UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

DEPARTAMENTUL CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

PROIECT

SISTEME AVANSATE DE

BAZE DE DATE

COORDONATOR ȘTIINȚIFIC:

IPATE FLORENTIN EUGEN

BOBE RADU

STUDENT:

VÎRTOPEANU SEBASTIAN-FILIP

BUCUREȘTI

2024

UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

DEPARTAMENTUL CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

CENTRU RECOLTARE SÂNGE

COORDONATORI ȘTIINȚIFICI:

IPATE FLORENTIN EUGEN

BOBE RADU

STUDENT:

VÎRTOPEANU SEBASTIAN-FILIP

BUCUREȘTI

2024

Sisteme Avansate de Baze de Date: Temă de casă

Cuprins

I. Exercițiu 1: Atribute Repetitive și Transformarea lor în Design-ul Logic

1. Atribut repetitiv al unei entități

Atributul repetitiv: Emailuri. Un donator poate avea mai multe adrese de e-mail. În modelul entitate-legătură, acest lucru este reprezentat printr-un atribut repetitiv.

Problema: Într-o bază de date relațională, fiecare tabel trebuie să aibă o structură fixă, iar atributul repetitiv încalcă această regulă.

Soluția: Crearea unui tabel separat pentru e-mail-uri (Email_Donator). Acest tabel are o cheie străină (ID_Donator), care face legătura cu tabelul principal Donator. Astfel, acest atribut multivaloare va deveni un tabel dependent de tabelul care conține cheia straină.

```
CREATE TABLE Donator (
    ID_Donator INT PRIMARY KEY,
    Nume VARCHAR(100),
    Prenume VARCHAR(100)
   -- alte atribute necesare
);
```

```
  CREATE TABLE Email_Donator (
        ID_email INT PRIMARY KEY,
        ID_Donator INT,
        Email VARCHAR(100),
        FOREIGN KEY (ID_Donator) REFERENCES Donator(ID_Donator)
  );
```

2. Atribut repetitiv al unei relații mulți-la-mulți

Context: În cadrul unei sesiuni de donare, un donator poate fi supus mai multor teste de sânge. Pentru a gestiona această complexitate:

Problemă: Avem nevoie să reprezentăm relația între donări și teste, unde o singură donare poate implica multiple teste, iar un test poate fi aplicat la mai multe donări. De asemenea, trebuie să gestionăm multiple rezultate ale testelor pentru o singură donare.

Soluție

- Crearea unui tabel **Donare_Teste**: Acest tabel asociativ face legătura între Donare și **Teste_De_Sange**. Tabelul asociază donările cu testele, dar nu stochează rezultatele testelor.
- Crearea unui tabel separat **Rezultate_Test**: Acest nou tabel va stoca toate rezultatele testelor. Fiecare înregistrare din Donare_Teste poate avea multiple rezultate asociate în Rezultate_Test. Astfel, putem urmări eficient multiplele rezultate ale testelor pentru fiecare donare. Astfel, cream un tabel dependent de tabelul asociativ corespunzător relației N:M.

```
CREATE TABLE Teste_De_Sange (
      ID_Test INT PRIMARY KEY,
      Nume_Test VARCHAR(100)
      -- alte atribute necesare
);
```

```
CREATE TABLE Donare (
      ID_Donare INT PRIMARY KEY,
      ID_Donator INT,
      Data_Donare DATE,
      FOREIGN KEY (ID_Donator) REFERENCES Donator(ID_Donator)
);
```

```
CREATE TABLE Donare_Teste (
      ID_Donare INT,
      ID_Test INT,
      Rezultat_Test VARCHAR(100),
      PRIMARY KEY (ID_Donare, ID_Test, Rezultat_Test),
      FOREIGN KEY (ID_Donare) REFERENCES Donare(ID_Donare),
      FOREIGN KEY (ID_Test) REFERENCES Teste_De_Sange(ID_Test)
);
```

3. Transformarea atributelor repetitive

Email_Donator: În loc să avem o listă de email-uri în tabelul Donator, avem un tabel separat care permite asocierea fiecărui donator cu unul sau mai multe email-uri. Astfel, structura bazei de date rămâne flexibilă și scalabilă.

