Planificarea Tranzacțiilor
Gestionarea Concurenței

2

Concurenţa – Planificare vacanţa (studiu de caz)

- Sistem de rezervari vacanţe, 2-3 clienti cauta rezervari in perioada vacantei (o saptamana, in aprilie)
- Clientul C1 – gaseste un hotel, locuri_disponibile > 4 (operatie Read- ok)
  - Nu este hotarat, mai cauta oferte, discuta cu familia
- Clientul C2 – gaseste acelasi hotel, locuri_disponibile > 4 (operatie Read – ok)
  - Presat de timp, plateste avansul, rezerva 4 locuri
    - Sistemul actualizeaza numarul de locuri(operatie Write -ok)
    - Observatie: in acest moment Clientul C1 nu mai are informatii corecte/valide

Concurenţa – Planificare vacanţa(studiu de caz)

- Clientul C1 revine si plateste pentru 4 locuri, vrea sa le reserve

  - Sistemul actualizeaza baza de date!!!
  - Ultimele locuri (cele 4) au fost vandute de 2 ori!

Dirty reads – Aplicatie bancara

Scenariu: O bază de date bancară și două tranzacții concurente care operează asupra aceluiași cont bancar.

Situația inițială: Contul lui Alex are un sold de 1000 RON.

- Tranzacția 1 (T1) – Transfer de 500 RON
  - T1 citește soldul contului lui Alex: 1000 RON.
  - T1 scade 500 RON din sold (nou sold: 500 RON).
  - T1 nu a confirmat (commit) încă modificarea.
- Tranzacția 2 (T2) – Verificare sold pentru retragere 600RON
  - Înainte ca T1 să facă commit, T2 citește soldul contului lui Alex și vede 500 RON.
  - T2 folosește această informație pentru a decide dacă poate aproba un alt transfer sau o retragere.
  - T2 poate aproba retragerea a 600 RON

Problema dirty read

- Dacă T1 face rollback (anulează modificarea), soldul real revine la 1000 RON, dar T2 a luat deja o decizie bazată pe o informație greșită (citită înainte de commit).

Unrepetable reads – Rezervare hotel

Scenariu: sistem de rezervări hoteliere, un utilizator verifică disponibilitatea unei camere, dar între două citiri succesive, datele se modifică.

- Tranzacția 1 (T1) – Verificarea disponibilității
  - T1 citește disponibilitatea camerei #121 (Presidential Suit) → Disponibilă.
  - T1 decide să o rezerve, dar nu finalizează încă rezervarea.
- Tranzacția 2 (T2) – Alt utilizator rezervând camera
  - T2 efectuează o rezervare pentru camera #121.
  - T2 modifică statutul camerei la Ocupată.
  - T2 finalizează tranzacția cu commit.
- Tranzacția 1 (T1) – Confirmare finală
  - T1 citește din nou disponibilitatea camerei #121.
  - Acum vede că este ocupată, deși inițial era disponibilă.

Problema unrepeatable read: T1 a citit aceeași informație de două ori, dar a primit rezultate diferite!

Blinds writes– Magazin online

Scenariu: magazin online, două tranzacții modifică stocul unui produs, dar fără să ia în considerare modificările anterioare.

- Tranzacția 1 (T1) – Primirea unui nou lot de produse
  - T1 nu citește stocul actual.
  - T1 adaugă 50 unități și setează stocul la 150 unități (presupunând că inițial era 100).
  - T1 face commit.
- Tranzacția 2 (T2) – Corecție a stocului după o verificare fizică
  - T2 nu citește stocul actual.
  - T2 observă o eroare și setează manual stocul la 110 unități.
  - T2 face commit.

Problema blind writes:

Dacă T2 face commit după T1, , sistemul afișează 110 unități, noul lot de produse se pierde. Stocul corect ar fi trebuit să fie 160 unități (dacă T2 ar fi luat în considerare adaosul făcut de T1).

Solutie: interzicerea accesului concurent?

- Interzicerea accesului concurrent -> intarzieri foarte mari deoarece cererile ulterioare pot fi procesate doar dupa ce S-AU INCHEIAT cererile initiale.
  - Rezervare vacanta amanata / unii nu pot cumpara pentru ca altii nu s-au decis, pot ramane locuri libere, etc
  - Nu se pot face plati
  - Nu se pot face rezervari hotel
  - Cumparaturile online – oprite de facto

Planificarea tranzacțiilor

Planificarea tranzacțiilor (scheduling) în baze de date se referă la ordonarea și gestionarea execuției tranzacțiilor multiple pentru a asigura **consistența**, **izolarea** și **integritatea** datelor.

Scopul principal este de a evita conflictele și anomaliile care pot apărea atunci când mai multe tranzacții accesează și modifică aceleași date simultan.

Tipuri de Planificare

- **Planificare serială (Serial Schedule)**
  - Tranzacțiile sunt executate una după alta, fără suprapunere.
  - Este sigură și garantează consistența datelor, dar poate fi ineficientă din cauza lipsei de paralelism.

- **Planificare intercalată (Interleaved Schedule / Concurrent Schedule)**
  - Tranzacțiile sunt executate simultan, intercalând operațiunile lor.
  - Crește performanța sistemului, dar poate duce la conflicte (**RWconflict**, **Wwconflict, etc**).

- **Planificare serializabilă (Serializable Schedule)**
  - Este o planificare intercalată care asigură un rezultat echivalent cu o planificare serială.
  - Poate fi implementată prin metode precum **blocarea în două faze (2PL)** , **timestamp ordering, etc**.

