# Tranzacţii

Mihai Budiu — mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

Martie 1998

**Subiect:** Tranzacţiile atomice: proprietăţi, implementări, utilizare

**Cunoştinţe necesare:** programare, concurenţă

**Cuvinte cheie:** atomicitate, durabilitate, independenţă, consistenţă, *log* (înregistrare), *lock* (încuietoare), *timestamp*

## Cuprins

# 1 Unelte

La o adică un calculator nu este altceva decît o unealtă. O unealtă care este folosită cu foarte mult succes în rezolvarea a fel de fel de probleme. Hai să luăm totuşi calculatorul ca pe un scop în sine, şi să aruncăm o scurtă privire asupra naturii muncii din domeniu, fie ea scriere de software sau proiectare de hardware (permiteţi-mi să contrag totul la aceste două categorii). Aş vrea să observ că poate mai pregnant decît în alte domenii tehnice, munca din calculatoare constă tot din creaţia unor unelte.

Ce sunt sistemele de operare? Nişte unelte care măresc productivitatea utilizării unui calculator. Ce sunt limbajele de programare? Unelte care înlesnesc exprimarea unor operaţii complexe. Ce sunt reţelele de calculatoare? Unelte care mijlocesc schimbul de informaţii. Procedurile sunt unelte care rezolvă subprobleme, etc. Şi tot aşa, la scară macro sau micro, peste tot numai unelte. Un bun specialist în proiectarea aplicaţiilor sau sistemelor are la dispoziţie o paletă amplă de unelte din care le alege pe cele mai potrivite scopurilor sale.

Unele probleme sunt de nerezolvat fără unelte potrivite; e ca şi cum ai încerca să înţelegi teoria relativităţii cunoscînd numai aritmetica de clasa a doua. Uneltele nu numai că dau chei pentru soluţionarea dificultăţilor, ci crează şi un cadru ordonat în care să ne exprimăm gîndurile. Este de neconceput scrierea de software la ora actuală fără suportul conceptului de "proces": un program care se execută în izolare. Ori procesul nu este altceva decît o altă unealtă construită de către sistemul de operare; o abstracţie, dar una extrem de utilă.

În acest articol voi prezenta tot o unealtă. Una mai puţin cunoscută, şi care a fost aplicată în relativ puţine contexte, deşi este extrem de expresivă şi utilă. Dar lucrurile se vor schimba cu siguranţă în ceea ce o priveşte, pentru că domeniul ei de aplicaţii se lărgeşte permanent, de la bazele de date, în care a fost (şi este) folosită iniţial pînă la sistemele de operare sau aplicaţiile distribuite. Este vorba de *tranzacţii*, numite şi "tranzacţii atomice".

# 2 Ce sunt tranzacţiile

Dintru început trebuie spus că subiectul este extrem de generos. Există cărţi întregi scrise despre tranzacţii, există varietăţi extrem de exotice de tranzacţii, dar evident, în spaţiul unui astfel de articol nu putem să ne aventurăm prea departe. Vom fi deci relativ "tradiţionali", discutînd despre unul dintre cele mai simple modele de tranzacţii, cam aşa cum este el întîlnit în majoritatea cărţilor de baze de date.

Ce e o tranzacţie? O tranzacţie este o unealtă pusă la îndemînă unui programator prin care acesta poate exprima anumite proprietăţi ale părţilor unor programe. Principala proprietate pe care tranzacţiile o pun la dispoziţie este *atomicitatea*. Cuvîntul atom, spuneau manualele de fizică de liceu pe vremea mea, însemna în greaca veche "indivizibil", "care nu poate fi spart în bucăţi".

O tranzacţie îi permite unui programator să grupeze mai multe operaţii elementare într-un tot indivizibil, să construiască deci noi operaţii, dar într-un fel care să nu ne permită să discernem structura lor interioară (adică faptul că sunt compuse din mai multe bucăţele). Vom vedea imediat că acest lucru poate fi interpretat în mai multe feluri, depinzînd de contextul la care ne gîndim.

# 3 Proprietăţi ale tranzacţiilor

Înainte de a vedea cum arată sau cum se implementează tranzacţiile o să le trecem în vedere proprietăţile. În literatură sunt evidenţiate patru proprietăţi esenţiale, ale căror iniţiale sunt ACID.

Să reţinem însă că anumite implementări ale tranzacţiilor nu oferă toate aceste proprietăţi, probabil pentru că domeniul de utilizare al nu are neapărată nevoie de unele din proprietăţi. Vom vedea de asemenea că proprietăţile acestea nu sunt complet independente, şi că anume mecanisme de implementare pot oferi simultan mai multe proprietăţi; o oarecare independenţă există între proprietăţi.

## 3.1 Atomicitate

O tranzacţie grupează mai multe instrucţiuni laolaltă într-o "macro-instrucţiune" care se comportă ca o unitate indivizibilă. Putem vedea această proprietate în două feluri, care se numesc respectiv atomicitate şi independenţă. Primul punct de vedere este discutat în această secţiune, iar al doilea ceva mai tîrziu.

Un grup de instrucţiuni dintr-o tranzacţie se comportă ca o singură entitate atunci cînd se întîmplă o catastrofă. Chiar dacă am grupat 10 instrucţiuni şi curentul cade după ce am executat numai primele 5, sistemul de tranzacţii trebuie să ne asigure că după reluarea execuţiei efectul celor cinci instrucţiuni nu este vizibil. O întrerupere catastrofală trebuie să lase sistemul într-o stare corectă, care ar fi putut surveni în urma unei execuţii normale. Cu alte cuvinte dacă am grupat nişte instrucţiuni într-o tranzacţie atunci se execută ori toate ori niciuna: "totul sau nimic".

Să luăm un exemplu ipotetic: un program de contabilitate care calculează plăţile făcute angajaţilor. Programul trebuie ca pentru fiecare angajat să marcheze salariul de plătit şi să scadă aceeaşi sumă din budgetul întreprinderii (îmi cer scuze dacă o dau în bară cu terminologia contabilă; ideea contează). Programul ar putea arăta aşa:

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

Dacă presupunem că valorile modificate sunt înregistrări pe disc, într-o bază de date, atunci dacă curentul cade undeva la mijlocul buclei anumite salarii vor fi fost plătite, dar suma nu a fost scăzută din budget. Programul nu poate fi reluat de la început, pentru că anumite salarii au fost plătite, dar altele nu.

