PL/SQL

—

Algoritmi Procedurali Intr-o Lume Declarative

Theodor Moroianu

November 26, 2020

Contents

1 Introducere

În acest scurt referat o să prezint implementarea si testarea performantelor a doi algoritmi clasici pe trei platforme diferite: Serverul Oracle PL/SQL, un interpretator de Python, si un executabil implementat in C++.

Mai precis, o sa prezint eficienţa functiei recursive care calculeaza al K-lea termen *Fibbonacci*, respectiv a functiei care calculeaza numarul de numere prime intr-un interval, folosind ciurul lui Eratostene.

Scopul limbajului SQL nu este de-a rula in mod eficient algoritmi procedurali, dar avand in vedere ca standardul PL/SQL ne permite acest lucru consider ca este un experiement interesant, al carui rezultat are implicatii asupra posibilitatii folosirii mediului PL/SQL pentru implementarea unor algoritmi mai complecşi.

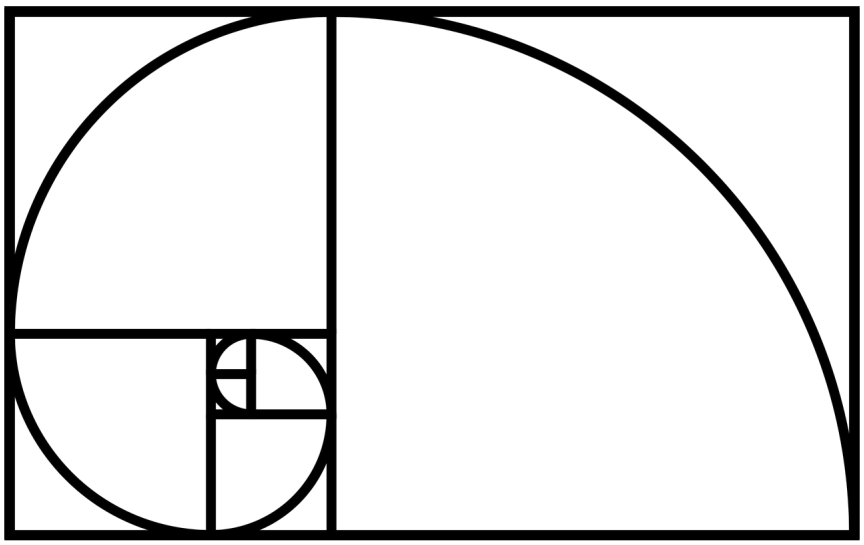

Figure 1: Reprezentarea numerelor Fibbonacci

2 Modul de Testare al Eficientei

Pentru a testa eficienta celor doi algoritmi intr-un mediu PL/SQL, am decis sa rulez aceiasi algoritmi in limbaje de programare diferite.

Astfel, o sa rulam functii echivalente in trei medii:

- PL/SQL, cu server de Back-End *Oracle Express Edition XE*
- *Python*, cu interpretatorul *Python 3.8*
- $C++$, cu compilatorul $MinGW$.

De notat ca pentru ca performantele calculatoarelor pe care sunt testati algoritmii sa nu joace un rol toate testele sunt facute pe acelasi calculator, cu un procesor $i7 - 7700$, $16GB$ de RAM, un SSD si fara procese care sa consume resurse pe fundal.

3 Codul de Python

Codul de Python care calculeaza al k-lea termen Fibbonacci este urmatorul:

```python
# Calcuelaza al k-lea numar fibbonacci (modulo 1000000)
def Fibb(n: int):
    if n <= 1:
        return n
    return (Fibb(n - 1) + Fibb(n - 2)) % 1000000000
```

Functia care calculeaza numarul de numere prime este acesta:

```python
# Numar de numere prime in intervalul 2-n
def CountPrimes(n: int):
    ciur = [False for i in range(n + 1)]
    raspuns = 0
    for i in range(2, n + 1):
        if not ciur[i]:
            for j in range(i, n + 1, i):
                ciur[j] = True
            raspuns += 1
    return raspuns
```

4 Codul de C++

Codul in C++ este similar cu cel de Python, dar sintaxa este putin diferita. Astfel, implementarile sunt:

```cpp
// Calculeaza al k-lea numar fibbonacci (modulo 1000000)
int Fibb(int n)
{
    if (n <= 1)
        return n;
    return (Fibb(n - 1) + Fibb(n - 2)) % 1000000000;
}

// Numar de numere prime in intervalul 2-n
int CountPrimes(int n)
{
    vector <bool> ciur(n + 1);
    int raspuns = 0;
    for (int i = 2; i <= n; i++) {
        if (!ciur[i]) {
            for (int j = i; j <= n; j += i)
                ciur[j] = 1;
            raspuns++;
        }
    }
    return raspuns;
}
```

5 Codul de PL/SQL

Functiile sunt scrise in PL/SQL, pentru ca SQL pur nu ne permite operatii procedurale. Astfel, functia de calculare al celui de-al K lea numar Fibbonacci este:

```sql
-- Functie care calculeaza al k-lea numar fibbonacci, modulo 10^9
CREATE OR REPLACE FUNCTION Fibb
    (n          INTEGER)
RETURN INTEGER
IS
BEGIN
    IF n <= 1 THEN
        RETURN n;
    END IF;
    RETURN MOD(Fibb(n - 1) + Fibb(n - 2), 1000000000);
END;
/
```