Planificare seriala

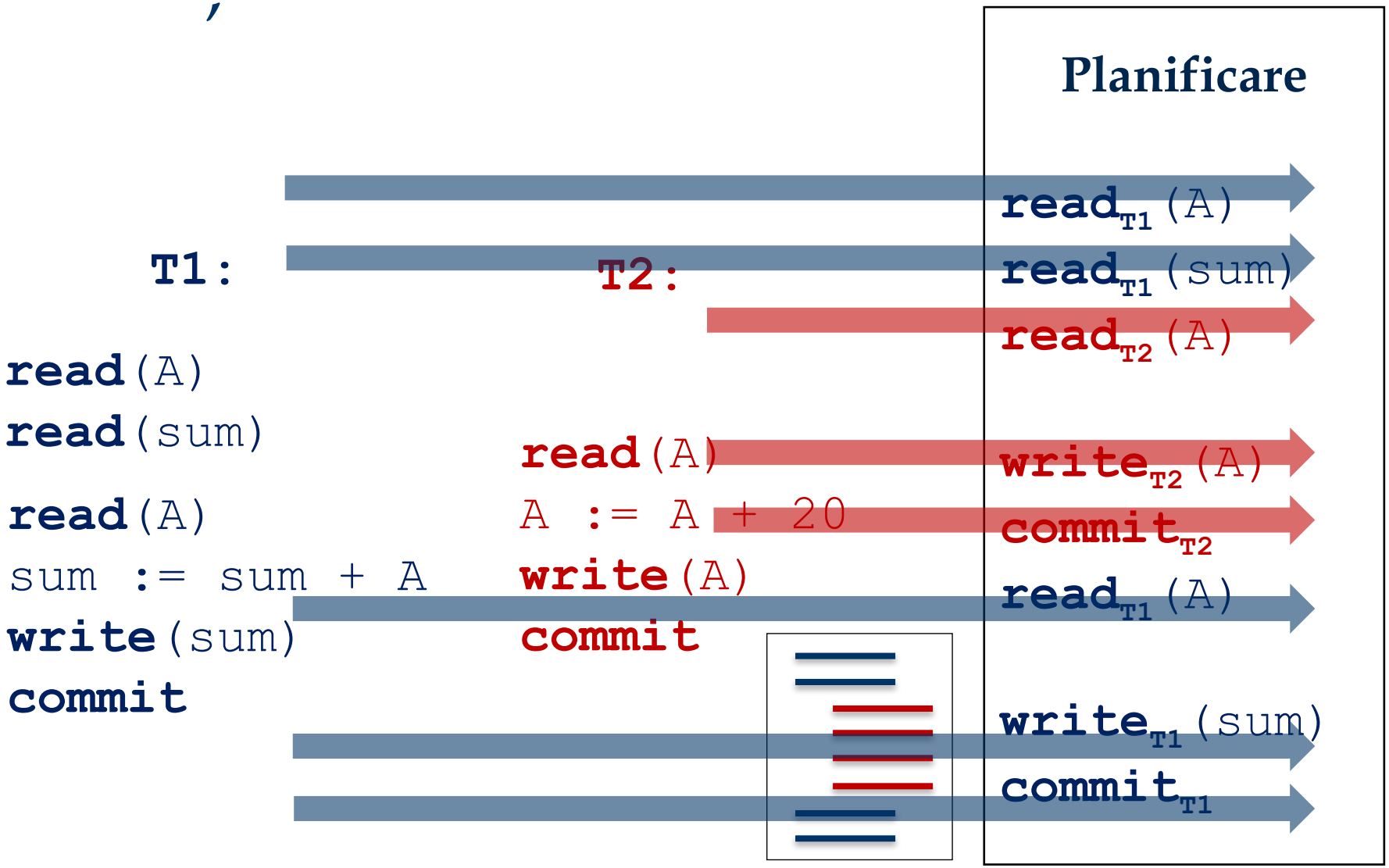
Planificarea serială este un tip de execuție a tranzacțiilor într-o bază de date, unde tranzacțiile sunt executate una după alta, fără suprapunere. Cu alte cuvinte, o tranzacție trebuie să se termine complet înainte ca următoarea să înceapă.

(*Read* / *Write* / *Abort* / *Commit*)

a *n* tranzacții

astfel încât ordinea instrucțiunilor
fiecărei tranzacții să se păstreze

Planificarea tranzacțiilor



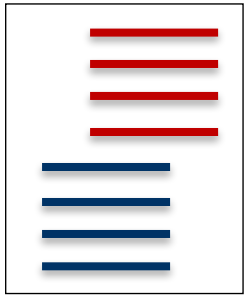
Planificarea tranzacțiilor

T1:

```
read(A)
read(sum)

read(A)
sum := sum + A
write(sum)
commit
```

T2:

```
read(A)
A := A + 20
write(A)
commit
```

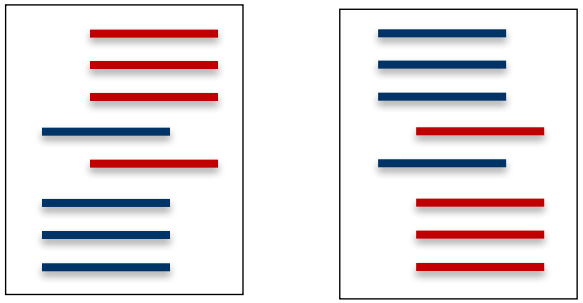


planificare serială

planificarea ce nu intercalează acțiuni ale mai multor tranzacții.

Planificarea tranzacțiilor

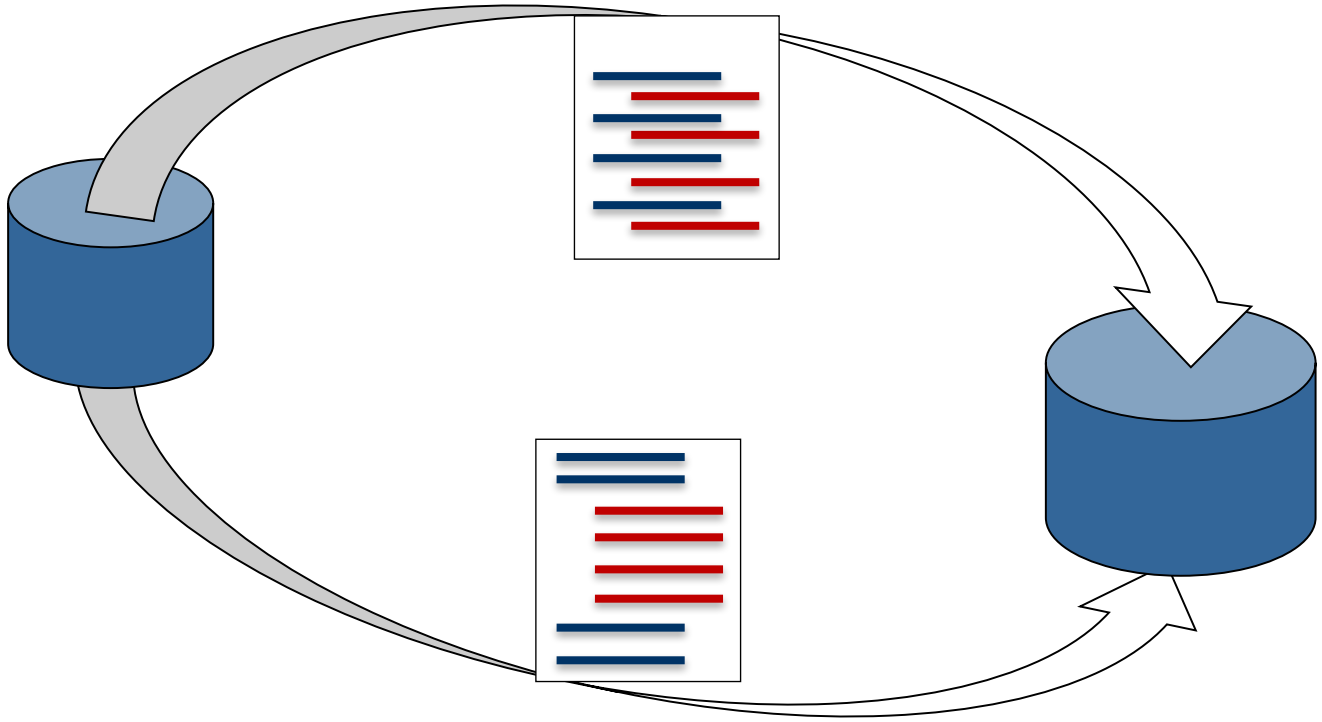
```
T1:                    T2:

                       read(A)
                       A := A + 20
                       write(A)
  read(A)
                       commit
  read(sum)
  read(A)
  sum := sum + A
  write(sum)
  commit
```



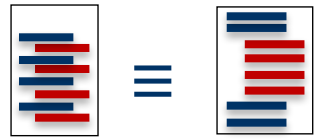
planificare non-serială

acțiunile mai multor tranzacții concurente se interpătrund.

