex
Category	function
User Friendly Category	other
Start	8,670.577 ms
Wall Duration	1,589.848 ms

Tabel 2

5. Comentarii asupra rezultatelor

Criteriile studiate au la bază diferențele dintre utilizarea expresiilor regulate pentru găsirea atributelor fiecărui animal prin id-ul personal sau folosirea unei funcții scrise de la zero exact pentru contextul de folosire. Totodată, comparația dintre tabelele **1.3** și **2.1** dorește a găsi diferența de viteză dintre utilizarea unei structuri repetitive bazate pe iteratorul din *STL* și o structură bazată pe index.

- ### Tabel 1.1:

  Primul test definește punctul de start al comparației și demonstrează chiar pe un număr foarte mic de elemente (10) că diferențele de performanță dintre funcțiile personalizate și cele în care se utilizează expresii regulate sunt considerabile și nu ar trebui ignorate în momentul în care se vrea a se optimiza la maxim timpul de execuție. Se observă diferențe chiar de 13 ori între varianta custom de găsire a tipului de animal și gruparea acestora în *map* și cea ce utilizează clasa *regex*, aceste funcții fiind cele mai complexe dintre funcțiile de validare.

- Tabel 1.2:

În cel de-al doilea test se poate observa că diferențele între variațiile de implementare devin mai vizibile. Observăm că ultima analiză făcută, cea care validează tipul de sex al animalului (și în care diferența dintre cele două implementări era de 50%, în testul din tabelul anterior) are acum o diferență de 3600% în favoarea implementării personalizate.

- Tabel 1.3 și Tabel 2:

Pentru 40000 de elemente diferențele încep cu adevărat să devină resimțite de către utilizator din punct de vedere al timpului de așteptare, iar acest lucru duce în final la pierderea unui număr considerabil de oameni care utilizează aplicația. Cel mai semnificativ test este tot funcția care se ocupă de gruparea animalelor, unde varianta cu *regex* durează cinci secunde, comparativ cu jumătatea de secundă de așteptare necesară celeilalte implementări.

În testele efectuate *for*-ul bazat pe indice nu este întotdeauna mai eficient. Pentru funcțiile de dimensiuni mici diferențele sunt oricum nesemnificative, deci nu se poate stabili o variantă optimă de a structura codul. Totuși, pe testul de grupare a tipurilor de animale diferența este notabilă între cele două abordări și poate constitui motivul pentru care s-ar alege o variantă în detrimentul celeilalte, în contextul respectiv (*for*-ul bazat pe index este constant mai rapid cu 200 ms).

6. Concluzii

Astfel, observăm că fiecare detaliu de implementare poate deveni important de analizat utilizând un profiler vizual, exemplul de mai sus fiind unul minimal pentru a demonstra importanța acestuia. Un profiler este, în

general, utilizat pentru a observa lucrul pe mai multe fire de execuție, pe un număr ridicat de funcții și pe o perioadă îndelungată pentru a putea stabili cu exactitate de unde apare o problemă de performanță și unde s-ar putea face îmbunătățiri. Totuși, se observă că, deși varianta cu iterator este cea recomandată în general, chiar și utilizarea unui *for* bazat pe indice poate duce la câștigarea de timp în contexte critice și merită a fi luat în considerare în momentul profilingului.

7. Referințe

[1] Basic Instrumentation Profiler (github.com) - Yan Chernikov

[2] VISUAL BENCHMARKING in C++ (how to measure performance visually) - YouTube - Yan Chernikov