Tranzactii

441A:

- Cealicu Alexandru Ciprian
- Popa Cristiana
- Tudor Alexandru
- Anton Daniel
- Ganciu Cristiana Valentina
- Neagoe Mirela

Cuprins:

Capitolele au fost realizate membrii grupei astfel:

1. Caracteristici si prezentare generala - Cealicu Alexandru Ciprian
2. Proprietati si operatii - Popa Cristiana
3. Gestiunea - Tudor Alexandru
4. Controlul accesului concurent:

-Serializabilitate, schedule, graful dependentelor – Neagoe Mirela

- Metode de implementare - Anton Daniel
- Anomalii de executie - Ganciu Cristiana Valentina

5. Erori si recuperarea datelor - Cealicu Alexandru Ciprian

Pentru o mai buna organizare, am scris la fiecare subcapitol de cine este realizat si bibliografia folosita. La sfarsitul lucrarii am pus bibliografia pentru toata lucrarea.

1. Caracteristici si prezentare generala

Cealicu Alexandru Ciprian

Tranzactia este comunicarea sau transferul intre doua entitati sau obiecte – facandu-se referinta in schimbul de obiecte de valoare, servicii, bani sau in cazul de fata, informatie.

Tranzactia permite programatorului sa grupeze mai multe operatiuni intr-una singura, indivizibila. In acest fel se pot contrui operatii care sa nu permita discernerea structurii lor interne.

Tranzactiile informationale se pot impartii in mai multe categorii, in functie de nivelul la care sunt folosite:

- Tranzactiile intre procese pe o masina virtuala (VM370, CMS, timesharing)
- Tranzactiile pe sisteme de distributie (prin retea, intre doua calculatoare)
- Tranzactii la nivel de Sistem de Operare

In cadrul acestor tranzactii pot aparea diferite probleme ca: problema de concurenta, problema de inconsistenta a memoriei, inconsistenta bazei de date.

Procesarea tranzactiilor mentine un sistem (in general o baza de date sau un sistem de fisiere) intr-o stare cunoscuta, consistenta, asigurandu-se ca toate operatiunile facute pe acel sistem si care sunt interdependente sunt fie toate duse la bun sfarsit fara erori fie sunt toate anulate fara eroare. Spre exemplu, o tranzactie bancara ce consta in transferul a 500$ dintr-un cont al unui client in alt cont al aceluiasi client. Aceasta tranzactie reprezinta o singura operatiune din punctul de vedere al bancii insa are de fapt cel putin doua operatiuni distincte, din punctul de vedere al sistemului informational: debitarea unui cont cu 500$ si creditarea celuilalt cont cu 500$. In cazul in care debitarea se incheie cu succes insa creditarea este oprta datorita unei erori (sau vice-versa) atunci jurnalul bancar va contine erori, iar balanta zilnica nu va fi buna.

Trebuie sa existe o cale prin care sa se poata asigura o metoda in care fie cele doua operatiuni sunt incheiate cu succes, fie la aparitia unei erori niciuna dintre operatiuni sa nu fie efectuata astfel incat in jurnalele bancii sa nu apara inconsistente. Tranzactiile ofera o astfel de modalitate de lucru.

Se permite astfel ca mai multe operatii individuale sa fie “legate” intr-una singura, automat, aceasta fiind indivizibila. Sistemul de procesare al tranzactiilor asigura fie ca toate operatiunile sunt completate fara erori, fie ca nici una dintre ele nu este efectuata (in

eventualitatea aparitiei unei probleme). Daca unele dintre operatiuni sunt efectuate si o eroare apare ulterior, sistemul de procesare lanseaza un “roll-back” pentru toate operatiunile (inclusiv cele bune), astfel stargand toate modificarile facute in cadrul tranzactiei si resaurand consistenta sistemului (starea cunoscuta in care se afla inainte de efectuarea tranzactiei). In cazul in care toate operatiunile unei tranzactii se incheie cu succes, se opereaza “commit” (confirmarea modificarilor) si toate modificarile sunt acceptate (modificarile sunt facute permanente). Nu se mai poate face “rollback” in cazul in care s-a efectuat un “commit” (pentru aceeasi tranzactie).

Procesarea tranzactiilor protejeaza sistemul impotriva oricaror tipuri de erori ce ar putea sa lase o tranzactie partial efectuata (erori hardware sau software). Astfel de erori ar lasa sistemul intr-o stare incerta, cu date corupte. In cazul in care sistemul sufera o eroare majora (crash, fatal error), se garanteaza ca toate operatiunile facute in cadrul uneia sau mai multor tranzactii, ce sunt neconfirmate, sunt anulate. Tranzactiile sunt procesate intr-o ordine strict cronologica. Daca tranzactia n+1 foloseste aceiasi portiune din sistem (de ex: aceleasi inregistrari dintr-o baza de date, aceeasi tabela etc.) ca si tranzactia n, atunci tranzactia n+1 nu poate incepe pana ce tranzactia n nu s-a terminat cu succes, iar modificarile facute de aceasta sunt confirmate (“commit”). Inainte de a se efectua un “commit” pentru o tranzactie, toate celalalte tranzactii care afecteaza aceeasi parte a sistemului trebuie sa fie confirmate. Nu pot exista “gauri” in secventa acestui model.

De asemenea, tranzactiile pot fi sincrone sau asincrone. In cazul tranzactiilor sincrone, un pachet n+1 nu poate fi timis daca nu s-a primit confirmarea pentru pachetul n. Aceasta confirmare poate fi un simplu “OK” sau un CRC (cyclic redundancy check).

Tranzactiile asincrone permit ca un pachet n+1 sa poata sa fie trimis chiar daca nu s-a primit confirmare pentru pachetul n (in cazul acestor tipuri de tranzactii nu se cere confirmare). Astfel, se poate primi pachetul n+1 inaintea pachetului n.

Tranzactiile sincrone se folosesc de exemplu in comunicarea server – client, iar cele asincrone sunt folosite in comunicarea de tip “broadcast” (un terminal transmite unul sau mai multe pachete de informatie mai multor terminale conectate cu acesta).

Se pot evidentia urmatoarele beneficii in cazul folosirii tranzactiilor:

- Programare usoara in prezenta concurentei: daca vrem sa respectam invarianti in prezenta unor procese concurente, atunci inchidem portiunile de cod care corup invariantii in tranzactii si alte programe vor vedea numai stari consistente;

- Tratament al erorilor grave hardware/software (crashes): nu mai trebuie sa ne facem griji ce se intampla daca apare o astfel de eroare; chiar daca se vor pierde date, calculatorul va putea functiona corect, pentru ca datele ramase sunt consistente. De asemenea, in cazul in care s-a executat “commit”, rezultatul a devenit permanent in pofida catastrofelor;
- Tratament simplificat al erorilor: daca survine o conditie de eroare programatorul poate executa un simplu “abort” pentru a reveni la starea initiala corecta, fara a trebui sa refaca “manual” toate valorile modificate.

Bibliografie:

1. http://technet.microsoft.com
2. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/
3. Principles of Transaction Processing, 1997, Morgan Kaufmann
4. Lorentz Jäntschi, Mădălina Văleanu, Sorana Bolboacă, Programarea rapidă a aplicațiilor pentru baze de date relaționale

2. Proprietati si operatii

Popa Cristiana

Exista 4 proprietati importante ale tranzactiilor, care de pot aplica la orice tip de tranzactie, la orice nivel. Aceste proprietati se rezuma la acronimul ACID, acesta insemnand:

- Atomicitate (Tranzactiile nu pot fi intrerupte de erori. In cazul unei erori toate operatiunile din cadrul tranzactiei sunt anulate – "rollback")
- Consistenta (Tranzactiile pastreaza structurile de date corecte)
- Independenta (Programele executate in paralel nu interfera)
- Durabilitate (Rezultatul unei tranzactii este permanent (modificari confirmate – "commit")

Atomicitate:

Tranzactiile grupeaza mai multe instructiuni laolalta intr-o "macro-instructiune" care se comporta ca o unitate indivizibila.

