

Managementul tranzacțiilor

Nicolae-Cosmin Vârlan

January 13, 2014

Tranzacții

O *tranzacție* este o secvență de operații ce formează o unitate logică de lucru.

Example

De exemplu, pentru a transfera \$50 dintr-un cont într-altul poate fi definită următoarea tranzacție:

$$T_i : \begin{array}{l} \text{read}(A); \\ A := A - 50; \\ \text{write}(A); \\ \text{read}(B); \\ B := B + 50; \\ \text{write}(B); \end{array}$$

Operații considerate

Baza de date este situată pe disc. Anumite porțiuni se pot afla în memoria RAM (cele pe care se operează).

Sunt considerate două operații principale:

- ▶ $\text{read}(X)$ - care transferă elementul X din baza de date în bufferul local (ce aparține tranzacției ce execută operația de read);
- ▶ $\text{write}(X)$ - care transferă elementul X din bufferul local tranzacției înapoi în baza de date.

În realitate, o operație de tip write nu va scrie imediat pe HDD.

Proprietatile tranzacțiilor (ACID)

- ▶ *Atomicity* - toate acțiunile dintr-o tranzacție trebuie să fie efectuate sau, în caz contrar, baza de date trebuie să revină la starea originală;
- ▶ *Consistency* - o tranzacție executată singură va modifica baza de date într-un mod consistent. Dacă două tranzacții sunt executate simultan, este posibil ca rezultatul să nu fie cel așteptat;
- ▶ *Isolation* - chiar dacă mai multe tranzacții sunt executate concurent trebuie ca sistemul să asigure ca oricare ar fi două dintre acestea T_i și T_j , din punctul de vedere al lui T_i , fie a început după ce T_j a fost executată în întregime, fie prima linie din T_j va fi executată după ce ultima sa linie s-a terminat de executat;
- ▶ *Durability* - după ce tranzacția s-a încheiat, schimbările rămân permanente în sistem (chiar în cazul unui crash);

Consistency

O tranzacție executată singură va modifica, baza de date într-un mod consistent. Dacă două tranzacții sunt executate simultan, este posibil ca rezultatul să nu fie cel așteptat.

Asigurarea consistenței este obligația programatorului.

În exemplul considerat, suma $A + B$ trebuie să rămână constantă :

```
 $T_i$  :  
    read(A);  
    A:=A-50;  
    write(A);  
    read(B);  
    B:=B+50;  
    write(B);
```

Atomicity

Toate acțiunile dintr-o tranzacție trebuie să fie efectuate sau, în caz contrar, baza de date trebuie să revină la starea originală. Inconsistențele în baza de date nu trebuie să fie vizibile în exterior.

În exemplul considerat, dacă luăm $A = 1000$ și $B = 2000$, trebuie ca suma $A + B$ să fie aceeași după terminarea tranzacției.

Ce se întâmplă dacă în timpul execuției tranzacției T_i are loc o pană de curent ?

```
 $T_i$  :  
    read(A);  
    A:=A-50;  
    write(A);  
    read(B);  
    B:=B+50;  
    write(B);
```

Durability

Dupa ce executia s-a terminat si utilizatorul a fost informat ca tranzactia s-a efectuat cu succes, nimic nu poate aduce baza de date in starea anterioara tranzactiei sau intr-o stare inconsistenta.

T_i :

```
read(A);  
A:=A-50;  
write(A);  
read(B);  
B:=B+50;  
write(B);
```

- ▶ Updateurile sunt efectuate inainte sa se termine tranzactia.
- ▶ Informatii suficiente sunt inregistrate pentru a putea reface baza de date in caz de esec (recovery-management component).

Isolation

Problema apare atunci când sunt executate mai multe tranzacții simultan, operațiile putând în unele cazuri să se întrepătrundă într-un mod ce ar da naștere la o stare inconsistentă.

Spre exemplu, în timpul transferului dintre A și B sistemul trece printr-o stare inconsistentă (după ce s-a rescris A și înainte de rescrierea lui B). O a doua tranzacție care ar citi A și B în această stare inconsistentă ar putea considera valori eronate. Mai mult, dacă această updatează B înainte ca T_i să citească pe B , se poate ca baza de date să rămână într-o stare inconsistentă după executia lui T_i .

O soluție este executarea tranzacțiilor în mod serial.

Concurrency-control component.

Starea unei tranzacții

- ▶ *Active* - starea initiala, tranzactia sta in aceasta stare in timpul executiei;
- ▶ *Partially committed* - dupa executia actiunii finale;
- ▶ *Failed* - dupa descoperirea ca executia normala nu poate continua (din cauza unor esecuri in hardware);
- ▶ *Aborted* - tranzactia a fost rolled back;
- ▶ *Committed* - dupa finalizarea cu succes.

Spunem despre o tranzactie ca este terminata daca se afla fie in starea de "Aborted" sau in "Committed".

Starea unei tranzacții

În cazul de eșec tranzacția poate fi:

- ▶ restarted (o tranzacție repornită este considerată o nouă tranzacție)
- ▶ killed (atunci când eșecul nu este unul hardware ci în logica tranzacției sau din cauza că datele nu au fost găsite)

Observație externă scrie ! - numai în starea committed.

Pentru rollback, se poate scrie (temporar) anumite informații privind tranzacția, scrierea definitivă având loc atunci când starea a devenit "commit".

Anumite situații sunt mai greu de prevăzută (de exemplu în cazul unui bancomat).

Implementarea atomicității

Atomicitatea poate fi realizată prin utilizarea de copii de siguranță (**shadow copy**) a bazei de date.

Se presupune ca numai o singură tranzacție este activă la un moment dat.

Atomicitatea tranzacției se reduce la atomicitatea scrierii pe disc a unui pointer către fișierul reprezentând versiunea curentă a bazei de date (asigurat de SO).

Sistemul a inspirat și anumite editoare de text (e.g. Word) care oferă o copie “de siguranță” pe care o salvează la intervale de timp pentru a restaura în cazul unei probleme.

Minusuri: baze de date mari + tranzacții concurente.

Execuții concurente - motivatie

Avantajele execuțiilor concurente:

- ▶ utilizarea eficientă a resurselor;
- ▶ reducerea timpului de așteptare.

Sistemul de baze de date trebuie să controleze felul în care interacționează tranzacțiile pentru a nu fi afectată consistența acestora (Concurrency control schemes. . .).

Execuții concurente

Fie următoarele două tranzacții:

T_1 :	read(A);		read(A);
	A:=A - 50;		temp:= A * 0.1;
	write(A);		A := A - temp;
	read(B);	T_2 :	write(A);
	B:=B + 50;		read(B);
	write(B);		B:=B + temp;
			write(B);

Sa presupunem ca $A = 1000$ si $B = 2000$.

O posibilită de execuție (schedule 1)

T_1	T_2
read(A); A:=A - 50; write(A); read(B); B:=B + 50; write(B);	read(A); temp:= A * 0.1; A := A - temp; write(A); read(B); B:=B + temp; write(B);

O posibilită de execuție (schedule 2)

T_1	T_2
	read(A);
	temp := A * 0.1;
	A := A - temp;
	write(A);
	read(B);
	B := B + temp;
	write(B);
read(A);	
A := A - 50;	
write(A);	
read(B);	
B := B + 50;	
write(B);	

Schedule (program)

Un *schedule* reprezintă o modalitate de a organiza secvența dintr-una sau mai multe tranzacții.

Un schedule trebuie să păstreze ordinea acțiunilor din fiecare tranzacție și trebuie să conțină toate acțiunile existente în tranzacțiile implicate.

Cele două programări prezentate sunt seriale deoarece întâi este efectuată o tranzacție în întregime apoi este efectuată cea de-a doua. Pentru n tranzacții pot fi construite $n!$ programări seriale diferite.

O posibilitate de execuție (schedule 3) - suma $A+B=\text{constanta}$

T_1	T_2
read(A); $A := A - 50$; write(A);	read(A); $\text{temp} := A * 0.1$; $A := A - \text{temp}$; write(A);
read(B); $B := B + 50$; write(B);	read(B); $B := B + \text{temp}$; write(B);

Schedule 4 (concurrency control component)

T_1	T_2
read(A); $A := A - 50$;	read(A); $temp := A * 0.1$; $A := A - temp$; write(A);
write(A); read(B); $B := B + 50$; write(B);	read(B); $B := B + temp$; write(B);

Serializability

Care programari asigura consistenta și care nu ?

Consideram doar două operații: *read* / *write*

Schedule 3 va fi scris sub formă:

T_1	T_2
read(A); write(A);	
	read(A); write(A);
read(B); write(B);	
	read(B); write(B);

Conflict serializability

Fie un schedule S având două instrucțiuni consecutive I_i și I_j din două tranzacții diferite: T_i și T_j ($i \neq j$).

- ▶ dacă I_i și I_j se referă la componente diferite (de exemplu acțiunea lui I_i peste A în timp ce I_j acționează asupra lui B), putem interschimba I_i cu I_j .
- ▶ dacă I_i și I_j acționează peste același element atunci:
 - ▶ dacă $I_i = \text{read}(Q)$ și $I_j = \text{read}(Q)$, atunci I_i și I_j **sunt interschimbabile**;
 - ▶ dacă $I_i = \text{read}(Q)$ și $I_j = \text{write}(Q)$, atunci I_i și I_j **NU sunt interschimbabile** (T_j influențează T_i);
 - ▶ dacă $I_i = \text{write}(Q)$ și $I_j = \text{read}(Q)$, atunci I_i și I_j **NU sunt interschimbabile** (T_i influențează T_j);
 - ▶ dacă $I_i = \text{write}(Q)$ și $I_j = \text{write}(Q)$, atunci I_i și I_j **NU sunt interschimbabile** (starea finală a DB);

Conflict serializability

Două acțiuni I_i și I_j din două tranzacții diferite: T_i respectiv T_j sunt în **conflict** dacă ele se referă la aceeași înregistrare și macar una dintre cele două acțiuni este de tip write.

Ordinea a două acțiuni ce nu sunt în conflict poate fi interschimbata fără a afecta rezultatul final.

Conflict serializability

Schedule 5 este echivalent cu schedule 3:

T_1	T_2	T_1	T_2
read(A);		read(A);	
write(A);		write(A);	
	read(A);		read(A);
read(B);			write(A);
	write(A);	read(B);	
write(B);		write(B);	
	read(B);		read(B);
	write(B);		write(B);

Schedule 5

Schedule 3

Conflict serializability

Sunt Schedule 1 si schedule 2 echivalente ?

T_1	T_2	T_1	T_2
read(A);			read(A);
write(A);			write(A);
read(B);			read(B);
write(B);			write(B);
	read(A);	read(A);	
	write(A);	write(A);	
	read(B);	read(B);	
	write(B);	write(B);	

Schedule 1

Schedule 2

Conflict serializability

Sunt Schedule 1 si schedule 3 echivalente ?

T_1	T_2	T_1	T_2
read(A);		read(A);	
write(A);		write(A);	
read(B);			read(A);
write(B);			write(A);
	read(A);	read(B);	
	write(A);	write(B);	
	read(B);		read(B);
	write(B);		write(B);

Schedule 1

Schedule 3

Conflict serializability

Deoarece Schedule 3 este echivalent din punctul de vedere al conflictelor cu Schedule 1 care este sub forma seriala, spunem despre acesta ca este serializabil din punctul de vedere al conflictelor (Conflict serializability).

Nu toate programările sunt serializabile (schedule 7):

T_3	T_4
read(Q);	
	write(Q);
write(Q);	

Sunt Schedule 8 și schedule $\langle T_1, T_5 \rangle$ echivalente ?

T_1	T_5	T_1	T_5
read(A)		read(A)	
$A := A - 50$		$A := A - 50$	
write(A)		write(A)	
	read(B)	read(B)	
	$B := B - 10$	$B := B + 50$	
	write(A)	write(B)	
read(B)			read(B)
$B := B + 50$			$B := B - 10$
write(B)			write(A)
	read(A)		read(A)
	$A := A + 10$		$A := A + 10$
	write(A)		write(A)

Schedule 8

Schedule $\langle T_1, T_5 \rangle$

View Equivalence

Doua schedule S si S' in care aceeasi multime de tranzacții este utilizata sunt *view equivalent* daca urmatoarele conditii sunt indeplinite:

- ▶ Pentru fiecare element Q , daca in S tranzacția T_i citește valoarea initiala a lui Q atunci T_i citește valoarea initiala a lui Q si in S' ;
- ▶ Daca in S tranzacția T_i citește valoarea lui Q scrisa de catre T_j atunci, si in S' , T_i va citi valoarea lui Q dupa ce aceasta a fost scrisa de T_j ;
- ▶ Pentru fiecare Q , tranzacția care efectueaza ultima $write(Q)$ in S va fi ultima ce efectueaza $write(Q)$ si in S' .

Schedule 1 si Schedule 2, desi sunt consistente, ele nu sunt echivalente din punctul de vedere al VE (View Equivalence). S1 este VE cu S3.

T_1	T_2
$\text{read}(A);$ $A := A - 50;$ $\text{write}(A);$ $\text{read}(B);$ $B := B + 50;$ $\text{write}(B);$	$\text{read}(A);$ $\text{temp} := A * 0.1;$ $A := A - \text{temp};$ $\text{write}(A);$ $\text{read}(B);$ $B := B + \text{temp};$ $\text{write}(B);$

Schedule 1

T_1	T_2
$A := A - 50;$ $\text{write}(A);$ $\text{read}(B);$ $B := B + 50;$ $\text{write}(B);$	$\text{read}(A);$ $\text{temp} := A * 0.1;$ $A := A - \text{temp};$ $\text{write}(A);$ $\text{read}(B);$ $B := B + \text{temp};$ $\text{write}(B); \text{read}(A);$

Schedule 2

T_1	T_2
$\text{read}(A);$ $A := A - 50;$ $\text{write}(A);$ $\text{read}(B);$ $B := B + 50;$ $\text{write}(B);$	$\text{read}(A);$ $\text{temp} := A * 0.1;$ $A := A - \text{temp};$ $\text{write}(A);$ $\text{read}(B);$ $B := B + \text{temp};$ $\text{write}(B);$

Schedule 1

T_1	T_3
$\text{read}(A);$ $A := A - 50;$ $\text{write}(A);$ $\text{read}(B);$ $B := B + 50;$ $\text{write}(B);$	$\text{read}(A);$ $\text{temp} := A * 0.1;$ $A := A - \text{temp};$ $\text{write}(A);$ $\text{read}(B);$ $B := B + \text{temp};$ $\text{write}(B);$

Schedule 3

View Serializable

Conceptul de View equivalent conduce la conceptul de View Serializable (daca programarea este echivalenta din punct de vedere al view-ului cu una seriala).

Exista programari care nu sunt CS(Conflict Serializable) dar sunt VS(View Serializable)

Schedule 9:

T_3	T_4	T_6
read(Q);	write(Q);	
write(Q);		write(Q)

Conflict Serializable vs View Serializable

Orice programare CS este și VS.

Există programări care sunt VS dar nu sunt și CS.

Recoverability

Am presupus concurența fără a considera un eventual eșec al componentei hardware. Ce se întâmplă în cazul în care avem mai multe tranzacții concurente și în timpul rularii uneia dintre ele sistemul cedează ?

O programare este recuperabilă dacă care pentru fiecare pereche de tranzacții T_i și T_j a.i. T_j citește date scrise în prealabil de T_i operația de *commit* apare întâi la T_i și apoi la T_j .

Nerecuperabilă (deoarece T_8 va face roll-back și T_9 are alte date):

T_8	T_9
read(A); write(A);	
read(B);	read(A); (T_9 face <i>commit</i>)

Cascadeless Schedules

T_{10}	T_{11}	T_{12}
read(A); read(B); write(A);	read(A); write(A);	read(A)

Dacă T_{10} nu apuca să facă commit înainte ca T_{11} să citească pe A , în cazul în care T_{10} trebuie să facă roll-back, și T_{11}, T_{12} vor trebui să facă același lucru.

O programare este *cascadeless schedule* dacă pentru orice pereche T_i, T_j în care T_j citește date scrise de T_i , acesta (T_i) a făcut *commit* înainte ca aceste date să fie citite de către T_j .

Testarea serializării

În proiectarea sistemelor ce control concurent trebuie să testăm dacă programările generate sunt serializabile.

O metoda simplă pentru determinarea CS (Conflict Serializable) este prin construirea unui graf orientat denumit *graf al precedentelor*.

Fiecare tranzacție va fi reprezentată printr-un nod. Avem muchie de la T_i la T_j dacă:

- ▶ T_i execută write(Q) înainte ca T_j să execute read(Q);
- ▶ T_i execută read(Q) înainte ca T_j să execute write(Q);
- ▶ T_i execută write(Q) înainte ca T_j să execute write(Q);

Dacă graful nu are cicluri atunci programarea poate fi serializată. Serializarea se face prin sortare topologică.

Bibliografie

Capitolele 15, 16 din Silberschats-Korth-Sudarshan *Database System Concepts, Fourth Edition*