

508.666



Biblioteca UTCN  
CLUJ-NAPOCA  
2002  
508668

UNIV. TEHNICA CLUJ-N.  
BIBLIOTECA  
Nr. 508.668 / 2003

VERIFICAT  
2002

ROBERT DOLLINGER

# UTILIZAREA SISTEMULUI SQL SERVER (SQL 7.0 SQL 2000)

# CUPRINS

## PREFĂTĂ

<b>1. TIPURI DE DATE ȘI VARIABILE</b>	11
Tipuri de date în SQL Server .....	11
Variabile.....	11
Declararea unei variabile.....	19
Atribuirea de valori unei variabile .....	20
Exerciții și probleme .....	21
	22

<b>2. OPERATORI ȘI FUNCȚII</b> .....	24
Operatori.....	24
Funcții SQL.....	24
Funcții scalare .....	31
Funcții matematice .....	32
Funcții care operează asupra sirurilor de caractere .....	32
Funcții pentru manipularea valorilor de tip dată calendaristică .....	34
Funcții sistem (conversii) .....	37
Exerciții și probleme .....	39
	42

<b>3. TABELE ȘI INDECȘI</b> .....	43
3.1. Instrucțiunea CREATE TABLE .....	43
3.1.1. Definirea unei coloane .....	43
3.1.2. Constrângeri .....	44
3.1.2.1. Exprimarea constrângерilor la nivel de coloană .....	47
3.1.2.2. Exprimarea constrângерilor la nivel de tabelă .....	48
3.2. Instrucțiunea ALTER TABLE .....	51
3.3. Instrucțiunea DROP TABLE .....	53
3.4. Utilizarea instrucțiunilor CREATE TABLE și ALTER TABLE .....	58
3.5. Indeci .....	58
3.5.1. Instrucțiunea CREATE INDEX .....	63
3.5.2. Instrucțiunea DROP INDEX .....	64
3.6. Exerciții și probleme .....	65
	67

Toate drepturile asupra acestei ediții sunt rezervate s.c. Casa de Editură Albastră s.r.l.

<b>4. INSTRUCȚIUNI DE CONTROL</b>	
4.1. Comentarii .....	69
4.2. Instrucțiunile BEGIN și END .....	69
4.3. Instrucțiunea IF...ELSE .....	70
4.4. Instrucțiunea WHILE .....	71
4.5. Instrucțiunea GOTO .....	72
4.6. Instrucțiunea WAITFOR .....	74
4.7. Instrucțiunea RETURN .....	76
4.8. Fișiere de comenzi (batch-uri) .....	77
4.8.1. Comanda GO .....	79
4.9. Exerciții și probleme .....	81
<b>5. SINTAXA FRAZEI SELECT ÎN SQL SERVER</b>	
5.1. Clauza SELECT .....	83
5.2. Clauza INTO .....	83
5.3. Clauza FROM .....	85
5.4. Clauza WHERE .....	88
5.5. Clauza GROUP BY .....	89
5.6. Clauza HAVING .....	89
5.7. Operatorul UNION .....	89
5.8. Clauza ORDER BY .....	90
5.9. Funcții de agregare .....	91
5.10. Exerciții și probleme .....	97
<b>6. FORMULAREA INTEROGĂRILOR ÎN LIMBAJUL SQL (PARTEA I)</b>	99
6.1. Interogări simple .....	99
6.2. Cuplarea a mai multe relații .....	101
6.3. Folosirea operatorului IN .....	102
6.4. Folosirea operatorului EXISTS .....	106
6.5. Exerciții și probleme .....	110
<b>7. FORMULAREA INTEROGĂRILOR ÎN LIMBAJUL SQL (PARTEA A II-A)</b>	112
7.1. Folosirea clauzelor GROUP BY și HAVING .....	112
7.2. Exerciții și probleme .....	119
<b>8. VALORI NULL ȘI OPERAȚII DE CUPLARE EXTERNĂ</b>	120
8.1. Valori Null .....	120
8.1.1. Funcția ISNULL .....	122
8.2. Operații de cuplare externă - interogări .....	125
8.3. Exerciții și probleme .....	131
<b>9. FUNCȚII CASE. OPERAȚII DE INSERARE, ȘTERGERE ȘI ACTUALIZARE</b>	132
9.1. Funcții (expresii) CASE .....	132
9.2. Operații de inserare, ștergere și actualizare .....	135
9.2.1. Instrucțiunea INSERT .....	135
9.2.2. Instrucțiunea DELETE .....	137
9.2.3. Instrucțiunea UPDATE .....	139
9.3. Exerciții și probleme .....	143
<b>10. VEDERI. SQL DINAMIC</b>	145
10.1. Vederi .....	145
10.1.1. Instrucțiunea CREATE VIEW .....	146
10.1.2. Instrucțiunea ALTER VIEW .....	149
10.1.3. Instrucțiunea DROP VIEW .....	149
10.1.4. Modificarea datelor din vederi .....	150
10.2. SQL dinamic .....	153
10.3. Exerciții și probleme .....	159
<b>11. PROCEDURI STOCATE</b>	161
11.1. Instrucțiunea CREATE PROCEDURE .....	162
11.2. Instrucțiunea EXECUTE .....	165
11.3. Instrucțiunea ALTER PROCEDURE .....	167
11.4. Instrucțiunea DROP PROCEDURE .....	167
11.5. Exemple de utilizare a procedurilor stocate .....	168
11.6. Exerciții și probleme .....	172
<b>12. CURSOARE SQL SERVER</b>	174
12.1. Instrucțiunea DECLARE CURSOR .....	175
12.2. Instrucțiunea OPEN .....	175
12.3. Instrucțiunea FETCH .....	176
12.4. Variabila sistem @@FETCH_STATUS .....	177
12.5. Instrucțiunea CLOSE .....	178
12.6. Instrucțiunea DEALLOCATE .....	178

## PREFĂTĂ

<b>13. TRIGGERE</b>	184
13.1. Instrucțiunea CREATE TRIGGER	185
13.2. Instrucțiunea DROP TRIGGER	185
13.3. Proprietăți ale triggerelor	188
13.4. Triggere multiple	189
13.5. Triggere recursive	190
13.6. Triggeruri imbricate	190
13.7. Exemplu de utilizare a triggerelor	191
13.8. Exerciții și probleme	191

## BIBLIOGRAFIE

200

SQL în dezvoltarea de aplicații folosind sistemul SQL Server pentru gestionarea bazelor de date. Prezentarea este abordată, în primul rând, din punctul de vedere al celui care dezvoltă aplicații și este pus în situația de a scrie cod care să fie executat de către SQL Server, dar poate fi utilă și administratorilor de baze de date, implicați în proiectarea și configurarea bazei de date, sau utilizatorilor simpli care pot exploata o bază de date formulând direct interogări SQL. Ca și cunoștințe preliminare necesare, din partea cititorului, se presupun cunoscute elementele fundamentale ale teoriei bazelor de date relationale.

Volumul acoperă versiunile SQL Server 7.0 și SQL Server 2000, punând în evidență elementele care diferă de la o versiune la cealaltă, acolo unde este cazul.

Un accent deosebit este pus pe limbajul de interogare SQL, prezentarea frazei SELECT și formularea de interogări de la cele mai simple până la interogări complexe. Această parte a lucrării poate fi studiată ca atare, independent de sistemul SQL Server, accentul fiind pus pe sintaxa standard SQL'92, la care SQL Server se aliniază. Acolo unde apar deosebiri ale sintaxei SQL Server față de standardul SQL'92, ele sunt menționate și comentate, fiind oferite mai multe variante de rezolvare a interogărilor, dintre care cel puțin una este suportată de către SQL Server.

Pe lângă limbajul de interogare, mai sunt prezentate o serie de elemente și concepte care sunt în mai mare măsură legate de sistemul SQL Server. Astfel sunt prezentate elementele de bază ale dialectului TSQL (varianta SQL pentru sistemele SQL Server) cum ar fi: tipuri de date, variabile, instrucțiuni de control, operatori, funcții, precum și concepte și instrumente de lucru mai evoluțate cum sunt: vederi, proceduri stocate, triggere, curseare, SQL dinamic și altele.

Fiecare capitol abordează câte o tematică relativ independentă, ori câte două subiecte corelate, ilustrate printr-un număr mare de exemple reprezentative și însotite de un set de exerciții și probleme propuse spre rezolvare.

Cărtea este organizată în 13 capitole astfel:

- Capitolul 1. Tipuri de date și variabile – prezintă, în prima parte, tipurile de date din SQL Server, proprietățile acestora și precedența tipurilor de date. Partea a doua se referă la variabile: declararea de variabile și initializarea acestora prin instrucțiunile SET sau SELECT.
- Capitolul 2. Operatori și funcții – trece în revistă categoriile de operatori din SQL Server: aritmetică, de atribuire, pe biți, de comparare, logici, de

concatenare siruri de caractere, operatori unari. Sunt prezentate regulile de evaluare a expresiilor complexe și precedența operatorilor. Ciforii începători ar putea amâna studiul paragrafelor referitoare la operatorii logici: **ALL**, **ANY**, **BETWEEN**, **IN**, **LIKE**, **OR** și **SOME** până după înțelegerea frazei **SELECT** care este prezentată în capitolul 5. Partea a două a capitolului prezintă cele mai importante categorii de funcții scalare și anume: funcțiile matematice, funcțiile care operează asupra sirurilor de caractere și funcțiile pentru manipularea valorilor de tip dată calendaristică.

**Capitolul 3. Tabele și indecesi** – ne arată cum se poate crea și modifica o tabelă. Sunt prezentate constrângările care se pot impune unei tabele și modul în care acestea se pot defini. Pentru exemplificare este prezentată crearea structurii de tabele pentru baza de date numită *Agenti*. Această bază de date este ulterior referată pe tot parcursul lucrării în exemplele din capitolile care urmează. Capitolul se încheie cu prezentarea tipurilor de indecesi din **SQL Server** arătând proprietățile, respectiv modul de creare și distrugere al acestora.

**Capitolul 4. Instrucțiuni de control. Fișiere de comenzi (batch-uri).** – instrucțiunile de control care în **TSQL** vin să întregescă funcționalitatea ca unitate independentă de compilare și comanda **G.O.**

**Capitolul 5. Sintaxa frazei SELECT în SQL Server** – prezintă sintaxa frazei **SELECT** și funcțiile de agregare.

**Capitolul 6. Formularea interrogărilor în limbajul SQL (Partea I)** – prezintă sintaxa frazei **SELECT** și ilustrează modul de folosire al operatorilor **IN** și **EXISTS**.

**Capitolul 7. Formularea interrogărilor în limbajul SQL (Partea II-a)** – prezintă sintaxa frazei **SELECT** și ilustrează modul de folosire (sau evitare) a clauzelor **GROUP BY** și **HAVING**.

**Capitolul 8. Valorii NULL și operații de cuplare externă** – prezintă proprietățile valorilor **NULL** și exemple referitoare operațiile de cuplare externă: **FULL JOIN**, **RIGHT JOIN**, **LEFT JOIN**.

**Capitolul 9. Funcții CASE. Operații de inserare, stergere și actualizare.** – prezintă funcțiile **CASE** și exemple de utilizare a acestora, respectiv instrucțiunile **INSERT**, **DELETE** și **UPDATE**.

**Capitolul 10. Vederi. SQL dinamic.** – abordează, în prima parte, conceptul de înțelegere frazei **SELECT** care este prezentată în capitolul 5. Partea a două a capitolului prezintă cele mai importante categorii de funcții scalare și anume: funcțiile matematice, funcțiile care operează asupra sirurilor de caractere și funcțiile pentru manipularea valorilor de tip dată calendaristică.

**Capitolul 11. Proceduri stocate** – introduce conceptul de procedură stocată și problemele legate de aceasta: create/modificare, execuție, transmiterea parametrilor, exemple de proceduri.

**Capitolul 12. Curseare SQL Server** – deși în **SQL Server** este încurajată soluționarea problemelor prin folosirea pe cât posibil a frazelor **SQL**, sunt situații când este nevoie să se recurgă la mijloace navigaționale. Acestea sunt reprezentate în **SQL Server** prin conceptul de cursor. Prin curseare se poate realiza accesul la date, tuplă cu tuplă, și realizează o prelucrare individualizată a fiecărei tuple. Sunt prezentate și exemplele cursorelor, încărcarea datelor în cursor, prelucrarea propriu-zisă, închiderea și dealocarea cursorului.

**Capitolul 13. Trigger** – ultimul capitol este dedicat unei categorii speciale de proceduri stocate și anume triggerelor. Principala utilizare a triggerelor este pentru a impune asupra unei baze de date o serie de contrângeri pentru care sistemul nu oferă posibilități declarative de exprimare. Deși se bazează pe principii generale comune, implementarea triggerelor difere sensibil de la un sistem la altul. În acest capitol este prezentat modul de lucru cu trigger în **SQL Server 7.0** și **2000**. Sunt prezentate triggerurile de tip **INSTEAD OF** care au apărut odată cu versiunea **SQL Server 2000**.

În încheiere doresc să aduc mulțumiri tuturor celor care, prin comentarii și idei și-au adus contribuția la îmbunătățirea materialului prezentat în această carte. Pe de altă parte, pornind de la convingerea că este întotdeauna loc pentru mai bine, aşteptăm observațiile și sugestiile cititorului interesat la adresa e-mail:

*robert.dollinger@cs.utcluj.ro*

# 1. TIPURI DE DATE SI VARIABILE

## 1.1. Tipuri de date în SQL Server

Orice obiect din SQL Server care conține o dată (atribut, variabilă, expresie, parametru de procedură, funcție ori procedură stocată care returnează o valoare) are asociat un *tip de date* prin care se definește ce fel de valori poate stoca obiectul. Specificarea unui tip de date presupune definirea a mai multe caracteristici ale obiectului:

- *natura* datelor conținute de obiect (caracter, întreg, binar etc.);
- *dimensiunea* valorilor stocate (în esență numărul de octetti necesari pentru stocare);
- *precizia* (numai pentru valori numerice, indică numărul total de cifre zecimale care pot fi acceptate pentru un tip de date);
- *scala* (numai pentru valori numerice, numărul de zecimale acceptate într-o anumită reprezentare).

SQL Server oferă un set de *tipuri de date de bază* care definesc tipurile de valori acceptate de acest sistem. Pe lângă acestea există posibilitatea definirii de *tipuri de date utilizator*. Un tip utilizator este întotdeauna definit în termeni de unui tip de bază căruia îl adaugă unele caracteristici suplimentare menite să descrie mai bine o anumită clasă de valori (de ex. întregi pozitivi). La definirea unui tip de date utilizator se specifică numele tipului de date creat, tipul de bază din care provine, dacă acceptă sau nu valori NULL și constrângerile suplimentare pe care le satisfac valorile tipului utilizator.

Tipurile de date de bază existente în SQL Server sunt:

### 1. TIPURI NUMERICE EXACTE

Definesc tipuri numerice întregi de diferite dimensiuni. Acestea sunt:

- |               |   |
|---------------|---|
| <b>bigint</b> | - întregi reprezentați pe 8 octeți, interval de valori: -<br>2 <sup>63</sup> la 2 <sup>63</sup> -1. |
| <b>int</b>    | - întregi reprezentați pe 4 octeți, interval de valori: -<br>2 <sup>31</sup> la 2 <sup>31</sup> -1. |

### **smallint**

- întregi reprezentați pe 2 octeți, interval de valori: -  
 $2^{15}$  la  $2^{15}-1$ .

### **tinyint**

- întregi pozitivi reprezentați pe un octet, interval de valori: 0 la 255.

#### **Observație:**

Tipul **bigint** a fost introdus începând cu versiunea SQL 2000 și este destinat pentru reprezentarea valorilor numerice exacte foarte mari. Tipul principal pentru reprezentarea întregilor în SQL Server este **int**.

## **2. TIPUL BIT**

Este implementat ca un întreg care poate conține valorile 0, 1 sau NULL. În aplicații este folosit pentru reprezentarea valorilor logice care pot lua valoările TRUE sau FALSE, eventual pot fi nedefinibile.

### **bit**

- valoare întreagă 0, 1 ori NULL.

#### **Observație:**

Tipul bit are o reprezentare optimizată în cadrul bazei de date, astfel că într-o tabelă există până la 8 attribute de tip bit (ele sunt împachetate într-un singur octet, dacă sunt între 9 și 16 attribute atunci ele ocupă 2 octeți s.a.m.d.). Dacă attributele acceptă și valori NULL, atunci spațiul necesar pentru stocare este mai mare.

## **3. TIPURI ZECIMALE**

Definesc tipuri numerice pentru reprezentarea valorilor zecimale având precizia și scala fixe.

### **decimal[(p[,s])]**

- interval de valori - $10^{38} + 1$  la  $10^{38} - 1$ .

*p* (precizia) - este o valoare între 1 și 38; specifică numărul maxim de cifre zecimale pe care le poate avea

numărul atât la stânga cât și dreapta punctului zecimal.

### **s (scala)**

- specifică numărul de zecimale. Este o valoare cuprinsă între 0 și valoarea lui *p*. Implicit este 0.

### **Exemple de constante zecimale:**

123,43, 5,6, 7,32

#### **Observație:**

Numărul de octeți folosiți pentru reprezentarea unei valori **decimal** sau **numeric** depinde de precizia acestuia (de ex. pentru *p>28* poate ajunge la 17 octeți).

## **4. TIPURI MONETARE**

Definesc tipuri de date pentru reprezentarea valorilor ce reprezintă sume de bani într-o monedă specificată.

### **money**

- reprezentare pe 8 octeți, interval de valori: - $2^{63} - 1$ , cu acuratețe de 4 zecimale din unitatea monetară de bază.

### **smallmoney**

- reprezentare pe 4 octeți, interval de valori: - $2^{31} - 1$ , cu acuratețe de 4 zecimale din unitatea monetară de bază.

#### **Exemple de constante de tip monetar:**

\$100.00, 200

## **5. TIPURI NUMERICE APROXIMATIVE**

Definesc tipuri numerice cu reprezentare în virgulă mobilă.

### **float [(n)]**

- este un număr flotant în intervalul - 1.79E + 308

la 1.79E + 308.

Pараметrul *n* este un întreg de la 1 la 53 reprezentând numărul de biți folosiți pentru reprezentarea mantisei și este determinant pentru precizia și dimensiunea reprezentării numărului flotant, după cum se arată în tabelul de mai jos:

Parametrul <i>n</i>	Precizie	Reprezentare	Corespondență
1-24	7 cifre zecimale	4 octeți	flotant simplă precizie
25-53	15 cifre zecimale	8 octeți	flotant dublă precizie

**real**

- este un număr flotant în intervalul -3.40E + 38 la 3.40E + 38; reprezentarea este pe 4 octeți, este sinonim cu float(24) și corespunde reprezentării flotante în simplă precizie.

#### Exemple de constante de tip flotant:

1023.1E2, 100E-1

### 6. TIPURI DATĂ CALENDARISTICