ALGORITMI PARALELI ȘI DISTRIBUIȚI: Tema 2
Mini Database Engine

Termen de predare: 1 Decembrie 2017

Titulari curs: *Valentin Cristea, Ciprian Dobre, Elena Apostol*

Responsabili temă: **Cristian Chilipirea, Radu-Ioan Ciobanu, Valeriu Stanciu**

Updatat pe 18/11, 23:20 şi 27/11, 00:10

Obiectivele temei

Scopul acestei teme este implementarea unui software minimalist de management al unei baze de date ce se va folosi de tehnologia multi-threading pentru a obține timpi de execuție scăzuți.

Sistemul de management al bazei de date va avea suport pentru multiple tabele asupra cărora se pot efectua operațiile insert, select și update. Operațiile pe tabele diferite vor fi complet independente (nu vor exista operații de tip join). Mai mult, operațiile de citire (select) din aceeași tabelă se pot efectua în paralel.

Acest software va folosi thread-uri multiple si va fi implementat în limbajul Java.

Un punct important al acestei teme va fi obținerea de scalabilitate cu privire la numărul de thread-uri și demonstrarea acesteia.

Operații suportate

Va trebui să porniți implementarea sistemului de management al bazei de date de la interfața oferită în scheletul de cod (în fișierul **MyDatabase.java**). Această interfață conține funcțiile:

- `initDb(numWorkerThreads)` - inițializează o bază de date cu `numWorkerThreads` workeri
- `stopDb()` - oprește serviciul de bază de date
- `createTable(tableName, columnNames, columnTypes)` - crează tabela `tabelName` ce va conține coloanele `columnName` cu tipurile aferente `columnTypes`
- `select(tableName, operations, condition)` - citește din tabela `tableName` și întoarce o listă cu atâtea coloane câte elemente sunt în lista `operations`; fiecare coloană va conține un număr de elemente dependent de tipul operației, condiție și elementele din tabelă
- `update(tableName, values, condition)` - scrie în tabela `tableName` valorile `values` doar în locațiile în care `condition` este `true`
- `insert(tableName, values)` - scrie la sfârșitul tabelei o nouă linie cu valorile `values`
- `startTransaction(tableName)` - începe o tranzacție; toate operațiile din interiorul unei tranzacții se vor executa atomic (ca o singură operație)
- `endTransaction(tableName)` - finalizează tranzacția.

Parametrii acestor funcții și tipurile lor sunt:

- `tableName` - numele tabelei asupra căreia se va executa funcția (`String`)
- `columnNames` - numele coloanelor din tabelă (`String[]`)
- `columnTypes` - tipurile coloanelor din tabelă (`String[]`, cu valori acceptate "string", "int" sau "bool")
- `condition` - o condiție (dată ca `String`) de tipul "columnName comparator value" (cu spaţiu între componente):
  - `comparator` poate fi "<", ">" sau "=="
  - `value` poate fi un număr, o valoare booleană sau un simplu șir de caractere, în funcție de tipul coloanei
  - în cazul în care condiția nu conține niciun caracter, operația de select sau insert se va aplica asupra întregii coloane
  - exemple de condiții: "grade $>$ 5", "studentName $==$ Ioana', "passed $==$ true', etc.
  - operaţiile de "<" şi ">" se pot aplica doar pentru coloane cu valori de tip "int".
- `values` - o listă de valori (`ArrayList<Object>`, se potrivește cu numărul și tipul coloanelor din tabelă)
- `operations` - variabilă care reprezintă operațiile pentru select (`String[]`):
  - poate fi o listă de nume de coloane, semnificând faptul că doar acele coloane trebuie returnate
  - poate fi o listă de oricâte funcții de agregare: "min", "max", "sum", "count", "avg"; fiecare din aceste funcții primește numele unei coloane ca parametru (de exemplu, "min(columnName)"); cu excepţia "count" (care se poate aplica indiferent de tipul de date din coloană), celelalte funcţii de agregare se pot aplica doar pe valori de tip "int"
  - poate fi o combinaţie de nume de coloane şi funcţii de agregare.

Toate aceste funcții ar trebui implementate ca blocante. Vor întoarce un rezultat doar când acesta este gata. Tratarea erorilor este la latitudinea voastră, dar se recomandă aruncarea de excepţii.

Arhitectură și hint-uri

Pentru a putea evidenția implementarea folosind thread-uri multiple a codului vostru, trebuie să simulăm mai mulți clienți care trimit cereri software-ului scris de voi. Astfel, vom porni mai multe thread-uri care apelează funcțiile de select, insert sau update. Aceste thread-uri vor fi pornite de modulul de test, dar sunteți responsabili de sincronizarea operațiilor pe care le execută din interiorul celor 3 funcții.

În funcție de nevoi, cele 3 funcții (select, insert, update) ar trebui să creeze job-uri care să fie executate de un set de thread-uri pe care le controlați (creați, închideți, sincronizați). Pentru a facilita controlul asupra acestor thread-uri, se pot folosi funcțiile de inițializare și închidere a bazei de date din interfața oferită. Un exemplu de astfel de job ar fi atunci când select trebuie să filtreze elementele în funcție de o condiție. În acest caz, s-ar putea crea mai multe job-uri care efectuează în paralel această operație: pentru 4 thread-uri, thread-ul 1 se uită în primul sfert din tabelă, thread-ul 2 în al doilea sfert, etc.

În implementarea temei, vă recomandăm să țineți cont de următoarele sugestii:

- identificați care dintre operațiile pe baza de date sunt citiri și care scrieri, pentru că apoi puteți folosi paradigma cititori-scriitori pentru sincronizare
- un tabel din baza de date poate fi reprezentat ca o listă de coloane sau o listă de linii, dar una din metode va avea un avantaj mai mare dat de liniile de cache
- operațiile pe tabele diferite se pot executa în paralel
- thread-urile care execută job-uri pot fi controlate de un Executor Service.

Punctaj, rulare și upload

Punctajul temei va fi distribuit astfel:

- **20 puncte** - implementarea secvențială
- **20 puncte** - consistența datelor la rularea paralelă
- **30 puncte** - scalabilitate pentru operații pe tabele diferite
- **30 puncte** - scalabilitate pentru o singură operație (un singur select sau update).

Arhiva temei va fi în format ZIP și trebuie să aibă toate fișierele în rădăcină (nu vor fi directoare). Arhiva va conţine codul sursă pentru temă, împreună cu un fişier **build.xml** reprezentând paşii de compilare şi împachetare pentru aplicaţie, pentru sistemul de build Ant (tutorial pentru Ant găsiţi aici). De asemenea, arhiva trebuie să mai conțină un fișier readme (în orice format text) în care să descrieți pe scurt implementarea.

Atenție! O temă fără readme sau readme necorespunzător va primi automat 0 puncte!

Un exemplu de conținut al arhivei găsiți mai jos (unde **MyFileX.java** reprezintă eventuale fișiere noi adăugate de voi):

```
.
+-- build.xml
+-- ConsistencyReaderThread.java
+-- ConsistencyWriterThreads.java
+-- Main.java
+-- MyDatabase.java
+-- MyFile1.java
+-- MyFile2.java
+-- readme.txt
+-- ScalabilityTestThread.java
```

Atenție! La testarea temei, toate fișierele din schelet vor fi suprascrise cu cele inițiale, deci nu le modificați! Va trebui să creați voi fișierul **Database.java**, plus orice alte fișiere pentru eventuale clase auxiliare.

În urma rulării comenzii `ant compile jar`, va trebui să rezulte în rădăcină un fişier numit **database.jar**, care va putea fi rulat cu comanda `java -jar database.jar`. Această comandă va rula suita de teste de sanity, consistență și scalabilitate. Pentru testele de scalabilitate, se printează la consolă duratele operațiilor de insert, update și select (în milisecunde).

Atenție! Testul de scalabilitate creează foarte multe obiecte pe care le adaugă în baza de date. Dacă primiți excepții de memorie insuficientă, vă recomandăm să micșorați numărul de obiecte generate din clasa **ScalabilityTestThread**. Pentru a verifica totuși scalabilitatea, puteți rula jar-ul pe cluster-ul facultății (ca la tema 1).