Donare_Teste și **Rezultate_Test**: Această abordare ne permite să gestionăm complexitatea relației mulți-la-mulți între donări și teste într-un mod eficient. Prin crearea tabelului intermediar **Donare_Teste**, evităm redundanțele și clarificăm relația dintre donări și teste. Mai mult, adăugarea tabelului **Rezultate_Test** îmbunătățește această structură, permițându-ne să stocăm multiple rezultate ale testelor pentru fiecare asociere între o donare și un test. Astfel, putem gestiona informațiile într-un mod mai detaliat și structurat, fără a supraîncărca niciun tabel cu prea multe informații. Această structură modulară asigură eficiența și claritatea în managementul datelor.

```
CREATE TABLE Donare_Teste (
      ID_Donare_Test INT PRIMARY,
      ID_Donare INT,
      ID_Test INT,
      FOREIGN KEY (ID_Donare) REFERENCES Donare(ID_Donare),
      FOREIGN KEY (ID_Test) REFERENCES Teste_De_Sange(ID_Test)
);
```

```
CREATE TABLE Rezultate_Test (
      ID_Rezultat INT PRIMARY KEY AUTO_INCREMENT,
      ID_Donare_Test INT,
      Rezultat VARCHAR(100),
      Data_Test DATE,
      FOREIGN KEY (ID_Donare_Test) REFERENCES Donare_Teste(ID_Donare_Test)
);
```

II. Exercițiu 2: Relații în Modelul Entitate-Legătură

1. Exemplu de relație de tip 3

Relația de tip 3: O astfel de relație implică mai mult de două entități și poate fi exemplificată prin interacțiunea dintre Donator, Donare și Personal Medical.

- **Entitate 1: Donator** (persoana care donează sânge).
- **Entitate 2: Donare** (sesiunea specifică de donare de sânge).
- **Entitate 3: Personal Medical** (persoana care asistă la donare, de exemplu un medic sau un asistent medical).

Descrierea relației: Fiecare **Donare** este o interacțiune unică între un **Donator** și un membru al Personalului Medical. În această situație, avem o relație ternară în care fiecare **Donare** implică exact un **Donator** și exact un membru al **Personalului Medical**.

```
CREATE TABLE Donator (
      ID_Donator INT PRIMARY KEY,
      Nume VARCHAR(100),
      Prenume VARCHAR(100)
      -- alte atribute necesare
);
```

```
CREATE TABLE Personal_Medical (
      ID_Personal INT PRIMARY KEY,
      Nume VARCHAR(100),
      Functie VARCHAR(100)
      -- alte atribute necesare
);
```

```
CREATE TABLE Donare (
      ID_Donare INT PRIMARY KEY,
      Data_Donare DATE
      -- alte atribute necesare
);
```

```
CREATE TABLE Donare_Detalii (
      ID_Donare INT,
      ID_Donator INT,
```

```
        ID_Personal INT,
        Cantitate_Donata DECIMAL(5, 2),
        -- am folosit CLOB pentru ca m-am gandit ca observatiile
        -- pot fi de o dimensiune destul de mare
        Observatii CLOB,
        Status_Donare VARCHAR2(100),
        Data_Ora_Inceput TIMESTAMP,
        Data_Ora_Sfarsit TIMESTAMP
        FOREIGN KEY (ID_Donare) REFERENCES Donare(ID_Donare),
        FOREIGN KEY (ID_Donator) REFERENCES Donator(ID_Donator),
        FOREIGN KEY (ID_Personal) REFERENCES Personal_Medical(ID_Personal)
);
```

Un alt exemplu ar fi o relație de tip 3 care ia loc între **Donator**, **Personal Medical** și **Salon**(prin salon nu ne referim doar la locul de recoltare, ci si la locul de depozitare), aceste 3 entități intâlnindu-se în tabela de legătura donație, asta deoarece o donație poate exista doar dacă cunoaștem toate entitățile.

2. Exemplu de relație aparentă de tip 3 care se „sparge” în relații mulți-la-mulți

```
CREATE TABLE Teste_De_Sange (
        ID_Test INT PRIMARY KEY,
        Nume_Test VARCHAR(100)
        -- alte atribute necesare
);
```

```
CREATE TABLE Rezultate (
        ID_Rezultat INT PRIMARY KEY,
        Descriere_Rezultat VARCHAR(100)
        -- alte atribute necesare
);
```

```sql
CREATE TABLE Donator_Teste (
	ID_Donator INT,
	ID_Test INT,
	FOREIGN KEY (ID_Donator) REFERENCES Donator(ID_Donator),
	FOREIGN KEY (ID_Test) REFERENCES Teste_De_Sange(ID_Test)
);
```

```sql
CREATE TABLE Teste_Rezultate (
	ID_Test INT,
	ID_Rezultat INT,
	FOREIGN KEY (ID_Test) REFERENCES Teste_De_Sange(ID_Test),
	FOREIGN KEY (ID_Rezultat) REFERENCES Rezultate(ID_Rezultat)
);
```

III. Exercițiul 3: Exemple de Normalizare a Tabelului Relațional

1. Tabel în FN1, dar nu în FN2

- FN1 (Prima Formă Normală) presupune că toate valorile din tabel sunt atomice (indivizibile) și fiecare înregistrare are o cheie unică.
- FN2 (A Doua Formă Normală) necesită ca tabelul să fie în FN1 și toate atributele non-cheie să fie dependente de întreaga cheie.