Soluţia este să executăm toate aceste instrucţiuni într-o singură tranzacţie, care atunci cînd este întreruptă inainte de terminare va face ca modificările făcute să dispară. Cum, vom vedea mai tîrziu.

## 3.2 Independenţa

Independenţa se referă la un context în care mai multe programe acţionează simultan asupra aceluiaşi set de date (activitate concurentă). Un program nu trebuie să vadă rezultatele parţiale ale altui program, pentru că acestea nu ar fi corecte.

Ca exemplu să considerăm un program care implementează o mărire de salariu (ura!), ceva de genul:

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

Ce se poate întîmpla dacă acest program se execută "simultan" cu cel anterior? O posibilitate este ca acest program să pornească în execuţie după cel precedent, să ajungă la acelaşi indice undeva şi apoi să o ia înainte. În acest fel unora dintre angajaţi le vor fi plătite salarii indexate (celor din urmă), iar altora nu. Total necinstit, şi nu numai necinstit, ci arbitrar, pentru că rezultatul depinde de momentul la care se execută programele şi de viteza lor relativă.

Dacă fiecare din aceste fragmente de program ar fi fost o tranzacţie aşa ceva nu s-ar fi putut întîmpla: toate măririle ar fi fost executate ori înainte ori după toate plăţile.

## 3.3 Consistenţa; programarea cu invarianţi

Să aruncăm o privire la totalitatea variabilelor unui program. La un anumit moment de timp ele vor avea fiecare o valoare. Dar nu orice combinaţie de valori este posibilă. De exemplu, pentru un program care sortează un vector de numere, în vector se vor afla tot timpul aceleaşi numere, poate în altă ordine. Această regulă (mai precis acest predicat) se numeşte un *invariant*, pentru că trebuie să fie tot timpul adevărată, indiferent de faza de execuţie a programului. Pentru primul program de mai sus un invariant este că (*) "suma de plătit şi budgetul curent trebuie să aibă aceeaşi sumă" (balanţa de plăţi este zero).

Tehnica invarianţilor este extrem de importantă pentru ingineria programării. Un tip de date abstract nu este altceva decît o serie de proceduri care menţin nişte invarianţi foarte precişi despre anumite structuri de date.

Dacă atunci cînd scriem un program scriem procedurile în aşa fel încît atunci cînd datele de intrare respectă un invariant să avem garanţia că şi datele de ieşire respectă acelaşi invariant, atunci putem obţine demonstraţii de corectitudine pentru programe. Prin inducţie matematică. (Schiţă de demonstraţie: să presupunem că datele de intrare respectă un invariant. Dacă toate procedurile chemate păstrează acest invariant, atunci prin inducţie după numărul de proceduri chemate demonstrăm imediat că invariantul este întotdeauna adevărat. Deci este adevărat şi atunci cînd programul se termină.) De altfel tehnica invarianţilor este cheia demonstraţiilor de corectitudine din metoda post-condiţiilor a lui Hoare. Dar asta este o cu totul altă mîncare de peşte; am vrut doar să arăt că paradigma este extrem de puternică.

Ei bine, tranzacţiile sunt una din metodele pentru a programa uşor cu invarianţi. Luînd din nou programul din primul exemplu, care plătea salariile, este evident că invariantul (*) nu este adevărat în interiorul buclei, unde variabilă `dePlatit` are o valoare nenulă, dar budgetul a rămas încă neschimbat. Invariantul (*) este adevărat înainte şi după textul prezentat. De aceea dacă "îmbrăcăm" programul într-o tranzacţie obţinem invariantul adevărat întotdeauna, pentru că pentru universul exterior nu mai există de fapt o buclă `for` cu mii de instrucţiuni, ci o singură operaţie atomică.

### 3.4 Durabilitatea

Ultima proprietate a tranzacţiilor este legată de programarea cu structuri de date persistente (lucru adevărat şi pentru atomicitate, descrisă mai sus). Un program Pascal simplu se execută de fiecare dată la fel, plecînd de la aceleaşi valori iniţiale. Dar un program care operează cu un fişier, sau o bază de date, are structuri de date care se află pe disc şi care supravieţuiesc "morţii" programului. Operaţia de creştere a salariului va avea, dacă este executată de două ori, rezultate diferite, pentru că a doua creştere se aplică peste prima, pentru că rezultatele primeia au fost salvate pe disc. Noţiunea de persistenţă este prezentă explicit în puţine limbaje de programare, dar este crucială în baze de date, unde în mod implicit orice operaţie se efectuează asupra unor date "permanente".

Durabilitatea tocmai la asta se referă: atunci cînd o tranzacţie se termină rezultatele ei trebuie să fie permanente, chiar în cazul unor catastrofe (ex. căderea curentului). Atunci cînd cumpăraţi un bilet de avion, se execută o tranzacţie care se termină cu tipărirea biletului. Cum ar fi dacă biletul ar fi tipărit, dar datele despre vînzare nu ar fi încă pe disc şi curentul ar cădea? La reluarea execuţiei pentru baza de date biletul ar fi ca ne-vîndut, deci s-ar putea să vă treziţi doi pe un loc. Aşa ceva este intolerabil (iar într-un sistem cu mii de terminale şi o reţea globală, cum au companiile de aviaţie, malfuncţiile hardware sunt statistic frecvente şi normale), aşa că tranzacţiile trebuie să ofere şi durabilitate.

Pe scurt iată proprietăţile tranzacţiilor:

| Nume | Proprietate |
|---|---|
| Atomicitate | Catastrofele nu pot întrerupe o tranzacţie "la mijloc" |
| Consistenţă | Tranzacţiile păstrează structurile de date corecte |
| Independenţă | Programe executate în paralel nu interferă |
| Durabilitate | Rezultatul unei tranzacţii este permanent |

## 4 Operaţiile tranzacţiilor

Să vedem cum se manifestă tranzacţiile pentru un programator. Operaţiile esenţiale elementare (atomice) pe care programatorul le are la dispoziţie atunci cînd operează asupra unei baze de date sunt citirile şi scrierile unei valori din baza de date, precum şi felurite operaţii aritmetice. Tranzacţiile se manifestă prin 3 noi instrucţiuni la dispoziţia programatorului:

| Operaţie | Semnificaţie |
|---|---|
| Begin | Indică începutul unei tranzacţii |
| End | Indică sfîrşitul tranzacţiei |
| Abort | Sfîrşit nereuşit de tranzacţie; modificările dispar |
| Read | Citeşte valoarea unei variabile persistente |
| Write | Scrie valoarea unei variabile persistente |

Vom folosi în mod explicit `Write` şi `Read` pentru a indica operaţiile asupra valorilor persistente; pe lîngă valori persistente (din baza de date) mai avem de-a face în exemple şi cu valori locale fiecărui proces (ca variabilele locale unei funcţii din limbajele de programare).