Atunci cand are loc o eroare, un grup de instructiuni dintr-o tranzactie se comporta ca o singura entitate . Daca am grupat o serie de 10 instructiuni si se intrerupe curentul dupa ce am executat numai primele 5, sistemul de tranzactii trebuie sa ne asigure ca efectul celor 5 instructiuni nu este vizibil dupa reluarea executiei. Intreruperea catastrofala (crash, fatal error) va lasa sistemul intr-o stare corecta, care ar fi putut surveni in urma unei executii normale. Altfel spus, daca intr-o tranzactie grupam niste instructiuni acestea se executa ori toate ori niciuna.

Putem considera un exemplu ipotetic: platile catre angajatori facute de catre un program de contabilitate. Programul ar putea arata asa:

```
platit := 0;
for i:=1 to angajati do begin
      dePlata[i] := dePlata[i] + salariu[i];
      platit := platit + salariu[i]
end;
budget := budget - platit;
```

Daca presupunem ca valorile modificate sunt inregistrari pe disc, intr-o baza de date, atunci in cazul unei erori (software sau hardware) undeva la mijlocul buclei anumite salarii vor fi fost platite, dar suma nu a fost scazuta din budget. Programul nu poate fi reluat de la inceput,

pentru ca anumite salarii au fost platite, dar altele nu. Solutia este sa executam toate aceste instructiuni intr-o singura tranzactie, care atunci cand este intrerupta inainte de terminare va face ca modificarile facute sa dispara.

- ***Consistenta:***

  Luand in considerare toate variabilele unui program, la un anumit moment de timp ele vor avea fiecare o valoare. Dar nu orice combinatie de valori este posibila. De exemplu, pentru un program care sorteaza un vector de numere, in vector se vor afla tot timpul aceleasi numere, posibil in alta ordine. Aceasta regula se numeste un *invariant*, pentru ca tot timpul trebuie a fie adevarata Pentru exemplul de program de mai sus un invariant este: (*) - “suma de platit si budgetul curent trebuie sa aiba aceeasi suma” (balanta de plati este zero).

  Tehnica invariantilor este foarte importanta pentru ingineria programarii. Un tip de date abstract nu este altceva decat o serie de proceduri care mentin niste invarianti foarte precisi despre anumite structuri de date. Folosirea tranzactiilor este o metoda usoara de a programa cu invarianti.

- ***Independenta:***

  Aceasta proprietate se refera la un context in care mai multe programe actioneaza simultan asupra aceluiasi set de date (activitate concurenta). Un program nu trebuie sa poata accesa partiale ale altui program, pentru ca acestea nu ar fi corecte. Pentru a exemplifica putem considera un program care implementeaza o marire de salariu:

```
for i:=1 to angajati do
    salariu[i] := salariu[i] * 110/100; { marire 10 la suta :( }
```

  Ce consecinte ar putea avea executarea acestui program in mod simultan cu cel anterior? Una din aceste consecinte ar putea fi ca acest program sa porneasca in executie dupa cel precedent, sa ajunga la acelasi indice undeva si apoi sa o ia inainte. Astfel unii angajati vor primi salarii indexate (celor din urma), iar altii nu. Acest lucru ar duce la o inconsistenta in balanta platilor acelei companii. Este posibila evitarea acestui lucru daca toate fragmentele de program sunt incluse intr-o tranzactie.

- ***Durabilitatea:***

  Se refera la programarea cu structuri de date persistente. Un program simplu este executat de fiecare data la fel, avand ca punct de plecare aceleasi valori initiale. Insa un program

care opereaza cu o baza de date sau un fisier are structuri de date care se afla pe disc si care pot trece peste o eroare datorata programului. In cazul executiei de doua ori a programului salarial, operatia de crestere a salariului poate avea rezultate diferite, intr-ucat a doua crestere este aplicata peste prima, iar rezultatele primei executii sunt salvate pe disc. Pesristenta este foarte importanta in baze de date, unde toate operatiile se efectueaza asupra unor date “permanente”.

La acest lucru se refera durabilitatea, la o tranzactie terminata, ale carui rezultate trebuie sa fie permanente, chiar in cazul unor erori fatale (ex: Intreruperea curentului).

Operatii

Principalele operatii pe care programatorul le poate folosicand opereaza asupra unei baze de date sunt de operatiile de citire si de scrie a unei valori din baza de date, precum si operatiile aritmetice.

Operatiile:

1. **begin**: marcheaza inceputul executiei unei tranzactii.
2. **read sau write**: operatii de citire sau scriere a articolelor in baza de date, executate in cadrul unei tranzactii.
3. **end**: marcheaza terminarea operatiilor de scriere sau citire din baza de date, tranzactia se poate termina; este necesara verificarea inainte de validarea (commit) tranzactiei.
4. **commit**: anunta terminarea cu succes a tranzactiei, validarea modificarilor efectuate in baza de date si vizibilitatea modificarilor efectuate pentru alte tranzactii; din acest moment, modificarile efectuate nu mai pot fi anulate, nici pierdute printr-o defectare ulterioara a sistemului (daca mecanismul de refacere al SGBD-ului functioneaza corect).
5. **rollback** (sau abort): anunta ca tranzactia a fost abandonata si ca orice efect pe care aceasta l-a avut asupra bazei de date trebuie anulat (printr-o “rulare inapoi” a operatiilor).
6. **undo**: este similara operatiei rollback, dar se aplica unei singure operatii, nu unei intregi tranzactii.
7. **redo**: specifica faptul ca unele operatii ale unei tranzactii trebuiesc executate din nou pentru a se putea valida intreaga tranzactie.

Vom folosi in mod explicit Write si Read pentru a indica operatiile asupra valorilor persistente; pe langa valori persistente (din baza de date) vom intalni si valori locale fiecarui proces (ca variabilele locale unei functii din limbajele de programare).
Unul din exemplele de mai sus va fi scris deci astfel intr-un sistem care ofera tranzactii:

```
Begin Transaction
for i:=1 to nr_angajati do begin
    x := Read(sal[i]); { x e o variabila locala }
    x := x * 110/100; { marire 10 la suta }
    Write(x, sal[i]);
End Transaction
```

Tot ceea ce se afla intre Begin Transaction si End Transaction se comporta ca o singura instructiune. Modificarile tuturor salariilor vor deveni in mod normal (daca nu se produce vreo eroare) vizibile instantaneu la executia comenzii End Transaction. Sfarsitul cu succes al unei tranzactii se numeste *Commit*.

Instructiunea Abort este utila atunci cand s-a apare ceva neprevazut si tranzactia nu poate fi terminata cu succes. Cand este executata comanda Abort toate modificarile facute asupra datelor persistente se pierd definitiv. Tranzactia se considera terminata.

Un exemplu de utilizare a tranzactiilor in cadrul sistemelor de operare in care instructiunea Abort este utilizata: codul ipotetic al operatiei move, care muta un fisier dintr-un director intr-altul si eventual ii schimba numele:

```
procedure move_fis(num_Vechi, num_Nou)
begin
    Begin Transaction;
    dir_Nou := extrage_Dir(num_Nou);
    fis_Nou := extrage_NumFis(num_Nou);
    is := cauta(dir_Nou, fis_Nou);
    if (is) then Abort;  { fisierul nou exista deja }
    dir_Vechi := extrage_Dir(num_Vechi);
    fis_Vechi := extrage_NumFis(num_Vechi);
    is := cauta(dir_Vechi, fis_Vechi);
    if (not is) then Abort; { fisierul vechi nu exista }
    cont_Fisier := fisier(num_Vechi);
    eroare := sterge(dir_Vechi, fis_Vechi);
    if (eroare) then Abort;
    eroare := creaza(dir_Nou, fis_Nou);
    if (eroare) then Abort;
```

```
    fisier(num_Nou) := cont_Fisier;
    End Transaction;
end;
```

(Functiile cauta, sterge, etc. sunt asemanatoare cu Read si Write de mai sus deoarece opereaza tot cu structuri persistente, insa sunt putin mai complicate, ele putand fi sintetizate din mai multe operatii de acest fel.)