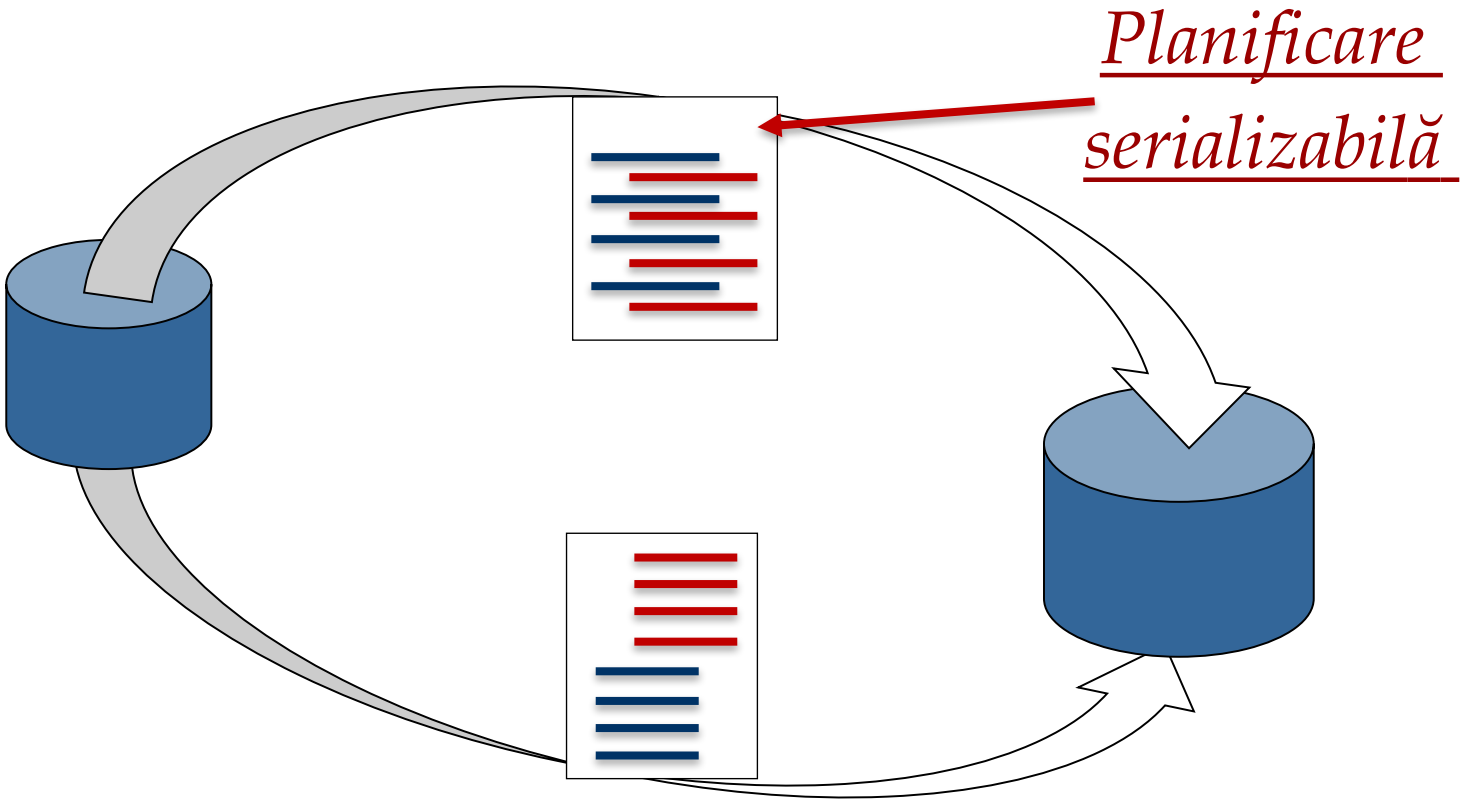
Planificarea tranzacțiilor

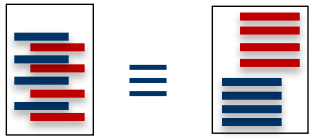


Planificări echivalente



Planificarea tranzacțiilor

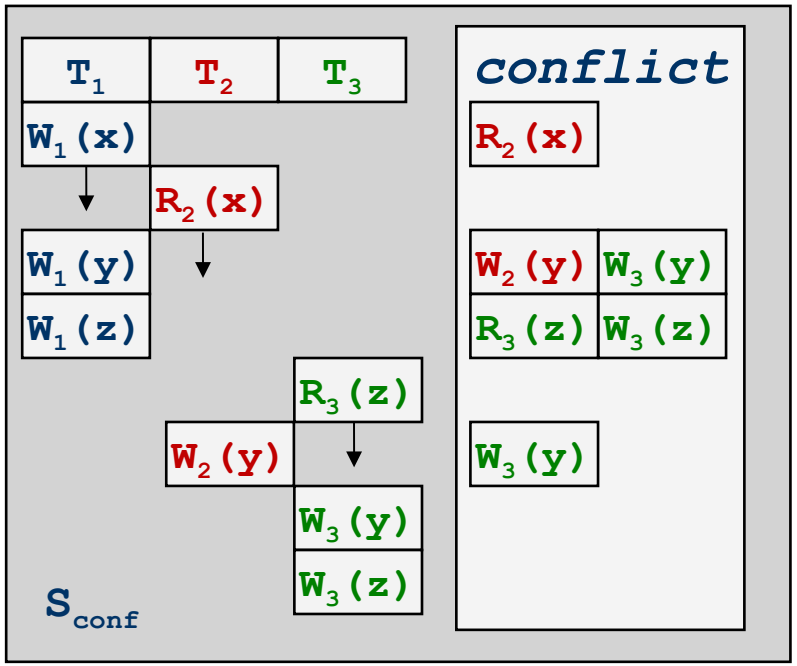




Planificarea tranzacțiilor

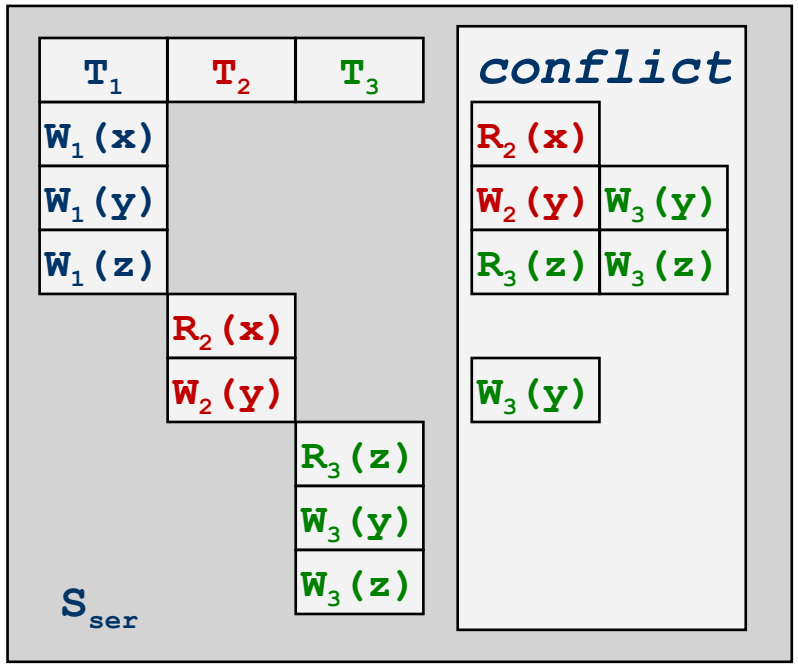
- Acțiuni ce nu se pot interschimba într-o planificare:
  - Acțiunile aparținând aceleiaşi tranzacții
  - Acțiuni aplicate de diferite tranzacții *aceluiaşi obiect,* dacă cel puțin una dintre ele este o operație de **write**. (*acțiuni conflictuale*!)

Conflict-serializabilitate



conflict-serializabil!!!

≡



Planificare seriala

- 2 planificări sunt *conflict-echivalente* dacă:
  - Implică acţiunile aceloraşi tranzacţii
  - Fiecare pereche de acţiuni conflictuale este ordonată în acelaşi mod.