Unul din exemplele de mai sus va fi scris deci astfel într-un sistem care oferă tranzacţii:

```
Begin Transaction
for i:=1 to angajati do begin
```

```
        x := Read(salariu[i]); { x e o variabila locala, nepersistenta }
        x := x * 110/100; { marire 10 la suta }
        Write(x, salariu[i]);
End Transaction
```

Tot ceea ce se află între `Begin Transaction` şi `End Transaction` se va comporta ca o singură instrucţiune (lungă). Modificările tuturor salariilor vor deveni în mod normal (dacă nu se produce vreo eroare) vizibile instantaneu la execuţia comenzii `End Transaction`. Sfîrşitul cu succes al unei tranzacţii se numeşte *Commit*.

Instrucţiunea `Abort` este utilă atunci cînd ceva neplăcut s-a întîmplat şi tranzacţia nu se poate termina cu succes. Toate modificările făcute asupra datelor persistente, de la execuţia lui `Begin` sunt pierdute pentru eternitate atunci cînd se execută `Abort`. Tranzacţia se consideră terminată.

Iată un exemplu de utilizare a tranzacţiilor în cadrul sistemelor de operare în care instrucţiunea `Abort` este utilizată: codul ipotetic al operaţiei `move`, care mută un fişier dintr-un director într-altul şi eventual îi schimbă numele:

```
procedure move(numeVechi, numeNou)
begin
        Begin Transaction;
        directorNou := extrageDirector(numeNou);
        fisierNou := extrageNumeFisier(numeNou);
        este := cauta(directorNou, fisierNou);
        if (este) then Abort;  { fisierul nou exista deja }

        directorVechi := extrageDirector(numeVechi);
        fisierVechi := extrageNumeFisier(numeVechi);
        este := cauta(directorVechi, fisierVechi);
        if (not este) then Abort; { fisierul vechi nu exista }

        continutFisier := fisier(numeVechi);
        eroare := sterge(directorVechi, fisierVechi);
        if (eroare) then Abort;

        eroare := creaza(directorNou, fisierNou);
        if (eroare) then Abort;
        fisier(numeNou) := continutFisier;
        End Transaction;
end;
```

(Funcţiile `cauta`, `sterge`, etc. operează tot cu structuri persistente, deci sunt asemănătoare cu `Read` şi `Write` de mai sus, numai că sunt puţin mai complicate; ele pot fi sintetizate din mai multe operaţii de acest fel.)

Ideea este că pe disc operaţiile atomice sunt foarte simple: ştergerea unui nume dintr-un director, crearea unui nume într-un alt director, asocierea unui nume cu un conţinut. Ca procedura `move` să aibă succes toate operaţiile componente trebuie să se efectueze cu succes; dacă nu *nici una* nu trebuie să se efectueze: cum ar fi ca fişierul să fie şters din directorul vechi dar să nu apară în cel nou pentru că utilizatorul, de pildă, nu are drept de scriere în directorul cel nou?

Este însă foarte greu să ne asigurăm că toate condiţiile sunt adevărate simultan, mai ales în prezenţa activităţii concurente. Dacă mai multe procese încearcă simultan să facă `move` la un acelaşi rezultat, atunci degeaba unul din ele verifică înainte de mutare dacă rezultatul există, pentru că rezultatul ar putea să apară în timp ce el şterge numele vechi.

Cu tranzacţiile problema se rezolvă automat: dacă toate condiţiile au fost îndeplinite `End` face modificările dintr-odată; altfel nici o modificare nu este vizibilă, şi `Abort` este executat. Este o diferenţă între `End Transaction` şi *Commit*: instrucţiunea `End` anunţă sfîrşitul tranzacţiei şi *încercarea* de a face modificările permanente. Faptul că rezultatele operaţiilor devin permanente se numeşte *Commit*. Un `End` însă poate să nu reuşească să facă `Commit`, şi atunci se va transforma în `Abort`.

Deci `End` nu înseamnă "succes", ci "aş vrea să termin cu succes". Dacă este posibil `End` rezultă în *Commit* (care înseamnă într-adevăr "succes"), altfel `End` devine un `Abort`.

### 4.1 Sinucidere şi crimă

Trebuie spus că există două feluri în care o tranzacţie se poate termina cu eşec:

- Cînd programul execută instrucţiunea `Abort` tranzacţia se "sinucide";
- Cînd sistemul care implementează tranzacţii observă că nu poate oferi garanţiile promise, el poate ucide o tranzacţie spontan. Aceasta este o crimă.

Vom vedea mai jos exemple de contexte în care sistemul ar avea nevoie să "ucidă" tranzacţii. Ideea este că promisiunile făcute sunt mai importante decît execuţia codului; codul poate fi eventual re-executat, dar promisiunile nu pot fi în nici un caz încălcate. În general atunci cînd o tranzacţie este ucisă programul care a invocat-o (sau utilizatorul) este informat de acest lucru şi poate decide dacă să încerce din nou execuţia sau nu, în funcţie de motive.

## 5 La ce sunt utile tranzacţiile?

Deja ne-am făcut o idee despre utilitatea tranzacţiilor. Să recapitulăm:

- Programare uşoară în prezentă concurenţei: dacă vrem să respectăm invarianţi în prezenţa unor procese concurente, atunci închidem porţiunile de cod care "strică" invarianţii în tranzacţii şi alte programe vor vedea numai stări consistente;
- Tratament al malfuncţiilor hardware/software (crashes): nu mai trebuie să ne facem griji ce se întîmplă dacă pică un calculator; chiar dacă se vor pierde date, calculatorul va putea funcţiona corect, pentru că datele rămase sunt consistente. De asemenea, dacă ni s-a promis că ceva s-a executat, rezultatul a devenit permanent în pofida catastrofelor;
- Tratament simplificat al erorilor: dacă survine o condiţie de eroare programatorul poate executa un simplu `Abort` pentru a reveni la starea iniţială corectă, fără a trebui să refacă "manual" toate valorile modificate.

# 6 Serializabilitate

Chiar dacă intuitiv noţiunea de independenţă poate părea clară, trebuie să înţelegem foarte bine ce înseamnă. Această secţiune este consacrată numai acestui scop. Pentru că independenţa nu are sens în absenţa paralelismului, discutăm despre un sistem în care se pot executa simultan mai multe procese cu tranzacţii.