Exemplu:

Să luăm un tabel **ProgramariDonare** care stochează informații despre programările donatorilor și personalul medical care le supervizează.

```sql
CREATE TABLE ProgramariDonare (
	ID_Programare INT,
	ID_Donator INT,
	Data_Programare DATE,
	ID_Personal INT,
	Nume_Personal VARCHAR(100),
	Functie_Personal VARCHAR(100),
	PRIMARY KEY (ID_Programare, ID_Donator, Data_Programare)
```

```
        -- Presupunem că Nume_Personal și Functie_Personal sunt dependente
--doar de ID_Personal, nu de întreaga cheie
    );
```

Problemă: Nume_Personal și Functie_Personal sunt dependente doar de ID_Personal, nu de întreaga cheie (ID_Programare, ID_Donator, Data_Programare).

Soluție:

Pentru a aduce tabelul în FN2, trebuie să eliminăm dependența parțială prin separarea datelor despre personalul medical într-un alt tabel.

```
CREATE TABLE PersonalMedical (
    ID_Personal INT PRIMARY KEY,
    Nume_Personal VARCHAR(100),
    Functie_Personal VARCHAR(100)
);
```

```
CREATE TABLE ProgramariDonare_FN2 (
    ID_Programare INT PRIMARY KEY,
    ID_Donator INT,
    Data_Programare DATE,
    ID_Personal INT,
    FOREIGN KEY (ID_Personal) REFERENCES PersonalMedical(ID_Personal)
);
```

2. Tabel în FN2, dar nu în FN3

- FN3 (A Treia Formă Normală) presupune ca tabelul să fie în FN2 și că toate atributele non-cheie sunt mutual independent.

Exemplu:

Să considerăm un tabel **DonatoriDetalii** care include informații despre donatori și orașul lor.

```
CREATE TABLE DonatoriDetalii (
      ID_Donator INT PRIMARY KEY,
      Nume_Donator VARCHAR(100),
      Oras VARCHAR(100),
      Cod_Postal VARCHAR(10)
      -- presupunem că Cod_Postal este dependent de Oras
);
```

Problemă: Cod_Postal este dependent de Oras, nu de cheia primară.

Soluție:

Pentru a aduce tabelul în FN3, trebuie să eliminăm dependențele tranzitive prin separarea datelor despre orașe într-un alt tabel.

```
CREATE TABLE Orase (
   Oras VARCHAR(100) PRIMARY KEY,
   Cod_Postal VARCHAR(10)
);
```

```
CREATE TABLE DonatoriDetalii_FN3 (
   ID_Donator INT PRIMARY KEY,
   Nume_Donator VARCHAR(100),
   Oras VARCHAR(100),
   FOREIGN KEY (Oras) REFERENCES Orase(Oras)
);
```

IV. Exercițiul 4: Exemplificarea Dependentei Multivaloare într-un Tabel Relațional

Dependența multivaloare sau multidependența într-un tabel relațional este un concept puțin diferit de dependența funcțională. În timp ce dependența funcțională implică o relație unu-la-unu sau unu-la-mulți între coloane, dependența multivaloare se referă la o situație în care există o relație mulți-la-mulți între coloane, independent de cheia primară.

Un exemplu este fiecare donator poate alege să doneze la mai multe centre, iar fiecare centru de donare poate fi asociat cu mai mulți membri ai personalului medical.

Exemplu:

Tabel: **Donator_Centru_Personal**

```
CREATE TABLE Donator_Centru_Personal (
      ID_Donator INT,
      Centru_Donare VARCHAR(100),
      ID_Personal INT
);
```

În acest tabel, presupunem următoarele:

- Un **Donator** (reprezentat prin **ID_Donator**) poate dona la mai multe centre (de exemplu, datorită programului).
- Un **Donator** poate alege ca donația să fie recoltată de diferite cadre medicale.
- Combinarea **ID_Personal** cu **Centru_Donare** nu este unică pentru un **ID_Donator**.

Aici, relația dintre ID_Personal și Centru_Donare este o dependență multivaloare. Nu există o dependență funcțională directă între aceste două coloane; un anumit personal nu determină un centru de donare specific, și invers. În schimb, fiecare donator poate asocia mai multe combinații de cadre medicale și centre de donare, creând astfel o relație mulți-la-mulți între aceste două atribute.

