

# SQL DDL: Limbaj de definire a datelor

---

Seminar 1



# Limbajul SQL: DDL

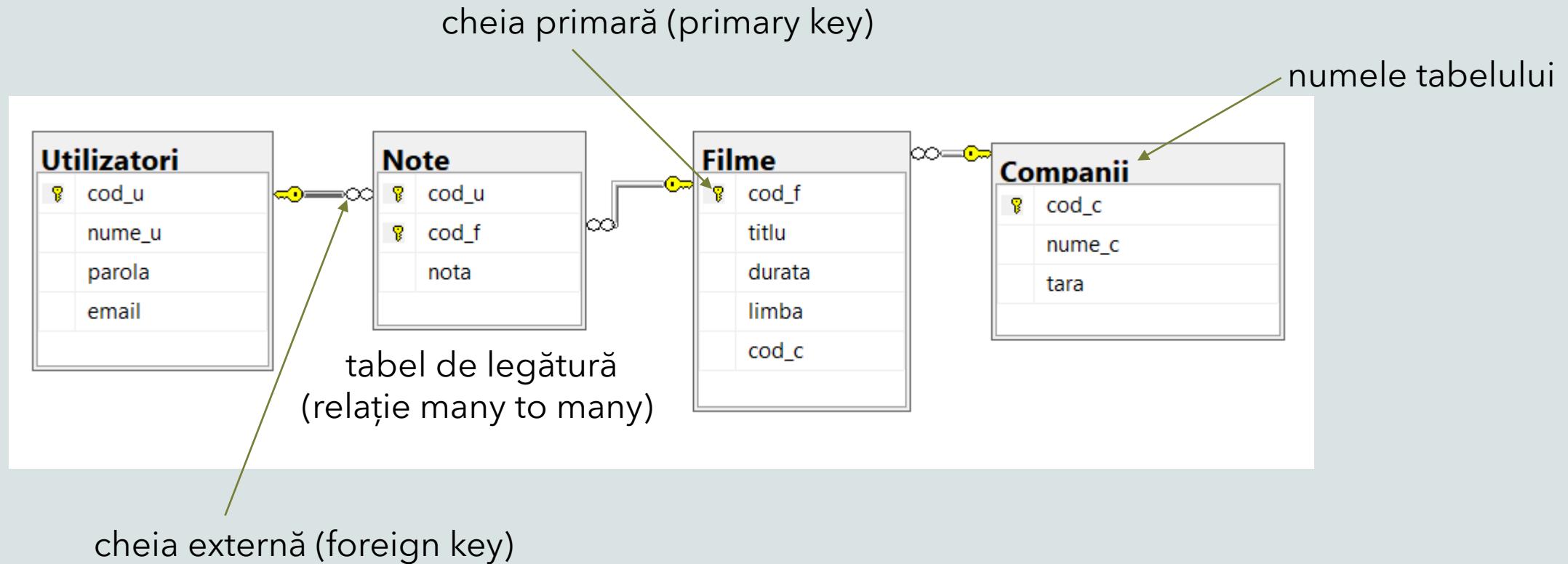
---

**DDL** (*Data Definition Language* - **Limbaj de definire a datelor**) conține comenzi pentru creare/modificare/ștergere bază de date, tabele, indecsi, stabilire relații între tabele, constrângeri

# Problemă

- Să se creeze baza de date a unei aplicații care gestionează notele date de către utilizatori unor filme. **Entitățile** de interes pentru **domeniul problemei** sunt: *Utilizatori, Companii și Filme*. Fiecare companie are un nume și o țară. O companie poate produce mai multe filme, dar fiecare film este produs de către o singură companie. Fiecare film are un titlu, o durată, o limbă și este produs de către o singură companie. Utilizatorii pot da note mai multor filme, iar un film poate primi note de la mai mulți utilizatori. Fiecare utilizator are un nume de utilizator, o parolă și o adresă de email. Numele de utilizator și adresa de email sunt unice. Pentru numele de utilizator nu se poate specifica valoarea *NULL*. Fiecare utilizator poate da fiecarui film o singură notă. Valoarea notei este cuprinsă între 1 și 10.

# Rezolvare: diagrama bazei de date relaționale



# Rezolvare: Cod Transact-SQL

- În primul rând, este necesară crearea unei noi baze de date, utilizând instrucțiunea ***CREATE DATABASE***:

--Crearea bazei de date cu numele *ProblemaFilme*

**CREATE DATABASE ProblemaFilme;**

**GO**

# Rezolvare: Cod Transact-SQL

- După ce am creat baza de date, este necesar să ne conectăm la aceasta înainte de a crea tablelele folosind instrucțiunea **USE**:
  - Conectarea la baza de date

```
USE ProblemaFilme;
```
- !!!Dacă **nu** ne conectăm la baza de date *ProblemaFilme*, instrucțiunile de creare a tabelelor ce urmează a fi executate vor crea aceste table în contextul unei alte baze de date (de obicei în baza de date *master*)

# Rezolvare: Cod Transact-SQL

- Urmează crearea primului tabel din baza de date *ProblemaFilme*, folosind instrucțiunea ***CREATE TABLE***:

--Crearea tabelului Utilizatori

```
CREATE TABLE Utilizatori
```

```
(cod_u INT PRIMARY KEY IDENTITY,
```

Constrângere PRIMARY KEY

Proprietate IDENTITY

```
nume_u VARCHAR(100) NOT NULL UNIQUE,
```

Constrângere UNIQUE

```
parola VARCHAR(100),
```

Constrângere NOT NULL

```
email VARCHAR(100)
```

Tipul de date

```
); Denumirea coloanei
```

# Observații

- Fiecare tabel va conține mai multe coloane, iar fiecare **coloană** va avea un **tip de date**
- Fiecare tabel va avea un **identificator unic**, asigurat prin definirea unei **constrângeri PRIMARY KEY** pe tabel (aceasta poate fi definită pe una sau mai multe coloane)
- În cazul în care avem o **constrângere PRIMARY KEY** definită pe o singură coloană de tipul *INT*, putem folosi **proprietatea IDENTITY** pentru a genera valori unice în mod automat pentru această coloană
- Dacă **nu** dorim să permitem introducerea de valori *NULL* pentru o coloană, aplicăm **constrângerea NOT NULL** pe coloana respectivă

# Observații

- **Constrângerea UNIQUE** asigură unicitatea valorilor la nivelul coloanei sau combinației de coloane pe care este definită
- Se pot defini mai multe **constrângeri UNIQUE** pe un tabel, dar doar o **singură constrângere PRIMARY KEY**
- Nu se pot specifica valori duplicate sau *NULL* pentru coloanele pe care a fost definită o **constrângere PRIMARY KEY**

# Rezolvare: Cod Transact-SQL

- În continuare, se va crea tabelul *Companii*, executând instrucțiunea de mai jos:

--Crearea tabelului Companii

```
CREATE TABLE Companii  
  (cod_c INT PRIMARY KEY IDENTITY(1,1),  
   nume_c NVARCHAR(100),  
   tara NVARCHAR(100)  
);
```

# Rezolvare: Cod Transact-SQL

- Apoi, se va crea tabelul *Filme*, executând instrucțiunea de mai jos:

--Crearea tabelului Filme

```
CREATE TABLE Filme  
(cod_f INT PRIMARY KEY IDENTITY,  
titlu NVARCHAR(200),  
durata TIME,  
limba NVARCHAR(100),  
cod_c INT FOREIGN KEY REFERENCES Companii(cod_c) ON UPDATE CASCADE  
ON DELETE CASCADE  
);
```

Constrângere FOREIGN KEY

# Observații

- **Constrângerea FOREIGN KEY** se folosește pentru a crea relații între tabele
- O coloană pe care este definită o constrângere **FOREIGN KEY** este conectată la o coloană pe care este definită o constrângere **PRIMARY KEY** (de obicei dintr-un alt tabel)
- Constrângerea **FOREIGN KEY** este folosită pentru a preveni acțiuni care ar distruge legăturile dintre cele două tabele, dar și pentru a împiedica introducerea unor date invalide care nu se regăsesc în coloana pe care este definită constrângerea **PRIMARY KEY**
- Nu se pot face modificări în tabelul care conține constrângerea **PRIMARY KEY** dacă aceste modificări distrug legături spre date din tabelul care conține constrângerea **FOREIGN KEY**

# Observații

- La crearea unei constrângeri **FOREIGN KEY**, se pot specifica acțiuni care să aibă loc în cazul operațiilor de **modificare** sau **ștergere** a valorilor din coloana pe care este definită constrângerea **PRIMARY KEY** din celălalt tabel:
  - NO ACTION (default)
  - CASCADE
  - SET NULL
  - SET DEFAULT

# Rezolvare: Cod Transact-SQL

- În final, se va crea tabelul de legătură (denumit Note) dintre *Filme* și *Utilizatori*, în care vor fi stocate notele date filmelor de către utilizatori:

--Crearea tabelului Note

```
CREATE TABLE Note
```

```
(cod_u INT,
```

```
cod_f INT,
```

```
nota INT,
```

```
CONSTRAINT fk_UtilizatoriNote FOREIGN KEY (cod_u) REFERENCES Utilizatori(cod_u),
```

```
CONSTRAINT fk_FilmeNote FOREIGN KEY (cod_f) REFERENCES Filme(cod_f),
```

```
CONSTRAINT pk_Note PRIMARY KEY (cod_u, cod_f) ← Constrângere PRIMARY KEY
```

```
); ← compusă din două coloane
```

Constrângere FOREIGN KEY

# Rezolvare: Cod Transact-SQL

- În enunțul problemei se specifică faptul că atât **numele utilizatorului** cât și **adresa de email** trebuie să fie **unice**
- Dacă am omis specificarea unei constrângeri **UNIQUE** la crearea tabelului, aceasta se poate crea și **ulterior** cu ajutorul instrucțiunii **ALTER TABLE**:

/\*Crearea unei constrângeri UNIQUE pe coloana email din tabelul Utilizatori după crearea tabelului\*/

ALTER TABLE Utilizatori

ADD CONSTRAINT uq\_email UNIQUE (email);

# Rezolvare: Cod Transact-SQL

- În enunțul problemei se menționează faptul că **nota** dată de către fiecare utilizator trebuie să aibă o **valoare** cuprinsă între **1** și **10**
- Această restricție se poate asigura cu ajutorul unei constrângeri **CHECK**, care poate fi impusă **ulterior** creării tabelului cu ajutorul instrucțiunii **ALTER TABLE**:

```
/*Crearea unei constrângeri CHECK pe coloana nota din tabelul Note  
după ce a fost creat tabelul*/
```

```
ALTER TABLE Note
```

```
ADD CONSTRAINT ck_nota CHECK (nota>=1 AND nota<=10);
```

# Observații

- **Constrângerea CHECK** se folosește pentru a limita intervalul de valori ce se pot introduce pentru o anumită coloană
- Se poate defini pe o coloană, iar în acest caz limitează valorile ce pot fi introduse pentru coloana respectivă
- Se pot crea și **constrângeri CHECK** care conțin restricții pentru mai multe coloane

# Modificări la nivelul structurii bazei de date

- În anumite cazuri, vor fi necesare **modificări** la nivelul structurii bazei de date
- Dacă dorim să **adăugăm** o **coloană nouă** în tabelul *Note* în care vom stoca data și ora la care a fost adăugată nota, vom folosi din nou instrucțiunea **ALTER TABLE**:  
--Adăugarea unei coloane noi în tabelul *Note*

```
ALTER TABLE Note
```

```
ADD data_si_ora_adaugarii DATETIME;
```

# Modificări la nivelul structurii bazei de date

- Dacă dorim să setăm o **valoare implicită** pentru o **coloană** (care să fie introdusă în mod automat atunci când nu se specifică o valoare pentru coloana respectivă la adăugarea unei înregistrări), vom folosi o constrângere **DEFAULT**:

--Adăugarea unei constrângeri DEFAULT pe coloana data\_si\_ora\_adaugarii

ALTER TABLE Note

```
ADD CONSTRAINT df_data_si_ora_adaugarii DEFAULT GETDATE() FOR  
data_si_ora_adaugarii;
```

- !!!În exemplul de mai sus, este folosită funcția sistem **GETDATE()** pentru generarea valorii隐含的, deoarece se dorește folosirea datei și orei curente la momentul adăugării înregistrării în tabel

# Modificări la nivelul structurii bazei de date

- Dacă dorim să modificăm tipul de date al unei coloane, vom folosi din nou instrucțiunea **ALTER TABLE**:

--Modificarea tipului de date al coloanei *titlu* din tabelul *Filme*

```
ALTER TABLE Filme
```

```
ALTER COLUMN titlu NVARCHAR(220);
```

# Modificări la nivelul structurii bazei de date

- Dacă dorim să eliminăm o constrângere dintr-un tabel, vom folosi instrucțiunea

***ALTER TABLE:***

*/\*Eliminarea constrângerii DEFAULT definită pe coloana data\_si\_ora\_adaugarii  
din tabelul Note\*/*

**ALTER TABLE Note**

**DROP CONSTRAINT df\_data\_si\_ora\_adaugarii;**

# Modificări la nivelul structurii bazei de date

- Dacă dorim să ștergem o coloană dintr-un tabel, vom folosi instrucțiunea

***ALTER TABLE:***

--Ștergerea coloanei data\_si\_ora\_adaugarii din tabelul Note

`ALTER TABLE Note`

`DROP COLUMN data_si_ora_adaugarii;`

# Modificări la nivelul structurii bazei de date

- Dacă dorim să modificăm numele bazei de date, vom folosi instrucțiunea

***ALTER DATABASE:***

--Modificarea numelui bazei de date *ProblemaFilme*

**ALTER DATABASE** ProblemaFilme

**MODIFY Name=NoteFilme;**

# Ștergerea unui tabel din baza de date

- Dacă dorim să ștergem un tabel din baza de date, vom folosi instrucțiunea

**DROP TABLE:**

--Ștergerea tabelului Note

`DROP TABLE Note;`

# Ştergerea bazei de date

- Dacă dorim să ştergem baza de date, vom folosi instrucțiunea **DROP DATABASE**:  
--Ştergerea bazei de date *NoteFilme*  

```
USE master;  
  
DROP DATABASE NoteFilme;
```
- !!!Este important să fim conectați la o altă bază de date în momentul în care ştergem baza de date *NoteFilme* de pe server, în caz contrar va apărea o eroare și ştergerea acesteia va eșua.

# Problemă propusă

- Să se creeze o bază de date care stochează informații despre un parc de distracții. Entitățile de interes pentru domeniul problemei sunt: categorii de vizitatori, vizitatori, secțiuni și atracții. Fiecare atracție din parcul de distracții are un nume, o descriere, o vârstă minimă recomandată și aparține unei singure secțiuni. Fiecare secțiune are un nume și o descriere. O secțiune poate conține mai multe atracții, dar fiecare atracție aparține unei singure secțiuni. Fiecare vizitator are un nume, o adresă de email și aparține unei singure categorii de vizitatori. Fiecare categorie de vizitatori are un nume. O categorie de vizitatori conține mai mulți vizitatori, dar fiecare vizitator aparține doar unei singure categorii. Fiecare vizitator poate vizita mai multe atracții, iar fiecare atracție poate fi vizitată de mai mulți vizitatori. Un vizitator poate da o singură notă fiecărei atracții pe care a vizitat-o. Nota este un număr real cuprins între 1 și 10.