Vom defini o proprietate mai tare decît cea de independenţă, numită *serializabilitate* (serializability). Serializabilitatea este o proprietate e *executării* tranzacţiilor care implică independenţă. Vom vedea deci că dacă sistemul run-time care implementează tranzacţiile garantează că orice execuţie a unui set de tranzacţii este serializabilă, atunci tranzacţiile sunt independente.

## 6.1 *Schedule*

Cum am văzut, fiecare tranzacţie este în realitate compusă din mai multe instrucţiuni, pe care le putem considera atomice. Atunci cînd două tranzacţii se execută "simultan", de fapt avem de-a face cu o execuţie *întreţesută* a instrucţiunilor care le compun. O ordine de execuţie a instrucţiunilor din mai multe tranzacţii se numeşte *schedule* (nu am nici o traducere rezonabilă pentru acest termen; numele românesc cel mai apropiat ar fi "planificare").

Să vedem un exemplu foarte simplu; să notăm tranzacţiile cu T1, T2, etc. Iată două posibile tranzacţii care citesc variabila persistentă `salariu` într-o variabilă locală, o modifică şi apoi o pun la loc:

```
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;
```

Aceste tranzacţii conţin fiecare cîte 3 instrucţiuni; există foarte multe "schedules" posibile pentru executarea lor. Iată unul dintre ele:

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

Există două "schedules" speciale: cel în care toate operaţiile lui T1 se execută înainte de începutul lui T2, şi cel în care toate se execută după sfîrşitul lui T2. Aceste două "schedules" se numesc *seriale*, pentru că sunt formate din executarea "în serie" a unei tranzacţii şi apoi a celeilalte.

Toate celelalte "schedules" constau din amestecări ale instrucţiunilor lui T1 cu T2, ca în exemplul de mai sus. Un "schedule" se numeşte *serializabil* dacă produce *exact aceleaşi rezultate finale* ca unul dintre "schedule"-le seriale. Citiţi cu atenţie definiţia asta. Rezultat final înseamnă exact aceleaşi valori ale variabilelor persistente.

Ce putem spune despre "schedule"-ul din exemplul de mai sus? Este sau nu serializabil? Nu este! Să vedem un exemplu de date pentru care asta e evident: dacă iniţial `salariu` avea valoarea 100, atunci schedule-ul de mai sus va da valoarea finală 300 (verificaţi). Pe de altă parte execuţia T1 urmat de T2 dă valoarea 310, iar T2 urmat de T1 dă valoarea 330.

> Sistemul care implementează tranzacţii trebuie să excludă posibilitatea executării instrucţiunilor unor tranzacţii într-o ordine (schedule) care nu este serializabilă.

## 6.2 Graful dependenţelor

Să încercăm să ne dăm seama pentru care motive "schedule"-ul de mai sus nu este serializabil. Putem să ne gîndim la serializabilitate şi în următorul fel: pentru anumite operaţii ordinea de executare nu contează. De exemplu dacă citim o valoare în două tranzacţii nu contează în ce ordine citim, pentru că vom obţine acelaşi rezultat. Pe de altă dacă scriem două valori într-o singură variabilă persistentă (cum se întîmplă mai sus, cu tranzacţiile T1 şi T2), atunci ordinea scrierilor este extrem de importantă pentru rezultatul final, pentru că ultima scriere dă valoarea rezultatului.

Operaţiile care se pot schimba între ele fără a afecta valoarea rezultatului se spune că *comută*. Toate citirile comută între ele, aşa cum comută operaţii de orice fel asupra variabilelor locale şi operaţii care se efectuează asupra unor variabile persistente *diferite*. Cu alte cuvinte dacă eu scriu în variabila $a$ iar tu citeşti $b$ nu contează prea tare în ce ordine facem asta, pentru că rezultatul va fi acelaşi.

Un criteriu de serializabilitate este următorul: dacă putem schimba într-un "schedule" între ele instrucţiuni care comută şi obţinem un "schedule" serial, atunci avem de-a face cu un "schedule" serializabil. Altfel nu.

Hai să vedem care instrucţiuni din T1 şi T2 nu comută în schedule-ul dat ca exemplu. "T1 3)" nu comută cu "T2 1)", pentru că "T1 3)" scrie în `salariu` iar "T2 1)" citeşte din `salariu`. De asemenea, "T1 3)" nu comută cu "T2 3)", pentru că amîndouă scriu în aceeaşi valoare. Păi atunci e clar de ce acest schedule nu e serializabil: instrucţiunea "T1 3)" nu poate fi nici "împinsă" înainte de T2 nici după, din cauza acestor dependenţe.

Formal putem exprima acest lucru printr-un *graf de dependenţe* care caracterizează un *schedule*. În graful (orientat) al dependenţelor nodurile sunt tranzacţii. Între două tranzacţii avem un arc dacă prima execută o operaţie care nu comută cu o operaţie executată ulterior de a doua. Testul de serializabilitate atunci devine foarte simplu:

> Un "schedule" este serializabil dacă în graful dependenţelor asociat lui nu există cicluri.

Verificarea prezenţei ciclurilor într-un graf se poate face în timp linear în numărul de noduri. Mai mult, o sortare topologică a grafului ne indică ordinea serială echivalentă cu cea dată.

Graful dependenţelor pentru "schedule"-ul de mai sus este următorul:

```
   T2 scrie salariu dupa ce T1 scrie
/---->---\
|        |
|        V
T1       T2
^        |
|        |
\---<----/
   T1 scrie salariu dupa ce T2 citeste
```

Avem deci un algoritm posibil pentru a implementa tranzacţii izolate: le lăsăm să se execute pînă cînd una vrea să se termine; atunci verificăm dacă graful asociat execuţiei are cicluri, şi dacă are atunci executăm `Abort` pentru (cel puţin) una din tranzacţii.

În secţiunea următoare intrăm în zona *implementării* tranzacţiilor. Vom vedea cum se implementează partea care oferă proprietatea de "Independenţă". Cum se implementează celelalte proprietăţi (durabilitate, etc.) mai tîrziu.

## 6.3 Controlul accesului concurent

Există două tipuri mari de implementări de sisteme tranzacţionale din punct de vedere al serializabilităţii:

- Sisteme cu control "optimist" asupra concurenţei: acestea afirmă că avem rar conflicte, deci e mai ieftin să executăm tranzacţiile şi la sfîrşit să facem `Abort` dacă execuţia nu a fost serială; atunci fiecare operaţie se face rapid;
- Sisteme cu control "pesimist", care îşi iau toate precauţiile de la început, în aşa fel încît pur şi simplu execuţii ne-serializabile sunt imposibile.