- O planificare este *conflict serializabilă* dacă e conflict echivalentă cu o planificare serială

Serializabilitate

- Obiectivul *serializabilității* este găsirea unei planificări non-seriale care permite execuția concurentă a tranzactiilor fără ca acestea să interfereze, şi astfel să conducă la o stare a unei baze de date la care se poate ajunge şi printr-o execuție serială.
- Garantarea serializabilității tranzacțiilor concurente este importantă deoarece previne apariția inconsistențelor generate de interferența tranzițiilor.

Conflict-serializabilitate

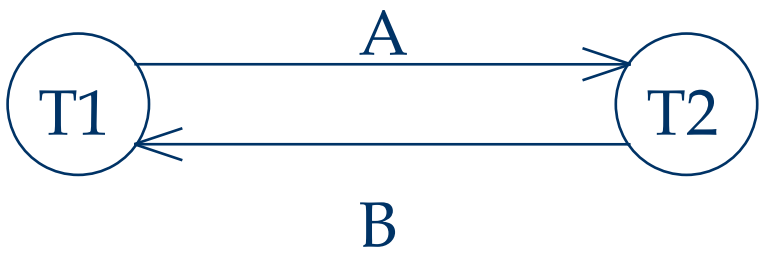
- Graf de precedență:
  - Graf orientat
  - Un nod per tranzacție
  - Arc între T_i şi T_j dacă o acțiune/operație de citire/modificare din T_j se realizează după o acțiune/operație conflictuală din T_i.

- Teoremă : O planificare este conflict- serializabilă dacă şi numai dacă graful său de precedendență nu conține circuite

Exemplu

- Planificare ce nu este conflict-serializabilă:

T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)



Graf de precedență

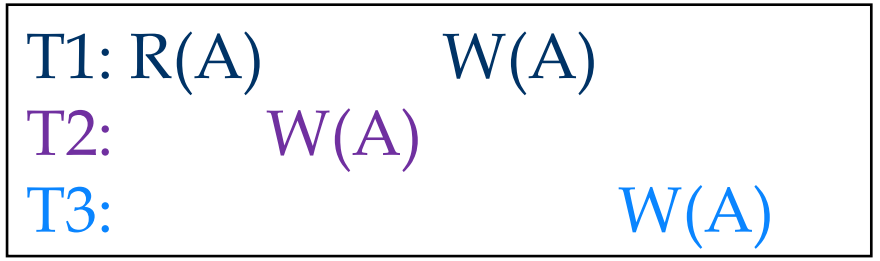
- Graful conține un circuit. Rezultatul lui T1 depinde de T2, şi invers.

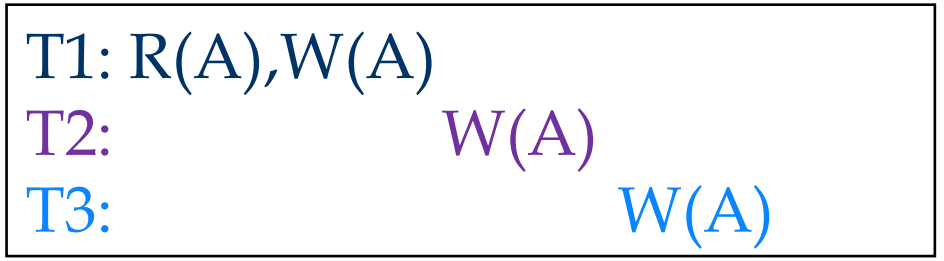
Algoritm de Testare a Conflict-Serializabilității lui **S**

1. Pentru fiecare tranzacție T_i din **S** de crează un nod etichetat T_i în graful de precedență.
2. Pentru fiecare **S** unde T_j execută un Read(x) după un Write(x) executat de T_i crează un arc (T_i,T_j) în graful de precedență
3. Pentru fiecare caz în **S** unde T_j execută un Write(x) după un Read(x) executat de T_i crează un arc (T_i,T_j) în graful de precedență
4. Pentru fiecare caz în **S** unde T_j execută un Write(x) după un Write(x) executat de T_i crează un arc (T_i,T_j) în graful de precedență
5. **S** este conflict serializabil ddacă graful de precedență nu are circuite