Functia care calculeaza numarul de numere prime, implementata cu ajutorul tipului de date colectie $Vector$, este:

```
-- Functie care calculeaza numarul de numere prime in intervalul [2-n]
CREATE OR REPLACE FUNCTION CountPrimes
    (n          INTEGER)
RETURN INTEGER
IS
    TYPE vector IS VARRAY(100000000) OF BOOLEAN;
    ciur    vector := vector();
    ans     BINARY_INTEGER;
    indice  BINARY_INTEGER;
BEGIN
    ans := 0;
    FOR i IN 1..n LOOP
        ciur.extend;
        ciur(i) := False;
    END LOOP;
    FOR i IN 2..n LOOP
        IF ciur(i) = False THEN
            ans := ans + 1;
            indice := i;
            WHILE indice <= n LOOP
                ciur(indice) := True;
                indice := indice + i;
            END LOOP;
        END IF;
    END LOOP;
    RETURN ans;
END;
/
```

Apelul celor doua functii de PL/SQL se face cu ajutorul urmatoarelor blocuri anonime:

```
DECLARE
    ans     INTEGER;
    val     INTEGER := 10;
BEGIN
    ans := Fibb(val);
    DBMS_OUTPUT.PUT_LINE('Fibb(' || val || ') = ' || ans);
END;
/
```

```
DECLARE
    ans     INTEGER;
    val     INTEGER := 10000000;
BEGIN
    ans := CountPrimes(val);
    DBMS_OUTPUT.PUT_LINE('CountPrimes(' || val || ') = ' || ans);
END;
/
```

6 Verificare de Corectitudine

Primul lucru de verificat este ca pentru ambii algoritmi cele 3 implementari dau acelasi rezultat. Acest lucru se verifica usor cu ajutorul unor teste relativ mari.

Verificare in Python:

```
print(CountPrimes(1000))
# Afiseaza '168'
print(Fibb(20))
# Afiseaza '6765'
```

Verificare in C++:

```
cout << CountPrimes(1000) << '\n';
// Afiseaza '168'
cout << Fibb(20) << '\n';
// Afiseaza '6765'
```

Verificare in SQL:

```
DECLARE
    ans     INTEGER;
    val     INTEGER := 1000;
BEGIN
    ans := CountPrimes(val);
    DBMS_OUTPUT.PUT_LINE('CountPrimes(' || val || ') = ' || ans);
END;
/
-- Afiseaza 'CountPrimes(1000) = 168'
DECLARE
    ans     INTEGER;
    val     INTEGER := 20;
BEGIN
    ans := Fibb(val);
    DBMS_OUTPUT.PUT_LINE('Fibb(' || val || ') = ' || ans);
END;
/
-- Afiseaza 'Fibb(20) = 6765'
```

Faptul ca cele 3 implementari dau in ambele cazuri acelasi raspuns ne ofera o oarecare garantie ca implementarile sunt corecte.

7 Benchmark-uri

Putem acum sa ne concentram atentia asupra timpului de executie a celor 3 algoritmi. Am rulat de mai multe ori fiecare test, luand valoarea mediana pentru o precizie mai buna.

Timpul de executie pentru calculearea celui de-al K-lea numar Fibbonacci:

K	$C++$	$Python$	SQL
20	23 ms	116 ms	38 ms
25	25 ms	161 ms	89 ms
30	29 ms	413 ms	657 ms
35	79 ms	706 ms	7.5 s
40	593 ms	27.8 s	82.2 s

Timpul de executie pentru calcularea numerelor prime pana in N:

N	$C++$	$Python$	SQL
10^3	21 ms	119 ms	25 ms
10^4	32 ms	131 ms	37 ms
10^5	52 ms	160 ms	47 ms
10^6	68 ms	368 ms	474 ms
10^7	527 ms	3.2 s	5.6 s
$2 * 10^7$	1 s	6.1 s	12.4 s
$3 * 10^7$	1.6 s	9.6 s	18.6 s

8 Analiza Datelor

C++ este de departe castigator in ambele cazuri. Notez ca nici macar nu a fost compilat cu flag-uri de optimizare, pentru a putea fi comparabil cu ceilalti (codul de C++ compilat cu "$-O2$" rula pe toate testele in cateva milisecunde).

Motivul pentru aceasta discrepanţă uriasa este datorat faptului ca C++ este un limbaj compilat. Apelul de functii in C++ este de asemenea foarte optimizat.

Astfel, din datele extrase putem intelege:

- C++ este mai rapid decat Python, care la randul sau este mai rapid decat SQL.
- Desi SQL este pe ultimul loc in toate testele, acesta are o eficienta surprinzator de buna, considerand ca nu a fost gandit pentru efectuarea operatilor procedurale.
- Recursivitatea este deosebit de lenta in SQL. Subprogramul SQL recursiv este de 140 de ori mai lent decat subprogramul echivalent in C++, dar cel fara recursivitate de doar 11 ori.

- Cea mai importanta observatie pe care am putut sa o fac este ca implementarea acestor algoritmi este posibila in PL/SQL, fara complicatii prea mari la implementare, si timpul de executie al acestora, desi mai mare decat in limbaje specializate precum C++ sau Python, este moderat, si cererile se executa in timp util, chiar pe teste foarte mari.

9 Concluzie

Cand am inceput acest proiect ma asteptam la rezultate mult mai slabe, si am fost pozitiv surprins de viteza serverului Oracle Express Edition, care a reusit sa efectueze operatii complet diferite decat cele standard in SQL intr-un timp moderat.

Motivatia din spatele acestui proiect era de-a vedea daca este fezabil sa includ in proiectul final din cadrul cursului de SGBD algoritmi mai complecşi.

Consider rezultatele pe care le-am obtinut un suces, si o confirmare a posibilitatii implementarii algortimilor in proceduri PL/SQL.