Secţiunile următoare dau exemple din fiecare tip.

### 6.3.1 Încuietori

Încuietorile (*locks*) sunt o metodă pentru controlul pesimist al accesului. Ideea este că atunci cînd faci operaţii asupra unei valori interzici accesele altor tranzacţii la acea valoare; ştim că accesele la valori diferite sunt serializabile.

**Tipuri de încuietori** Pentru a nu limita excesiv activitatea concurentă în sistem de obicei se folosesc mai multe tipuri de încuietori. Există variante extrem de sofisticate, dar noi vom discuta doar una, în care există încuietori separate pentru citire şi scriere. O încuietoare pentru scriere este mai "exclusivă" decît una pentru citire, pentru că putem executa mai multe citiri simultan, dar o scriere nu poate fi simultană cu o altă operaţie din altă tranzacţie.

Manipularea încuietorilor se poate face fie explicit de către programator (cînd e nevoie de control fin) fie în ascuns de către sistemul tranzacţional. În orice caz, încuietorile sunt asociate valorilor persistente şi se manipulează cu următoarele operaţii (primele două sunt fundamentale, iar următoarele auxiliare):

| Operaţie | Semnificaţie |
|---|---|
| lock(date, operaţie) | "încuie" aceste date pentru "operaţie" |
| unlock(date) | eliberează restricţiile asupra acestor date |
| upgrade(date) | întăreşte o încuietoare |
| degrade(date) | slăbeşte o încuietoare |

Cum funcţionează încuietorile? De obicei există o entitate (proces) specială numit "lock manager" care ţine minte care din date sunt încuiate, de cine şi în ce fel. Atunci cînd o tranzacţie vrea să încuie o valoare se petrec următoarele lucruri:

- Dacă datele nu sunt încuiate atunci încuietoarea este pusă şi execuţia continuă în mod normal;
- Dacă pe date există deja o încuietoare:
  - Cînd încuietoarea precedentă era pentru citire iar cea cerută este tot pentru citire o a doua încuietoare este pusă, după care execuţia continuă iar accesul la date este permis;
  - Cînd încuietoarea precedentă indică un conflict cu cea curentă atunci cererea de încuiere este refuzată.

Există două acţiuni posibile cînd încuierea este refuzată; alegerea uneia dintre ele poate fi o politică a celui care a implementat baza de date, sau poate fi la alegerea utilizatorului:

- Cel care cere o încuiere imposibilă este pur şi simplu blocat din execuţie pînă cînd valoarea se "descuie", iar atunci execuţia este automat continuată;
- Cel care cere o încuiere imposibilă primeşte un cod de eroare şi decide ce trebuie să facă (de pildă să `Abort`-eze tranzacţia curentă).

Terminarea unei tranzacţii (cu *Commit* sau `Abort`) eliberează întotdeauna toate încuietorile.

O tranzacţie este *obligată* să încuie datele înainte 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 în două faze** E interesant că doar punînd încuietori în jurul acceselor la date nu e suficient. Ca să ne convingem, consideraţi tranzacţiile T1 şi T2 în forma în care exact fiecare acces este protejat, ca în exemplul următor 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 uşor de observat că o astfel de încuiere permite execuţii neserializabile (chiar execuţia indicată mai sus este posibilă). Nu ajunge deci să încuiem, trebuie să respectăm un *protocol* de încuiere. Adică un set de reguli pentru toată lumea.

Cel mai simplu şi mai des folosit protocol este cel în două faze (two-phase locking; 2PL). Acest protocol se poate enunţa astfel:

> În prima fază se pot numai obţine încuietori (adică se execută instrucţiuni `lock` şi `upgrade`).
>
> În a doua fază se pot numai dealoca încuietori (`unlock` sau `degrade`).

Dacă facem un grafic al numărului de încuietori posedate de o tranzacţie, el trebuie să arate cam ca în figura următoare: să aibă o fază de creştere (growing phase) şi una de descreştere (shrinking phase).

```
                   growing  shrinking
                  ^        ----
                  |       /    \
numar de          |      /      \____
incuietori        |  __/             |
                  | /                 \
                  0----------------------->
                                 timp
```

Acest protocol *garantează* serializabilitatea. Este un exerciţiu interesant să demonstrăm acest lucru. Iată o schiţă a demonstraţiei.

Să presupunem (prin absurd) că o serie de tranzacţii T1, T2, ... Tn, care respectă protocolul 2PL, au avut o execuţie ne-serializabilă. Asta înseamnă că în graful lor de dependenţe există un ciclu, T1 $\rightarrow$ T2 $\rightarrow$ ... Tn $\rightarrow$ T1. Ce înseamnă că avem un arc de la T1 la T2? Înseamnă că T1 a operat asupra unei valori înainte de T2, cu o operaţie care nu comută cu cea a lui T2. Dar noi ştim că T1 nu are voie să opereze asupra unei valori ne-încuiate. Asta înseamnă că T2 a încuiat acea valoare *după* ce T1 a descuiat-o. Arcul T2 $\rightarrow$ T3 indică un lucru asemănător, şi anume că T3 a încuiat o valoare (nu necesar aceeaşi) *după* ce T2 a descuiat-o. Din aproape în aproape obţinem că, în fine, T1 a încuiat o valoare după ce T1 a descuiat o altă valoare, ceea ce este absurd. Concluzia iniţială era deci greşită. În concluzie 2PL garantează serializabilitate.

Să observăm ca 2PL *nu* este acelaşi lucru cu serializabilitatea, ci că 2PL doar o *implică*.

**2PL strict** Există un dezavantaj al lui 2PL şi o implementare care îl evită. Să considerăm următorul scenariu: o tranzacţie T1 încuie o valoare $x$, o modifică şi apoi o descuie. T2 vine la rînd, încuie şi citeşte $x$. Dacă acum T1 vrea să execute `Abort`, valoarea lui $x$ trebuie pusă cum era înainte ca T1 să o fi modificat. Dar T2 a citit-o deja! Asta înseamnă nimic altceva decît că dacă

T1 execută `Abort`, T2 trebuie să fie "ucisă" de sistem, pentru a respecta proprietatea de izolare! E clar că lanţul poate fi mai lung: T3 poate a citit la rîndul ei $x$, şi atunci trebuie ucisă şi ea.