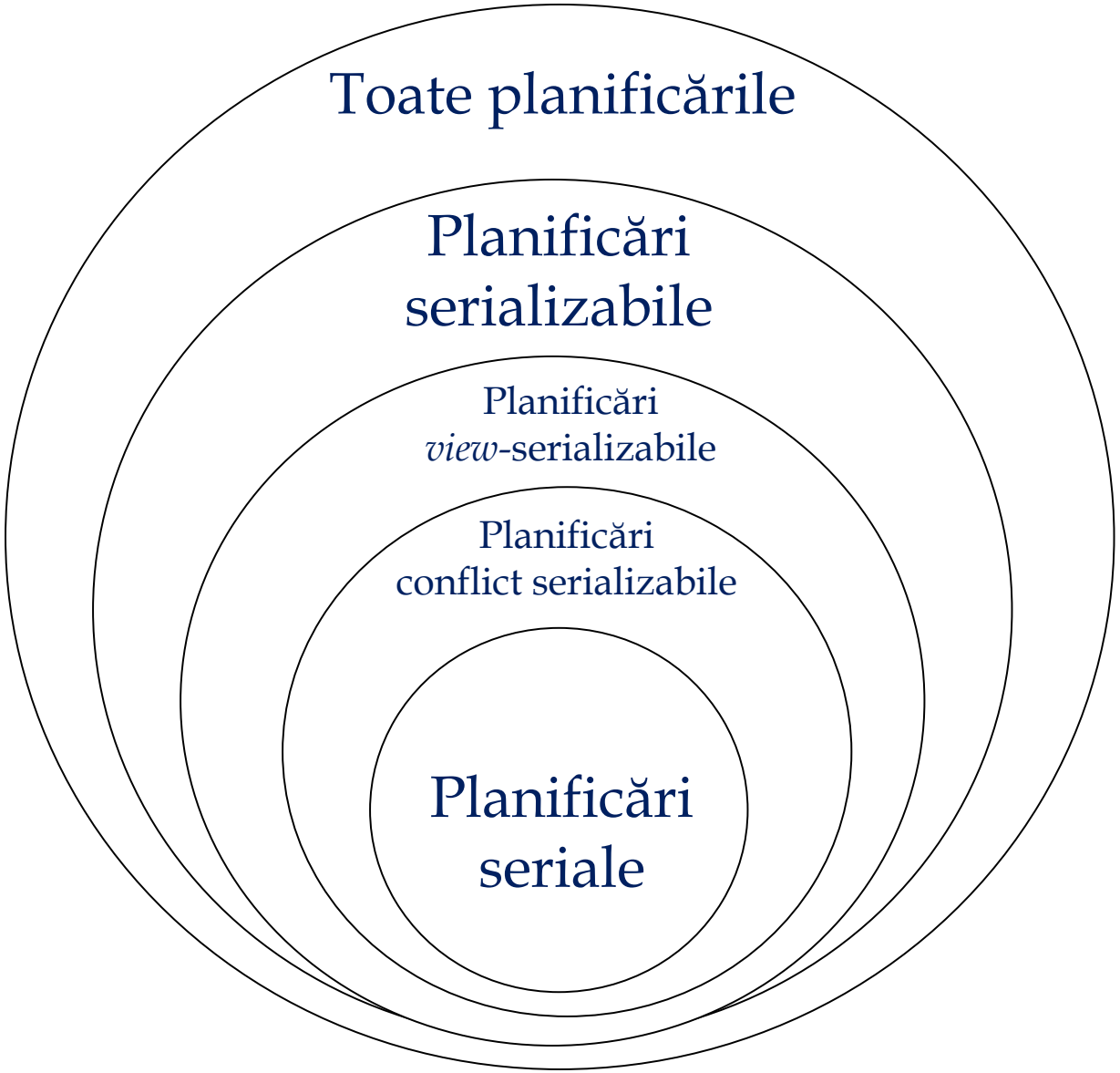
View - serializabilitatea

- Planificările S_1 şi S_2 sunt *view*-echivalente :
  - Dacă T_i citeşte valoarea inițială a lui A în S_1, atunci T_i de asemenea citeşte valoarea inițială a lui A în S_2
  - Dacă T_i citeşte valoarea lui A modificată de T_j în S_1, atunci T_i de asemenea citeşte valoarea lui A modificată de T_j în S_2
  - Dacă T_i modifică valoarea finală a lui A în S_1, atunci T_i de asemenea modifică valoarea finală a lui A în S_2





Planificarea tranzacțiilor



Planificare serializabilă

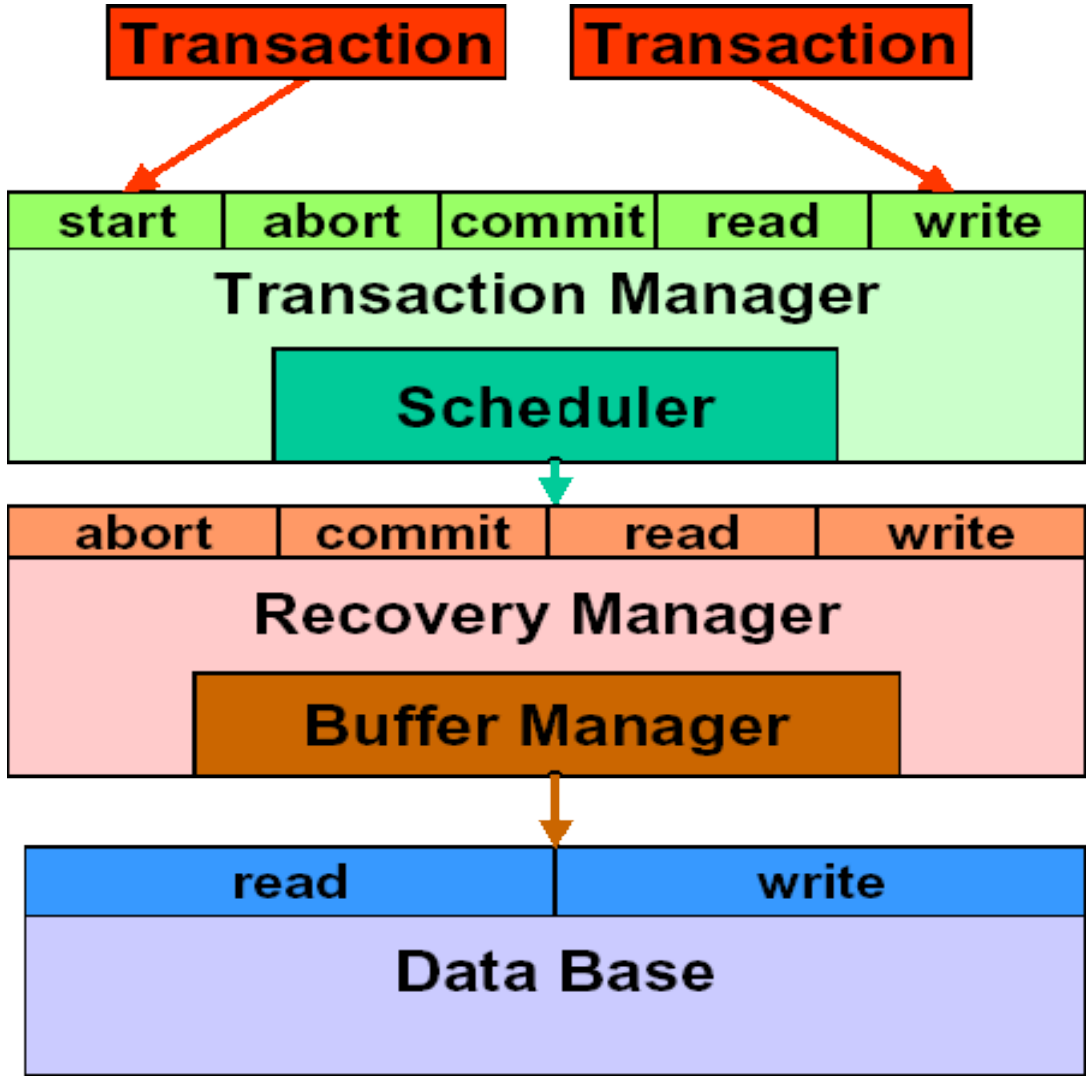
... dar care nu este nici *conflict-serializabilă* nici *view-serializabilă*

T1:	T2:
read(A)	
A := A-50	
write(A)	
	read(B)
	B := B - 10
	write(B)
read(B)	
B := B + 50	
write(B)	
	read(A)
	A := A + 10
	write(A)

Serializabilitate în practică

- In practică, un SGDB nu testează serializabilitea unei planificări date. Acest lucru nu este practic deoarece intercalarea operațiilor mai multor tranzacții concurente poate fi dictată de OS şi prin urmare este dificil de impus.
- Abordarea DBMS esta să folosească protocoale specifice care sunt cunoscute că generează planificări serializabile.
- Aceste protocoale pot afecta gradul de concurență, însă elimină cazurile conflictuale.

Execuția tranzacțiilor



Anomalii ale execuției concurente

- *Reading Uncommitted Data* (conflict WR, "*dirty reads*"):

T1: R(A), W(A), R(B), W(B), **A**
T2: R(A), W(A), **C**

- *Unrepeatable Reads* (conflict RW):

T1: R(A), R(A), W(A), **C**
T2: R(A), W(A), **C**

- *Overwriting Uncommitted Data* (Conflict WW, "*blind writes*"):

T1: W(A), W(B), **C**
T2: W(A), W(B), **C**

Phantom Reads

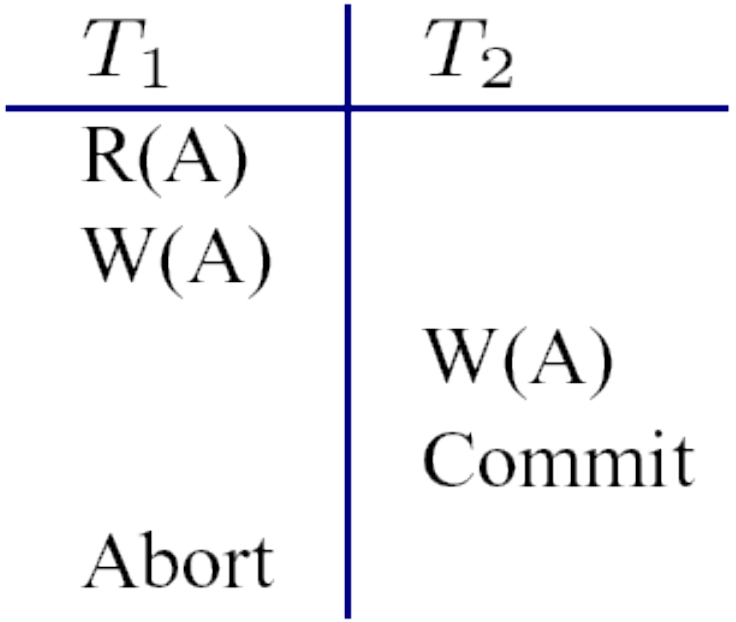
- O tranzacție re-execută o interogare şi găseşte că o altă tranzacție comisă a inserat înregistrări adiționale ce satisfac condițiile interogării
  - Dacă înregistrările au fost modificate sau şterse, este vorba de conflictul *unrepeatable read*

Exemplu:

- T_1 execută `select * from Students where age < 25`
- T_2 execută `insert into Students values(12,'Jim', 23, 7)`
- T_2 se comite
- T_1 execută `select * from Students where age < 25`

Planificări recuperabile

- Într-o *planificare recuperabilă* tranzacțiile pot **doar citi** date care a fost **deja comise**
- Există posibilitatea apariției **blind write**



- Care ar trebui sa fie valoarea lui A dupa **abort**??