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-database-transact-sql?view=sql-server-ver16&tabs=sqlpool>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql?view=sql-server-ver16&tabs=sqlpool>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/drop-database-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/alter-table-transact-sql?view=sql-server-ver16>

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/statements/drop-table-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/relational-databases/tables/primary-and-foreign-key-constraints?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/relational-databases/tables/specify-default-values-for-columns?view=sql-server-ver16>

# SQL DML: Limbaj de manipulare a datelor

---

Seminar 2



# Limbajul SQL: DML

---

**DML** (*Data Manipulation Language* - **Limbaj de manipulare a datelor**) conține instrucțiuni pentru inserare, actualizare, ștergere și interogare a datelor stocate într-o bază de date relațională

# Limbajul SQL: DML

---

Cele mai folosite **instructiuni DML** sunt:

**INSERT** - inserează înregistrări noi

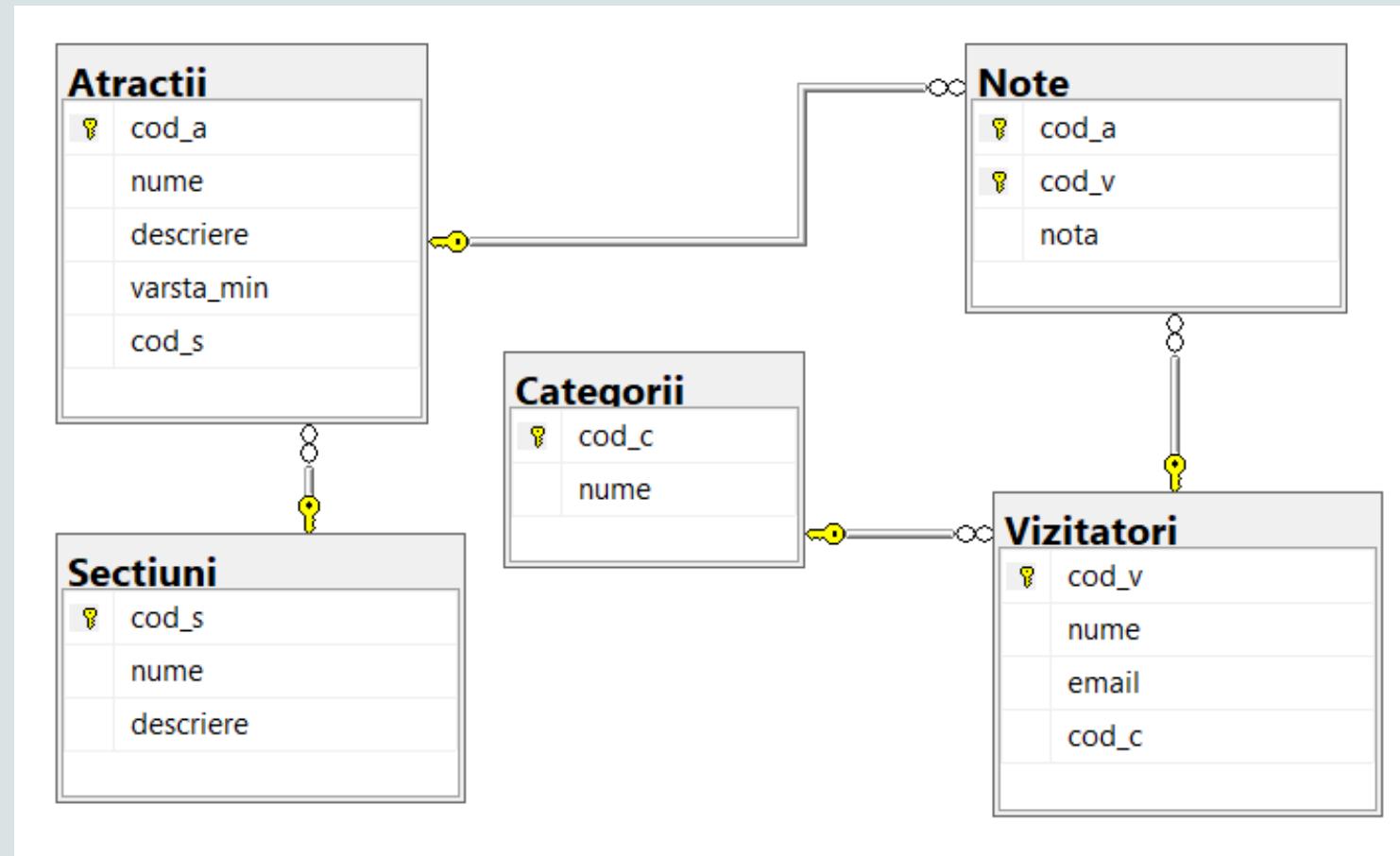
**UPDATE** - actualizează înregistrări

**DELETE** - sterge înregistrări

**SELECT** - extrage înregistrări

# Limbajul SQL - DML

- Se dă o bază de date având următoarea structură:



# Limbajul SQL - DML

- Pentru a adăuga date într-un tabel, vom folosi instrucțiunea **INSERT**
- Exemplu:

--Adăugarea unei înregistrări noi în tabelul *Sectiuni*

```
INSERT INTO Sectiuni (nume, descriere) VALUES ('sectiunea 1',
'cea mai mare sectiune');
```

--Adăugarea unei înregistrări noi în tabelul *Atractii*

```
INSERT INTO Atractii (nume, descriere, varsta_min, cod_s) VALUES
('rollercoaster', 'cel mai rapid din parc',12,1);
```

# Observații

- Specificarea coloanelor după numele tabelului este optională
- Prin specificarea coloanelor controlăm asocierile coloană-valoare, deci nu ne bazăm pe ordinea în care apar coloanele atunci când a fost creat tabelul sau când structura tabelului a fost modificată ultima dată
- Dacă **nu** specificăm o valoare pentru o coloană, **SQL Server** va verifica dacă există o valoare implicită pentru coloana respectivă iar dacă nu există și coloana nu permite *NULL* atunci inserarea **nu** va avea loc

# Limbajul SQL - DML

- Pentru a actualiza îngreșările într-un tabel, vom folosi instrucțiunea **UPDATE**
- Exemplu:
  - Modificarea unei îngreșările din tabelul *Sectiuni*

```
UPDATE Sectiuni SET descriere='cea mai veche sectiune' WHERE nume='sectiunea 1';
```
- !!! Omiterea clauzei **WHERE** va rezulta în actualizarea tuturor îngreșărilor din tabel

# Limbajul SQL - DML

- Pentru a șterge îngreșări dintr-un tabel, vom folosi instrucțiunea **DELETE**
- Exemplu:
  - Ștergerea unei îngreșări din tabelul *Sectiuni*

```
DELETE FROM Sectiuni WHERE nume='sectiunea 1';
```
- !!!Omiterea clauzei **WHERE** va rezulta în ștergerea tuturor îngreșărilor din tabel

# Limbajul SQL - DML

- Dacă dorim să extragem înregistrări din baza de date, vom folosi instrucțiunea **SELECT**, iar **rezultatul interogării** va fi afișat într-un **tabel rezultat** (*result-set*)
- Exemplu:

--Returnarea tuturor înregistrărilor din tabelul *Sectiuni*

```
SELECT * FROM Sectiuni;
```

/\*Returnarea tuturor înregistrărilor din tabelul *Sectiuni*, specificând explicit numele coloanelor\*/

```
SELECT cod_s, nume, descriere FROM Sectiuni;
```

# Limbajul SQL - DML

- Dacă dorim să extragem doar valorile distincte dintr-o coloană (sau combinație de coloane) vom folosi cuvântul cheie **DISTINCT**
- Exemplu:

--Returnarea tuturor valorilor distincte din coloana *varsta\_min* din tabelul *Atractii*

```
SELECT DISTINCT varsta_min FROM Atractii;
```

# Limbajul SQL - DML

- Dacă dorim ca o interogare să returneze doar înregistrări care îndeplinesc anumite criterii, vom folosi clauza **WHERE**
- Exemplu:

/\*Returnarea numelui și descrierii atracțiilor care au vârsta minimă recomandată egală cu 12\*/

```
SELECT nume, descriere FROM Atractii WHERE varsta_min=12;
```

/\*Returnarea numelui și vârstei minime recomandate ale atracțiilor care au numele diferit de 'Castelul Negru'\*/

```
SELECT nume, varsta_min FROM Atractii WHERE nume<>'Castelul Negru';
```

# Limbajul SQL - DML

- Dacă dorim să returnăm toate atracțiile care au vârstă minimă recomandată **mai mare sau egală cu 14**, vom executa următoarea interogare:

```
SELECT * FROM Atractii WHERE varsta_min>=14;
```

- Dacă dorim să returnăm toate atracțiile care au vârstă minimă recomandată **mai mică sau egală cu 16**, vom executa următoarea interogare:

```
SELECT * FROM Atractii WHERE varsta_min<=16;
```

# Limbajul SQL - DML

- Dacă dorim să returnăm toate atracțiile care au vârstă minimă recomandată strict mai mare decât 11 și strict mai mică decât 16, vom executa următoarea interogare:

```
SELECT * FROM Atractii WHERE varsta_min>11 AND varsta_min<16;
```

--Varianta cu operatorul BETWEEN, care specifică un interval închis de valori

```
SELECT * FROM Atractii WHERE varsta_min BETWEEN 12 AND 15;
```

- Dacă dorim să returnăm toate atracțiile care au vârstă minimă recomandată **în afara intervalului închis [14,18]**, vom executa următoarea interogare:

```
SELECT * FROM Atractii WHERE varsta_min NOT BETWEEN 14 AND 18;
```

- !!!Tipul de date al coloanei `varsta_min` este **int**

# Limbajul SQL - DML

- Dacă dorim să returnăm toate atracțiile care au vârstă minimă recomandată **egală** cu 12, 14 sau 16 vom executa următoarea interogare:

```
SELECT * FROM Atractii WHERE varsta_min IN (12,14,16);
```

- Dacă dorim să returnăm toate atracțiile care au vârstă minimă recomandată **diferită** de 12, 14 sau 16 vom executa următoarea interogare:

```
SELECT * FROM Atractii WHERE varsta_min NOT IN (12,14,16);
```

# Limbajul SQL - DML

- Pentru a specifica şabloane de căutare într-o coloană, vom folosi operatorul **LIKE**
- Dacă dorim să returnăm toate înregistrările din tabelul *Vizitatori* pentru care numele începe cu litera A, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE nume LIKE 'A%';
```

- Dacă dorim să returnăm toate înregistrările din tabelul *Vizitatori* pentru care numele se termină cu litera a, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE nume LIKE '%a';
```

# Limbajul SQL - DML

- Dacă dorim să returnăm toate înregistrările din tabelul *Vizitatori* pentru care numele conține 'ana', vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE nume LIKE '%ana%';
```

- Dacă dorim să returnăm toate înregistrările din tabelul *Vizitatori* pentru care numele se termină cu 'na' și este format din 3 caractere, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE nume LIKE '_na';
```

!!!Caracterul \_ înlocuiește un singur caracter

!!!Caracterul % înlocuiește zero sau mai multe caractere

# Limbajul SQL - DML

- Dacă dorim să returnăm toate înregistrările din tabelul *Vizitatori* pentru care numele începe cu litera A, B sau C, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE nume LIKE '[ABC]%' ;
```

- Dacă dorim să returnăm toate înregistrările din tabelul *Vizitatori* pentru care numele **nu** începe cu litera A, B sau C, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE nume LIKE '[^ABC]%' ;
```

# Limbajul SQL - DML

- Dacă dorim să extragem toate înregistrările din tabelul *Vizitatori* pentru care coloana *email* are valoarea *NULL*, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE email IS NULL;
```

- Dacă dorim să extragem toate înregistrările din tabelul *Vizitatori* pentru care coloana *email* are valoarea **diferită** de *NULL*, vom executa următoarea interogare:

```
SELECT * FROM Vizitatori WHERE email IS NOT NULL;
```

# Limbajul SQL - DML

- Dacă dorim să extragem numele categoriei, numele vizitatorului și adresa de email pentru toți vizitatorii care aparțin unei categorii, vom executa următoarea interogare:

```
SELECT C.nume AS categorie, V.nume, V.email FROM Categorii C,  
Vizitatori V WHERE C.cod_c=V.cod_c;
```

Alias pentru coloană                                  Alias pentru tabel



- Interogarea poate fi rescrisă utilizând **INNER JOIN**:

```
SELECT C.nume AS categorie, V.nume, V.email FROM Categorii C INNER  
JOIN Vizitatori V ON C.cod_c=V.cod_c;
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele categoriilor și adresa de email a vizitatorilor, **inclusiv și categoriile care nu au vizitatori asociați** (dar **nu** și vizitorii care nu aparțin unei categorii), vom executa următoarea interogare:

```
SELECT C.nume, V.email FROM Categoriile C LEFT JOIN Vizitatori V ON  
C.cod_c=V.cod_c;
```

- Dacă dorim să afișăm numele categoriilor și adresa de email a vizitatorilor, **inclusiv și vizitorii care nu aparțin unei categorii** (dar **nu** și categoriile care nu au vizitatori asociați), vom executa următoarea interogare:

```
SELECT C.nume, V.email FROM Categoriile C RIGHT JOIN Vizitatori V ON  
C.cod_c=V.cod_c;
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele categoriilor și adresa de email a vizitatorilor, **inclusiv atât vizitatorii care nu aparțin unei categorii, cât și categoriile care nu au vizitatori asociați**, vom executa următoarea interogare:

```
SELECT C.nume, V.email FROM Categorii C FULL JOIN Vizitatori V  
ON C.cod_c=V.cod_c;
```

# Limbajul SQL - DML

- Dacă dorim să realizăm un calcul pe o mulțime de valori și să returnăm o singură valoare, vom utiliza **funcții de agregare**
- Funcțiile de agregare se utilizează de obicei împreună cu clauzele **GROUP BY** și **HAVING**
- Exemple de funcții de agregare: **COUNT()**, **SUM()**, **AVG()**, **MIN()**, **MAX()**
- Dacă dorim să returnăm numărul total de înregistrări din tabelul *Categorii*, vom executa următoarea interogare:

```
SELECT COUNT(*) FROM Categorii;
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele categoriilor și numărul de vizitatori pentru fiecare categorie care are **cel puțin un vizitator**, vom executa următoarea interogare:

```
SELECT C.nume, COUNT(cod_v) nr_vizitatori FROM Categorii C INNER  
JOIN Vizitatori V ON C.cod_c=V.cod_c GROUP BY C.cod_c, C.nume;
```

- Dacă dorim să afișăm numele categoriilor și numărul de vizitatori pentru fiecare categorie, vom executa următoarea interogare:

```
SELECT C.nume, COUNT(cod_v) nr_vizitatori FROM Categorii C LEFT  
JOIN Vizitatori V ON C.cod_c=V.cod_c GROUP BY C.cod_c, C.nume;
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele atracției și media aritmetică a notelor primite pentru toate atracțiile care au primit note, vom executa următoarea interogare:

```
SELECT A.nume, AVG(nota) medie_note FROM Atractii A INNER JOIN  
Note N ON A.cod_a=N.cod_a GROUP BY A.cod_a, A.nume;
```

- Dacă dorim să afișăm numele atracției și media aritmetică a notelor primite pentru toate atracțiile care au primit note și au media aritmetică strict mai mare decât 9, vom executa următoarea interogare:

```
SELECT A.nume, AVG(nota) medie_note FROM Atractii A INNER JOIN  
Note N ON A.cod_a=N.cod_a GROUP BY A.cod_a, A.nume HAVING  
AVG(nota)>9;
```

# Subinterrogări

- O subinterrogare este o interogare încorporată într-o altă interogare
- Se poate folosi o subinterrogare în clauza **WHERE** pentru a găsi înregistrările dintr-un tabel care se potrivesc cu înregistrările din alt tabel **fără** a folosi **JOIN**
- Dacă dorim să afișăm numele tuturor categoriilor care au cel puțin un vizitator, vom executa următoarea interogare:

```
SELECT nume FROM Categoriile WHERE cod_c IN (SELECT cod_c FROM Vizitatori);
```

# Subinterrogări

- Dacă dorim, putem rescrie interogarea folosind **INNER JOIN**:

```
SELECT DISTINCT C.nume FROM Categorii C INNER JOIN Vizitatori V ON  
C.cod_c=V.cod_c;
```

- Putem rescrie interogarea folosind operatorul **EXISTS**:

```
SELECT C.nume FROM Categorii C WHERE EXISTS(SELECT * FROM  
Vizitatori V WHERE V.cod_c=C.cod_c);
```

!!!Operatorul EXISTS returnează valoarea TRUE dacă rezultatul subinterrogării conține cel puțin o înregistrare

# Subinterrogări

- O subinterrogare în clauza **WHERE** poate fi folosită și pentru a găsi înregistrările din primul tabel care **nu** au potriviri în cel de-al doilea tabel
- Dacă dorim să afișăm numele tuturor categoriilor care **nu** au niciun vizitator, vom executa următoarea interogare:

```
SELECT nume FROM Categorii WHERE cod_c NOT IN (SELECT cod_c FROM Vizitatori WHERE cod_c IS NOT NULL);
```

- Dacă dorim, putem rescrie interogarea folosind operatorul **NOT EXISTS**:

```
SELECT C.nume FROM Categorii C WHERE NOT EXISTS(SELECT * FROM Vizitatori V WHERE V.cod_c=C.cod_c);
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele atracțiilor care au primit **cel puțin o dată** nota 9, vom executa următoarea interogare:

```
SELECT nume FROM Atractii WHERE cod_a = ANY(SELECT cod_a FROM Note WHERE nota=9);
```

- Dacă dorim, putem să rescriem interogarea folosind operatorul **IN**:

```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=9);
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele atracțiilor care **nu** au primit nota 9 (dar au primit cel puțin o notă):

```
SELECT nume FROM Atractii WHERE cod_a <> ALL(SELECT cod_a FROM Note WHERE nota=9) AND cod_a IN (SELECT cod_a FROM Note);
```

- Dacă dorim, putem să rescriem interogarea folosind operatorul **NOT IN**:

```
SELECT nume FROM Atractii WHERE cod_a NOT IN (SELECT cod_a FROM Note WHERE nota=9) AND cod_a IN (SELECT cod_a FROM Note);
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele atracțiilor care **nu** au primit **niciodată** nota 9, dar au primit **cel puțin o dată** nota 10, vom executa următoarea interogare:

```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=10)
```

```
EXCEPT
```

```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=9);
```

# Limbajul SQL - DML

- Dacă dorim să afișăm numele atracțiilor care au primit **cel puțin o dată** nota 9 sau **cel puțin o dată** nota 10, vom executa următoarea interogare:

```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=10)
```

```
UNION [ALL]
```

```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=9);
```

- **!!!UNION ALL** include înregistrări duplicate
- **!!!UNION nu** include înregistrări duplicate

# Limbajul SQL - DML

- Dacă dorim să afișăm numele atracțiilor care au primit **cel puțin o dată** nota 9 și **cel puțin o dată** nota 10, vom executa următoarea interogare:

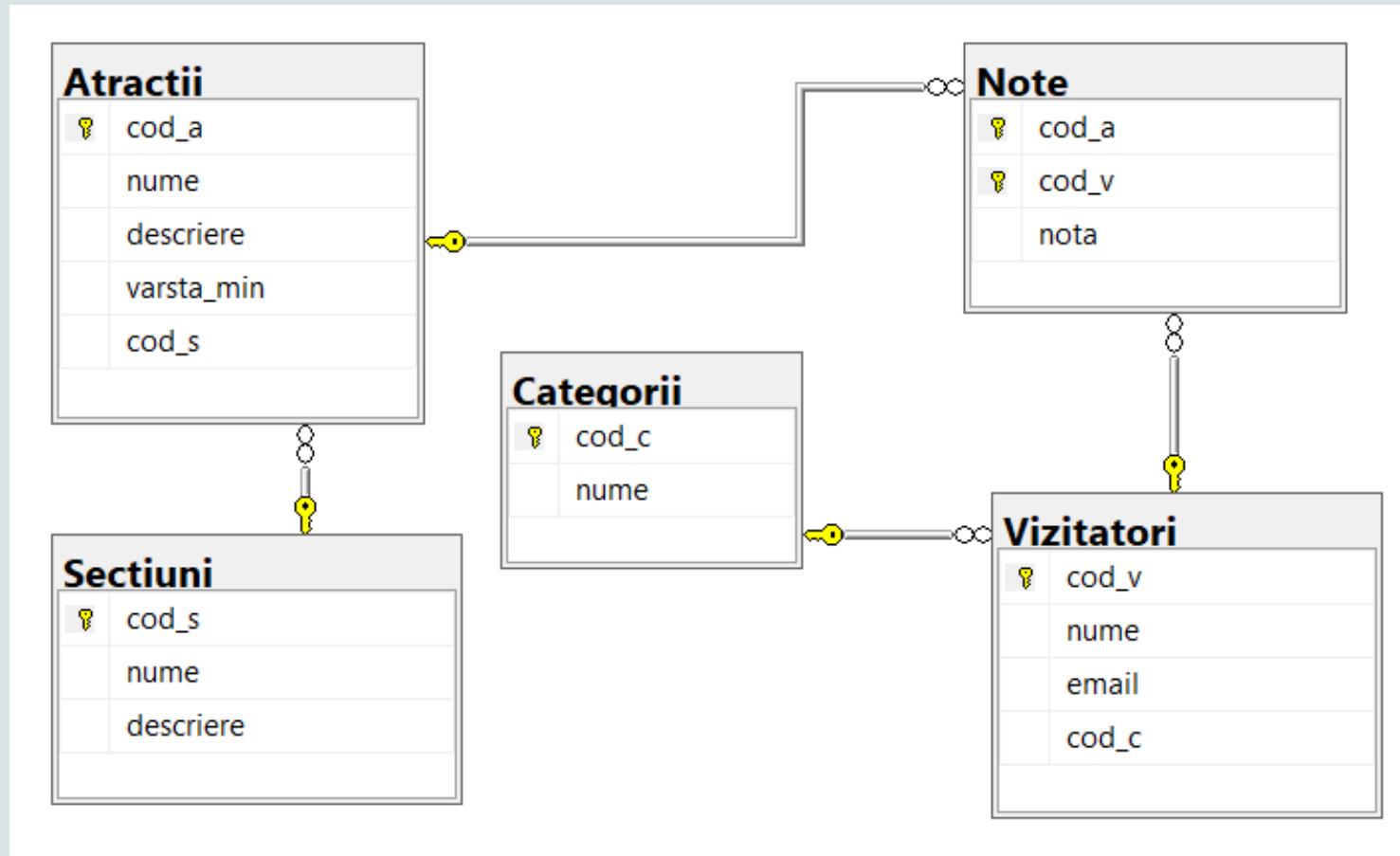
```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=10)  
INTERSECT
```

```
SELECT nume FROM Atractii WHERE cod_a IN (SELECT cod_a FROM Note WHERE nota=9);
```

- !!!Cu ajutorul operatorilor **UNION**, **INTERSECT** și **EXCEPT** se pot îmbina rezultatele a două sau mai multe interogări într-un singur *result-set* (fiecare interogare trebuie să conțină același număr de coloane, iar tipurile coloanelor trebuie să fie compatibile)

# Problemă propusă

- Se dă baza de date cu următoarea structură:



# Problemă propusă

- Cerințe:
- 1) Populați fiecare tabel cu câte 7 înregistrări
- 2) Actualizați câte o înregistrare din fiecare tabel
- 3) Ștergeți o înregistrare din tabelul *Note*
- 4) Scrieți o interogare care afișează toate înregistrările din tabelul *Categorii* al căror nume este egal cu 'pensionari' sau 'copii'
- 5) Scrieți o interogare care afișează toate înregistrările din tabelul *Sectiuni* al căror nume începe cu litera C

# Problemă propusă

- 6) Scrieți o interogare care afișează toate înregistrările din tabelul *Sectiuni* al căror nume se termină cu litera n și au cel puțin două caractere
- 7) Scrieți o interogare care afișează toți vizitatorii care nu au dat nicio notă nici unei atracții
- 8) Scrieți o interogare care afișează numele vizitatorilor, nota și numele atracției
- 9) Scrieți o interogare care afișează numele vizitatorilor și numărul de note pe care l-au dat atracțiilor (se vor include și numele vizitatorilor care nu au dat nicio notă)
- 10) Scrieți o interogare care afișează valorile distincte ale notelor date atracțiilor

# Problemă propusă

- 11) Scrieți o interogare care afișează numele secțiunii, numele și descrierea atracțiilor pentru toate secțiunile care au cel puțin o atracție asociată (se vor include și atracțiile care nu au o secțiune asociată)
- 12) Scrieți o interogare care afișează numele și vârsta minimă recomandată a atracției și numărul de note primite pentru toate atracțiile care au primit cel puțin 2 note
- 13) Scrieți o interogare care afișează numele categoriei, numele vizitatorului, nota, numele și descrierea atracției pentru toate categoriile care au numele diferit de 'adult' și au vizitatori asociați care au dat cel puțin o notă unei atracții

# Problemă propusă

- 14) Scrieți o interogare care afișează nota maximă primită de către fiecare atracție și numele atracției (se vor selecta doar acele atracții care au primit cel puțin o notă)
- 15) Scrieți o interogare care afișează nota minimă primită de către fiecare atracție și numele atracției (se vor selecta doar acele atracții care au primit cel puțin o notă)
- 16) Scrieți o interogare care afișează numele și adresa de email a vizitatorilor care nu aparțin niciunei categorii
- 17) Scrieți o interogare care afișează numele și descrierea atracțiilor care aparțin unei categorii (valoarea codului de categorie să fie diferită de *NULL*)

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/select-clause-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/select-examples-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/select-group-by-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/select-having-transact-sql?view=sql-server-ver16>

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/queries/from-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/where-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/is-null-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/like-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/between-transact-sql?view=sql-server-ver16>

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/exists-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/in-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/not-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/or-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/some-any-transact-sql?view=sql-server-ver16>

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/all-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/and-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-union-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-except-and-intersect-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/insert-transact-sql?view=sql-server-ver16>

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/queries/update-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/delete-transact-sql?view=sql-server-ver16>

# Proceduri stocate. Execuție dinamică. Limbaj de control al fluxului

---

Seminar 3



# Procedura stocată

---

- este o un **grup de instrucțiuni SQL** compilate într-un singur plan de execuție
- **acceptă** parametri de intrare și **poate returna** multiple valori ca parametri de ieșire
  - **contine** instrucțiuni de programare care efectuează operațiuni în baza de date, inclusiv **apeluri de proceduri**
  - **returnează** o valoare de stare care indică succes sau eroare



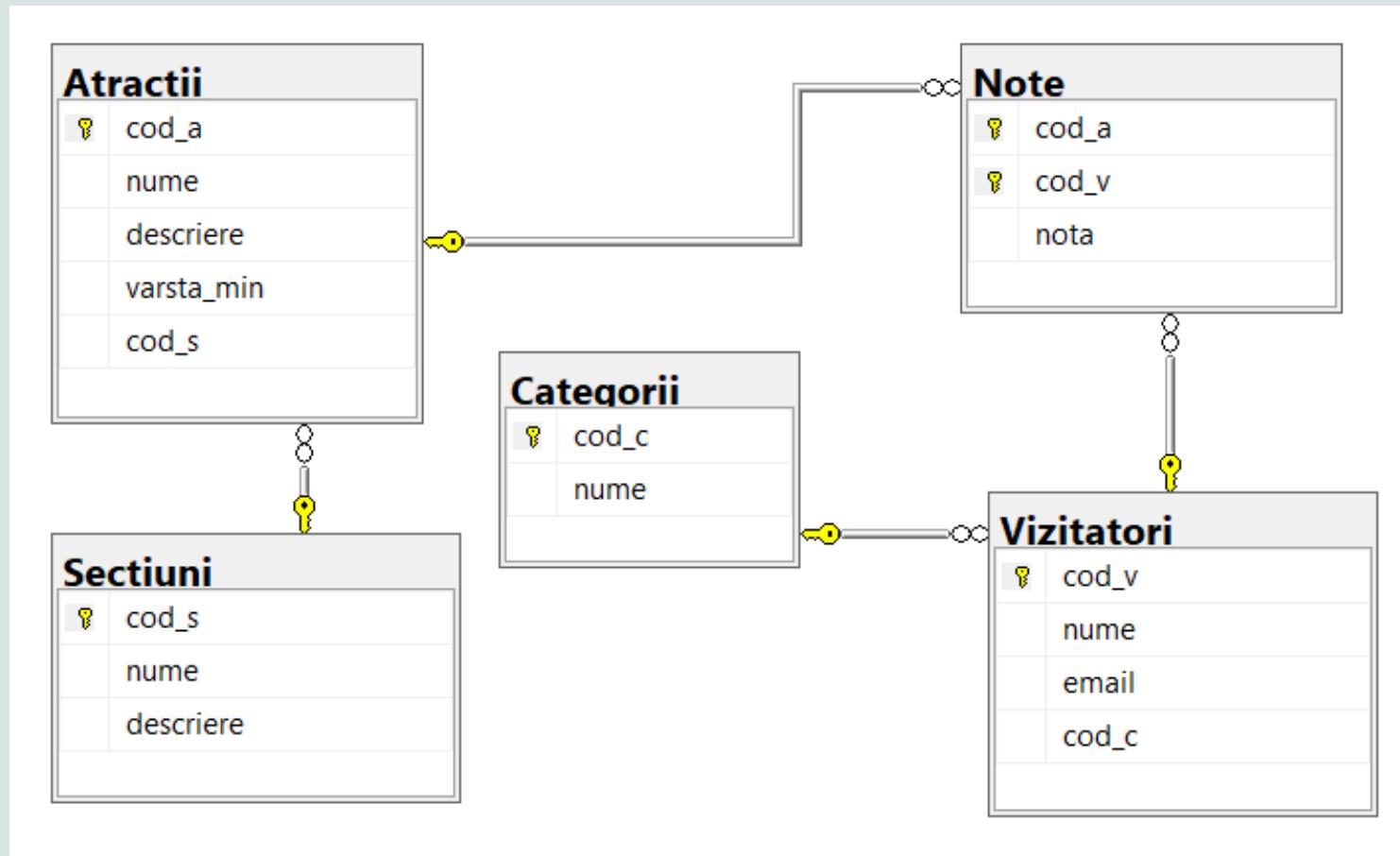
# Avantajele procedurilor stocate

---

- 1. Reducerea traficului pe rețea**
- 2. Control mai bun al securității**
- 3. Reutilizarea codului**
- 4. Întreținere simplificată**
- 5. Performanță îmbunătățită**
- 6. Posibilitatea de a încorpora cod pentru  
tratarea erorilor direct în interiorul procedurii  
stocate**

# Proceduri stocate

- Se dă o bază de date având următoarea structură:



# Proceduri stocate

- Următoarea procedură stocată va adăuga o constrângere de **valoare implicită** pentru coloana *varsta\_min* din tabelul *Atractii*:

```
CREATE PROCEDURE AdaugaConstrangereDefault
AS
BEGIN
    ALTER TABLE Atractii
        ADD CONSTRAINT df_varsta_min DEFAULT 12 FOR varsta_min;
END;
```

- Procedura stocată creată anterior se va apela în modul următor:

```
EXEC AdaugaConstrangereDefault;
```

# Proceduri stocate

- Următoarea procedură stocată va returna numele, descrierea și vârsta minimă recomandată pentru toate înregistrările din tabelul *Atractii* care au vârsta minimă recomandată egală cu cea furnizată prin intermediul **parametrului de intrare**:

```
CREATE PROCEDURE RetornaazaAtractiiCuVarstaMin @varsta_min INT  
AS  
BEGIN  
SELECT nume, descriere, varsta_min FROM Atractii  
WHERE varsta_min=@varsta_min;  
END;
```

# Proceduri stocate

- Procedura stocată creată anterior se va apela în modul următor:

```
EXEC ReturneazaAtractiiCuVarstaMin 12;
```

SAU

```
EXEC ReturneazaAtractiiCuVarstaMin @varsta_min=12;
```

# Proceduri stocate

- Putem **modifica** definiția procedurii stocate astfel încât să returneze prin intermediul unui **parametru de ieșire** numărul de atracții care au vârstă minimă recomandată egală cu cea furnizată prin intermediul **parametrului de intrare**:

```
ALTER PROCEDURE ReturneazaAtractiiCuVarstaMin @varsta_min INT,  
@nr_atractii INT OUTPUT  
AS  
BEGIN  
SELECT @nr_atractii=COUNT(*) FROM Atractii WHERE varsta_min=@varsta_min;  
END;
```

# Proceduri stocate

- Noua variantă a procedurii stocate (care conține un parametru de ieșire) va fi apelată în modul următor:

--Se declară o variabilă locală

```
DECLARE @nratractii AS INT;
```

--Se initializează variabila locală

```
SET @nratractii=0;
```

--Se apelează procedura stocată

```
EXEC ReturneazaAtractiiCuVarstaMin 12,@nr_atractii=@nratractii OUTPUT;
```

--Se afișează pe ecran valoarea parametrului de ieșire, stocată în variabila locală

```
PRINT @nratractii;
```

# Generarea mesajelor de eroare cu ajutorul RAISERROR

- Putem folosi **RAISERROR** pentru a genera mesaje de eroare și pentru a iniția procesarea erorilor pentru sesiune
- **Sintaxa:**

```
RAISERROR ( { msg_id | msg_str | @local_variable} {, severity, state} )
```

- **RAISERROR** poate referi un mesaj definit de utilizator stocat în **sys.messages catalog view** sau poate **construi un mesaj în mod dinamic**
- **Severity** reprezintă **nivelul de severitate** definit de utilizator asociat mesajului (utilizatorii pot specifica un nivel de severitate între 0 și 18)

# Generarea mesajelor de eroare cu ajutorul RAISERROR

- Putem modifica definiția procedurii stocate astfel încât să genereze un mesaj de eroare dacă nu se găsește nicio atracție pentru vârsta minimă recomandată furnizată prin intermediul parametrului de intrare:

```
ALTER PROCEDURE ReturneazaAtractiiCuVarstaMin @varsta_min INT,  
@nr_atractii INT OUTPUT  
AS  
BEGIN  
SELECT @nr_atractii=COUNT(*) FROM Atractii WHERE varsta_min=@varsta_min;  
IF (@nr_atractii=0)  
    RAISERROR('Nu a fost returnata nicio atractie!',16,1);  
END;
```

# Ștergerea procedurilor stocate

- Procedurile stocate se pot **șterge** cu ajutorul instrucțiunii **DROP PROCEDURE**:

--Exemplu de ștergere a unei proceduri stocate

```
DROP PROCEDURE RetornaazaAtractiiCuVarstaMin;
```

--Exemplu de ștergere a unei proceduri stocate în care numele procedurii este prefixat cu numele schemei (în acest caz este vorba de schema default, *dbo*)

```
DROP PROCEDURE dbo.ReturnazaAtractiiCuVarstaMin;
```

# Variabile globale

- În Microsoft SQL Server se pot utiliza **variabile globale**, care **nu** trebuie declarate (ele fiind **funcții sistem**)
- Numele variabilelor globale din Microsoft SQL Server începe cu **@@**
- Server-ul menține în permanentă valorile variabilelor globale, care reprezintă informații specifice server-ului sau sesiunii curente

# Exemple de variabile globale

- **@@ERROR** - conține numărul celei mai recente erori Transact-SQL (0 indică faptul că nu s-a produs nicio eroare)
- **@@IDENTITY** - conține valoarea câmpului IDENTITY al ultimei înregistrări inserate
- **@@ROWCOUNT** - conține numărul de înregistrări afectate de cea mai recentă instrucțiune
- **@@SERVERNAME** - conține numele instanței
- **@@SPID** - conține ID-ul de sesiune al procesului de utilizator curent
- **@@VERSION** - conține informații în legătură cu sistemul și compilarea curentă a server-ului instalat

# SET NOCOUNT

- **SET NOCOUNT ON** - oprește returnarea mesajului cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată
- **SET NOCOUNT OFF** - mesajul cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată va fi returnat ca parte din result set
- Variabila globală **@@ROWCOUNT** va fi modificată întotdeauna
- Dacă NOCOUNT este setat pe ON, performanța procedurilor stocate care conțin bucle Transact-SQL sau instrucțiuni care nu returnează multe date se va îmbunătăți (traficul pe rețea este redus)

# Execuție dinamică

- **EXEC** poate fi folosit pentru a executa cod SQL în mod dinamic
- **EXEC** acceptă ca parametru un sir de caractere și execută codul SQL din interiorul acestuia

--Exemplu de procedură stocată care execută cod SQL în mod dinamic

```
CREATE PROCEDURE RetornazaDateDinTabel @nume_tabel VARCHAR(100)
AS
BEGIN
    EXEC('SELECT * FROM '+@nume_tabel);
END;
```

# Execuție dinamică

- Dezavantajele principale ale execuției dinamice sunt problemele de performanță și posibilele probleme de securitate
- În locul instrucțiunii **EXEC** putem folosi procedura stocată **sp\_executesql**
- Procedura stocată **sp\_executesql** evită o mare parte din problemele generate de **SQL injection** și este uneori mult mai rapidă decât **EXEC**
- Spre deosebire de **EXEC**, **sp\_executesql** suportă doar siruri de caractere **Unicode** și **permite parametri de intrare și de ieșire**

# Execuție dinamică

- Exemplu de utilizare a procedurii stocate **sp\_executesql**:

```
DECLARE @sql NVARCHAR(100);

SET @sql=N'SELECT nume, descriere FROM Sectiuni WHERE nume<>@nume;';

EXEC sp_executesql @sql, N'@nume AS VARCHAR(100)', @nume='sectiunea 1';
```

# Limbaj de control al fluxului

- Transact-SQL oferă un set de cuvinte speciale, numit **limbaj de control al fluxului** care pot fi folosite pentru a controla **fluxul execuției** instrucțiunilor Transact-SQL, al blocurilor de instrucțiuni, al funcțiilor definite de utilizator și al procedurilor stocate
- În lipsa limbajului de control al fluxului, instrucțiunile Transact-SQL se execută în ordine secvențială
- BEGIN ... END - delimită un grup de instrucțiuni SQL care se execută împreună
- Blocurile BEGIN ... END pot fi încorporate

# Limbaj de control al fluxului

- **Sintaxa:**

BEGIN

{ sql\_statement | sql\_block}

END

- RETURN - ieșe necondiționat dintr-o interogare sau dintr-o procedură stocată
- Poate fi folosit în orice punct pentru a ieși dintr-o procedură, batch sau bloc de instrucțiuni

- **Sintaxa:**

RETURN [integer\_expression]

- Se poate folosi pentru a returna **status codes** - procedurile stocate returnează zero (success) sau o valoare întreagă diferită de zero (failure)

# Limbaj de control al fluxului

- Exemplu de procedură stocată care returneză **status codes**:

```
CREATE PROCEDURE VerificaVarstaMin @cod_a INT  
AS  
BEGIN  
IF((SELECT varsta_min FROM Atractii WHERE cod_a=@cod_a)=12)  
    RETURN 1;  
ELSE  
    RETURN 2;  
END;
```

# Limbaj de control al fluxului

- Procedura stocată creată anterior se va apela în modul următor:

```
DECLARE @status INT;  
  
EXEC @status=VerificaVarstaMin 1;  
  
SELECT 'Status'=@status;
```

# Limbaj de control al fluxului

- IF ... ELSE - condiționează execuția unei instrucțiuni SQL sau a unui bloc de instrucțiuni SQL
- **Sintaxa:**

```
IF Boolean_expression  
{ sql_statement | statement_block }  
[ ELSE  
{ sql_statement | statement_block } ]
```

# Limbaj de control al fluxului

- WHILE - setează o condiție pentru execuția repetată a unei instrucțiuni SQL sau a unui bloc de instrucțiuni SQL
- **Sintaxa:**

```
WHILE Boolean_expression  
{ sql_statement | statement_block | BREAK | CONTINUE }
```

# Limbaj de control al fluxului

- BREAK - ieșe din cea mai interioară buclă WHILE sau dintr-o instrucțiune IF... ELSE din interiorul unei bucle WHILE
- CONTINUE - cauzează reînceperea buclei WHILE, ignorând toate instrucțiunile care apar după CONTINUE
- GOTO - execută un salt în execuție la o porțiune din cod marcată printr-un *label*

Label: -- some SQL statements

```
GOTO Label;
```

# Limbaj de control al fluxului

- WAITFOR - blochează execuția unui batch, tranzacție sau procedură stocată până când un interval de timp sau timp specificat este atins sau o instrucțiune specificată modifică sau returnează cel puțin o înregistrare
- **Sintaxa:**

WAITFOR

```
{DELAY 'time_to_pass' | TIME 'time_to_execute' |
[ ( receive_statement ) | ( get_conversation_group_statement ) ]
[ , TIMEOUT timeout ]}
```

# Limbaj de control al fluxului

- În funcție de nivelul activității pe server, **timpul de așteptare** poate **varia**, deci poate fi mai mare decât timpul specificat în **WAITFOR**
- Exemple:

-- Execuția continuă la 22:00

```
WAITFOR TIME '22:00';
```

-- Execuția continuă peste 3 ore

```
WAITFOR DELAY '03:00:00';
```

# Limbaj de control al fluxului

- THROW - aruncă o excepție și transferă execuția unui bloc CATCH dintr-o construcție TRY ... CATCH
- **Severitatea excepției** este mereu setată pe valoarea **16**
- **Sintaxa:**

```
THROW [ { error_number | @local_variable },  
        { message | @local_variable },  
        { state | @local_variable } ] [ ; ]
```

- Exemplu:

```
THROW 50002, 'Înregistrarea nu există!', 1;
```

# Limbaj de control al fluxului

- TRY ... CATCH implementează tratarea erorilor pentru Transact-SQL și captează toate erorile de execuție care au o severitate mai mare decât 10 și care nu închid conexiunea la baza de date
- **Sintaxa:**

```
BEGIN TRY  
  
    { sql_statement | statement_block }  
  
END TRY  
  
BEGIN CATCH  
  
    [ { sql_statement | statement_block } ]  
  
END CATCH  
  
[ ; ]
```

# Limbaj de control al fluxului

- În interiorul unui bloc **CATCH** pot fi folosite următoarele funcții sistem pentru a obține informații despre eroarea care a cauzat execuția blocului **CATCH**:
  - **ERROR\_NUMBER()** - returnează numărul erorii
  - **ERROR\_SEVERITY()** - returnează severitatea erorii
  - **ERROR\_STATE()** - returnează *error state number*
  - **ERROR\_PROCEDURE()** - returnează numele procedurii stocate sau al trigger-ului în care a avut loc eroarea
  - **ERROR\_LINE()** - returnează numărul liniei care a cauzat eroarea
  - **ERROR\_MESSAGE()** - returnează mesajul erorii

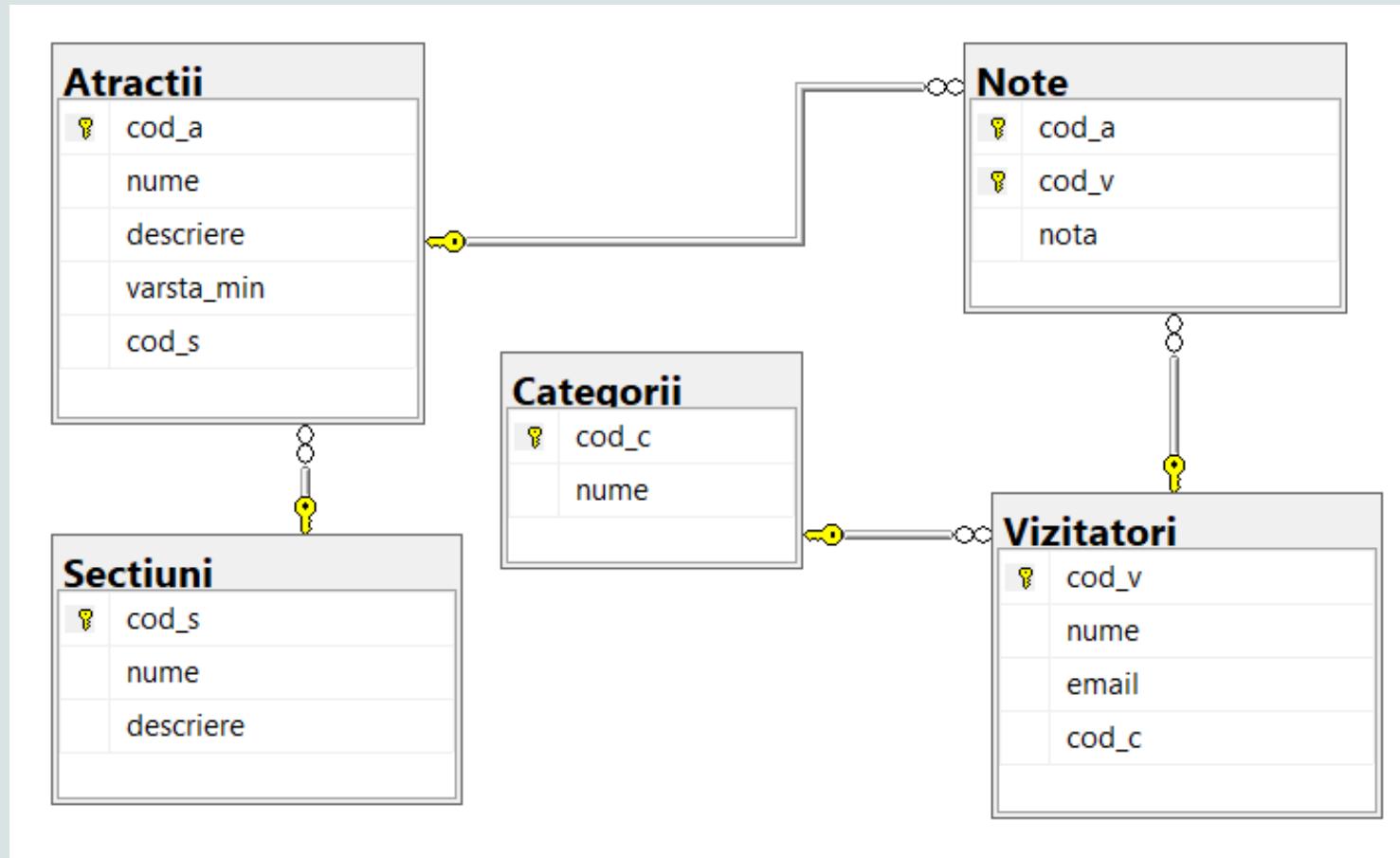
# Limbaj de control al fluxului

## Mesaje de eroare

- **Error number** (numărul erorii) este o valoare întreagă cuprinsă între 1 și 49999  
Pentru erorile custom (definite de către utilizator) valoarea este cuprinsă între 50000 și 2147483647
- **Error severity** (severitatea erorii) - 26 de nivele de severitate  
Erorile care au nivelul de severitate  $\geq 16$  sunt înregistrate în *error log* în mod automat  
Erorile care au nivelul de severitate cuprins între 20 și 25 sunt fatale și închid conexiunea
- Error message (mesajul erorii) - NVARCHAR(2048)

# Problemă propusă

- Se dă baza de date cu următoarea structură:



# Problemă propusă

- **Cerințe**
- 1) Să se creeze o procedură stocată care inserează o secțiune nouă în tabelul *Sectiuni*. Procedura va avea doi parametri de intrare: numele și descrierea secțiunii.
- 2) Să se creeze o procedură stocată care inserează o categorie nouă în tabelul *Categorii*. Procedura va avea un parametru de intrare: numele categoriei. Dacă există deja categoria dată ca parametru, se va afișa un mesaj pe ecran și categoria nu va fi adăugată încă o dată.
- 3) Să se creeze o procedură stocată care inserează o atracție nouă în tabelul *Atractii*. Procedura va avea 4 parametri de intrare: numele, descrierea, vârstă minimă recomandată și numele secțiunii în care se găsește atracția. Dacă nu există secțiunea dată ca parametru, aceasta va fi adăugată în tabelul *Sectiuni*.

# Problemă propusă

- 4) Să se creeze o procedură stocată care verifică dacă există un vizitator căruia îi corespunde adresa de email dată ca parametru de intrare. Dacă vizitatorul există, se va returna codul acestuia, dacă nu există se va genera un mesaj de eroare.
- 5) Să se creeze o procedură stocată care inserează o notă în tabelul *Note*. Procedura stocată va avea 3 parametri de intrare: codul atracției, codul vizitatorului și nota. Înainte de inserare, se va verifica dacă există codul atracției și codul vizitatorului în tabelele *Atractii* și *Vizitatori*. Dacă nu există, se va genera un mesaj de eroare. Dacă există, se verifică dacă nota este cuprinsă între 1 și 10. Se va returna un mesaj de eroare dacă nota nu are o valoare validă.

# Problemă propusă

- 6) Să se creeze o procedură stocată care actualizează adresa de email a unui vizitator din tabelul *Vizitatori*. Procedura stocată primește 2 parametri de intrare: codul vizitatorului și noua adresă de email.
- 7) Să se creeze o procedură stocată care returnează numele vizitatorului, adresa de email și numărul total de note pentru toți vizitorii care au dat cel puțin o notă unei atracții.
- 8) Să se creeze o procedură stocată care șterge o atracție din tabelul *Atractii*. Procedura stocată va avea un singur parametru de intrare: numele atracției. Înainte de ștergerea atracției, se va verifica dacă există note pentru acea atracție. În cazul în care există note date acelei atracții, se va afișa pe ecran un mesaj corespunzător și nu se va șterge atracția respectivă.

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/execute-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-executesql-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/set-nocount-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/variables-transact-sql?view=sql-server-ver16>

# Bibliografie

- <https://learn.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/functions/system-functions-transact-sql?view=sql-server-ver16>



# Functii definite de utilizator. View. Trigger. Cursoare

---

Seminar 4



# Funcții definite de către utilizator

---

- Microsoft SQL Server oferă posibilitatea de a crea funcții care pot fi mai apoi folosite în interogări
- Funcțiile definite de utilizator pot avea parametri de intrare și returnează o valoare
- În Microsoft SQL Server sunt disponibile trei tipuri de funcții definite de utilizator:
  - Funcții **scalare**
  - Funcții **inline table-valued**
  - Funcții **multi-statement table-valued**



# Functii definite de către utilizator

---

- Funcțiile scalare returnează o singură valoare
- Sintaxa pentru crearea unei funcții scalare:

```
CREATE FUNCTION scalar_function_name([@param_1  
datatype, @param_2 datatype, ... @param_n datatype])  
RETURNS datatype AS  
BEGIN  
    -- SQL Statements  
    RETURN value;  
END;
```



# Functii definite de către utilizator

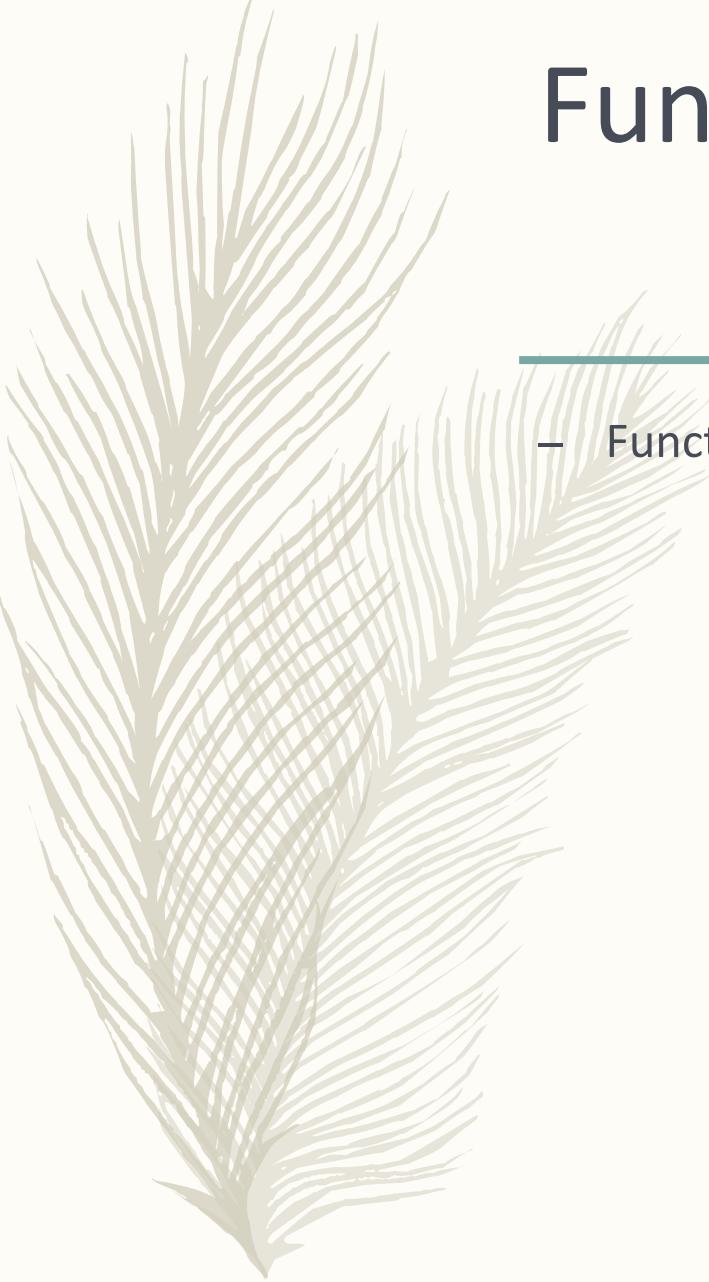
---

- Sintaxa pentru modificarea unei funcții scalare:

```
ALTER FUNCTION scalar_function_name([@param_1 datatype,  
@param_2 datatype, ... @param_n datatype])  
RETURNS datatype AS  
BEGIN  
    -- SQL Statements  
    RETURN value;  
END;
```

- Sintaxa pentru ștergerea unei funcții scalare:

```
DROP FUNCTION scalar_function_name;
```



# Functii definite de către utilizator

---

- Funcție care returnează numărul de cursuri care au un anumit număr de credite:

```
CREATE FUNCTION ufNrCrediteCursuri(@nrcredite INT)
RETURNS INT AS
BEGIN
DECLARE @nrcursuri INT=0;
SELECT @nrcursuri=COUNT(*) FROM Cursuri WHERE
nrcredite=@nrcredite;
RETURN @nrcursuri;
END;
```

---

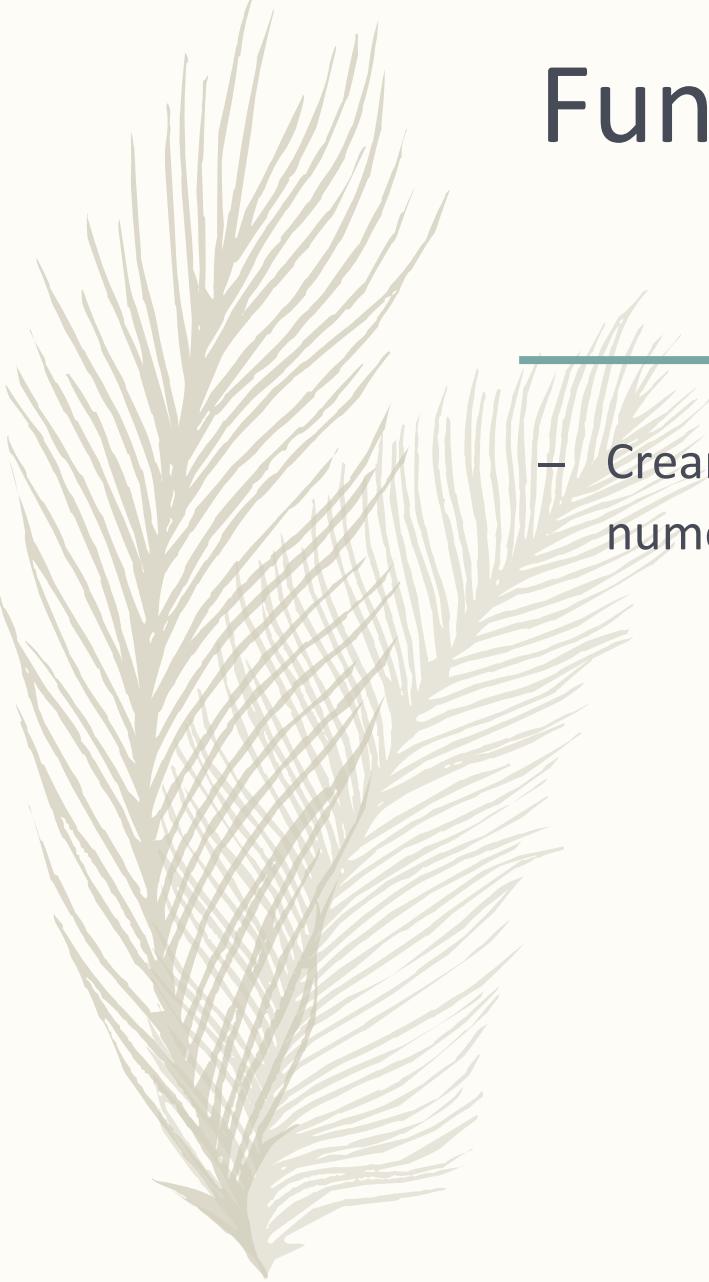
```
PRINT dbo.ufNrCrediteCursuri(6);
```



# Functii definite de către utilizator

---

- Funcțiile definite de utilizator de **tip inline table-valued** returnează un tabel în locul unei singure valori
- Pot fi folosite oriunde poate fi folosit un tabel, de obicei în clauza *FROM* a unei interogări
- O funcție definită de utilizator de tip **inline table-valued** conține o singură instrucțiune SQL
- O funcție definită de utilizator de tipul **multi-statement table-valued** returnează un tabel și conține mai multe instrucțiuni SQL, spre deosebire de o funcție **inline table-valued** care conține o singură instrucțiune SQL



# Functii definite de către utilizator

---

- Crearea unei funcții care primește ca parametru numărul de credite și returnează numele cursurilor cu acel număr de credite:

```
CREATE FUNCTION ufNumeCursuri(@nrcredite INT)
```

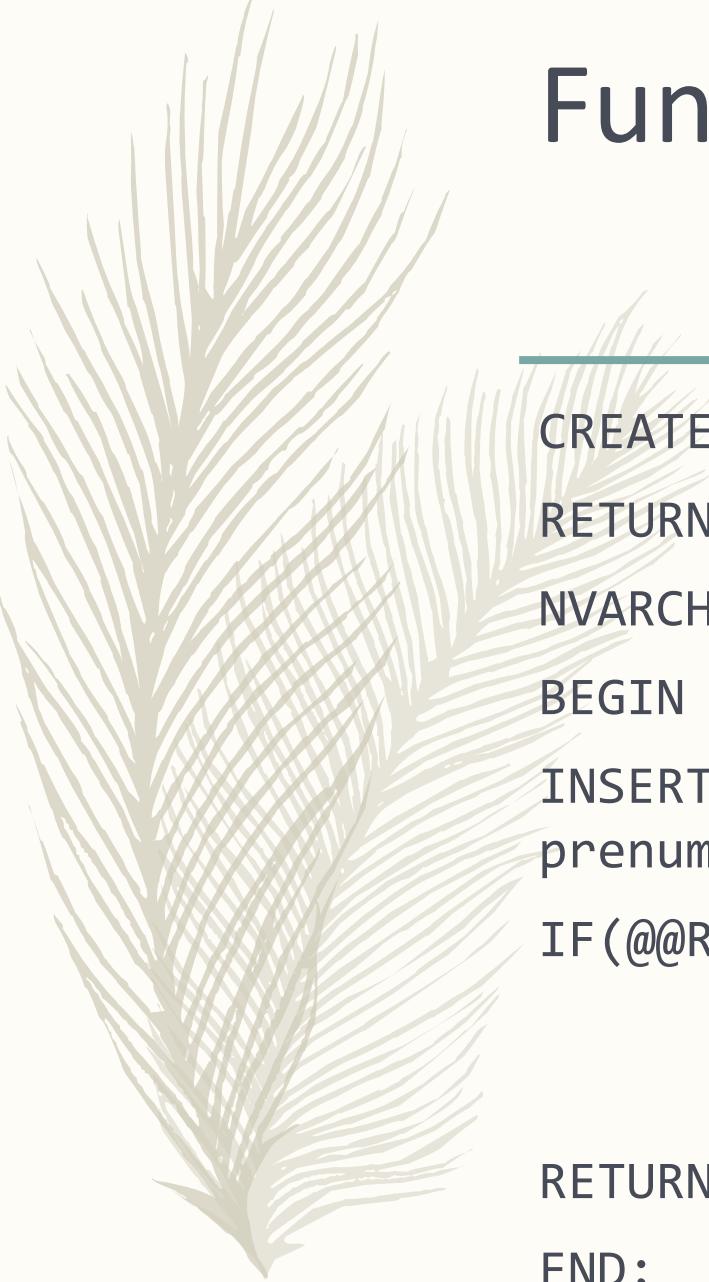
```
RETURNS TABLE
```

```
AS
```

```
RETURN SELECT nume FROM Cursuri WHERE  
nrcredite=@nrcredite;
```

---

```
SELECT * FROM dbo.ufNumeCursuri(6);
```



# Functii definite de către utilizator

---

```
CREATE FUNCTION ufPersoaneLocalitate(@localitate NVARCHAR(30))
RETURNS @PersoaneLocalitate TABLE (nume NVARCHAR(40), prenume
NVARCHAR(40)) AS
BEGIN
    INSERT INTO @PersoaneLocalitate (nume, prenume) SELECT nume,
prenume FROM Persoane WHERE localitate=@localitate;
    IF(@@ROWCOUNT=0)
        INSERT INTO @PersoaneLocalitate (nume, prenume) VALUES
(N'Nicio persoană din această localitate',N '');
    RETURN;
END;
```



# Funcții definite de către utilizator

---

- Funcția **multi-statement table-valued** definită anterior primește ca parametru o valoare ce reprezintă localitatea și returnează un tabel cu numele și prenumele persoanelor care au localitatea egală cu valoarea transmisă ca parametru
- În cazul în care nu este returnată nicio înregistrare care să corespundă localității transmise ca parametru, în variabila de tip tabel se va insera o înregistrare care conține un mesaj corespunzător
- Exemplu de apel al funcției:

```
SELECT * FROM dbo.ufPersoaneLocalitate(N'Sibiu');
```



# View

---

- Un **view** este un tabel virtual bazat pe result set-ul unei interogări
- Conține înregistrări și coloane ca un tabel real
- Un **view** nu stochează date, stochează definiția unei interogări
- Cu ajutorul unui **view** putem prezenta date din mai multe tabele ca și cum ar veni din același tabel
- De fiecare dată când un **view** este interogat, motorul bazei de date va recrea datele folosind **instrucțiunea SELECT** specificată la crearea **view-ului**, astfel că un **view** va prezenta întotdeauna **date actualizate**
- **Numele coloanelor** dintr-un **view** trebuie să fie **unice** (în cazul în care avem două coloane cu același nume provenind din tabele diferite, putem folosi un **alias** pentru una dintre ele)



# View

---

- Sintaxa pentru crearea unui view:

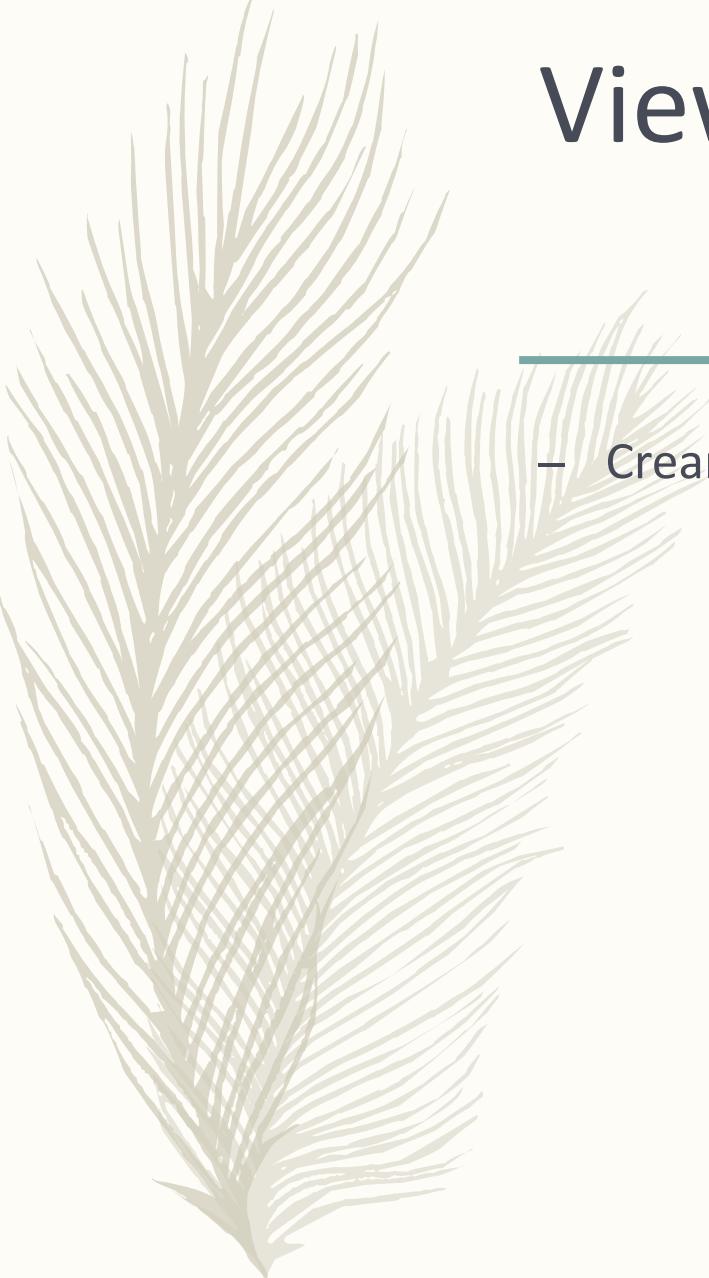
```
CREATE VIEW view_name AS  
<select_statement>;
```

- Sintaxa pentru modificarea unui view:

```
ALTER VIEW view_name AS  
<select_statement>;
```

- Sintaxa pentru ștergerea unui view:

```
DROP VIEW view_name;
```

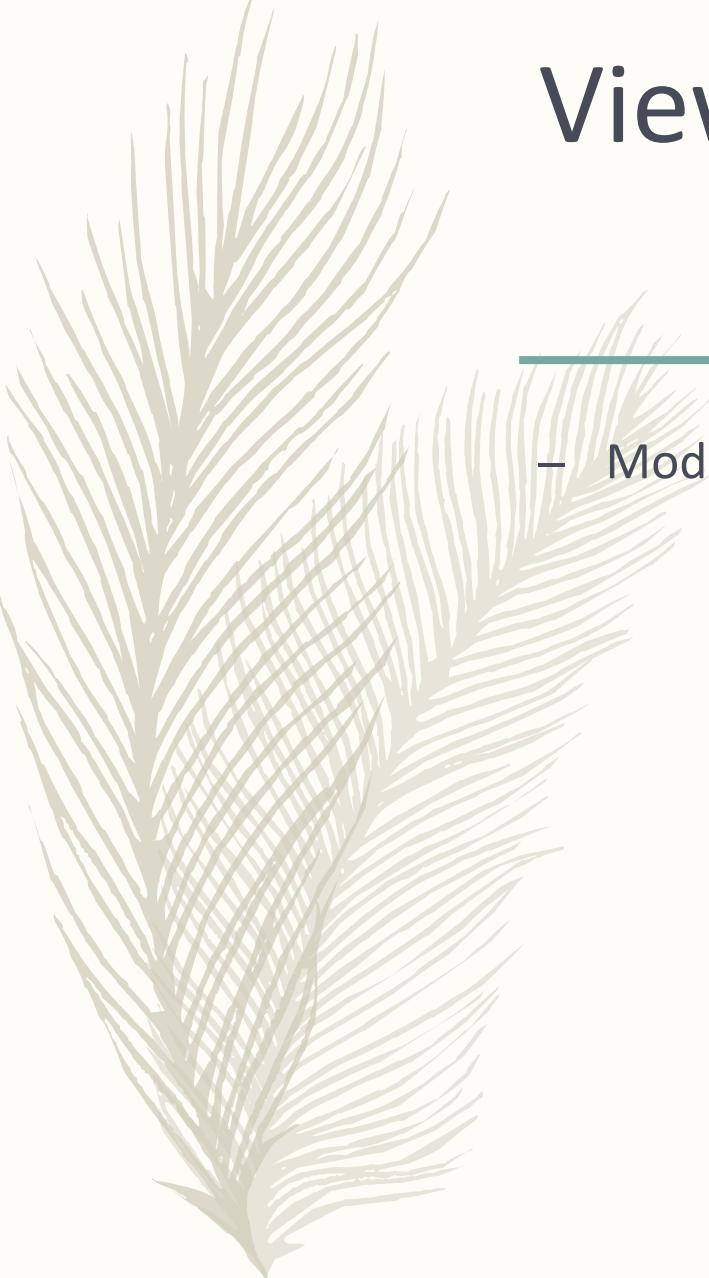


# View

---

- Crearea unui view care conține date din două tabele, *Categorii și Produse*:

```
CREATE VIEW vw_Produse AS  
SELECT P.nume, P.preț,  
C.nume AS categorie  
FROM Produse AS P  
INNER JOIN Categorii AS C  
ON P.id_cat=C.id_cat;
```



# View

---

- Modificarea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
ALTER VIEW vw_Produse AS  
SELECT P.nume, P.preț, P.cantitate,  
C.nume AS categorie  
FROM Produse AS P  
INNER JOIN Categorii AS C  
ON P.id_cat=C.id_cat;
```



# View

---

- Interogarea unui view care conține date din două tabele, *Categorii și Produse*:

```
SELECT nume, preț, cantitate, categorie
```

```
FROM vw_Produse;
```

SAU

```
SELECT * FROM vw_Produse;
```

- Exemplu de ștergere a unui view:

```
DROP VIEW vw_Produse;
```

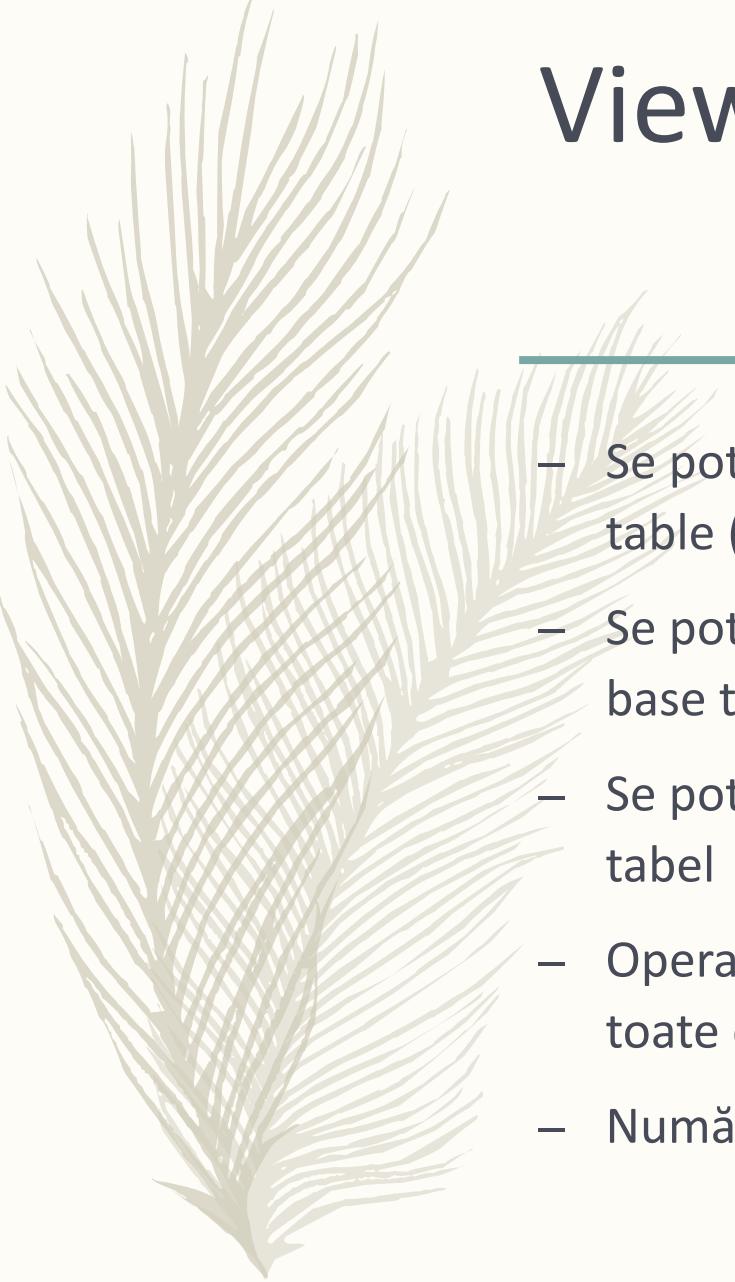


# View

---

- Nu se poate folosi clauza ORDER BY în definiția unui view (decât dacă se specifică în definiția view-ului clauza TOP, OFFSET sau FOR XML)
- Dacă dorim să ordonăm înregistrările din result set, putem folosi clauza ORDER BY atunci când interogăm view-ul
- Pentru a afișa definiția unui view, putem folosi funcția **OBJECT\_DEFINITION** sau procedura stocată **sp\_helptext**:

```
PRINT OBJECT_DEFINITION (OBJECT_ID('schema_name.view_name'));  
EXEC sp_helptext 'schema_name.view_name';
```



# View

---

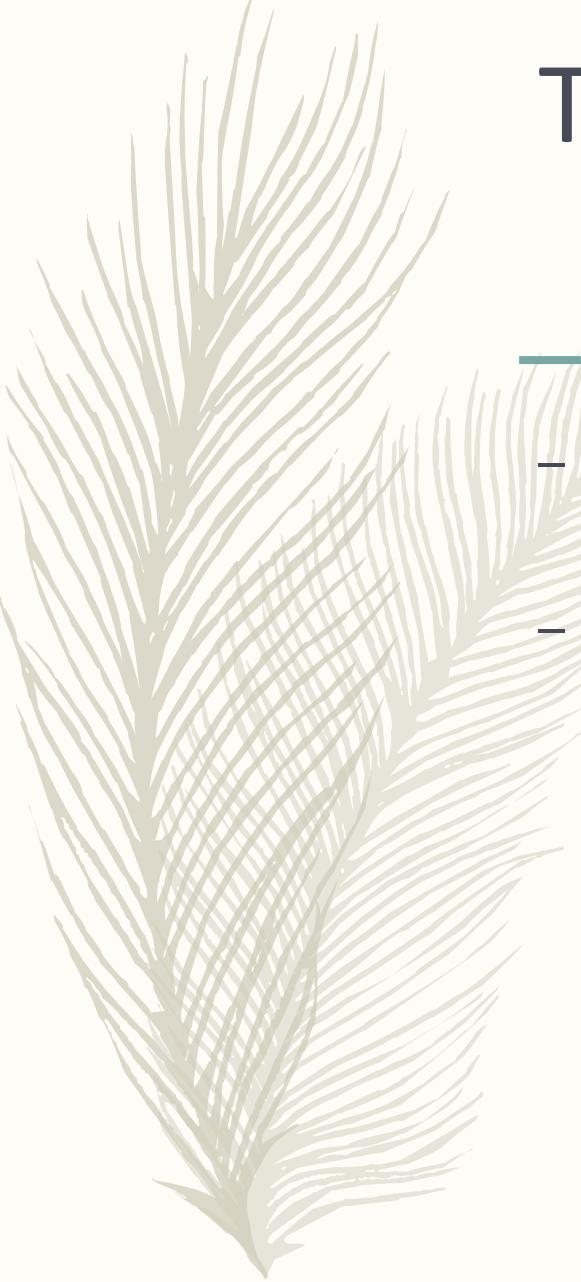
- Se pot inseră date într-un view doar dacă inserarea afectează un singur base table (în cazul în care view-ul conține date din mai multe tabele)
- Se pot actualiza date într-un view doar dacă actualizarea afectează un singur base table (în cazul în care view-ul conține date din mai multe tabele)
- Se pot șterge date dintr-un view doar dacă view-ul conține date dintr-un singur tabel
- Operațiunile de inserare într-un view sunt posibile doar dacă view-ul expune toate coloanele care nu permit valori NULL
- Numărul maxim de coloane pe care le poate avea un view este 1024



# Tabele sistem

---

- Tabelele sistem sunt niște tabele speciale care conțin informații despre toate obiectele create într-o bază de date, cum ar fi:
  - Tabele
  - Coloane
  - Proceduri stocate
  - Trigger-e
  - View-uri
  - Funcții definite de utilizator
  - Indecși



# Tabele sistem

---

- Tabelele sistem sunt gestionate de către server (nu se recomandă modificarea lor direct de către utilizator)
- Exemple:
  - sys.objects** – conține câte o înregistrare pentru fiecare obiect creat în baza de date, cum ar fi: procedură stocată, trigger, tabel, constrângere
  - sys.columns** – conține câte o înregistrare pentru fiecare coloană a unui obiect care are coloane, cum ar fi: tabel, funcție definită de utilizator care returnează un tabel, view
  - sys.databases** – conține câte o înregistrare pentru fiecare bază de date existentă pe server



# Trigger

---

- Trigger-ul este un **tip special de procedură stocată** care se execută automat atunci când un anumit eveniment DML sau DDL are loc în baza de date
- Nu se poate executa în mod direct
- Evenimente DML:
  - INSERT
  - UPDATE
  - DELETE
- Evenimente DDL:
  - CREATE
  - ALTER
  - DROP
- Fiecare trigger (DML) aparține unui singur tabel



# Trigger

---

- Sintaxa:

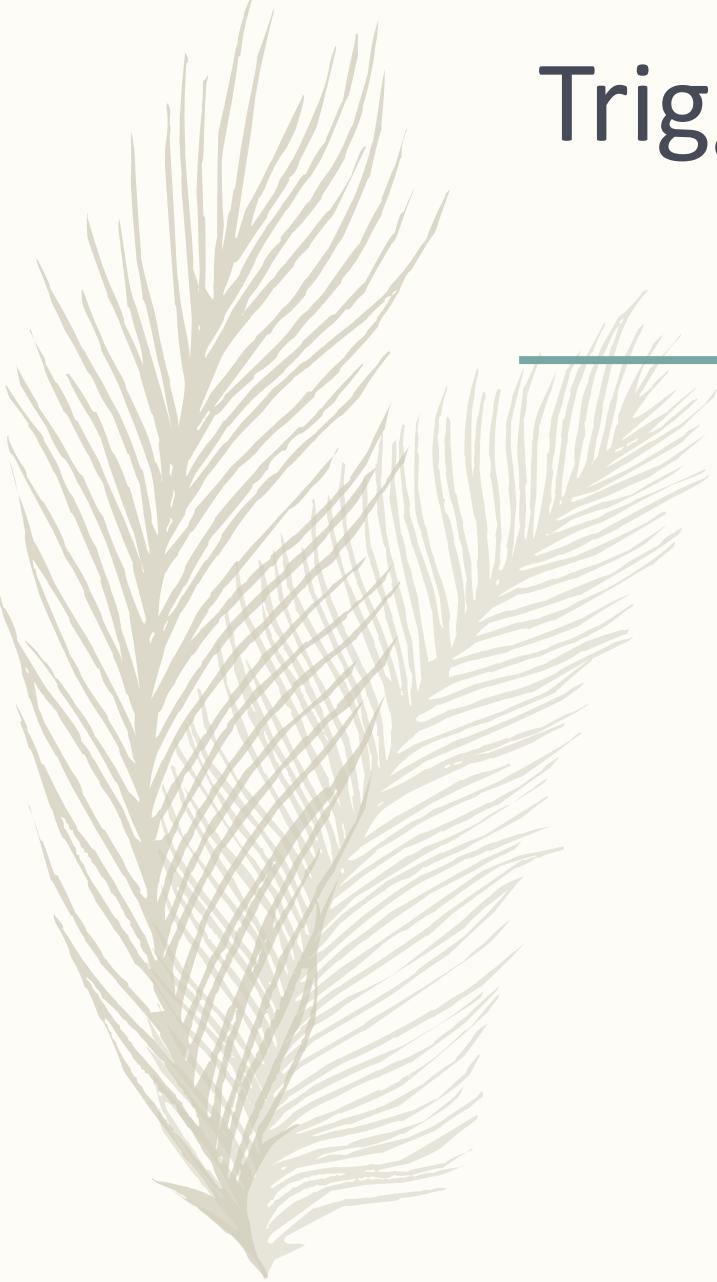
```
CREATE TRIGGER trigger_name  
ON { table | view }  
[ WITH <dml_trigger_option> [ ,...n ] ]  
{ FOR | AFTER | INSTEAD OF }  
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }  
[ WITH APPEND ]  
[ NOT FOR REPLICATION ]  
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME  
<method specifier [ ; ] > }
```



# Trigger

---

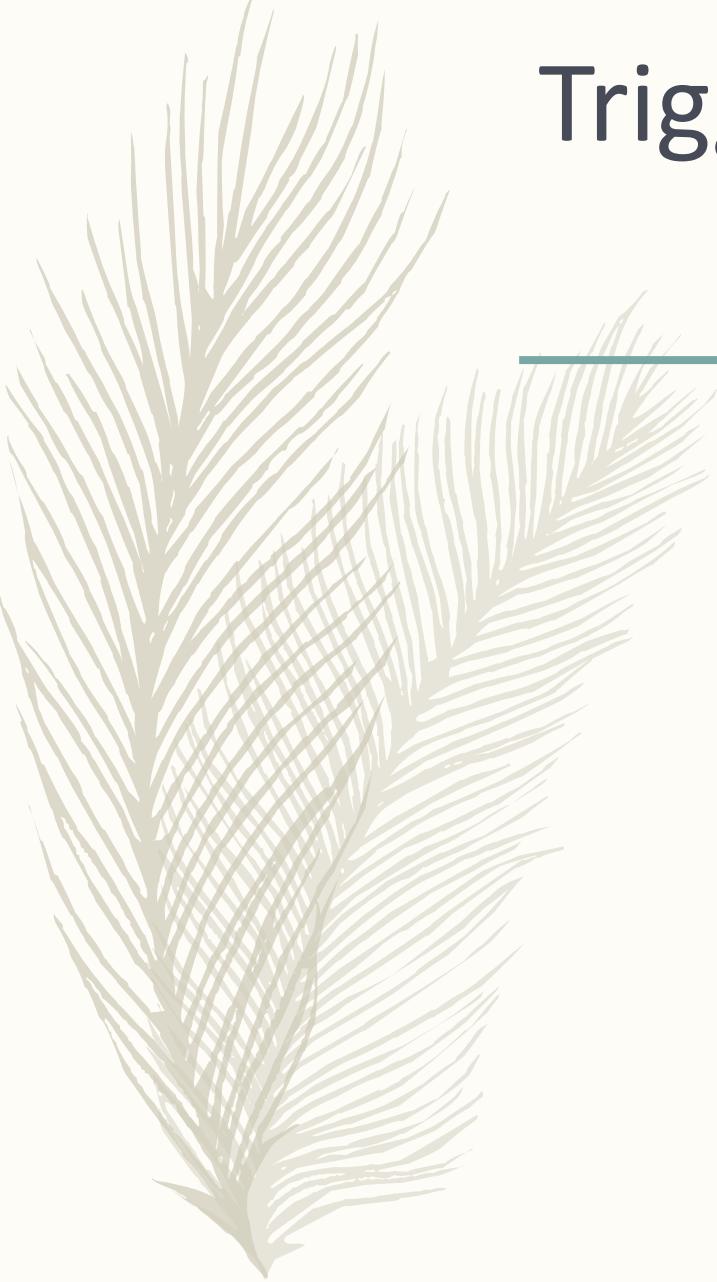
- Momentul execuției unui trigger
  - FOR, AFTER (se pot defini mai multe trigger-e de acest tip) – trigger-ul se execută după ce s-a executat evenimentul declanșator
  - INSTEAD OF – trigger-ul se execută în locul evenimentului declanșator
- Dacă se definesc mai multe trigger-e pentru aceeași acțiune (eveniment), ele se execută în ordine aleatorie
- Când se execută un trigger, sunt disponibile două tabele speciale, numite **inserted** și **deleted**



# Trigger - Exemplu

---

```
CREATE TRIGGER [dbo].[La_introducere_produs]
ON [dbo].[Produse]
FOR INSERT
AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO Arhivă_Cumpărare (nume, dată, cantitate)
    SELECT nume, GETDATE(), cantitate FROM inserted;
END;
```



# Trigger - Exemplu

---

```
CREATE TRIGGER [dbo].[La_ștergere_produs]
ON [dbo].[Produse]
FOR DELETE
AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO Arhivă_Vânzare (nume, dată, cantitate)
    SELECT nume, GETDATE(), cantitate FROM deleted;
END;
```



# Trigger - Exemplu

---

```
CREATE TRIGGER [dbo].[La_actualizare_produs]
ON [dbo].[Produse]
FOR UPDATE AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO Arhivă_Vânzare (nume, dată, cantitate) SELECT d.nume,
    GETDATE(), d.cantitate-i.cantitate FROM deleted d INNER JOIN
    inserted i ON d.cod_p=i.cod_p WHERE i.cantitate<d.cantitate;
    INSERT INTO Arhivă_Cumpărare (nume, dată, cantitate) SELECT i.nume,
    GETDATE(), i.cantitate-d.cantitate FROM deleted d INNER JOIN
    inserted i on d.cod_p=i.cod_p WHERE i.cantitate>d.cantitate;
END;
```



# Clauza OUTPUT

---

- Cu ajutorul clauzei **OUTPUT** avem acces la înregistrările modificate, șterse sau adăugate
- În exemplul de mai jos se actualizează numele persoanei care are *cod\_p=5* din tabelul *Persoane* și se stochează în tabelul *ModificăriNumePersoane* valoarea din coloana *cod\_p*, valoarea veche a numelui (*deleted.nume*), valoarea nouă a numelui (*inserted.nume*), data curentă (GETDATE()) și numele login-ului care a realizat modificarea (SUSER\_SNAME()):

```
UPDATE Persoane SET nume='Pop' OUTPUT inserted.cod_p,
deleted.nume, inserted.nume, GETDATE(), SUSER_SNAME()
INTO ModificăriNumePersoane (cod_p, nume_vechi,
nume_nou, data_modificării, nume_login) WHERE cod_p=5;
```



# Cursoare

---

- Sunt anumite situații în care procesarea unui result set este mai eficientă dacă se procesează pe rând fiecare înregistrare din result set
- Deschiderea unui cursor pe un result set permite procesarea result set-ului înregistrare cu înregistrare (se procesează o singură înregistrare la un moment dat)
- Cursoarele extind procesarea rezultatelor prin faptul că:
  - permit poziționarea la înregistrări specifice dintr-un result set
  - returnează o înregistrare sau un grup de înregistrări aflate la poziția curentă din result set



# Cursoare

---

- suportă modificarea înregistrărilor aflate în poziția curentă în result set
- suportă diferite niveluri de vizibilitate a modificărilor făcute de către alți utilizatori asupra datelor din baza de date care fac parte din result set
- permit instrucțiunilor Transact-SQL din script-uri, proceduri stocate și trigger-e accesul la datele dintr-un result set



# Cursoare

---

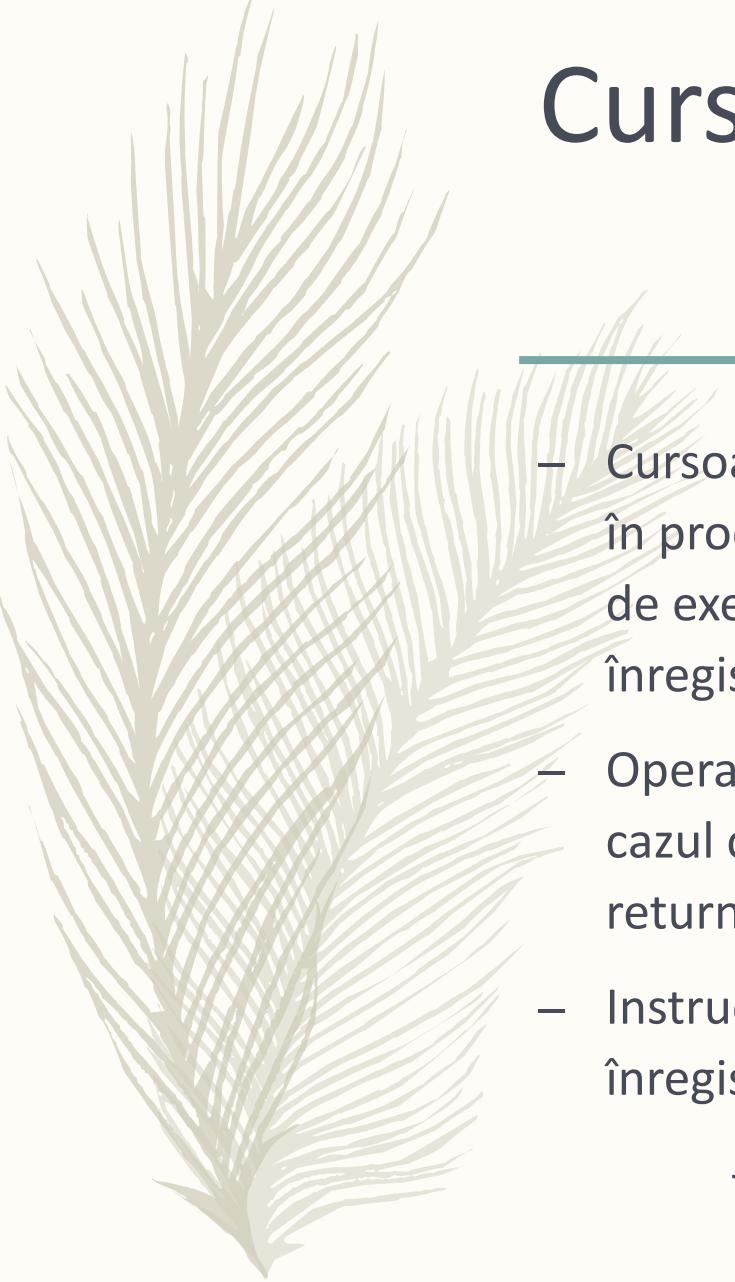
- Cursoarele Transact-SQL necesită anumite instrucțiuni pentru declarare, populare și extragere de date:
  - se folosește o instrucțiune **DECLARE CURSOR** pentru a declara cursorul și se specifică o instrucțiune **SELECT** care va produce result set-ul cursorului
  - se folosește o instrucțiune **OPEN** pentru a popula cursorul, care execută instrucțiunea **SELECT** încorporată în instrucțiunea **DECLARE CURSOR**
  - se folosește o instrucțiune **FETCH** pentru a extrage înregistrări individual din result set (de obicei **FETCH** se execută de multe ori, cel puțin o dată pentru fiecare înregistrare din result set)



# Cursoare

---

- dacă este cazul, se folosește o instrucțiune **UPDATE** sau **DELETE** pentru a modifica înregistrarea ( acest pas este opțional)
- se folosește o instrucțiune **CLOSE** pentru a închide cursorul și a elibera unele resurse (cum ar fi result set-ul cursorului și lock-urile de pe înregistrarea curentă)
- cursorul este încă declarat, deci poate fi deschis din nou folosind o instrucțiune **OPEN**
- se folosește o instrucțiune **DEALLOCATE** pentru a elimina referința cursorului din sesiunea curentă iar acest proces eliberează toate resursele alocate cursorului, inclusiv numele său (după acest pas, pentru a reconstrui cursorul este nevoie ca acesta să fie declarat din nou)
- cursoarele aflate în interiorul procedurilor stocate nu necesită închidere și eliminare, aceste instrucțiuni se execută automat când procedura stocată își încheie execuția



# Cursoare

---

- Cursoarele Transact-SQL sunt extrem de eficiente atunci când sunt încorporate în proceduri stocate și trigger-e deoarece totul este compilat într-un singur plan de execuție pe server, deci nu există trafic pe rețea asociat cu returnarea înregistrărilor
- Operațiunea de a returna o înregistrare dintr-un cursor se numește **fetch**, iar în cazul cursoarelor Transact-SQL se folosește instrucțiunea **FETCH** pentru a returna înregistrări din result set-ul unui cursor
- Instrucțiunea **FETCH** suportă un număr de opțiuni care permit returnarea unor înregistrări specifice:
  - **FETCH FIRST** – returnează prima înregistrare din cursor



# Cursoare

---

- **FETCH NEXT** – returnează înregistrarea care urmează după ultima înregistrare returnată
- **FETCH PRIOR** – returnează înregistrarea care se află înaintea ultimei înregistrări returnate
- **FETCH LAST** – returnează ultima înregistrare din cursor
- **FETCH ABSOLUTE n** – returnează a n-a înregistrare de la începutul cursorului dacă n este un număr pozitiv, iar dacă n este un număr negativ returnează înregistrarea care se află cu n înregistrări înaintea sfârșitului cursorului (dacă n este 0, nicio înregistrare nu este returnată)



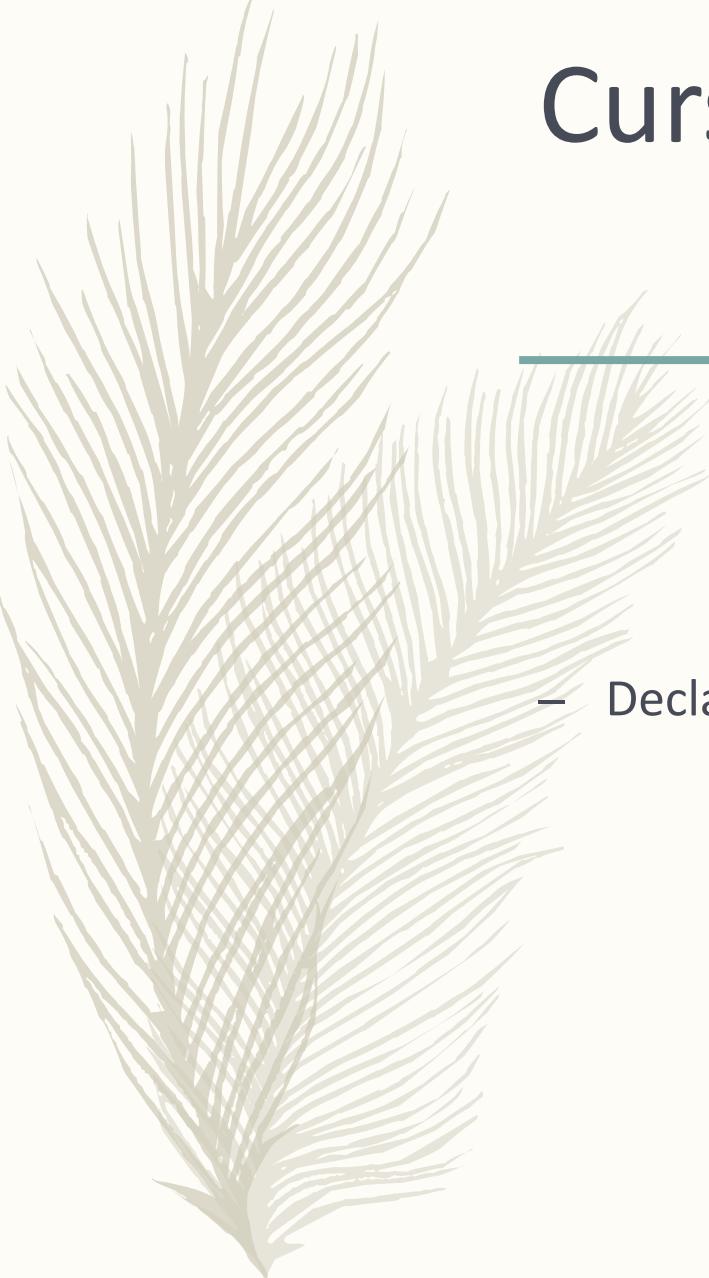
# Cursoare

---

- **FETCH RELATIVE n** – returnează a n-a înregistrare după ultima înregistrare returnată dacă n este pozitiv, iar dacă n este negativ returnează înregistrarea care se află înainte cu n înregistrări față de ultima înregistrare returnată (dacă n este 0, ultima înregistrare returnată va fi returnată din nou)

Comportamentul unui cursor poate fi specificat în două moduri:

- prin specificarea comportamentului cursoarelor folosind cuvintele cheie **SCROLL** și **INSENSITIVE** în instrucțiunea **DECLARE CURSOR** (SQL-92 standard)
- prin specificarea comportamentului unui cursor cu ajutorul tipurilor de cursoare

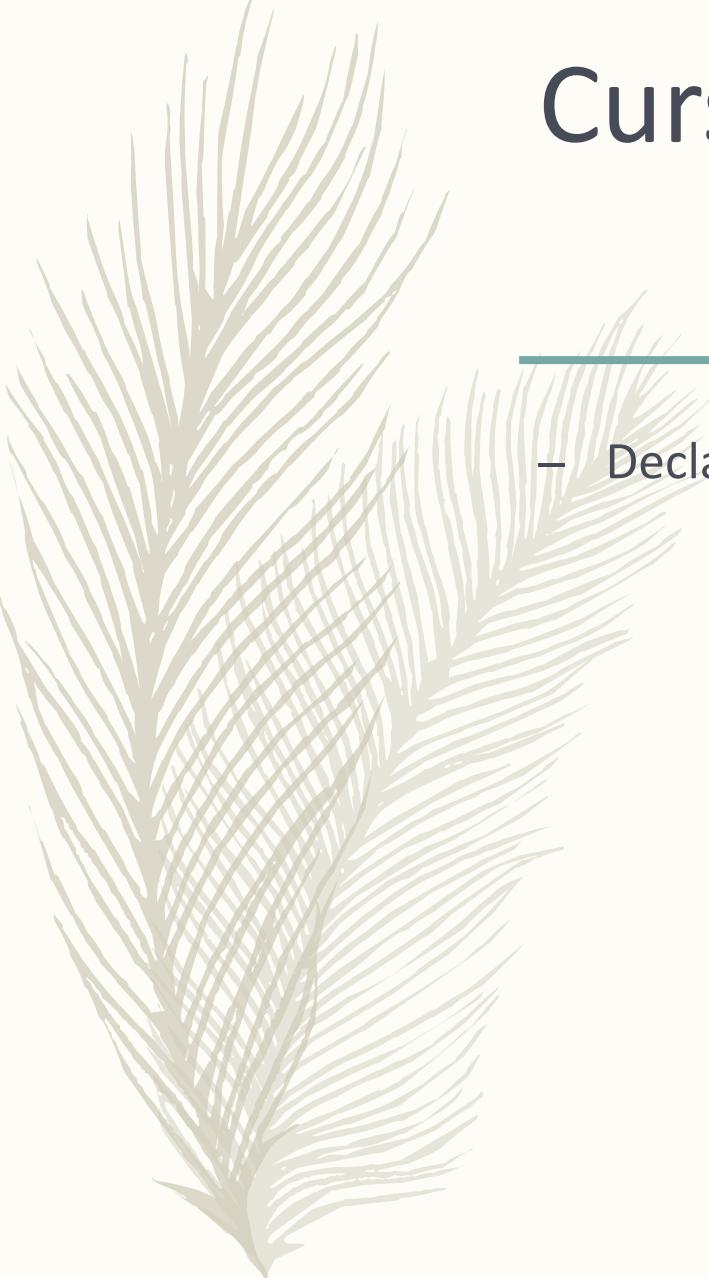


# Cursoare

---

- de obicei API-urile pentru baze de date definesc comportamentul cursoarelor împărțindu-le în patru tipuri de cursoare: forward-only, static (uneori denumit snapshot sau insensitive), keyset-driven și dynamic
- Declararea unui cursor – sintaxa ISO:

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ]
] } ]
```



# Cursoare

---

- Declararea unui cursor – sintaxa Transact-SQL:

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```



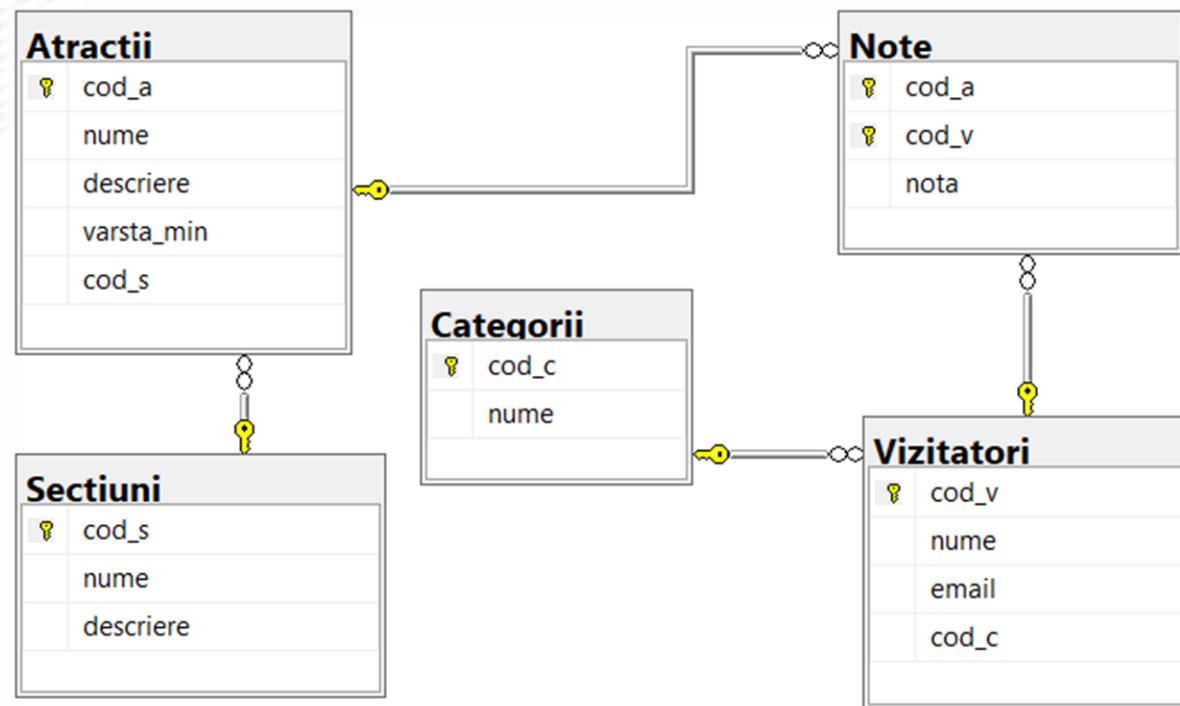
# Cursoare - Exemplu

---

```
DECLARE @nume NVARCHAR(50), @prenume NVARCHAR(50), @oraș NVARCHAR(50);
DECLARE cursorpersoane CURSOR FAST_FORWARD FOR
SELECT prenume, nume, oraș FROM Persoane;
OPEN cursorpersoane;
FETCH NEXT FROM cursorpersoane INTO @prenume, @nume, @oraș;
WHILE @@FETCH_STATUS=0
    BEGIN
        PRINT @prenume+ N' '+@nume+ N' s-a născut în orașul '+@oraș;
        FETCH NEXT FROM cursorpersoane INTO @prenume, @nume, @oraș;
    END
CLOSE cursorpersoane;
DEALLOCATE cursorpersoane;
```

# Problemă propusă

- Se dă o bază de date având următoarea structură:





# Problemă propusă

---

- Cerințe:
  - 1) Să se creeze o funcție scalară care verifică dacă numele unei categorii există în tabelul *Categorii*. Funcția primește ca parametru de intrare numele categoriei. Dacă numele există deja în tabel, funcția va returna valoarea 1, iar dacă numele nu există va returna valoarea 0.
  - 2) Să se creeze o funcție scalară care primește ca parametru numele unei categorii și returnează codul acesteia.
  - 3) Să se creeze o funcție inline table valued care primește ca parametru de intrare vârsta minimă recomandată și returnează numele atracțiilor, nota primită și adresa de email a vizitatorului, pentru toate atracțiile care au vârsta minimă recomandată egală cu cea dată ca parametru.



# Problemă propusă

---

- 4) Creați un trigger care împiedică execuția operațiilor de ștergere din tabelul *Categorii* și afișează un mesaj corespunzător.
- 5) Creați un view care afișează toate înregistrările din tabelul *Categorii* al căror nume este egal cu ‘pensionari’ sau ‘copii’.
- 6) Creați un view care afișează toate înregistrările din tabelul *Sectiuni* al căror nume începe cu litera C.
- 7) Creați o funcție de tip inline table valued care returnează toate înregistrările din tabelul *Sectiuni* al căror nume se termină cu o literă dată ca parametru de intrare și au cel puțin două caractere.
- 8) Creați un view care afișează numele vizitatorilor, nota și numele atracției.



# Bibliografie

---

- <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/queries/output-clause-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/relational-databases/cursors?view=sql-server-ver16>
- <https://learn.microsoft.com/en-us/sql/relational-databases/system-catalog-views/catalog-views-transact-sql?view=sql-server-ver16>