O astfel de situaţie foarte neplăcută (pentru că o mulţime de operaţii trebuie "şterse") se numeşte *cascaded abort* (abort în cascadă). (O altă consecinţă neplăcută este că T2 nu se poate termina înainte de T1, pentru că dacă T2 face *Commit* iar T1 `Abort` am încurcat-o, căci T2 a promis că valoarea lui $x$ este permanentă, dar nu avea voie s-o citească! Deci T2 trebuie să aştepte ca T1 să se termine.)

Soluţia este simplă: restrîngi concurenţa, dar nu laşi pe nimeni să vadă ce-ai modificat. Nu descui nimic pînă la sfîrşit, cînd descui totul dintr-o mişcare (de exemplu folosind `End Transaction`, care eliberează încuietorile). Graficul ar arăta atunci cam aşa:

```
                       growing       shrinking
                     ^       ---------
                     |      /        |
numar de             |     /         |
incuietori           |  __/          |
                     | /             |terminare
                     0----------------------->
                                    timp
```

**Blocarea (*deadlock*)** Încuietorile sunt foarte bune şi frumoase pentru a obţine serializabilitate, dar au un foarte, foarte mare dezavantaj.... Pot duce la situaţii de blocare totală, din care nu există ieşire. Aceasta se cheamă *deadlock*. Iată un exemplu foarte simplu de "schedule" în care două tranzacţii încuie nişte valori, după care nici una nu mai poate progresa nicicum, pentru că vrea valoarea încuiată de cealaltă:

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

Problema deadlock-ului este foarte interesantă şi complicată; prea complicată pentru articolul ăsta. Să menţionăm că există tehnici standard pentru a o trata:

- Prevenirea deadlock-ului, folosind protocoale speciale pentru a pune încuietori (ex: protocoale ierarhice);
- Evitarea deadlock-ului, alocînd resurse în mod special sau folosind tehnici bazate pe timp (vom vedea exemple mai jos);
- Detectarea şi repararea deadlock-ului: algoritmi care analizează cine pe cine aşteaptă pot detecta deadlock-ul şi pot ucide una dintre tranzacţiile implicate. Mecanismul de `Abort` se va ocupa de restul.

Algoritmii pentru detectarea deadlock-ului sunt extrem de sofisticaţi în sisteme distribuite, şi de aceea de obicei evitarea este tehnica folosită.

O metodă nu prea precisă constă în a omorî orice tranzacţie care a aşteptat mai mult de un anumit timp eliberarea unei încuietori; e clar atunci că nu se pot petrece deadlock-uri, dar s-ar putea ca nevinovaţi să piară, doar pentru că stăteau la o coadă prea lungă.

**Granularitatea încuietorilor** O problemă foarte importantă, pe care am eludat-o (şi o vom eluda şi mai departe) este *granularitatea* unei încuietori: cît de mare este o "valoare" pe care o încui? Încuietori mai mari restrîng paralelismul posibil, dar reduc costul: o tranzacţie ca cea din primul exemplu, care procesează toate salariile ar prefera să pună o singură încuietoare mare decît un milion mici. Astfel putem încuia:

- O singură valoare;
- O înregistrare (într-o bază de date relaţională) (un `record`);
- O întreagă relaţie (un fişier);
- O parte din baza de date;
- Întreaga bază de date.

Încuietori de mărimi diferite ridică şi alte probleme: de exemplu trebuie să fim grijulii să nu avem o tranzacţie care încuie o valoare pentru scriere şi o alta care încuie tot fişierul care conţine valoarea, pentru că atunci avem un conflict.

### 6.3.2 *Timestamps*

O altă metodă de control al activităţii concurente este de a impune tranzacţiilor de la început o anumită ordine de acces la date. De pildă tranzacţiile sunt etichetate cu ora la care au fost lansate, şi o tranzacţie mai "bătrînă" nu este lăsată cu nici un chip să facă o operaţie asupra unei valori atinse de o tranzacţie mai "tînără". Cu alte cuvinte se alege de la început unul dintre "schedule"-urile seriale cu care va fi echivalent cel al execuţiei curente.

Metoda foloseşte cîte două "etichete temporale" (timestamps) pentru *fiecare* valoare persistentă. O etichetă (RT: read time) indică ultima tranzacţie care a citit valoarea, iar o alta (WT: write time) indică ultima tranzacţie care a scris valoarea. Notînd eticheta unei tranzacţii T1 cu T(T1), protocoalele de acces la date ale unei tranzacţii 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 lăsa o tranzacţie mai veche să facă operaţii asupra unei valori numai dacă tranzacţii mai noi nu au făcut operaţii care nu comută. Cazul scrierii este interesant: dacă o tranzacţie veche scrie peste o valoare pe care a scris între timp o tranzacţie mai nouă (dar între cele două timpuri nimeni nu a citit valoarea), atunci valoarea scrisă de tranzacţia veche este pur şi simplu ignorată. Asta pentru că dacă tranzacţia veche ar fi scris înainte valoarea, nimeni nu ar fi apucat să o citească pînă cînd cea nouă ar fi supra-scris-o.

**"Timestamps" şi "locks"** Putem combina laolaltă încuietorile cu "timestamps" pentru a evita deadlock-ul. Există două mari soluţii care folosesc "timestamps" pentru a arbitra accesul la un lock:

**wait-die:**

- Cînd o tranzacţie vrea o încuietoare ţinută de o tranzacţie mai "tînără" atunci trebuie să aştepte;
- Cînd o tranzacţie vrea o încuietoare a unei "bătrîne", tînăra este ucisă.

**wound-wait:**

- Cînd o tranzacţie vine şi vrea o încuietoare ţinută de o tranzacţie mai tînără, tînăra este ucisă;
- Tranzacţiile tinere aşteaptă încuietorile ţinute de vîrstnice.

Amîndouă schemele dau prioritate tranzacţiilor care au trăit mai mult, în ideea că au făcut mai multă treabă şi e păcat să le omorîm. Wait-die poate produce livelock, sau "starvation" (cînd tranzacţii tinere sunt mereu repornite şi nu apucă niciodată un lock). Nu se pot produce deadlock-uri, pentru că în orice ciclu tranzacţia cea mai tînără trebuie să moară.