Pentru a gestiona corect aceste date într-o bază de date relațională și pentru a evita redundanța, ar fi ideal să se separe aceste informații în tabele diferite și să se utilizeze o structură de tabel de asociere pentru a reprezenta relația mulți-la-mulți.

V. Exercițiul 5: Indecși

a) Index de Tip Arbore B*

Structura Indexului de Tip Arbore B*

Indexurile de tip arbore B* (o variație a arborelui B) sunt folosite în bazele de date pentru a îmbunătăți viteza de acces la date. Structura unui astfel de index este similară cu cea a unui arbore B, dar cu diferența că nodurile sunt umplute mai mult (de obicei între 66% și 100%). Fiecare nod conține chei și referințe (pointeri) către nodurile copil sau date.

Utilizarea în Interogările SQL

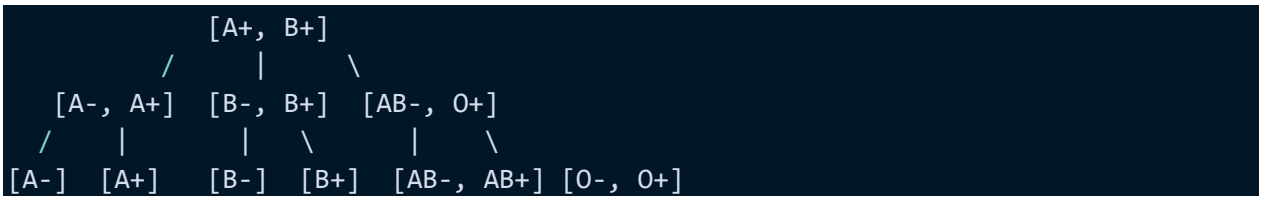
Să presupunem că avem un tabel Donari cu multe înregistrări și vrem să găsim rapid toate donările de un anumit tip de sânge. Creăm un index pe coloana Tip_Sange:

```
CREATE TABLE Donari (
      ID INT PRIMARY KEY,
      NumeDonator VARCHAR2(100),
      Tip_Sange VARCHAR2(3)
);

CREATE INDEX idx_tip_sange ON Donari(Tip_Sange);
CREATE INDEX idx_nume ON Donari(NumeDonator);
```

Cand executam o interogare de tipul:

```
SELECT * FROM Donari WHERE Tip_Sange = 'O+';
SELECT * FROM Donari WHERE NumeDonator= 'Sebi';
```



- Nodurile interne (de exemplu, [A+, B+]) conțin chei care ajută la navigarea prin arbore pentru a găsi valorile dorite.

- Nodurile frunze (de exemplu, [A-], [A+], [B-], [B+], etc.) conțin valorile efective ale indexului și referințe la rândurile din tabelul Donari unde aceste valori sunt întâlnite.

  Sistemul de gestiune a bazei de date (SGBD) va folosi indexul idx_tip_sange pentru a găsi rapid toate înregistrările cu tipul de sânge 'O+'. În loc să scaneze întregul tabel, SGBD-ul va traversa arborele B* pentru a localiza valorile dorite, ceea ce reduce semnificativ timpul de căutare. Arborele B* va fi parcurs începând de la rădăcină. În acest caz, am merge în dreapta arborelui, către nodul [AB-, O+], și apoi spre nodul frunză [O-, O+] pentru a găsi și accesa înregistrările corespunzătoare din tabel. Această abordare este susținută și de natură datelor, astfel un query care filtrează după prefix este des întâlnit.

b) Index de Tip Bitmap

Structura Indexului de Tip Bitmap

Indexurile de tip bitmap sunt eficiente pentru coloane cu un număr redus de valori distincte (de exemplu, coloane de tip boolean sau cu valori enumerate). Un index bitmap pentru o anumită valoare a coloanei conține un bit pentru fiecare rând din tabel: bitul este setat pe 1 (true) dacă rândul are valoarea respectivă, și pe 0 (false) în caz contrar.

Utilizarea în Interogările SQL

Să presupunem că avem un tabel Donari cu o coloană EligibilDonare care indică dacă donatorul este eligibil pentru donare.

```
CREATE TABLE Donari (
      ID INT PRIMARY KEY,
      NumeDonator VARCHAR2(100),
      EligibilDonare VARCHAR2(3)
);

CREATE BITMAP INDEX idx_eligibil_donare ON Donari(EligibilDonare);
```

Cand executam o interogare de tipul:

```
SELECT * FROM Donari WHERE EligibilDonare = 'da';
```

SGBD-ul va folosi indexul bitmap idx_eligibil_donare pentru a identifica rapid toate rândurile unde EligibilDonare este ’da’. Acest lucru se realizează prin verificarea rapidă a bitilor din index, ceea ce este mult mai rapid decât scanarea întregului tabel.