Pe disc operatiile atomice sunt foarte simple: stergerea unui nume dintr-un director, crearea unui nume intr-un alt director, asocierea unui nume cu un continut. Pentru ca procedura move sa aiba succes toate operatiile componente trebuie sa se efectueze cu succes; altfel nici una nu trebuie sa se efectueze

Este destul de dificil sa ne asiguram ca toate conditiile sunt adevarate simultan, mai ales in prezenta activitatii concurente. Daca mai multe procese incearca simultan sa faca move la un acelasi rezultat, atunci degeaba unul din ele verifica inainte de mutare daca rezultatul exista, pentru ca rezultatul poate sa apara in timp ce el sterge numele vechi.

Folosind tranzactiile problema este rezolvata automat: daca toate conditiile au fost indeplinite End face modificarile dintr-odata; altfel nici o modificare nu este vizibila, si Abort este executat. Este o diferenta intre End Transaction si *Commit*, instructiunea End anunta sfarsitul tranzactiei si incercarea de a face modificarile permanente. Faptul ca rezultatele operatiilor devin permanente se numeste *Commit*. Un End insa poate sa nu reuseasca sa faca Commit, si atunci se va transforma in Abort.

Exista doua feluri in care o tranzactie se poate termina cu esec:

- Cand programul executa instructiunea Abort tranzactia se anuleaza de la sine;
- Cand sistemul care implementeaza tranzactii observa ca nu poate oferi garantiile promise, el poate anula o tranzactie spontan.

Respectarea integritatii datelor este mai importanta decat executia codului; codul poate fi re-executat, dar datele trebuie sa fie in permanenta consistente. In general atunci cand o tranzactie este anulata, programul care a invocat-o (sau utilizatorul) este informat de acest lucru si poate decide daca sa incerce din nou executia sau nu, in functie de motive.

Implementarea lui *Abort (rollback)* si *Commit*

Exista doua metode pentru a *modifica* valorile persistente (baza de date):

1. Modificarile se fac pe *copii* ale valorilor din baza de date (acestea se numesc “date umbra” (shadow-data)). Tranzactia citeste toate valorile interesante din baza de date (poate folosi fie incuietori, fie validare), dupa care modifica toate valorile local. Cand a

terminat cu succes, (*Commit*) trimite toate valorile la server-ul care tine baza de date, unde ele trebuie efectuate i*n mod atomic*, toate deodata (*redo*). Daca tranzactia decide sa faca Abort nu trimite nici o valoare, ci doar elibereaza incuietorile.

2. Tranzactia poate alege sa modifice valorile direct in baza de date. in cazul asta *Commit* e foarte simpla: eliberezi incuietorile. Abort insa are nevoie de un mecanism special, prin care variabilele trebuie readuse la valorile initiale (*undo*).

In fiecare caz una dintre operatiile de terminare este mai grea decat cealalta. Ambele probleme pot fi rezolvate folosind *aceeasi* unealta, in moduri usor diferite. Aceasta unealta este setul de inregistrari, mai precis numit *log*, sau *audit*.

Bibliografie:

1. http://technet.microsoft.com/en-us/library/aa213068(SQL.80).aspx (Proprietati ale tranzactiilor
2. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/
3. Morgan Kaufmann, *Principles of Transaction Processing*, 1997

3.Gestiunea bazelor de date

Tudor Alexandru

Gestiunea

Bază de date: Este o colecție partajată de date legate logic, proiectată pentru a satisface necesitățile unui sistem informatic. Cu alte cuvinte datele sunt strânse într-o colectie unică şi sunt folosite simultan de mai mulți utilizatori. Redundanța datelor este controlată prin normalizare, ceea ce implică o redundanță minimă.

O astfel de bază de date are nevoie de *un sistem de gestiune a bazei de date*.

Acest sistem de gestiune a bazei de date, prescurtat SGBD are urmatoarele obiective:

- independența fizică;
- independeța logică;
- manipularea datelor de către neinformaticieni;
- eficacitatea accesului la date;
- administrarea centralizată a datelor;
- neredundanța datelor;
- coerența datelor;
- partajabilitatea datelor;
- securitatea şi confidențialitatea datelor.

1. Independența fizică presupune realizarea independenței structurilor de stocare în raport cu structurile de date din lumea reală.

2. Independența logică a datelor se referă la posibilitatea adăugării de noi articole sau extinderea structurii conceptuale, fără ca aceasta să impună rescrierea programelor existente.

3. Manipularea datelor de către neinformaticieni presupune utilizarea unui limbaj cât mai apropiat de limbajul natural, ceea ce permite exploatarea cu uşurință a bazei de date de către utilizatorii finali.

4. Eficacitatea accesului la date se realizează prin:

- limbaje de manipulare a datelor;
- limbaj neprocedural, care permite utilizatorului să descrie ceea ce vrea să obțină fără a da modul în care poate să obțină.

5. Administrarea centralizată a datelor presupune definirea structurii datelor şi a modului de stocare a acestora, permițând organizarea coerentă şi eficace a informației.

6. Neredundanța datelor presupune neduplicarea fizică a datelor. Sunt şi situații în care, pentru micşorarea timpului de acces şi a răspunsului la solicitări, să se accepte o anumită redundanţă a datelor.

7. Coerența datelor presupune satisfacerea constrângerilor statice sau dinamice, locale sau generale.

8. Partajabilitatea datelor presupune utilizarea datelor de mai multe aplicații ce efectuează operații asupra bazelor de date.

Tranzacția este o unitate logică de tratament, care, aplicată la o stare coerentă a bazei de date, generează o nouă stare coerentă a bazei de date.

SGBD asigură gestiunea tranzacțiilor şi a acceselor consecvente pentru evitarea cazurilor de interblocare, în care una sau mai multe tranzacții aşteaptă eliberarea datelor ținute de celelalte tranzacții.

9. Securitatea şi confidențialitatea datelor presupune protecția la accesul neautorizat sau rău intenționat, prin mecanisme care permit identificarea şi autentificarea utilizatorilor.

SGBD trebuie să asigure securitatea fizică şi logică a bazelor de date şi să garanteze că doar utilizatorii autorizați efectuează operații asupra bazelor de date. Această funcție complexă presupune:

Gestiunea autorizațiilor;

Controlul validității operațiilor;

Protecția datelor împotriva accesului neautorizat (parolă, criptare etc.) şi în cazul defecțiunilor. Aceste defecțiuni pot să apară datorită unor manipulări incorecte, unor incidente fizice sau logice, iar SGBD permite menținerea şi repunerea bazei într-o stare coerentă în cazul apariției unei avarii.

Stocarea datelor

Stocarea datelor trebuie sa fie facutab astfel incat recuperarea lor sa fie foarte efcienta. Datele sunt stocate in „compartimente” denumite „magazii” (baze de date). Intreg sistemul de sticare trebuie sa aiba in vedere un sistem de salvare si recuperare a datelor (backup & recovery). Datorita frecventei ridicate cu care se apeleaza aceste doua ultime servicii, stocarea si recuperarea datelor trebuie sa fie de o acuratete foarte mare. Baza de date stocheaza diferite tipuri de date, protejand totusi aceste date impotriva accesului utilizatorilor neautorizati. Ele sunt concepute

folosind structuri ierarhice, de retea sau relationale, fiecare dintre ele fiind eficiente in anumite cazuri.

TPS-ul prezinta urmatoarele proprietati:

- ***Asezarea corecta a datelor:***

  Sistemul ar trebui sa fie conceput astfel incat sa permita mai multor utilizatori sa acceseze eficient aceleasi date.

- ***Tranzactii scurte:***

  Tranzactiile scurte permit o procesare rapida. Acest lucru evita problemele de concurenta si eficientizeaza sistemul.

- ***Backup in timp real:***

  O sesiune de backup ar trebui programata in itpul perioadelor cu activitate scazula pentru a preveni incetinirea (lag) serverului.