Planificare recuperabilă

- O planificare este recuperabilă dacă pentru oricare tranzacție T comisă, comiterea lui T se efectuează după comiterea tuturor tranzacțiilor de la care T a citit un element.

Next:

- Controlul concurenței bazat pe blocări:
  - Two-Phase locking
  - Strict Two-Phase Locking
- Deadlocks
  - Prevenire (Wait-Die, Wound-Wait)
- Nivele de Izolare
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

Controlul concurenței bazat pe blocări

Blocare: O metodă utilizată pentru controlul accesului concurent la date.

Atunci când o tranzacție accesează un obiect al bazei de date, blocarea poate proteja obiectul respectiv de a fi accesat de o altă tranzacție pentru a preveni obținerea de rezultate incorecte.

Controlul concurenței bazat pe blocări

- **Blocare partajată** (*shared* sau *read lock*): Dacă o tranzacție blochează un obiect in mod partajat, ea poate citi acel obiect dar nu il poate modifica.

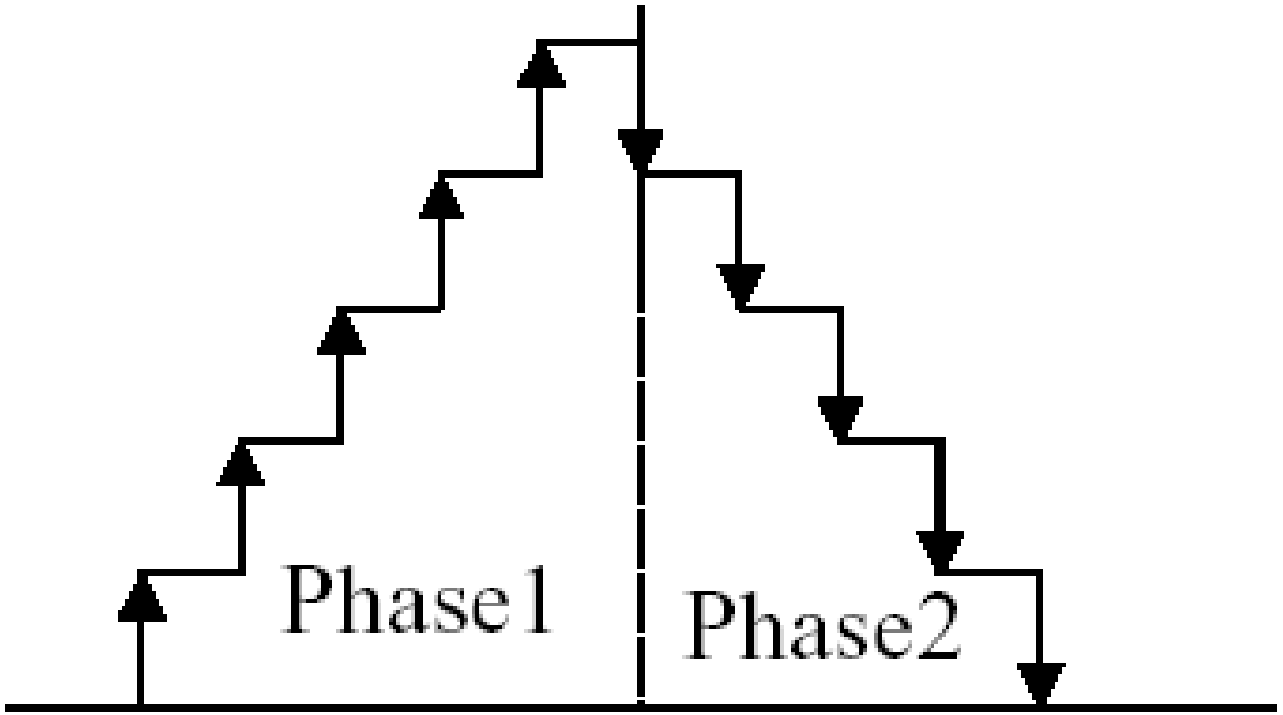
- **Blocare exclusivă** (*exclusive* sau *write lock*): Dacă o tranzacție blochează un obiect in mod exclusiv aceasta poate citi şi modifica valoarea obiectului.

Algoritmi bazați pe blocări

	Partajat	Exclusiv
Partajat	Da	Nu
Exclusiv	Nu	Nu

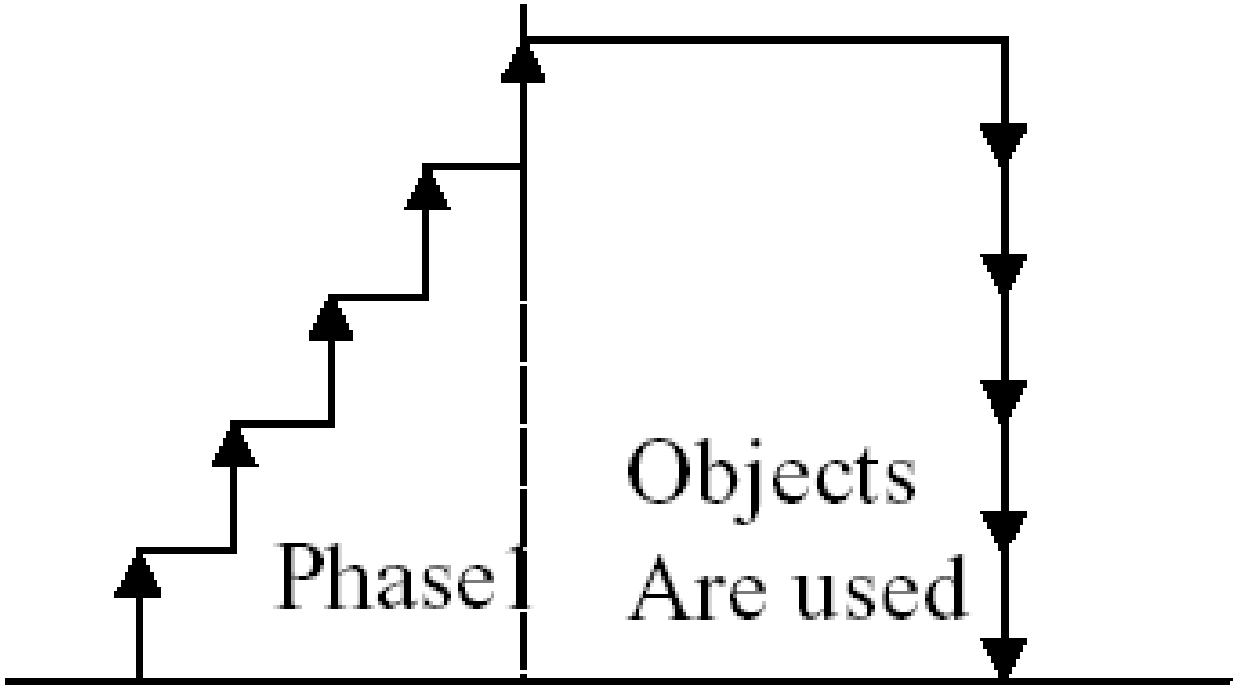
Protocol de blocare în două faze

- *2PL (2 Phase Locking) protocol*:
  - toate operațiile de blocare preced prima operație de deblocare în cadrul tranzacției.



Protocol strict de blocare în două faze

- *Strict 2PL protocol*:
  - Toate blocările sunt menținute de către tranzacție până imediat înainte de ***commit***



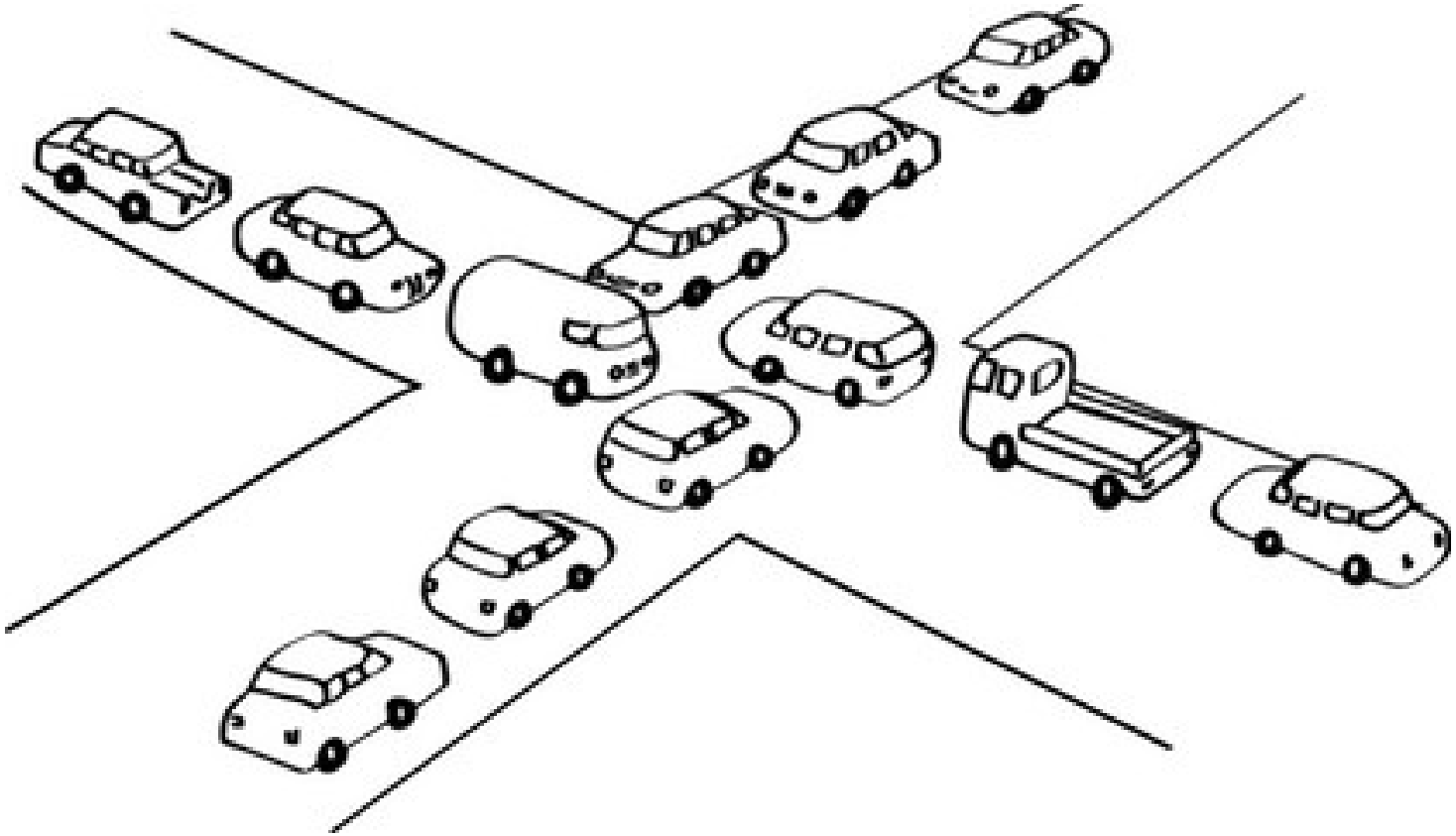
Gestionarea blocărilor

- Cererile de blocare şi deblocare de obiecte sunt gestionate de modulul *Lock Management*
- Tabelă de blocări :
  - Tranzacțiile care au cel puțin o blocare
  - Tipul de blocare (*shared* sau *exclusive*)
  - Pointer către o coadă de cereri de blocare
- Operațiile de blocare şi deblocare trebuie să fie atomice (!!)

Deadlock

Two or more transactions waiting for the same resources.

Deadlock

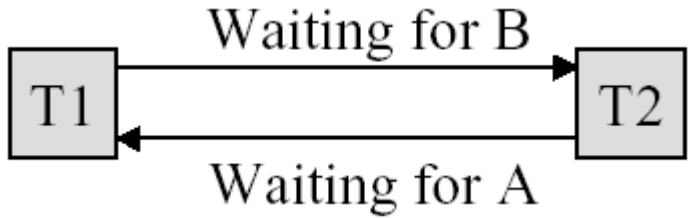


Detectare

Prevenire

Exemplu de *deadlock*

T1	T2
begin-transaction	
Write-lock(A)	begin-transaction
Read(A)	**Write-lock(B)**
A=A-100	Read(B)
Write(A)	B=B*1.06
Write-lock(B)	Write(B)
Wait	**write-lock(A)**
Wait	Wait
…	Wait
	…



Prevenire *deadlock* prin *timestamp*

- Tranzacțiile mai vechi au prioritatea mai mare
- T_i doreşte accesul la un obiect blocat de T_j :
  - *Wait-Die*: Dacă T_i are prioritate mai mare, T_i aşteaptă după T_j; altfel T_i se termină
  - *Wound-Wait*: Dacă T_i are prioritate mai mare, T_j se termină; altfel T_i aşteaptă

- Dacă o tranzacție eliminată
  se reporneşte ulterior,
  va avea *timestamp*-ul original


time is up!

Prevenirea *deadlock*-ului

- Ierarhia blocărilor: Se stabileste o ierarhie pentru obținerea blocărilor, se asigura că tranzacțiile solicită întotdeauna blocările în aceeași ordine - ajută la prevenirea așteptării circulare.

- Timpuri limită: Implementați timpuri limită pentru cererile de blocare. Dacă o tranzacție nu poate obține o blocare într-un interval de timp specificat, eliberează blocările curente și încearcă din nou. Acest lucru previne menținerea blocărilor pe termen nelimitat și apariția interblocărilor.

- Detectarea interblocărilor: Implementați un mecanism de detectare a interblocărilor care verifică periodic condițiile de așteptare circulară.

- Timp limită pentru blocare: Stabiliți un timp limită pentru cât timp o tranzacție poate deține o blocare. Dacă tranzacția depășește acest timp, eliberează blocările. Acest lucru ajută la evitarea situațiilor în care o tranzacție menține o blocare pentru o perioadă extinsă, ceea ce poate duce la interblocări.

Prevenirea *deadlock*-ului

- Ordinea tranzacțiilor: Impuneți o ordine consecventă în care tranzacțiile obțin blocările. Aceasta poate fi bazată pe ID-urile tranzacțiilor sau pe alte criterii. – se reduc șansele apariției condițiilor de așteptare circulară.

- Reducerea timpului tranzacțiilor: Minimizați timpul în care o tranzacție menține blocările. Descompuneți tranzacțiile complexe în unități mai mici sau eliberați blocările atunci când nu mai sunt necesare pe parcursul unei tranzacții.

- Design-ul bazei de date: Optimizați schema bazei de date și interogările pentru a minimiza necesitatea blocărilor. Luați în considerare denormalizarea sau alte tehnici de ajustare a performanței pentru a reduce concurența pentru resurse.

- Educația utilizatorilor: Educați dezvoltatorii și utilizatorii despre strategiile de prevenire a interblocărilor. Încurajați practicile care reduc probabilitatea interblocărilor, cum ar fi obținerea blocărilor într-o ordine consecventă.

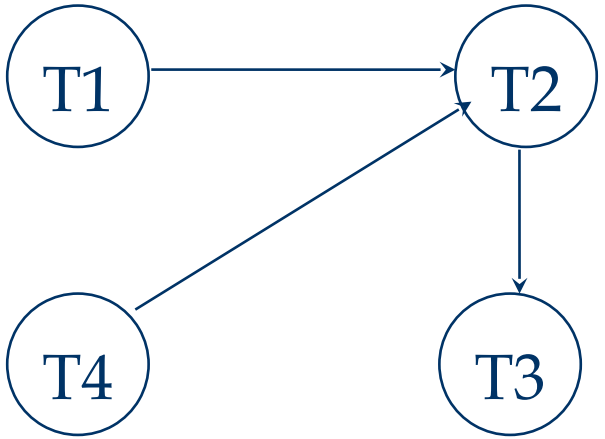
Detectarea *deadlock*-ului

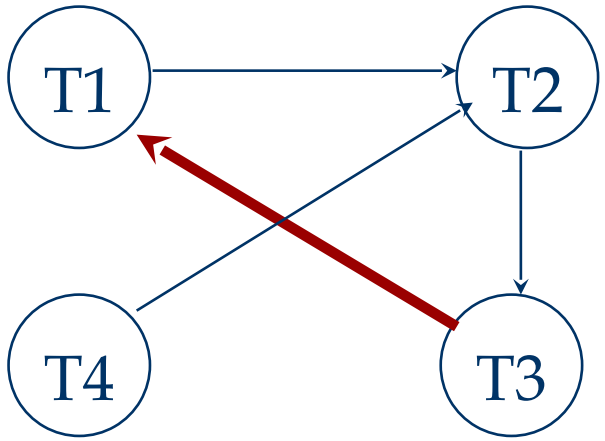
- Se crează un graf de aşteptare:
  - Nodurile sunt tranzacţii
  - Există un arc de la T_i la T_j dacă T_i aşteaptă după T_j să elibereze un obiect blocat
- Dacă este un circuit în acest graf atunci a apărut un *deadlock*.
- Periodic SGBD verifică dacă au apărut circuite în graful de aşteptare

Detectare *deadlock*

- Exemplu:

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)





Recuperarea după *deadlock*

- Cum se alege tranzacția victimă a unui *deadlock*?
  - Durata execuției unei tranzacții
  - Numărul obiectelor modificate de către tranzacție
  - Numărul obiectelor ce urmează să fie modificate de către tranzacție

- Politica de alegere a "*victimei*" trebuie să aibă în vedere echitatea: să nu fie aleasă de fiecare dată aceeaşi tranzacție ca victimă

Nivele de izolare: Read Uncommited

Caracteristici:

- O tranzacție poate citi date care nu au fost încă confirmate (commit) de alte tranzacții.
- Permite cel mai mare grad de paralelism, dar și cel mai mare risc de inconsistență a datelor.
- **Utilizare:** În scenarii unde performanța este prioritară față de consistență (ex. analize rapide de date).

Nivele de izolare: Read Commited

- O tranzacție T poate citi doar date comise.
- Totuși, un obiect citit de T poate fi modificat de o altă tranzacție în timp ce T este în curs de execuție.
- O tranzacție trebuie să obțină a blocare exclusiva înainte de a scrie un obiect.
- O tranzacție trebuie să obțină o blocare partajata înainte de a citi un obiect (adică, ultima tranzacție care a modificat obiectul trebuie să fie finalizată).
- Blocarile exclusive sunt eliberate la sfârșitul tranzacției.
- Blocarile partajate sunt eliberate imediat.

Nivele de izolare: Read Commited

Caracteristici:

- O tranzacție poate citi doar date care au fost confirmate (commit).
- Împiedică **dirty reads**, dar nu rezolvă alte probleme de concurență.
- **Utilizare:** Majoritatea bazelor de date folosesc acest nivel ca implicit (ex. Oracle, SQL Server)

Nivele de izolare: Repetable Read

- O tranzacție T poate citi doar date confirmate/comise.
- Nici un obiect citit de T nu poate fi modificat de o altă tranzacție în timp ce T este în curs de execuție (dacă T citește un obiect O de două ori, nici o tranzacție nu poate modifica O între citirile lui T).
- O tranzacție trebuie să obțină o blocare exclusiva înainte de a scrie un obiect.
- O tranzacție trebuie să obțină o blocare partajata a datelor înainte de a citi un obiect.
- Blocarile exclusive sunt eliberate la sfârșitul tranzacției.
- Blocarile partajate sunt eliberate la sfârșitul tranzacției.

Nivele de izolare: Serializable

- O tranzacție T poate citi doar date confirmate/comise.
- Niciun obiect citit de T nu poate fi modificat de o altă tranzacție în timp ce T este în curs de execuție.
- Dacă T citește un set de obiecte pe baza unui predicat de căutare, acest set nu poate fi modificat de alte tranzacții în timp ce T este în curs de execuție.
- O tranzacție trebuie să obțină blocari pe obiecte înainte de a le citi/scrie.
- O tranzacție obține, de asemenea, blocari pe seturi de obiecte care trebuie să rămână nemodificate.
- Dacă o interogare (SELECT * FROM … WHERE …) este executată de două ori într-o tranzacție, aceasta trebuie să returneze același set de răspunsuri.
- Blocarile sunt păstrate până la sfârșitul tranzacției.

Nivele de izolare: Serializable

- Cel mai strict nivel de izolare, asigură că execuția concurentă a tranzacțiilor este echivalentă cu o execuție serială.
- Împiedică toate problemele: dirty reads, non-repeatable reads și phantom reads.
- Implementat prin blocări stricte sau control bazat pe timestamp.
- Reduce semnificativ performanța, deoarece tranzacțiile trebuie să aștepte unele după altele pentru acces exclusiv la date.
- Utilizare: În aplicatii unde acuratețea datelor este crucială, cum ar fi contabilitatea bancară, etc.

Nivel de Izolare	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Performanță
Read Uncommitted	❌ Permise	❌ Permise	❌ Permise	🚀 Foarte rapid
Read Committed	✅ Blocate	❌ Permise	❌ Permise	⚡ Rapid
Repeatable Read	✅ Blocate	✅ Blocate	❌ Permise	🔹 Medie
Serializable	✅ Blocate	✅ Blocate	✅ Blocate	🐢 Foarte lent