Ambele tipuri de indexuri au avantajele și utilizările lor specifice și sunt alese în funcție de tipul datelor și de tipul de interogări executate cel mai frecvent pe tabelul respectiv. Indexurile de tip arbore B* sunt preferate pentru coloane cu multe valori distincte, în timp ce indexurile bitmap sunt ideale pentru coloane cu puține valori distincte.

VI. Exercițiul 6: Crearea și Confruntarea cu Erori în Vizualizări

Pentru a crea un exemplu de vedere (vizualizare) VIEW_EX care va genera o eroare atunci când încercăm să inserăm date în ea, putem folosi o vizualizare bazată pe o interogare complexă care nu permite inserții. Un astfel de exemplu este o vizualizare care implică o interogare cu join-uri sau funcții de agregare.

Să luăm ca exemplu un tabel Donari și un tabel Donatori, și să creăm o vizualizare care rezumă numărul de donări făcute de fiecare donator:

```
CREATE TABLE Donatori (
	ID_Donator INT PRIMARY KEY,
	Nume VARCHAR2(100),
	Prenume VARCHAR2(100)
);

CREATE TABLE Donari (
	ID_Donare INT PRIMARY KEY,
	ID_Donator INT,
	DataDonare DATE,
	FOREIGN KEY (ID_Donator) REFERENCES Donatori(ID_Donator)
);

CREATE VIEW VIEW_EX AS
SELECT D.Nume, D.Prenume, COUNT(*) AS Numar_Donari
FROM Donari DN
JOIN Donatori D ON DN.ID_Donator = D.ID_Donator
GROUP BY D.Nume, D.Prenume;
```

În această vizualizare, datele sunt agregate (folosind COUNT(*)) și grupate după numele și prenumele donatorilor.

Când încercăm să inserăm date în această vizualizare:

```
INSERT INTO VIEW_EX VALUES ('Ionescu', 'Ion', 5);
```

Vom întâmpina o eroare. Motivele pentru care această inserție eșuează sunt următoarele:

a) **Imposibilitatea de a determina rândurile țintă**: Deoarece vizualizarea este bazată pe un join și o funcție de agregare, SGBD-ul nu poate determina în mod unic rândurile din tabelele de bază pe care ar trebui să le actualizeze sau să le insereze.
b) **Lipsa unicității rândurilor**: Vizualizările care conțin funcții de agregare, cum ar fi `COUNT`, `SUM`, `AVG`, etc., sau care sunt rezultatul unei operații de grupare, nu permit inserția deoarece nu există o corespondență directă și unică între rândurile din vizualizare și cele din tabelele de bază.

VII. Exercițiul 7: Asigurarea Securității cu Vizualizări în Baze de Date

Exemplu de utilizare a unei vizualizări pentru a îmbunătăți securitatea datelor:

Să presupunem că avem un tabel DonatoriDetalii care conține informații sensibile despre donatori, cum ar fi numele, adresa și detaliile de contact. Vrem să permitem utilizatorilor să acceseze doar numele donatorilor, fără a expune restul informațiilor.

Creăm o vizualizare care include doar coloanele permise:

```
CREATE TABLE DonatoriDetalii (
      ID_Donator INT PRIMARY KEY,
      Nume VARCHAR2(100),
      Prenume VARCHAR2(100),
      CNP VARCHAR2(13),
      Parola VARCHAR2(100)
);

CREATE VIEW VizualizareDonatori AS
SELECT Nume, Prenume
FROM DonatoriDetalii;
```

Utilizatorii pot acum interoga VizualizareDonatori pentru a obține numele donatorilor, dar nu pot accesa informații sensibile CNP și parola, deoarece acestea nu sunt incluse în vizualizare. Aceasta limitează accesul la datele sensibile și ajută la protejarea confidențialității informațiilor.

Prin acest mecanism, vizualizările funcționează ca un strat de securitate, permițând expunerea doar a datelor necesare și restricționând accesul la informații sensibile sau critice.

În plus, prin Google poți indexa produsele prin intermediul unui view în baza de date, oferindu-le doar datele de care au nevoie într-un mod sigur și eficient.

VIII. Exercițiul 8: Variația Rezultatelor în Interogări Consecutive Identice

Un exemplu în care două interogări SELECT identice, executate consecutiv în aceeași sesiune de lucru pe același tabel, pot produce rezultate diferite este atunci când un alt proces sau tranzacție modifică datele din tabel între cele două interogări. Acest scenariu este comun în sistemele de baze de date care suportă concurența și tranzacțiile.

Exemplu:

Presupunem că avem un tabel ProgramariDonare care conține informații despre produse și cantitățile lor în stoc.