- ***Grad de normalizare mare:***

  Acest lucru reduce cantitatea de informatie redundanta pentru a imbunatatii viteza si concurenta. De asemea, este imbunatata viteza cu care se face backup-ul.

- ***Arhivarea datelor vechi:***

  Datele folosite rar sunt mutate in alte tabele sau baze de date de backup (sau scheme). Acest lucru mentine scazuta marimea bazei de date (a tabelelor) si imbunatateste timpii de backup.

- ***O buna configuratie hardware:***

  Configuratia hardware trebuie sa poata sa faca fata aceesului simultan din partea mai multor utilizatori si de asemenea un timp de raspuns mic.

In cadrul unui TPS se folosesc cinci tipuri de fisiere, utilizate in stocarea si organizarea datelor:

- ***Fisierul MASTER:***

  Contine informatii despre situatia organizatiei. Majoritatea tranzactiilor si a bazelor de date sunt stocate in fisierul master.

- ***Fisierul TRANSACTION:***

  Reprezinta totalitatea inregistrarilor tranzactiilor. Ajuta la aducerea la zi (update) a fisierului master si de asemenea serveste ca istoric al tranzactiilor.

- ***Fisierul REPORT:***

  Contine datele care au fost formatate in vederea prezentarii pentru utilizator.

- ***Fisierul WORK:***

  Este un fisier temporar folosit de catre sistem in timpul procesarii tranzactiilor.

- ***Fisierul PROGRAM:***

  Contine instructiunile pentru procesarea datelor.

Bibliografie:

1. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/
2. Principles of Transaction Processing, 1997, Morgan Kaufmann
3. Lorentz Jäntschi, Mădălina Văleanu, Sorana Bolboacă, Programarea rapidă a aplicațiilor pentru baze de date relaționale

---

4. Controlul accesului concurent

Serializabilitate, schedule, graful dependentelor

Neagoe Mirela

- ***Serializabilitate:***

Independenta este lipsita de sens in absenta paralelismului; astfel, se considera un system in care este posibila executia simultana a mai multor procese cu tranzactii.

Definitia serializabilitatii: este o proprietate a executarii tranzactiilor ce implica independenta. O tranzactie se poate numi independenta daca sistemul run-time ce implementeaza tranzactiile garanteaza serializabilitatea oricarei executii a unui set de tranzactii.

- ***Schedule:***

Tranzactia este compusa din mai multe instructiuni. Cand doua tranzactii sunt executate simultan se realizeaza o executie intretesuta a instructiunilor ce le compun. O ordine de executie a instructiunilor din mai multe tranzactii se numeste *schedule*.

Pentru exemplificare vom considera doua tranzactii ce citesc variabila persistenta salariu intr-o variabila locala, o modifica si o pun la loc. Notam tranzactiile cu T1 si T2.

Begin T1;

1) x := Read(salariu);
2) x := x * 110/100;
3) Write(salariu, x);

End T1; Begin T2;

1) y := Read(salariu);
2) y := y + 200;
3) Write(salariu, y);

End T2;

Tranzactiile contin cate 3 instructiuni; exista foarte multe “schedules” posibile pentru executarea lor::

Begin T1;	tt
1) x := Read(salariu);	tt
	ttBegin T2;
	tt1) y := Read(salariu);
2) x := x * 110/100;	tt
3) Write(salariu, x);	tt
	tt2) y := y + 200;
	tt3) Write(salariu, y);
End T1;	tt
	ttEnd T2;

Apar aici doua “schedules”: unul in care toate operatiile lui T1 se executa inaintea inceperii lui T2, si unul in care toate se executa la sfarsitul lui T2. Cele 2 “schedules” sunt *seriale*.

Toate “schedules” constau in amestecari ale instructiunilor T1 cu T2. Un “schedule” este *serializabil* daca produce *exact aceleasi rezultate finale* ca unul dintre “schedule”-le seriale. Rezultatul final reprezinta exact aceleasi valori ale variabilelor persistente.

“Schedule”-ul din exemplul de mai sus nu este serializabil.

Consideram un alt exemplu de date pentru care aceasta idée este evidenta: daca initial salariul avea valoarea 100, atunci schedule-ul de mai sus va da valoarea finala 300 (verificati). Executia T1 urmat de T2 da valoarea 310, iar T2 urmat de T1 va da valoarea 330.

Sistemul ce realizeaza implementarea tranzactiilor trebuie sa excluda posibilitatea executarii instructiunilor unor tranzactii intr-o ordine ce nu este serializabila.

- ***Graful dependentelor***

Putem considera ca pentru anumite operatii ordinea de executie nu este una stricta; daca citim o valoare in doua tranzactii nu are importanta ordinea in care realizam citirea deoarece rezultatul obtinut este acelasi. Daca scriem doua valori intr-o singura variabila persistenta, ordinea scrierilor este importanta pentru rezultatul final, ultima scriere oferind valoarea rezultatului.

Operatiile ce po fi schimbate intre ele fara afectarea valorii rezultatului *comuta*. Toate citirile comuta intre ele, asa cum comuta operatii de orice fel asupra variabilelor locale si operatii care se efectueaza asupra unor variabile persistente *diferite*. Daca scrierea se face in variabila *a* iar citirea in *b* nu conteaza in ce ordine se realizeaza acesti pasi, rezultatul fiind acelasi.

Unul dintre principiile serializabilitatii sustine ca daca se poate schimba intr-un "schedule" instructiunile intre ele si obtinem un "schedule" serial, atunci acesta este serializabil.

Astfel ramane de analizat care dintre instructiunile T1 si T2 nu comuta in acest exemplu. "T1 3)" nu comuta cu "T2 1)", pentru ca "T1 3)" scrie in salariu iar "T2 1)" citeste din salariu. "T1 3)" nu comuta cu "T2 3)", amandoua scriind in aceeasi valoare. Este evident de ce acest schedule nu e serializabil: instructiunea "T1 3)" nu poate fi nici impinsa inainte de T2 nici dupa, din cauza dependentelor.

Putem ilustra asta cu ajutorul unui *graf de dependente* ce caracterizeaza un *schedule*. In graful dependentelor nodurile sunt tranzactii. Intre doua dintre aceste tranzactii se poate avea un arc daca prima executa o operatie ce nu comuta cu o operatie executata ulterior celei de-a doua. In acest caz testul de serializabilitate devine simplu: un "schedule" este serializabil daca in graful dependentelor asociat lui nu exista cicluri.

Verificarea prezentei ciclurilor intr-un graf se realizeaza in timp liniar in numarul de noduri. O sortare topologica a grafului indica ordinea seriala echivalenta cu cea data.

Bibliografie:

1. http://www.cs.cmu.edu/~mihaib/articles/tranzactii/tranzactii-html.html
2. Principles of Transaction Processing, 1997, Morgan Kaufmann
3. Lorentz Jäntschi, Mădălina Văleanu, Sorana Bolboacă, Programarea rapidă a aplicațiilor pentru baze de date relaționale
4. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/

Metode de implementare

Anton Daniel

Sistemele tranzactionale pot fi impartite in doua tipuri mari de implementari din punct de vedere al serializabilitatii:

Sisteme cu control optimist asupra concurentei: acestea afirma ca avem rar conflicte, deci e mai ieftin sa executam tranzactiile si la sfarsit sa facem Abort daca executia nu a fost seriala; atunci fiecare operatie se face rapid si sisteme cu control pesimist, care isi iau toate precautiile de la inceput, in asa fel incat pur si simplu executii ne-serializabile sunt imposibile.

Incuietori

Incuietorile sunt o metoda pentru controlul pesimist al accesului. Ideea este ca atunci cand faci operatii asupra unei valori sa se interzica accesul altor tranzactii la valoarea respectiva; stim ca accesele la valori diferite sunt serializabile.

Tipuri de incuietori

In sistem se folosesc mai multe tipuri de incuietori pentru a nu limita activitatea concurenta. O incuietoare pentru scriere este mai restrictiva decat una pentru citire, pentru ca putem executa mai multe citiri simultan, dar o scriere nu poate fi simultana cu o alta operatie din alta tranzactie.

Manipularea incuietorilor se poate face fie explicit de catre programator fie ascuns de catre sistemul tranzactional. In orice caz, incuietorile sunt asociate valorilor persistente si se manipuleaza cu urmatoarele operatii (primele doua sunt fundamentale, iar urmatoarele auxiliare):

Operatie	Semnificatie
lock(date, operatie)	“incuie” aceste date pentru “operatie”
unlock(date)	elibereaza restrictiile asupra acestor date
upgrade(date)	intareste o incuietoare
degrade(date)	slabeste o incuietoare

Procesul special care tine minte ce date sunt incuiate, de cine si in ce fel poarta numele de lock manager. Atunci cand o tranzactie vrea sa incuie o valoare se petrec urmatoarele lucruri:

Daca datele nu sunt incuiate atunci incuietoarea este pusa si executia continua in mod normal;

Daca pe date exista deja o incuietoare:

Cand incuietoarea precedenta era pentru citire iar cea ceruta este tot pentru citire o a doua incuietoare este pusa, dupa care executia continua iar accesul la date este permis;

Cand incuietoarea precedenta indica un conflict cu cea curenta atunci cererea de incuiere este refuzata.

Exista doua cazuri cand incuierea este refuzata: fie asa a fost implementata baza de date fie aceasta este alegerea utilizatorului

Cel care cere o incuiere imposibila este pur si simplu blocat din executie pana cand valoarea se "descuie", iar atunci executia este automat continuata;

Cel care cere o incuiere imposibila primeste un cod de eroare si decide ce trebuie sa faca (de pilda sa Abort-eze tranzactia curenta).

Terminarea unei tranzactii (cu *Commit* sau Abort) elibereaza intotdeauna toate incuietorile.

O tranzactie este obligata sa incuie datele inainte de a le accesa. Un exemplu ar fi:

```
Begin T2;
   lock(salariu, readLock);  { incuie pentru citire }
   y := Read(salariu);
   y := y + 200;
   upgrade(salariu);        { transforma incuietoarea pentru scriere }
   Write(salariu, y);
   unlock(salariu);
End T2;
```

Protocolul in doua faze

Doar punand incuietori in jurul acceselor la date nu e suficient. Consideram tranzactiile T1 si T2 in forma in care exact fiecare acces este protejat, ca in exemplul urmator pentru T2:

```
Begin T2;
   lock(salariu, readLock);
   y := Read(salariu);
   unlock(salariu);
   y := y + 200;
   lock(salariu, writeLock);
```

Write(salariu, y);
unlock(salariu);
End T2;

E usor de observat ca o astfel de incuietoare permite executii neserializabile (chiar executia indicata mai sus este posibila). Nu ajunge deci sa incuiem, trebuie sa respectam un protocol de incuiere. Cel mai simplu si mai des folosit protocol este cel in doua faze (two-phase locking; 2PL). Acest protocol se poate enunta astfel:

In prima faza se pot numai obtine incuietori (adica se executa instructiuni lock si upgrade).

In a doua faza se pot numai dealoca incuietori (unlock sau degrade).

Numarul de incuietori pe care le are o tranzactie trebuie sa aiba o faza de crestere (growing phase) si una de descrestere (shrinking phase) in timp.

Acest protocol garanteaza serializabilitatea.

2PL strict

Exista un dezavantaj al lui 2PL si o implementare care il evita. Se consideram urmatorul scenariu:

O tranzactie T1 incuie o valoare x, o modifica si apoi o descuie. T2 vine la rand, incuie si citeste x. Daca acum T1 vrea sa execute Abort, valoarea lui x trebuie pusa cum era inainte ca T1 sa o fi modificat. Dar T2 a citit-o deja. Asta inseamna nimic altceva decat ca daca T1 executa Abort, T2 trebuie sa fie anulata de sistem, pentru a respecta proprietatea de izolare. Lantul poate fi mai lung: T3 poate a citit la randul ei x, si atunci trebuie anulata si ea.

O astfel de situatie foarte neplacuta (pentru ca o multime de operatii trebuie sterse) se numeste *cascaded abort* (abort in cascada). (O alta consecinta neplacuta este ca T2 nu poate fi terminata inainte de T1, pentru ca daca T2 face *Commit* iar T1 Abort se genereaza erori, caci T2 garanteaza ca valoarea lui x este permanenta, dar nu avea voie s-o citeasca. Deci T2 trebuie sa astepte ca T1 sa se termine.)

Solutia este simpla, se restrange concurenta, dar permite nimanui sa vada ce s-a modificat. Nu se descuie nimic pana la sfarsit, cand se descuie totul dintr-o miscare (de exemplu folosind End Transaction, care elibereaza incuietorile).

Blocarea (deadlock)

Incuietorile sunt foarte bune pentru a obtine serializabilitate, dar au un dezavantaj. Pot aparea situatii de blocare totala, din care nu exista iesire. Aceasta se cheama *deadlock*.

Iata un exemplu foarte simplu de “schedule” in care doua tranzactii incuie niste valori, dupa care nici una nu mai poate progresa nicicum, pentru ca vrea valoarea incuiata de cealalta:

T1	T2
lock(x, writeLock)	
	lock(y, writeLock)
lock(y, writeLock)	
refuz: blocat	lock(x, writeLock)
	refuz: blocat

Problema deadlock-ului este complicata. Exista tehnici standard pentru a o trata:

Prevenirea deadlock-ului, folosind protocoale speciale pentru a pune incuietori (ex: protocoale ierarhice);

Evitarea deadlock-ului, alocand resurse in mod special sau folosind tehnici bazate pe timp (vom vedea exemple mai jos);

Detectarea si repararea deadlock-ului: algoritmi care analizeaza cine pe cine asteapta pot detecta deadlock-ul si pot ucide una dintre tranzactiile implicate. Mecanismul de Abort se va ocupa de restul.

Algoritmii pentru detectarea deadlock-ului sunt extrem de sofisticati in sisteme distribuite, si de aceea se incearca cat mai mult evitarea acestui fenomen.

O metoda nu prea precisa consta in a anula orice tranzactie care a asteptat mai mult de un anumit timp eliberarea unei incuietori; e clar atunci ca nu se pot petrece deadlock-uri, dar s-ar putea ca anumite tranzactii sa fie anulate doar pentru ca stateau la o coada prea lunga.

Granularitatea incuietorilor

O problema foarte importanta este granularitatea unei incuietori: cat de mare este o valoare care se incuie. incuietori mai mari restrang paralelismul posibil, dar reduc costul. Astfel putem incuia:

O singura valoare;

O inregistrare (antr-o baza de date relationala) (un record);

O intreaga relatie (un fisier);

O parte din baza de date;

Intreaga baza de date.

Incuietori de marimi diferite ridica si alte probleme: de exemplu trebuie avut grija sa nu existe o tranzactie care incuie o valoare pentru scriere si o alta care incuie tot fisierul care contine valoarea, pentru ca atunci apare un conflict.

Timestamps

O alta metoda de control al activitatii concurente este de a impune tranzactiilor de la inceput o anumita ordine de acces la date. De pilda tranzactiile sunt etichetate cu ora la care au fost lansate, si o tranzactie mai veche nu este lasata cu nici un chip sa faca o operatie asupra unei valori atinse de o tranzactie mai recenta. Cu alte cuvinte se alege de la inceput unul dintre "schedule"-urile seriale cu care va fi echivalent cel al executiei curente.