**O metodă optimistă: algoritmul validării** O metodă care nu restrînge deloc accesul concurent este următoarea: cine are de făcut modificări strînge toate valorile de modificat în variabile locale, iar cînd a terminat verifică dacă poate să salveze modificările. Asta se poate dacă nimeni nu s-a atins între timp de valorile iniţiale. Dacă această fază (de *validare*) se termină cu succes toate valorile sunt salvate (printr-o operaţie atomică) în baza de date. Altfel tranzacţia face `Abort`.

Ca să verifice dacă nimeni nu s-a atins de valori între timp, fiecare valoare este etichetată cu un număr de *versiune*, care este incrementat la fiecare scriere. Dacă observi atunci cînd vrei să-ţi salvezi rezultatele că versiunea s-a schimbat faţă de atunci cînd ai citit datele, renunţi să le mai scrii.

Un astfel de protocol este folosit de sistemul de fişiere NFS (network file system): fiecare fişier pe server are o versiune. Cînd cineva şterge un fişier şi crează altul folosind aceleaşi porţiuni de pe disc (mai precis acelaşi inod — pentru detalii citiţi articolele mele mai vechi despre sistemul de fişiere din Unix, disponibile şi din pagina mea de web), versiunea se incrementează. Dacă un alt client vrea să facă operaţii pe un fişier care nu mai există, server-ul se prinde, pentru că clientul oferă versiunea veche a inod-ului. Astfel accese ilegale sunt interzise.

### 6.4 Izolare: recapitulare

Am văzut multe tehnici alternative pentru a implementa proprietatea de izolare a tranzacţiilor. Un anume sistem va implementa probabil una singură dintre ele.

| **Protocol** | **Calităţi** |
|---|---|
| 2PL | deadlock şi cascaded-abort posibile |
| 2PL strict | deadlock posibil; concurenţă redusă faţă de 2PL |
| Timestamp | restrînge "schedule"-le posibile; necesită spaţiu suplimentar pentru etichete; concurenţă mare |
| Validare | poate cauza abort-uri tardive; cere spaţiu suplimentar |
| Timestamp + lock | pot cauza "starvation"; nu au deadlock |

## 7 Implementarea lui `Abort` şi *Commit*

Dom'le, am tot vorbit de cum se implementează sincronizarea, şi am folosit mereu operaţiile `Abort` şi *Commit*, dar nu avem încă nici cea mai mică idee despre cum se pot implementa. Le-a venit timpul.

Există două metode mari pentru a *modifica* valorile persistente (baza de date):

1. Toate modificările se fac pe *copii* ale valorilor din baza de date (acestea se numesc "date umbră" (shadow-data)). Tranzacţia citeşte toate valorile interesante din baza de date (poate folosi fie încuietori, fie validare), după care modifică toate valorile local. Cînd a terminat cu succes, (*Commit*) trimite toate valorile la server-ul care ţine baza de date, unde ele trebuie efectuate *în mod atomic*, toate deodată (*redo*). Dacă tranzacţia decide să facă `Abort` nu trimite nici o valoare, ci doar eliberează încuietorile.

2. Tranzacţia poate alege să modifice valorile direct în baza de date. În cazul ăsta *Commit* e foarte simplă: eliberezi încuietorile. `Abort` însă are nevoie de un mecanism special, prin care variabilele trebuie readuse la valorile iniţiale (*undo*).

În fiecare caz una dintre operaţiile de terminare este mai grea decît cealaltă. Amuzant este că ambele probleme pot fi rezolvate folosind *aceeaşi* unealtă, în moduri uşor diferite. Această unealtă este setul de înregistrări, mai precis numit *log*, sau *audit*.

## 7.1 *Log*-ul

Un “log” este o secvenţă lineară, teoretic nesfîrşită, de înregistrări care descriu operaţii. Există două feluri de “log”:

1. “log”-ul de *intenţii*, sau “redo-log”: atunci cînd modificările se fac pe o copie personală a datelor, toate noile valori ale datelor sunt marcate în log. Pentru *Commit*, log-ul tranzacţiei este trimis la baza de date care re-execută (de aici numele de “redo”) toate operaţiile descrise în log, de data asta direct în baza de date, într-o manieră atomică.

2. “log”-ul “undo” ţine minte în schimb toate valorile *iniţiale* ale datelor, înainte de modificare. Dacă faci modificări marchezi în log cum erau valorile, iar dacă-ţi vine pofta să faci `Abort` atunci o iei de-a-ndoaselea în log şi refaci valorile cum erau iniţial. Această tehnică de parcurgere inversă cu des-facerea modificărilor se numeşte *Rollback*.

Vom vedea că, în mod surprinzător, log-ul se poate folosi nu numai pentru a implementa operaţiile de `Abort` şi *Commit*, ci şi pentru a obţine proprietăţile de atomicitate şi durabilitate în faţa catastrofelor!

Desigur că putem avea un log combinat, care ţine ambele feluri de valori, şi care permite atît “undo” cît şi “redo”. Iată un exemplu posibil de înregistrare dintr-un astfel de log:

| | |
|---|---|
| Tranzacţie | T232 |
| Ora | 12:01:22 EST |
| Valoare modificată | salariu |
| Valoare iniţială | 100 |
| Valoare finală | 200 |

În log trebuie marcate nu numai operaţiile asupra valorilor persistente, ci şi fiecare acţiune de început şi terminare de tranzacţie.

### 7.1.1 Puncte de control (*checkpoints*)

Problema principală cu log-ul este că acesta creşte permanent cu noi înregistrări. Niciodată nu se eliberează spaţiu. E clar, asta nu poate merge la infinit. O metodă care recuperează spaţiul inutil din log (de exemplu înregistrările tranzacţiilor care au făcut *Commit*) se bazează pe ceea ce se numeşte “puncte de control” (checkpoints). Aceasta este o formă specială de “garbage collecting”.

Să facem următoarea observaţie: dacă ştim starea iniţială a unui sistem şi toate modificările făcute asupra lui (în ordine) atunci putem deduce starea curentă. Exact asta este şi un log: lista tuturor modificărilor făcute asupra unui sistem. Dar de la un anumit rang încolo, descrierea modificărilor poate fi mai mare decît descrierea stării întregului sistem. Putem atunci “compacta” lista modificărilor ţinînd minte starea curentă a sistemului şi uitînd toate modificările făcute pînă atunci. Această “stare îngheţată” a sistemului se numeşte “checkpoint”.

