

SGBD - mărtă de curs

Ep. 19: TRANZACȚII

transacție = sevență de instrucțiuni care împreună sunt privile ca o singură funcție logică

→ lasă baza de date într-o stare consistentă

→ proprietăți ACID:

A: atomicitate (totul sau nimic) → delete, update, add sunt atomicice

C: consistență (garanțorie conținutului de integritate)

↳ cheile primare de ex. rămân la fel

I: izolare (m-ar trebui să nu opereze cu cîine ar putea intra în conflict)

D: durabilitate (acțiunile transacției executate persistă)

ex. dacă 2 transacții se exec.
în acel. timp, efectul asupra BD
ar trebui să fie același cu exec.
seriale ale acestora

→ anomalii:

1. Dirty reads (reading uncommitted data)

T₁: citește A, modifică A

T₂:

citește A, modifică A, commit

conflict WR (write read)

citește B, modifică B, abort

practic T₂ a citit ceea ce nu e OK (uncommitted)

A revine la valoarea inițială (rollback T₁) \Rightarrow T₂ nu e durabilă

2. Unrepeatable reads \rightarrow conflict RW

T₁: R(A)

R(A), w(A), commit

T₂: R(A), w(A), commit

modifică din exterior, 2 citiri consecutive dau 2 rez. diferențite

3. Blind writes \rightarrow conflict ww (overwriting uncommitted data)

T₁: w(A)

w(B), commit

trămâne cu ca 2 ișT₂

T₂: w(A), w(B), commit

B trămâne cu ca a 2 ișT₁

4. Phantom reads

DATA 10.11.2022 - 10:30

selected STUDENTS

T1: select STUDENTS

T2: add STUDENTS

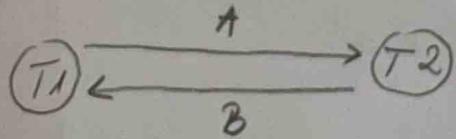
EP.20 : PLANIFICAREA TRANZACȚIILOR

- ordonarea secvențială a instrucțiunilor (read/write/abort/commit) a m tranzacții
- planificarea e ordinea read-writer și write-writer din 2 (sau mai multe) tranzacții
- poate fi serială (nu intercalată actiuni) sau non-serială (ultima op. a fiecăruia din tranzacții să se execute după prima op. a celeilalte tranzacții)
- dacă nu putem obține o planificare serializabilă (cu acq. efect ca execuția lor în rând) avem o problemă, m-ar trebui să permitem execuțarea
- op. conflictuale (nu se pot intereschimba între o tranzacție): cel puțin una e de separe cum rezolvăm? - cel mai la indemâna e ordinea în care se exec. op. conflictuale să facă
 - conflicte echivalente: fiecare permută de actiuni conflictuale e ordonată în același mod
 - o planificare e conflict serializable dacă e conflict ech. cu o plan. serială
- graf de precedență: orientat, un nod e o tranzacție, are i j dacă o op. de citire / modificare din i se realizează după o op. conflictuală din j

R(B), W(B)

ex.: T1: R(A), W(A)

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



⇒ concluzie: nu avem cum să găsim o planificare serială care să îibă acq. efect (⇒ nu e conflict serializable)
pt. că avem un circuit

> view echivalente (condiții):

- dacă în prima planificare T_i este primul pe A, în a doua totul trbuie să il citească primul
- dacă T_i este val. A modificat de T_j , aşa trbuie să fie și în a doua planificare
- dacă T_i e ultimul care modifica A în prima tran., aşa e și în a doua tran.
(nu sunt conflict ech., dar sunt view ech.)

$T_1:$	$R(A)$	$w(A)$	$T_1:$	$R(A), w(A)$
$T_2:$		$w(A)$	$T_2:$	$w(A)$
$T_3:$		$w(A)$	$T_3:$	$w(A)$

→ CONCLuzIE: toate planificările < planificări serializabile < planificări view-serializabile
< planificări conflict-serializabile < planificări seriale

Ep21: Controlul concurenței transacțiilor

→ avem 2 tipuri de blocări

1. shared / read lock : poate citi, dar nu poate modifica
2. exclusiv / write lock : poate citi și modifica

→ politici

1. POLITICA DE BLOCARE ÎN 2 FAZE

- dacă am blocat A, apoi B (apoi altel...) după ce am blocat ultimul urmează etapa de deblocare (!) în ordine inversă: întâi B, apoi A)

2. BLOCARE ÎN 2 FAZE STRICT → elimină toate anomalii

- difizit că nu sunt deblocate decât atunci când tran. se termină (deobicei, atomice)
- miercurăzi și mult capacitatea de a executa 2 tran. în acel. timp, de astă preferăm 2 PL, cănd se poate

- avem tabele de blocări: cu tran. care a blocat, tipul blocării, poziție și durată de blocare, tipuri
spre același obiect
- poate apărea deadlock: ne antepășește între ele
- * evitare (prevenire): - hierarhic între transacții (timestamp, cea mai veche e cea mai importantă)
- ex: wait-die: T_i are prioritate, T_i așteaptă după T_j ; altfel T_i se termină
wound-wait: T_i are prioritate, T_j se termină; altfel T_i așteaptă
- dacă o transacție eliminată (victima) se repornește ulterior, nu ar avea timestampul original
- de obicei SGBD face altceva, nici wait-die, nici wound-wait. Se vede cât timp a așteptat transacția după o resursă și dacă a trecut un timp se termină (prob. n-a ajuns în deadlock)
- * detectare: graf de așteptare (dacă avem circuit, alegem o victimă)

Ep. 22: Baze de date

distribuite

- sparte în mai multe fragmente, memorate în locații diferite
- avantaje: - autonomic locală
- performanță în accesare date
 - disponibilitate
 - modularitate (împărțit în sub-sisteme)
- dezavantaje: - unde vor fi stocate fragmentele diferențiate ale BD?
- în câte fragmente se sparg?
 - costuri de comunicare
 - procesare paralelă (avantaj)
 - ↳ interogări mai mici
- } PROIECTARE
- } PROCESARE INTEROGĂRI
- {
- serializabilitate
 - gestionarea deadlock
 - propagarea modificării
- ↳ inconsistență eventual
- } CONTROL CONCURENTĂ
- {
- multiple modalități de execuție
 - sincronizarea datelor
- ↳ stocarea pe replici diferenți a BD
- } PĂSTRARE CONSISTENȚĂ

tipuri de BD distribuite:

1. SGBD singular: transmitem spre un server, care gestionează mai multe fragmente
2. SGBD multiplu: serverul difuză pe fiecare fragment, comunică între ei printr-un manager
 - omogen: ex. QBW server pe tot
 - heterogen: niciun difuzor (MySQL pe altul)
trebuie tradus pentru fiecare

→ stocarea datelor - fragmentare

1. orizontală (anumite informații sunt într-un loc, altele în alt loc)
 - primară (toti studenții în diferite locuri)
 - derivată (pointeri spre grupe, specializări, etc)
↳ se duc pt. că suntem legate de datele primaria pe care am avut să le transmit
2. verticală: minim cheia primară comună pe tot

→ proprietăți:

1. completitudine: pt. oricărui elem. din relația mea trebuie să fie cel puțin un fragment fi care face parte din multimea de fragmente în care aparțin elem. x (toate elem. trebuie să fie și opuse)
2. disjunctivitate: nu există elemente doar de 2 ori (în 2 fragmente diferențiale)
3. reconstrucție: făcând un join nu obținem baza de date pe care am fragmentat-o

→ propagarea modificărilor

1. sincron: imediat după ce am făcut modificarea (se face pe tot)
→ transparent pt. utilizator

2. asincron: cu întârziere

→ tehnici de replicare sincronă:

1. citeste-orică/modifică-tot (ROWA)

→ informația e adusă de pe acercurul cel mai apropiat de mine
→ transm. nu se termină până nu se modifică toate replicile frag. respectiv
→ citiri rapide și scrieri mai lente

2. Votare

ex: dîm 10 copii alegem raport 7:4

eu nu trebuie să rupă/actualizez modificarea pe toate, decât peste \Rightarrow sunt momente când nu sunt la fel informațiile, deci atunci când citesc trebuie să o fac de pe rețea de 4 fragmente (dacă 3 au acel. info și una nu, înseamnă că ceea ce se întâmplă are informație veche)

\rightarrow nu e prăvăd utilizat

\rightarrow tehnici de replicare ASINCRONĂ

1. Peer to peer

- \rightarrow avem 10 copii (o parte din ei sunt copii - master aka sunt actualizate imediat)
- \rightarrow dacă 2 master-ură modifică în același timp, doar cea care modifică ultima va obține un căștigător
- \rightarrow de preferat când nu avem conflicte (ex: fragmente disjuncte)
- \rightarrow de la copia principală la copiile secundare, nu imediat

2. Site principal

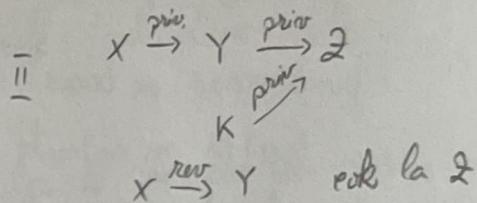
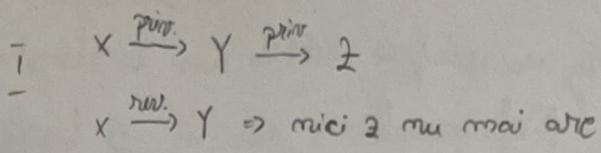
- \rightarrow o copie primară și mai mulți subscriberi
- \rightarrow subscriberii apeleză la copia principală ca să vadă dacă s-au făcut actualizări:
 - 2.1. pe bază de log
 - \rightarrow o tabelă change data table în copia principală (se reexecută și este nouă)
 - \rightarrow având ceea ce se transferă și, putin
 - \rightarrow text
 - 2.2. procedural
 - \rightarrow fac căte un snapshot al bazei de date
 - \rightarrow iau tot și o stocuz undeva, apoi când vinei subscriberii să văd dacă există un snapshot mai recent de ultima actualizare. Dacă este, îl copiez pe baza mea de date

Ep23: Securitatea bazelor de date

Acces Discretional

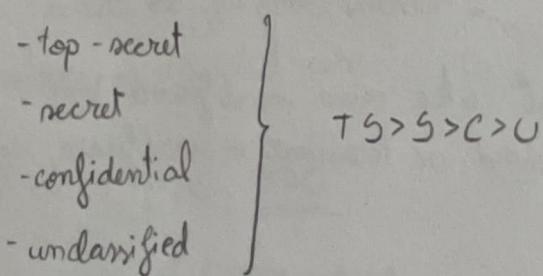
→ controlul discretional al accesului: se bazează pe conceptul drepturilor de acces (privilegii) p. obiectele bazei de date (tabeli & view-uri) și pe mecanisme de acordare și revocare de privilegii

- creatorul unui tabel primește toate privilegiile automat
- select, insert, delete, references (OFERIRE PRIVILEGIU) → GRANT
 - ↳ poate crea o tabelă care pointează la mîine
- [WITH GRANT OPTION] = poate oferi mai departe privilegii
- REVOCARE PRIVILEGIU → REVOKE



Ep24 : Acces Obligatoriu

- obiect: orice element din BD
- subiect: ceea ce interacționează cu BD
- celălalt mecanism nu este de strict
- Modelul Bell-LaPadula



- securitate simplă: S poate citi obiectul O $\Leftrightarrow \text{class}(S) \geq \text{class}(O)$
- proprietatea *: S poate modifica obiectul O $\Leftrightarrow \text{class}(S) < \text{class}(O)$
- clasa de obiect intră în clasa primară

Ep 25: Recuperarea datelor

- modulul de recuperare a datelor e responsabilul păstrarea prop. de:
 1. atomicitate: refac efectul acțiunilor trans. meconizate
 2. durabilitate: trans. comise "rezistă" erorilor și întreruperilor neașteptate
 - dacă se întrerupe sistemul în timpul exec. unei tranzacții, se refac starea de dimineață
- checkpoint: până unde suntem siguri că toate modificările sunt pe disk
 - { - s-a terminat înainte: ok
 - a început înainte, s-a terminat după checkpoint: roll forward (reluăm doar de la checkpoint înainte)
 - s-a început înainte, trebuia să se termine după sistem crash: rollback
- funcționează pe bază de log ca să facă REDO
 - se face la minute sau tranzacții
 - în absență unui checkpoint trebuie să merg până la prima intrare din log
 - dirty flag: s-au operat sau nu modificări pe acea pagină
- write-ahead logging: prima dată pun log-ul pe disk, după tranzacția (în caz că trebuie refăcută la o cădru de sistem)
- pin-count: câte tranzacții folosesc acea pagină în acel moment
dacă este > 0 nu poate fi alătură victimă (de scos) când memoria e plină
 - ↳ de buffer manager
 - o salvează pe disk
- conflict buffer manager - recovery manager → TRIVIAH
- pt. recovery manager: force & no-steal aka nicio modificare intermediară nu e pe hard disk și de fiecare dată când se termină o modificare, se ducă pe hard disk
- ideal: steal-no-force (performanță) → lăsăm control total la BM
BM poate salva modificări intermediare
RM salvează doar un commit

Ep 26: Recuperarea Datei în BDD

→ avem o transacție principală care le coordează pe restul (eventual o transacție pe fiecare fragment din BDD)

- dacă tran. principală are succes înseamnă că cele mici au avut
- dacă una mică eșuează, tran. mare eșuează și ne face rollback la tot

→ comitem în 2 faze (2PC) → ola cu generali și armate

1. coordonatorul trimite către site-write mai mici un mesaj: prepare
2. site-write trimit yes sau no la coordonator
3. dacă primește cel puțin un nu, înseamnă abort în log rec și transmite abort site-write. Altfel înseamnă commit în log rec și transmite commit site-write
4. Se transmite done coordonatorului (indiferent dacă în log e abort sau commit)
5. Coordonatorul scrie end în log după ce primește toate done-write

- nu e nevoie să fie ok, dar e cel mai folosit
- toate întrările din log au transaction ID și coordinator ID (pe acest caz distribuit)

→ 2PC - recuperare date

re-încărcă în log : dacă nu există end, dar există commit → redo
abort → undo

→ blocări : coordonatorul a trimis prepară, subordonatul trimite yes, dar coordonatorul erăt → toată transacția este blocată până își revine

→ reader : în loc de yes/no, înseamnă că subordonat nu face modificări, doar rulează select-write

→ avem și cu 3 faze : 3PC

can commit? → yes

pre commit → ACK

do commit → have committed

Ep 27: Algoritmul ARIES de recuperare a datelor

- LSN - log sequence number
 - creste impremntal (un fel de cheie primară)
- pageLSN - fiecare pagină de date conține un pageLSN aka identificatorul celei mai recente înregistrări făcute (pt. cele stocate pe disk)
- flushedLSN - tot ceea ce e până la acest identificator a ajuns pe hard disk
 - cel mai mare ID până la care toate modificările au ajuns pe disk
- întrări din LOG (câmpuri)
 - LSN - identificator
 - prevLSN : sunt într-o listă simplă întâlnită în log după acest prevLSN
 - transID - id de tranzacție care face modificarea
 - type : update, commit, abort, checkpoint, end, compensation log record (CLR)
 - pt. update mai avem și : pageID, length, offset, before-image, after-image
 - pt. UNDO modificări: delete, insert, update
- ! CLR : avem 5 UNDO-uri de făcut, n-au făcut 3 și iar dă crash
atunci pe cele 3 avem CLR și pt. UNDO nu li facem sau cănd le reuzim
- ! → log-ul îl duc pe disk la commit, chiar dacă modificările sunt doar în memoria interioară
- faze ale ARIES:
 1. analiză : ne parcurge log-ul de la cel mai recent checkpoint pînă final pt. identificarea tuturor tranz. actuale și a tuturor pag. modificate existente în buffer în timpul întreruperii
 2. redo : refac toate cele care au avut commit după checkpoint (de la prima la ultima)
 3. undo : folosind valoarea anterioră prezentă în intrare se anulează, dimineață în față
- ! nu se amestecă redo cu undo

Ep 28: Sortarea Externă

- Algoritmă clasică presupune că datele sunt stocate complet în mem. internă
 - se sortază pe bucăți, după care o interclasare
- Algoritm sortare externă

1. se împart datele în monotonii
 2. se interclasă monotonii într-un singur sir complet sortat
- monotonii inițiali se vor să fie cât mai lungi
 - se paralelizează cât mai mult citirea, procesarea și salvarea datelor
 - nr pagini de sortat \Rightarrow nr pagi = $\lceil \log_2 m \rceil + 1$
post. initializare
 - costul total (dpdV al transferurilor) : $2m (\lceil \log_2 m \rceil + 1)$
 - funcționarea pe ideea divide et impera

→ Arbore de selecție

- selecțarea celui mai mic elem. e consumator de timp
- vizi video că e un alg. simplu (pe la minut 25)
 - cele mai mici elem. sunt preluate din varful arborelui. elem. mai mari sunt impinsă în față.
 - se repetă până arboreli se găsesc (grupuri inițiale, din care extragem, sunt sortate)

→ Algoritm de sortare internă (min-huap)

↳ arbore cu ară minimul în rădăcina

- ex quicksort
- alt algoritm pe la minutul 30
- replacement selection alg. (min 34)
- meșteșugăm să creștem monotonii inițiale mai lungi

Ep 29: Operatorul JOIN (evaluarea lui)

(3)

→ Student, Courses, Evaluations Evaluations
 → join pe Students x Evaluations (R & S) Students
 ↳ join natural

→ conceptual ne opăndem că facem un produs cartezian (toate cu toate), după care filtrează doar cele p. care se întâmplă egalitatea de id-uri

→ în practică nu se întâmplă asta

M : pagini de memorie dim R , p_R înregistrări pe pag (100)

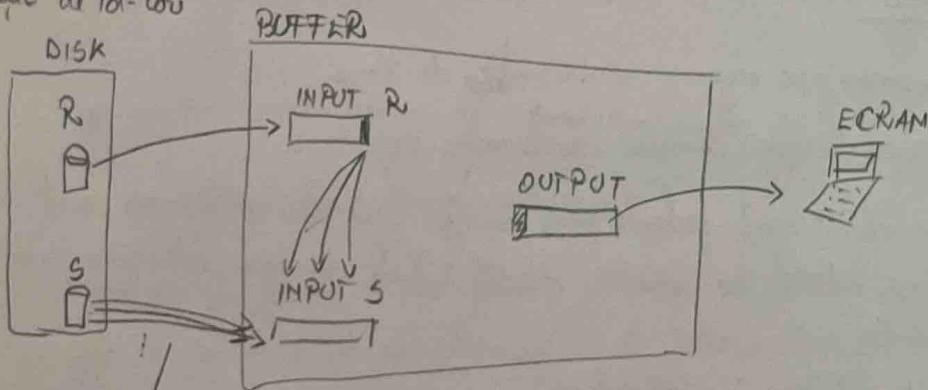
N : pagini de memorie dim S , p_S înreg. pe pag (80)

→ metrică folosită : nr pag citite/salvate

→ TEHNICI DE IMPLEMENTARE + OP. JOIN
 ITERARE

① Simplified nested loops join

- iterăm primul rând din tabelă R și p.d. fiecare iterăm primul rând din S și facem comparație de id-uri



de 500 ori (500 pg)
 apoi o două pag dim R și tot aşa

$$\text{- cost: } \boxed{|M + P_R * M * N|} = 1000 + 100 * 1000 * 500$$

- fără eficiență

② Page oriented nested loop join

- diferă că adăună o pagină din S, compară cu ea toate înregistrările de pe pagina din R, alia apoi adăună urm. pagină din S (dincolo mergea înregistrare cu înregistrare din R)

$$\text{- cost: } \boxed{M + M \cdot N} = 100 + 100 * 500$$

③ Block Nested Loops Join

- folosim mai multe pagini să citim din R (nu doar una)
- pt. toate înregistrările din paginile aduse din R facem comparația cu nimicuia pagină din S , apoi aduc urmă din S și tot așa
- cost :
$$\boxed{\text{nr. pag. tabela exterioară } R + \text{câte pagini mici buffer folosim pt cătarea concurentă}} \\ * \text{nr. pag. tabela internă } S$$

$$= 1000 + 10 * 500 \quad (\text{pp. avem } 102 \text{ pg. disponibile})$$

- generalizare a tehnicii precedente

II INDEXARE

Index Nested Loops Join

- exploatăm indexul
- ⇒ ne fac doar o căutare binară ca să găsim match-uri
- cost :
$$\boxed{M + ((M + P_R) * \text{cost găsire înregistrare dim } S)}$$

$$\text{cost de găsire} = \text{cost de căutare în index} + \text{cost cădere înregistrare}$$

$$\begin{aligned} &\text{acces direct: } 1 \cdot 2 \quad \text{cu Barbore: } 2^{-4} \\ &1 \cdot 2 + 1 = 2 \cdot 2 \end{aligned}$$

 $\Rightarrow 221 \ 000$

III PARTITIONARE

① Sort Merge Join

- sortăm R și memorăm temporar R' { pe hard disk, folosim bufferul }
- sortăm S și memorăm temporar S' { pe hard disk, folosim bufferul }
- citim prima pag din R' și prima din S' , apoi elem cu elem, nu oprim când găsim potrivirea
- cost :
$$\boxed{M \log_2 M + N \log_2 N + (M+N)} = 7500$$

$$\underbrace{\text{costul scanării (f. rar parte } \frac{1}{2} M+N)}$$

$$\text{- sortare în 2 pași}$$

$$\text{toate înregistrările au ac val pt. cheia externă}$$

② SMJ optimizat ≠ cau cel mai bun

- se căută și ultimul monotonii, dar nu în mai interclasă
- apoi facem merge cu prima pag dim fără monotonie
- cost: monotonii R = 1000 + 1000
monotonii S = 500 + 500
merge = 1000 + 500

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} = \underline{\underline{4500}}$$

! condiție importantă: să avem destul spațiu în buffer să citim prima pag. dim (pe cazul asta 16) fără merge

$B > \sqrt{L}$ → nr pag celei mai mari tabele se întâlnește în join

↳ nr pag disp. buffer

③ Hash join

- citim R, aplicăm o funcție de dispersie să o spargă în partitii ($B-1$ partitii)
- la fel pt. S
- citim partitiiile dim R una cu una

pt. fiecare aducem pag. cu pag. dim S, apoi facem comparații

$$- cost: \boxed{3(M+N)} = 4500$$

! part. cu R să încapă în mem. internă

$$\rightarrow B > \sqrt{M}$$

Ep 30: Factori de reducție

→ Catalogul unei baze de date

- nr. de înregistrări și nr. pagini dim fiecare tabelă
- nr. val. distințe ale cheilor de indexare și nr. pagini dim index
- înălțimea și val. minime și maxime ale cheilor (ex. pt. moto ar fi 4 și 10)

→ aceste cataloge sunt actualizate periodic

→ pot avea și alte date suplimentare (ex: cură histogramme)

→ estimarea dimensiunii și factorii de reducție

(cum la cate pagini nu ne așteptăm că urmăză să accesăm)

- ne ajută să alegem cea mai bună tehnică de întreagere

Ep 31: Evaluarea Operatorilor

selecție și proiecție

→ $\sigma = \text{selecție}$

→ dimensiunea rez. aproximativă de : dimensiunea lui $S * \text{factor de reducție}$

select * from Students S where S.smname < 'C%'

40 000 înregistrări.

30 lit. în alfabet \Rightarrow cam $\frac{3}{30} = \frac{1}{10}$ înregistrări obținem

→ dacă nu avem index, sortat : - scărim totă tabla

→ fără index, sortat : - costul e N (nr. pag. dim S)

- căutare binară

- cost $\log_2 N$

→ cu index pe atr. de selecție : fol. indexul pt. a det. înregistrările dim rez, apoi le returnăm
- costul depinde de nr. înr. returnate și de clustORIZARE

- costul de căciu : 1.2, 2-4

↓ ↓
direct B-arboru

- cost de aducere : clustORIZAT : 50

medclustORIZAT : 4000 (acă eventual sortăm și intrările im index)

→ condiții de selectie generale

- indexul trebuie să contină în prefix condițiile din întrebare
ex: $A=5 \text{ și } B=3$ avem un index pe A și B
dacă vă dări doar pe B nu va ajuta

→ ABORDĂRI ALE SELECTIILOR GENERALE

① găsirea celei mai reditive căi de acces

ex: index pe A și index pe B , cu folosim? cea care dă cele mai puține înr. (pagini)

② dacă sunt 2 sau mai mulți indexi

- se obține lista de rând ale înregistrărilor folosind fiecare index
- se intersectează rezultatele
- se va obține apărăm termenii rămași

→ ii : proiecție

ex: Select DISTINCT E.sid, E.cid from Evaluations E (aici ii cid, sid Evaluations)

- se elimină câmpurile medioite
- se elimină duplate

→ ABORDĂRI:

① proiecția bazată pe sortare (aici vom primi rez sortat după cele 2 câmpuri)

- eliminarea câmpurilor medioite: cost $N + T$ \downarrow nr pag E' (tabela temporară doar cu câmp. care nu trebuie)

- sortare E': $T \log_2 T$ (cost)

- scanare rez: cost T

- pot optimiza folosindu-mă de pașii pe care îi fac oricum la sortarea externe
(formarea de monotonie - pot extrage acolo câmpurile dorite - și interclasarea)

- cost: neoptimizat: 3500
optimizat: 1500

↳ eliminăm și dupliaj

② proiectie bazată pe funcție de dispersie

- fază de partitionare (pe baza celor 2 id-uri) - când le trimiți le trimiți direct cu cămpurile bune
- fază de eliminare a duplicatelor (folosind o altă f. de dispersie)
- cost: 1500

Ep 32: Optimizarea interopărilor

→ pași făcuți de modulul de execuție:

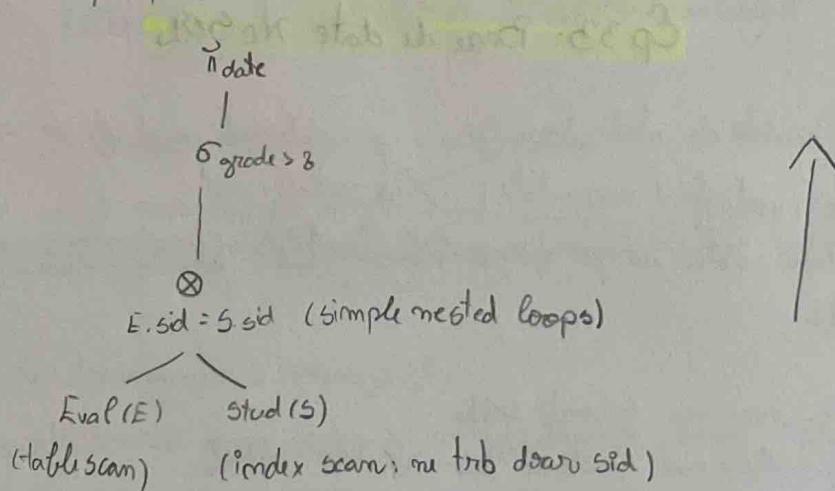
- parsare

optimizare { - selecție plan logic (un arbore cu tabele ca frunze și op. logice ca noduri)

 |- selecție plan fizic

 |- execuție interoparic : compilator, motor execuție, index/record mgt., buffer mgt., storage mgt., disk

- ex plan logic de execuție pt. select E.date from Eval E, Stud S where E.sid = S.sid and E.grade > 8



- SGBD încercă să schimbe între ei operatorii logici, încercă să îmbunătățească performanța ideal se alege cel mai bun plan posibil
în practică se elimină planurile cele mai proaste

→ SYSTEM R OPTIMIZER

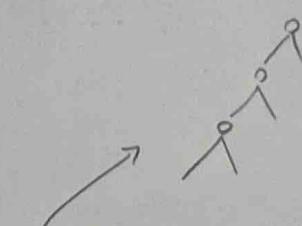
- cel mai utilizat

- funcționează în baza <10 join-uri

- ia în plus costul CPU (de utilizare a microprocesorului)

!- nu ia în considerare toate planurile, doar planurile left-deep join (permite ca rezultatul să fie transferat în pipeline către urm. op. fără stocarea temporară a rezultatului)

- exclude prod. cartesian (dacă poate)



- planul fizic aduce detalii de implementare la planul logic
- echivalențe în algebra relațională permit algoritmii unei ordini diferențiale a join-urilor și "împingerea" selectiilor și a predicilor în fața join-urilor
- selecție: $\sigma_{c_1 \dots c_m}(R) = \sigma_{c_1}(\dots(\sigma_{c_m}(R)))$ cascadă
 - $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$ comutativitate
 - proiecție: $\pi_{a_1}(R) = \pi_{a_1}(\dots(\pi_{a_m}(R)))$ cascadă
 - join: $R \otimes (S \otimes T) = (R \otimes S) \otimes T$ asociativitate
 $R \otimes S = S \otimes R$ comutativitate
 $\Rightarrow R \otimes (S \otimes T) = (T \otimes R) \otimes S$
 - ! atenție: $\sigma(R \otimes S) \neq \sigma(R) \otimes S$ (doar atributelor din R îm preleagă)

Ep 33: Baze de date NoSQL

- totatwile ab mai căută de utilizatorii BD pt. relational mult ab pe verticală performanțele
- scalabilitate (verticală / orizontală) \rightarrow restul
 - disponibilitate: datele să fie disponibile clientilor (five-nines availability)
 - performanță \rightarrow 99.999% din timp
- ce înseamnă NoSQL? - orice nu folosește SQL
- totuși nu include BD orientată obiect
 - not only SQL
 - ex. Cassandra folosește Java-like, CQL
 - nu fol. conceptele modelului relational pt. stocare și nu folosește limbajul SQL standard

DIFERENTE moduri de abordare

- * BD relationali - extragerea informației folosind operații de join
 - accelerarea procesării presupune indexare
 - sunt impuse prop. ACID

alternativa (Teorema CAP):

- consistență
- availability

- organirea rute alternativă în rețea pt. a obt. date din diverse moduri

- ! Se pot implementa 2 (a 3-a are de suferit)

* modelul de consistență BASE

- Basic Availability: pară că funcționează în marca majoritate a timpului
- Soft state: consistență la scriere nu e necesară. replicile nu trebuie să fie mutual consistentă
- Eventual consistent: BD va fi consistentă la un moment dat

READ REPAIR: atesează încă dăru seama că nu e actualizat atunci se repară

DELETION REPAIR: nu e controlat de utilizator momentul

→ Abordări bazate pe coloane (NoSQL) Cassandra

- BD relationale sunt bazate pe linii
- înf. multe situații aplicațiilor au nevoie de informații aflate pe o singură coloană

→ Abordări bazate pe documente : MongoDB

- sharding : partitionarea unei BD mari în părți mici, mai rapid de gestionat
- embedded document : cercare cu join
- avem insert, find, remove (filter), count, getCollectionName, getIndexNames
 - ! poate avea parametrii limit, skip, sort
- responsabilitatea validării datelor cade la programator (pot adăuga numPass în loc de NPass)
- ! - nu avem convenție legată de structură (nici măcar pe bază de tip)
- ! - indecesii au doar structură de arbori B