Metoda foloseste cate doua etichete temporale (timestamps) pentru fiecare valoare persistenta. O eticheta (RT: read time) indica ultima tranzactie care a citit valoarea, iar o alta (WT: write time) indica ultima tranzactie care a scris valoarea. Notand eticheta unei tranzactii T1 cu T(T1), protocoalele de acces la date ale unei tranzactii vor fi:

```
procedure read(date, tranzactie)
begin
  if T(tranzactie) >= RT(date) then begin
      RT(date) := T(tranzactie);
      return date;
  else
      Abort;
end;
procedure write(date, valoare, tranzactie)
begin
  if (T(tranzactie) >= max(RT(date), WT(date))) then begin
      date := valoare;
      WT(date) := T(tranzactie)
  end
  else if RT(date) > T(tranzactie) then Abort
  else if RT(date) <= T(tranzactie) and
        T(tranzactie) < WT(date) then
  ;    { nu face absolut nimic; ignora scrierea }
end;
```

Aceste proceduri vor lasa o tranzactie mai veche sa faca operatii asupra unei valori numai daca tranzactii mai noi nu au facut operatii care nu comuta. Cazul scrierii este interesant: daca o tranzactie veche scrie peste o valoare pe care a scris intre timp o tranzactie mai noua (dar intre cele doua timpuri nimeni nu a citit valoarea), atunci valoarea scrisa de tranzactia veche este pur si simplu ignorata. Asta pentru ca daca tranzactia veche ar fi scris inainte valoarea, nimeni nu ar fi apucat sa o citeasca pana cand cea noua ar fi suprascris-o.

"Timestamps" si "locks"

Putem combina laolalta incuietorile cu "timestamps" pentru a evita deadlock-ul. Exista doua solutii care folosesc "timestamps" pentru a arbitra accesul la un lock:

wait-die:

Cand o tranzactie vrea o incuietoare tinuta de o tranzactie mai recenta atunci trebuie sa astepte;

Cand o tranzactie vrea o incuietoare a uneia vechi, cea noua este anulata.

wound-wait:

Cand o tranzactie vine si vrea o incuietoare tinuta de o tranzactie mai tanara, tanara este anulata;

Tranzactiile tinere asteapta incuietorile tinute de varstnice.

Amandoua schemele dau prioritate tranzactiilor care au trait mai mult, in ideea ca au facut mai multa treaba si e pacat sa le omoram. Wait-die poate produce livelock, sau "starvation" (cand tranzactii tinere sunt mereu repornite si nu apuca niciodata un lock). Nu se pot produce deadlock-uri, pentru ca in orice ciclu tranzactia cea mai tanara trebuie sa moara.

O metoda optimista: algoritmul validarii

O metoda care nu restrange deloc accesul concurent este urmatoarea: cine are de facut modificari strange toate valorile de modificat in variabile locale, iar cand a terminat verifica daca poate sa salveze modificarile. Asta se poate daca nimeni nu s-a atins intre timp de valorile initiale. Daca aceasta faza (de validare) se termina cu succes toate valorile sunt salvate (printr-o operatie atomica) in baza de date. Altfel tranzactia face Abort.

Ca sa verifice daca nimeni nu s-a atins de valori intre timp, fiecare valoare este etichetata cu un numar de versiune, care este incrementat la fiecare scriere. Daca se observa in momentul cand se doreste sa se salveze rezultatele, ca versiunea s-a schimbat fata de momentul cand datele au fost citite, se renunta sa a le mai scrie.

Un astfel de protocol este folosit de sistemul de fisiere NFS (network file system): fiecare fisier pe server are o versiune. Cand cineva sterge un fisier si creaza altul folosind aceleasi portiuni de pe disc versiunea se incrementeaza. Daca un alt client vrea sa faca operatii pe un fisier care nu mai exista, server-ul detecteaza acest lucru, pentru ca acel clientul ofera versiunea veche a fisierului. Astfel accese ilegale sunt interzise.

Acestea au fost tehnici alternative pentru a implementa proprietatea de izolare a tranzactiilor. Un anume sistem va implementa probabil una singura dintre ele.

Protocol	**Calitati**
2PL	deadlock si cascaded-abort posibile
2PL strict	deadlock posibil; concurenta redusa fata de 2PL
Timestamp	restrange “schedule”-le posibile; necesita spatiu suplimentar pentru etichete; concurenta mare
Validare	poate cauza abort-uri tardive; cere spatiu suplimentar
Timestamp + lock	pot cauza “starvation”; nu au deadlock

Bibliografie:

1. http://www.cs.cmu.edu/~mihaib/articles/tranzactii/tranzactii-html.html
2. Principles of Transaction Processing, 1997, Morgan Kaufmann
3. Grupul BDASEIG, ASE, Baze de date – fundamente teoretice si practice, Ed. InfoMega, 2002
4. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/

Probleme si anomalii de executie a tranzactiilor bazelor de date

Ganciu Cristiana Valentina

Operatiile concurente sunt referite prin notiunea de tranzactie[1]. Aceasta reprezinta o unitate logica de lucru definita ca o singura executie a unui program. Programul poate sa contina o singura cerere exprimata intr-un limbaj de cereri sau o serie de instructiuni in limbajul gazda impreuna cu cereri. Se considera baza de date ca fiind alcatuita din elementele ce pot fi bloacate la un moment dat. Blocarea unui element permite accesul unei tranzactii la acel element si interzice accesul celorlalte tranzactii la elementul respectiv pana la terminarea tranzactiei curente, cand are loc o deblocare.

Pot aparea unele fenomene ce trebuie tratate de sistemul de baze, cum sunt **asteptarea la nesfarsit (livelock)** sau **interblocarea (deadlock)**.

Asteptarea la nesfarsit poate sa apara cand o tranzactie asteapta utilizarea unui element care este ocupat de noi tranzactii ce apar in sistem. Ea poate fievitata printr-o strategie de tip coada pentru fiecare element inparte. Au prioritate de folosire a unui element tranzactiile cu cea mai veche cerere pentruacel element.

Interblocarea apare cand unele tranzactii detin unele elemente cerute de alte tranzactii si, la randul lor, cer elemente detinute de alte tranzactii, ajungandu-se la o asteptare in cerc inchis in care nici o tranzactie nu mai poate continua.

Rezolvarea interblocarilor se poate face prinmai multe metode cum ar impunerea satisfacerii tuturor cererilor de elemente pentru o tranzactie inainte de a ase executa, impunerea unei ordini asupra elementelor si prezentarea cererilor de elemente in tranzactii in aceasta ordine, determinarea si tratarea interblocarilor prin intreruperea si reluarea unor tranzactii. Evitarea interblocarii e asigurata prin impunerea unor conditii asupra ordinii de efecuare a operatiilor fiecarei tranzactii in parte.

O alta problema este legata de rezultatul obtinut prin efectuarea anumitor tranzactii. Se numeste “programare a unei multimi de tranzactii” ordinea in care sunt executati pasii elementari ai tranzactiilor; pentru fiecare tranzactie in parte, ordinea pasilor ei coincide cu ordinea de executie din tranzactia respectiva.

[1] Octavian Basca, *Baze de date*, ED. All, 1997, pag. 137

Pentru pastrarea consistentei bazei de date, orice tranzactie trebuie sa se execute in totalitate, fie sa nu se execute deloc.

Erorile ce pot declansa procesul de restabilire a bazei de date se clasifica in erori locale, care sunt produse de tranzactie si afecteaza numai tranzactia respectiva, si erori globale care por afecta mai multe tranzactii active la un moment dat. Erorile globale pot fi erori de sistem(efectuarea eronata a unor comenzi, erori din sistemul de operare s.a.) sau erori de mediu(de ex. functionarea incorecta a unei portiuni din memoria interna). Erorile locale care tinde dependenta datelor vor fi prezentate in cele ce urmeaza.

Existenta unor legaturi logice sau dependente, intre atributele aceleiasi clase de entitati sau intre atributele unor clase de entitati diferite, determina aparitia unor anomalii la realizarea operatiilor concurente asupra bazelor de date, denumite anomalii la reactualizare: adaugare, stergere si modificare. Trebuie identificate campurile care depind de alte campuri.

Ex.

Fie schema din relatie:

Carte(Isbn *, Titlu, Editura, AdresaEditura)

In care se observa existenta unei dependente a atributului ISBN de atributul Editura. In ipoteza ca fiecare carte, identificata prin numarul Isbn, are o singura editura, atunci fiecare valoare a atributului Isbn determina valoarea corespunzatoare a atributului Editura. Dependenta intre cele doua atribute conduce la aparitia urmatoarelor anomalii:

-anomalii la adugare: nu se poate adauga in baza de date o carte noua pana nu se cunosc datele editurii;

-la stergere: se refera la cazul in care se peird coordonatele editurii in momentul stergerii tuturor publicatiilor acelei edituri;

-la modificare: apare datorita prezentei redundante a atributului Editura si AdresaEditura pentru fiecare publicatie inregistrata a unei anumite edituri. In cazul in care se modifica coordonatele unei edituri este necesaraparcurgerea intregii relatii pentru depistarea si corectarea tuturor aparitiilor editurii in cauza; in caz contrar, apare pericolul introducerii unor inconsistente in baza de date – aceeasi editura poate aparea cu adrese diferite.

Eliminarea anomaliilor prezentate se realizeaza prin operatia de normalizare a bazei de date. Normalizarea este o tehnica de realizare a unui set de relatii cu proprietati adecvate. E o

metoda formala, care poate fi utilizata pentru identificarea relatiilor, bazandu-se pe cheile si pe dependentele functionale dintre atributele acestora. Principalele scopuri urmarite in rocesul de normalizare sunt: eliminarea unor redundante, evitarea unor anomalii de reactualizare, realizarea unui proiect care sa reprezinte cat mai fidel modelul real(usor de inteles si, eventual de modificat, stabilirea unor constrangeri de integritate simple…).

Anomaliile depistate in exemplul enterior, datorita dependentelor din schema de relatie carte, se elimina prin descompunerea in doua scheme de relatie Carte si Editura.

Carte(Isbn *, Titlu, CodEditura)

Editura(CodEditura *,Editura, AdresaEditura)

Observatii[2]:

1. Prin normalizarea bazelor de date se asigura o integritate sporita a datelor.
2. Normalizarea mareste numarul de clase de entitati(tabele) si astfel se mareste considerabil timpul de regasire a informatiei, prin cuplarea unui numar mai mare de tabele. Cresterea timpului de rezolvare al unei interogari se considera un dezavantaj.
3. Tabelele suplimentare rezultate in urma normalizarii bazei de date s-ar putea sa nu fie necesare la o analiza mai atenta si atunci se recurge la denormalizarea bazei. Denormalizarea determina cresterea redundantei datelor, dar mai reduce numarul de tabele si timpul de rezolvare al interogarilor.

Pe masura ce relatia e transformata in forme superioare, devine din ce in ce mai restrictiva si mai expusa anomaliilor de reactualizare. Normalizarea este transformarea unei relatii(descompunerea ei) intr-o multime de relatii de un anumit tip (o colectie de relatii mai simple care evita anomaliile de adaugare, stergere si actualizare). Pentru cazul particular al descompunerii unei scheme de relatie R in doua proiectii R1 si R2 se poate formula o regula simpla pentru verificarea proprietatii de conservare continutului de informatie. Aceasta regula este data de teorema lui Ullman[3]:

Teorema:

Fie $\rho=\{R1,R2\}$ o descompunere a schemei de relatie R, atunci ρ constituie o descompunere fara pierdere de informatie, in raport cu un set dat de dependente functionale initiale, daca inurma descompunerii avem una dinurmatoarele dependente functionale:

[2] Colin Dan Maier, *Analiza, Proiectarea si implementarea bazelor de date, aplicatii in Visual FoxPro*, Ed. Albastra, Cluj Napoca, 2002, pag. 38

[3] Robert Dollinger, Baze de date di gestiunea tranzactiilor, Ed. Albastra, Cluj Napoca, 1998, pag. 148

$(R1 \cap R2) \rightarrow (R1\text{-}R2)$,

$(R1 \cap R2) \rightarrow (R2\text{-}R1)$

Din formularea teoremei de mai sus se constata ca proprietatea de conservare a informatiei depinde nu numai de descompunerea ρ, ci si de setul initial de dependente functionale existente in schema de relatie R.

S-au facut mai multe clasificari pentru a deosebi anumite calitati specifice ale unor relatii. Dintre acestea, cea mai frecvent utilizata este clasificarea in forme normale. Se spune ca o relatie este intr-o foema normala particulara daca satisface o multime data de constrangeri. Primele 3 forme normale au fost definite de Codd, iar a patra si a cincea de Fagin.

Concluzii:

Prin procesul de normalizare se realizeaza eliminarea din schemele de relatie a dependentelor cu scopul de a produce o schema relationala avand proprietati mai bune din punctul de vedere al redondantei datelor si al posibilelor anomalii ce por apare in cazul operatiilor de adaugare, stergere si actualizare.

Teoria normalizarii trebuie privita in primul rand ca un cadru de ghidare a proiectantului bazei de date in vederea realizarii de scheme relationale mai bune. Teoria normalizarii ofera proiectantului un instrument de lucru care permite o mai buna reflectare a semanticii lumii reale in modelele create.

Bibliografie:

1. Robert Dollinger,*Baze de date si gestiunea tranzactiilor*, Ed Albastra, Cluj Napoca, 1998

2. Collin Dan Maier, *Analiza, proiectarea si implementarea bazelor de date*, Ed. Albastra, 2002

3. Thommas Connoly, Begg Carolyn, Anne Strachan, *Baze de date – proiectare, implementare si gestionare*, Ed. Teora, 2001

4. Octavian Basca, *Baze de date*, Ed. All, 1997

5.Pastrarea consistentei si recuperarea datelor

Cealicu Alexandru Ciprian

- ***Erori (crash):***

Tranzactiile respecta proprietatile de atomicitate si durabilitate chiar si in momentul producerii unor erori grave (crash). Esential in acest caz este jurnalul (log), pe baza caruia se pot face operatiuni de tipul “undo – redo” (stergerea modificarilor temporare sau recuperarea modificarilor temporare).

Se doreste asigurarea a doua proprietati:

Durabilitate:

O tranzactie in cadrul careia s-au confirmat modificarile (commit) trebuie sa aiba efecte permanente, in orice situatie.

Atomicitate:

Dupa aparitia unei erori grave (crash) se executa un protocol de refacere a datelor (practic, revenirea la starea initiala – cea de dinaintea inceperii tranzactiei) denumit “recovery protocol”. Acest lucru permite reluarea operatiunilor in mod corect, prin anularea tranzactiilor ce nu erau confirmate la momentul aparitiei erorii. De asemenea, in timpul actiunii de refacere a sistemului se urmareste ca aceasta sa nu produca alte probleme, daune, erori.

Pentru a putea efectua acest lucru se respecta anumite reguli, poate cea mai importanta fiind:

Se raporteaza rezultatul unei tranzactii (succes sau nu) doar in cazul in care modificarile efectuate in cadrul acesteia sunt stocate pe un mediu stabil, sigur, de unde ulterior pot fi accesate. Acest mediu nu trebuie sa poata fi afectat de diferitele defectiuni sau erori ale sistemului. Ca exemplu, si in functie de sistem, mediul stabil poate fi memoria RAM, memoria NVRAM, hard-disk, banda magnetica si altele.

Regula este importanta deoarece anumite actiuni (reale) nu pot fi anulate, asa cum ar fi de exemplu tiparirea unui bilet de avion. In momentul in care clientul a luat acest bilet si plecat, biletul nu mai poate fi recuperat (evident, acest lucru este supus aparitiei de exceptii).

In acest fel se poate garanta durabilitatea sistemului (prin stocarea datelor modificate pe mediul stabil).

- ***Jurnalul (log):***

Un jurnal este o secventa liniara, teoretic indefinit de mare, in care se descriu operatiile prin introducerea de inregistrari in momentul efectuarii acestora. Jurnalele se pot imparti in doua categorii:

1. Jurnalul de intentii sau jurnalul de re-executare a operatiilor (redo-log). In momentul in case modificarile se fac in cadrul unei copii a datelor, noile valori sunt marcate si introduse in jurnal. In cazul efectuarii unei confirmari (commit), jurnalul tranzactiei este accesat de sistem si sunt re-executate toate operatiunile descrise in acesta. In acest caz operatiile sunt efectuate direct in sistem (baza de date) intr-o maniera atomica.
2. Jurnalul de revenire la starea initiala (undo) retine, evident, valorile initiale ale datelor ce au fost modificate in cadrul tranzactiei. Pentru fiecare modificare sunt retinute valorile initiale si in cazul in care operatiunea este anulata sistemul acceseaza acest jurnal si revine la starea initiala prin preluarea acestor valori (initiale). Tehnica de parcurgere inversa prin anularea modificarilor se numeste “rollback” (anularea modificarilor).

Jurnalul se poate folosi atat in cazul implementarii oepratiunilor “Abort” (anulare) si “Commit” (confirmare) cat si pentru respectarea proprietatilor de atomicitate si durabilitate in cazul aparitiei erorilor. Se poate folosi, evident, un jurnal care este o combinatie intre cele doua tipuri prin marcarea modificarilor (operatiile, tipul lor) care sunt efectuate asupra unor valori cat si prin valorile dinaintea inceperii tranzactiei cat si valorile rezultate in urma incheierii acesteia.

- ***Puncte de control (checkpoints):***

Problematica jurnalului consta in cresterea acestuia. Teoretic, acesta creste la infinit iar spatiul ocupat nu este eliberat. O metoda de recuperare a spatiului folosit de catre jurnal in mod inutil (criteriu stabilit subiectiv – in mod evident toate inregistrarile din jurnal sunt importante insa depinde pana la ce punct dorim sa ne intoarcem. Astfel, pe baza unor criterii, utilizatorul stabileste niste limite pana la care considera ca toate oepratiunile efectuate au fost corecte) se bazeaza pe principiul punctelor de control (check-points).

Daca se cunoaste starea initiala a sistemului si toate modificarile facute asupra lui (in ordine) atunci se poate deduce starea curenta a acestuia.

In cazul in care se cunoaste starea initiala a sistemului si toate modificarile efectuate (in ordine) se poate deduce starea curenta. Pe acest principiu lucreaza jurnalul. Insa, de la un anumit moment, cantitatea de date necesare descrierii modificarilor poate sa depaseasca cantitatea de

date necesare descrierii starii sistemului. In acel moment se considera modificarile ca fiind corecte si lista descrierilor modificarilor este compactata. Inghetarea sistemului in acest punct se numeste “checkpoint”.

Deoarece sistemul nu este niciodata intr-o stare stabila, in orice moment existand tranzactii in desfasurare ce vor avea rezultate incerte (din punctul de vedere al succesului operatiunilor efectuate) se adopta urmatoarea solutie:

Se va opri activitatea sistemului iar jurnalul este parcurs in ordine inversa. Informatiile despre tranzactiile care au fost terminate (cu succes) se vor sterge din jurnal, deoarece modificarile produse de acestea sunt deja in sistem (garantat prin proprietatea de durabilitate). In jurnal se mai pastraza doar informatiile despre tranzactiile curente neterminate. In general acest lucru se efectueaza tot cu ajutorul jurnalului – scriindu-se o inregistrare de tip “checkpoint”, urmand ca acesta sa fie accesat si toate inregistrarile active sa fie copiate. Spatiul de dinaintea punctului de control poate apoi fi reutilizat.

Aceasta tehnica este folosita si in sistemele de oeprare unde proceselor care ruleaza pentru o perioada indelungata li se aplica astfel de puncte de control. In cazul unei erori se reiau operatiunile incepand cu ultimul punct de control.

Un exemplu concludent se poate observa la incercarea de instalare a unei noi aplicatii in WindowsXP. In cazul in care incercarea este nereusita (si in sistem exista save-point) se poate face un recovery pana la acel save-point (ideal, acel save-point este starea sistemului inainte de incercarea de a instala aplicatia).

- ***Jurnal “write-ahead”:***

Pentru garantarea atomicitatii este necesara o metoda de identificare a tranzactiilor neterminate in momentul aparitiei erorii, pentru a le putea anula. Deoarece o eroare majora distruge practic valorile variabilelor nonpersistente, aceasta informatie – necesara pentru identificarea tranzactiilor in cauza – trebuie stocata pe un mediu stabil. Cea mai importanta regula este:

Operatiile pot fi efectuate in bazele de date numai daca inregistrarea din jurnal aferente acestor operatii au fost stocate pe un mediu stabil. In cazul in care ar aparea o eroare in timpul efectuarii unei tranzactii, iar aceasta nu a fost inregistrata in jurnal (sau inregistrarea din jurnal se pierde, nu poate fi recuperata) nu s-ar mai putea discerne modificarile efectuate de aceasta tranzactie in sistem.

- ***Protocolul de refacere (recovery):***

In cazul in care se respecta regula mentionata anterior rezulta ca exista destula informatie necesara refacerii sistemului (bazei de date) in eventualitatea aparitiei unei erori grave.
Algoritmul de refacere este compus din urmatoarele etape:

1. Se scaneaza jurnalul si se identifica tranzactiile care au generat eroarea (sau daca s-a pierdut conexiunea intre client si server – pentru tranzactiile pe baza de date sau retea).
2. Tranzactiile ce sunt inregistrate cu confirmare a modificarilor (commit) se re-executa in ordinea in care au fost efectuate.
3. Tranzactiile din jurnal care nu au fost terminate se vor executa (special) in ordine inversa si modificarile produse asupra sistemului de catre aceste tranzactii vor fi anulate.

Inregistrarile din jurnal trebuie sa satisfaca o proprietate suplimentara:

Actiunile acestora trebuie sa fie idempotente – acest lucru insemnand ca executarea unei actiuni aferente unei inregistrari din jurnal trebuie sa dea aceleasi rezultate de fiecare data. Astfel se evita aparitia problemelor in momentul re-executarii operatiunilor deja efectuate. Mai mult, se evita aparitia problemelor in cazul aparitiei unei doi erori.

Acest lucru ofera si posibilitatea cresterii performantei in implementarea tranzactiilor, nefiind necesar sa se inregistreze in jurnal de fiecare data cand se efectueaza o operatiune. Se vor introduce in jurnal inregistrari numai in momentul modificarii sistemului. Momentul cel mai important in care este necesara salvarea pe disc este cel de dinaintea terminarii tranzactiei, celalalte modificar se pot amana.[1]

Bibliografie:

1. Grupul BDASEIG, ASE, Baze de date – fundamente teoretice si practice, Ed. InfoMega, 2002
2. http://www.cs.cmu.edu/~mihaib/articles/tranzactii/tranzactii-html.html
3. Principles of Transaction Processing, 1997, Morgan Kaufmann
4. Lorentz Jäntschi, Mădălina Văleanu, Sorana Bolboacă, Programarea rapidă a aplicațiilor pentru baze de date relaționale

Bibliografie generala:

1. Collin Dan Maier, analiza, proiectarea si implementarea bazelor de date, Ed. Albastra, Cluj Napoca, 2002
2. Thomas Connoly, Begg Carolyn, Anne Strachan, Baze de date – proiectare, imlementare si gestionare, Ed. Teora, 2001
3. Octavian Basca, Baze de date, Ed. All,Bucuresti, 1997
4. Robert Dolliger, Baze de date si gestiunea tranzactiilor, Ed. Albastra, Cluj Napoca, 1998
5. Grupul BDASEIG, ASE, Baze de date – fundamente teoretice si practice, Ed. InfoMega, 2002
6. http://www.cs.cmu.edu/~mihaib/articles/tranzactii/tranzactii-html.html
7. Principles of Transaction Processing, 1997, Morgan Kaufmann
8. Lorentz Jäntschi, Mădălina Văleanu, Sorana Bolboacă, Programarea rapidă a aplicațiilor pentru baze de date relaționale
9. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/