Problema este mai grea decît pare: de fapt sistemul nu este niciodată într-o stare stabilă; tot timpul sunt în execuţie tranzacţii despre care nu ştim încă dacă se vor termina cu succes sau nu, deci nu putem vorbi de “o stare”. Soluţia (cu variante din ce în ce mai sofisticate şi performante) este grosso-modo următoarea: se opreşte toată activitatea din sistem, după care log-ul este parcurs în ordine inversă. Toate informaţiile despre tranzacţii care au fost terminate pot fi şterse din “log”, pentru că efectele acestora sunt deja în baza de date (proprietatea de durabilitate garantează acest lucru). În log se mai păstrează doar informaţii despre tranzacţiile curente neterminate. De obicei toate aceste lucruri se fac tot folosind log-ul: se scrie o înregistrare specială “checkpoint”, după care log-ul este scanat şi sunt copiate toate înregistrările încă “vii”. Tot spaţiul află înainte de checkpoint poate fi apoi reutilizat.

Tehnica aceasta este folosită şi în sisteme de operare, unde proceselor care rulează pentru foarte mult timp li se fac periodic astfel de puncte de control, pentru ca în cazul unei catastrofe să se reia calculele numai de la ultimul punct.

# 8 Catastrofe (crash)

Am văzut pînă acum cum se pot obţine proprietăţile de izolare (prin controlul accesului concurent) şi consistenţă (printr-un stil de programare care menţine invarianţii adevăraţi înafara tranzacţiilor).

În această secţiune vom introduce ultimul element suplimentar care complică viaţa celor ce implementează tranzacţii: catastrofele şi vom vedea cum tranzacţiile reuşesc să ofere proprietăţile de atomicitate şi durabilitate în faţa unor condiţii atît de adverse.

Scula esenţială este din nou “log”-ul (undo-redo).

Vrem să asigurăm deci două proprietăţi:

**Durabilitate:** o tranzacţie care a făcut *Commit* trebuie să aibă efectele permanente orice s-ar întîmpla.

**Atomicitate:** după un “crash” baza de date va executa un protocol special de refacere (recovery protocol) care-i va permite să reia operaţiunile într-un mod corect (adică tranzacţiile neterminate la ora crash-ului vor fi `Abort`-ate). Mai mult, un crash în timpul acţiunii de refacere trebuie să nu cauzeze daune de nici un fel!

Prima regulă pe care trebuie s-o respectăm este următoarea:

> Raportăm succesul unei tranzacţii *numai după ce* toate valorile modificate de ea au ajuns pe un mediu stabil (stable medium), care supravieţuieşte catastrofelor.

(În funcţie de importanţa sistemului, mediul stabil se poate defini ca memorie RAM, memorie NVRAM, disc, disc cu redundanţă, disc dublu (mirrored disk), bandă magnetică, bunker anti-atomic, etc. Depinde de ce fel de catastrofe vrem să ne ferim.)

Această regulă este adevărată pentru că nu există nici un *undo* pentru ceea ce i-am spus utilizatorului. Dacă am tipărit un bilet de avion atunci nu mai putem să-l luăm înapoi (cel puţin pînă învăţăm să dezvoltăm roboţi suficient de puternici…).

Astfel garantăm durabilitatea.

## 8.1 “Write-ahead log”

Pentru a garanta atomicitatea trebuie să avem o metodă de a identifica toate tranzacţiile care erau neterminate la momentul crash-ului şi de a face `Abort` cu ele. Pentru că un crash distruge absolut toate valorile variabilelor ne-persistente, toată informaţia necesară pentru identificare şi `Abort` *trebuie să fie pe un mediu stabil.* Regula de aur este:

> O operaţie poate fi efectuată în baza da date numai dacă înregistrarea din log care o descrie a ajuns pe un mediu stabil.

E uşor de înţeles de ce: dacă facem o modificare în baza de date şi aparatul crapă înainte să apucăm să scriem în log, nu avem nici o metodă după repornire să ne dăm seama că am scris într-adevăr ceva în baza de date sau nu.

### 8.2 Protocolul de refacere (recovery)

Dacă respectăm regula de aur atunci avem destulă informaţie ca să refacem baza de date după o catastrofă. Algoritmul are trei paşi mari:

1. Scanăm log-ul şi identificăm toate tranzacţiile care corespund clientului care a răposat (dacă a decedat chiar serverul considerăm toate tranzacţiile).
2. Pentru toate tranzacţiile care au o înregistrare *Commit* în log re-executăm acţiunile din log în ordinea în care au fost făcute.
3. Pentru toate tranzacţiile neterminate din log executăm acţiunile în ordine inversă, ştergînd modificările lor.

Ca treaba să meargă cum trebuie, este necesar ca înregistrările din log să satisfacă o proprietate suplimentară: acţiunile lor trebuie să fie *idempotente*. Asta înseamnă că executarea unei acţiuni din log de mai multe ori trebuie să dea exact acelaşi rezultat ca executarea o singură dată. În acest fel nu ne doare dacă cumva re-executăm acţiuni care au mai fost făcute. De asemenea, nu avem probleme dacă pică din nou curentul în timpul acţiunii de refacere: o a doua refacere va da aceleaşi rezultate.

O înregistrare de log de genul "A este incrementat cu 2" nu este idempotentă, pentru că fiecare nouă execuţie schimbă valoarea lui A. Exemplul de mai sus însă este.

Să observăm că "regula de aur" ne oferă şi cheia creşterii performanţei în implementarea tranzacţiilor: nu trebuie să scriem în log de fiecare dată cînd facem ceva: trebuie să scriem în log numai înainte de a modifica baza de date. Putem astfel strînge serii întregi de modificări în log, pe care le scriem dintr-un singur foc. Se ştie că pentru accesul la disc mai mult costă să începi decît să faci treaba, deci costul este practic acelaşi pentru una sau mai multe înregistrări. Singurul moment cînd trebuie neapărat să salvăm pe disc este exact înainte de terminarea tranzacţiei; toate celelalte modificări pot fi amînate.

## 9 Încheiere

Ar mai fi foarte multe lucruri de spus despre tranzacţii; foarte interesant este protocolul pentru tranzacţii în medii distribuite, numit "Two-phase commit protocol", 2PC. Tranzacţiile ierarhice (nested) sunt o altă paradigmă aplicată cu succes. Sisteme de fişiere bazate pe log-uri sunt folosite cu succes în sisteme Unix pentru refaceri extrem de rapide după căderi.

Tranzacţiile sunt o sculă foarte utilă pentru programator, care cu preţul unei oarecare scăderi în performanţă oferă o serie de proprietăţi foarte utile ale datelor. Este de aşteptat să devină o paradigmă din ce în ce mai întîlnită în arsenalul sculelor programatorilor.