Considerăm următoarea secvență de evenimente:

```
CREATE TABLE ProgramariDonare (
      ID_Programare INT PRIMARY KEY,
      ID_Donator INT,
      Data_Programare DATE,
      Status_Programare VARCHAR(100)
);
```

Într-o sesiune, executăm o interogare SELECT pentru a afla statusul a unui anumite programări:

```
SELECT Status_Programare FROM ProgramariDonare WHERE ID_Programare = 1;
```

Între timp, într-o altă sesiune sau proces, se execută o actualizare asupra aceluiași tabel, schimbând cantitatea în stoc pentru produsul respectiv:

```
UPDATE ProgramariDonare
SET Status_Programare = 'Anulată'
WHERE ID_Programare = 1;
COMMIT;
```

În sesiunea originală, executăm din nou aceeași interogare SELECT:

```
SELECT Status_Programare FROM ProgramariDonare WHERE ID_Programare = 1;
```

În acest scenariu, prima interogare SELECT va returna statusul înainte de actualizare, iar a doua interogare SELECT va returna statusul după actualizare. Deși interogările sunt identice, rezultatele sunt diferite din cauza modificării intervenite în tabel între cele două execuții. Acest lucru nu ar fi fost posibil dacă foloseam SERIALIZABLE, care prezvine atât Nonrepeatable Reads, cât și Phantom Reads.

O altă variantă este ordonarea random a rândurilor astfel orice rezultat al unei cereri este diferit.

```
SELECT * FROM ProgramariDonare
ORDER BY DBMS_RANDOM.VALUE();
-- am observat ca alte persoane folosesc SYS_GUID();
```

IX. Exercițiul 9: Interblocarea

Interblocarea (deadlock) în baze de date se întâmplă atunci când două sau mai multe tranzacții încearcă să blocheze resursele/obiecte pe care le folosește cealaltă, creând un

cerc vicios în care niciuna nu poate progresa. Acest subiect este des întâlnit și în sistemele de operare, unde există algoritmi de organizarea a resurselor pentru procese.

Exemplu:

Să presupunem că avem două tranzacții, T1 și T2, care operează pe două tabele Donații și Donatori.

- **T1** pornește și pune o blocare pe tabelul Donații pentru a face o actualizare.
- Apoi, **T1** încearcă să pună o blocare pe tabelul Donatori pentru o altă actualizare.
- **T2** începe și blochează Donații pentru a face o actualizare.
- Ulterior, **T2** încearcă să blocheze tabelul Donatori pentru a efectua și el o actualizare.
- **T1** așteaptă ca T2 să elibereze Donatori, iar T2 așteaptă ca T1 să elibereze Donații.

Niciuna dintre tranzacții nu poate progresa, deoarece fiecare așteaptă ca cealaltă să elibereze resursa blocată.

Sistemele moderne de gestionare a bazelor de date detectează adesea astfel de interblocări și le rezolvă prin întreruperea uneia dintre tranzacții, permițând celeilalte să continue. Tranzacția întreruptă va primi o eroare de interblocare și va avea opțiunea de a reîncerca operațiunea.

Interblocarea poate fi evitată prin utilizarea unor tehnici precum ordonarea consecventă a blocărilor resurselor sau prin implementarea unor mecanisme de timeout pentru blocări.

X. Exercițiul 10: Triggere Constraint

Să presupunem că avem două tabele: Donatori și ProgramariDonari. Vrem să ne asigurăm că un donator nu poate avea mai mult de un număr specificat de programări într-o perioadă determinată, de exemplu, pentru a respecta normele de siguranță.

```
CREATE TABLE Donatori (
      ID_Donator INT PRIMARY KEY,
      Nume VARCHAR(100),
      Prenume VARCHAR(100),
      -- alte atribute necesare
);
```

```
CREATE TABLE ProgramariDonari (
	ID_Programare INT PRIMARY KEY,
	ID_Donator INT,
	Data_Programare DATE,
	FOREIGN KEY (ID_Donator) REFERENCES Donatori(ID_Donator)
);
```

```
CREATE OR REPLACE TRIGGER VerificaLimitaProgramari
BEFORE INSERT OR UPDATE ON ProgramariDonari
FOR EACH ROW
DECLARE
	NumarProgramari INT;
	LimitaProgramari INT := 3;
-- presupunem o limită de 3 programări într-un anumit interval
BEGIN
-- Calculăm numărul de programări pentru donatorul respectiv
	SELECT COUNT(*) INTO NumarProgramari
	FROM ProgramariDonari
	WHERE ID_Donator = :NEW.ID_Donator
	AND Data_Programare BETWEEN ADD_MONTHS(:NEW.Data_Programare, -12)
AND :NEW.Data_Programare;

-- Verificăm dacă limita a fost depășită
	IF NumarProgramari > LimitaProgramari THEN
		RAISE_APPLICATION_ERROR(-20002, 'Limita de programări pentru
donator a fost depășită.');
	END IF;
END;
```

Descriere:

- Trigger-ul **VerificaLimitaProgramari** este activat după fiecare inserție în tabelul ProgramariDonari.

- El verifică numărul total de programări pentru donatorul respectiv în ultimul an (12 luni) și compară acest număr cu limita stabilită.
- Dacă numărul de programări depășește limita stabilită, trigger-ul generează o eroare și împiedică inserția noii programări.
- Acest exemplu ilustrează cum un trigger poate fi folosit pentru a impune reguli complexe de afaceri care nu pot fi realizate prin simpla utilizare a CONSTRAINT-urilor standard în definiția tabelului. În acest caz, trigger-ul asigură respectarea unei politici de sănătate și siguranță, limitând numărul de programări de donare de sânge pentru un donator într-o anumită perioadă.

XI. Exercițiul 11: Gestionarea Excepțiilor în Blocuri Multiple

În acest exemplu, vom avea două blocuri PL/SQL: un bloc intern și unul extern. Blocul intern va genera o excepție, iar blocul extern va trata excepția în cazul în care aceasta nu este tratată în blocul intern.

```
DECLARE
    e_donator_neeligibil EXCEPTION;

    -- Procedură externă pentru procesul de donare
    PROCEDURE proces_donare IS

        -- Procedură internă pentru verificarea eligibilității donatorului
        PROCEDURE verifica_eligibilitate IS
        BEGIN
            RAISE e_donator_neeligibil;
        EXCEPTION
            WHEN e_donator_neeligibil THEN
                DBMS_OUTPUT.PUT_LINE('Donator neeligibil.');
                RAISE;
        END verifica_eligibilitate;

    BEGIN
        verifica_eligibilitate;
    EXCEPTION
        WHEN e_donator_neeligibil THEN
            DBMS_OUTPUT.PUT_LINE('Procesul de donare nu poate continua pentru
acest donator.');
    END proces_donare;
```

```
BEGIN
    proces_donare;
EXCEPTION
    WHEN e_donator_neeligibil THEN
        DBMS_OUTPUT.PUT_LINE('Tratarea excepției la nivelul apelului
principal.');
END;
```

Descrierea Logicii:

- **Blocul Intern**: În verifica_eligibilitate, generăm intenționat o excepție (e_donator_neelegibil). Această excepție poate fi tratată direct în blocul intern sau poate fi propagată către blocul extern. În exemplul nostru, linia care ar trata excepția este comentată, permițând propagarea excepției.
- **Blocul Extern**: Dacă excepția din bloc_intern nu este tratată (sau este propagată folosind RAISE;), atunci controlul este transferat către secțiunea EXCEPTION a process_donare, unde excepția este tratată.
- **Execuția Globală**: Începem execuția prin apelarea bloc_extern din blocul anonim PL/SQL.

Prin acest exemplu, putem observa cum controlul programului și tratamentul excepțiilor sunt transferate între blocuri diferite în PL/SQL, oferind flexibilitate și control în gestionarea erorilor. În funcție de necesități, excepțiile pot fi tratate local sau pot fi propagate pentru a fi gestionate la un nivel superior, permițând o mai bună organizare a logicii de eroare în programele complexe.

XII. Exercițiul 12: Exceptii

1. Ridicarea unei Excepții Predefinite

Excepțiile predefinite sunt cele care sunt deja definite în PL/SQL, cum ar fi **NO_DATA_FOUND**, **TOO_MANY_ROWS**, etc. Acestea sunt automat ridicate de Oracle atunci când apar anumite condiții în timpul execuției programului.

Exemplu cu **NO_DATA_FOUND**:

```
DECLARE
      v_nume_donator VARCHAR2(100);
BEGIN
-- Încercăm să găsim un donator cu un ID specific
```

```
            SELECT Nume INTO v_nume_donator FROM Donatori WHERE ID_Donator =
999;

      -- Verificăm dacă numele este NULL, care este o verificare în plus
      -- fata de verificarea cand nu este niciun rand
            IF v_nume_donator IS NULL THEN
                  RAISE NO_DATA_FOUND;
            END IF;

      EXCEPTION
            WHEN NO_DATA_FOUND THEN
        -- Ridicăm o eroare personalizată pentru donatorul fără nume sau care  --
--lipsește
                  RAISE_APPLICATION_ERROR(-20001, 'Niciun donator gasit cu ID-ul
999, sau donatorul nu are nume.');
      END;
```

Explicație:

- **Interogarea SELECT**: Interogăm tabela Donatori pentru a găsi numele unui donator cu ID-ul 999.
- **Verificare pentru Nume NULL**: Dacă interogarea găsește un rând, dar numele donatorului este NULL, utilizăm RAISE pentru a ridica manual excepția NO_DATA_FOUND.
- **Blocul EXCEPTION**: Când excepția NO_DATA_FOUND este ridicată (fie automat de Oracle, fie manual de noi), folosim RAISE_APPLICATION_ERROR pentru a genera o eroare personalizată. Acest lucru permite afișarea unui mesaj clar care indică fie că nu a fost găsit niciun donator cu ID-ul specificat, fie că donatorul găsit nu are un nume.
- **Personalizarea Mesajului de Eroare**: Codul de eroare -20001 și mesajul de eroare specific oferă informații clare despre natura problemei.

2. Ridicarea unei Excepții Definite de Utilizator

Excepțiile definite de utilizator sunt cele pe care le definim în codul nostru pentru a gestiona cazuri specifice care nu sunt acoperite de excepțiile predefinite.

Exemplu cu **Excepție Definită de Utilizator**:

```
      DECLARE
            e_valoare_negativa EXCEPTION;
            v_donatie DECIMAL := -100;
```

```
	BEGIN
		IF v_donatie < 0 THEN
			RAISE e_valoare_negativa;
		END IF;
	EXCEPTION
		WHEN e_valoare_negativa THEN
			RAISE_APPLICATION_ERROR(-20002, 'Valoarea donației nu poate fi
negativă.');
	END;
```

Explicație:

- Aici, definim o excepție e_valoare_negativa.
- Dacă valoarea donației (v_donatie) este negativă, ridicăm această excepție folosind RAISE.
- În blocul EXCEPTION, capturăm excepția și folosim RAISE_APPLICATION_ERROR pentru a oferi un mesaj de eroare specific.
- Prin aceste exemple, putem vedea cum instrucțiunea RAISE poate fi utilizată pentru a semnala și a gestiona condiții excepționale, fie prin capturarea și personalizarea excepțiilor predefinite, fie prin definirea și gestionarea excepțiilor specifice aplicației. Aceasta permite scrierea unui cod mai clar și mai robust, care poate comunica mai eficient erorile și situațiile excepționale.

XIII. Concluzii

În urma acestui proiect putem trage următoarele concluzii:

- Atribute Repetitive și Transformarea lor: Am discutat despre transformarea atributelor repetitive din modelul entitate-legătură în design-ul logic al bazelor de date relaționale, evidențiind necesitatea de a separa aceste atribute pentru a menține normalizarea și integritatea datelor.
- Relații în Modelul Entitate-Legătură: Am explorat relațiile de tip 3 și descompunerea relațiilor aparent ternare în relații mulți-la-mulți, subliniind complexitatea modelării datelor în baze de date relaționale.
- Normalizarea Tabelului Relațional: Am demonstrat procesul de normalizare a bazelor de date, de la FN1 la FN3, ilustrând importanța reducerii redundanței și a îmbunătățirii integrității datelor.
- Dependența Multivaloare: Am arătat cum dependența multivaloare poate fi reprezentată în baze de date și cum diferă de dependența funcțională.

- Utilizarea Indexurilor: Am discutat despre diferite tipuri de indexuri (arbore B* și bitmap) și cum sunt folosite în interogări SQL pentru a îmbunătăți performanța.
- Vizualizările și Erorile de Inserție: Am explorat cum anumite tipuri de vizualizări (de exemplu, cele care implică agregări) sunt read-only și nu permit inserții, demonstrând limitările acestora.
- Securitatea prin Vizualizări: Am evidențiat cum vizualizările pot fi folosite pentru a proteja datele sensibile, oferind acces doar la informațiile necesare.
- Concurența și Rezultatele Variabile: Am ilustrat cum modificările datelor între interogări consecutive pot duce la rezultate diferite.
- Interblocările: Am oferit un exemplu de cum interblocările pot apărea în sistemele de baze de date și cum pot fi gestionate.
- Triggere și Condiții Complexă: Am prezentat cum triggerele pot fi folosite pentru a implementa logica de afaceri complexă și constrângeri care nu pot fi realizate prin CONSTRAINT-uri standard.
- Propagarea Excepțiilor în PL/SQL: Am discutat despre gestionarea și propagarea excepțiilor în blocuri multiple în PL/SQL, evidențiind controlul fluxului de eroare.
- Utilizarea Instrucțiunii RAISE: Am exemplificat cum instrucțiunea RAISE este folosită pentru a semnala condiții excepționale în PL/SQL, atât pentru excepțiile predefinite, cât și pentru cele definite de utilizator.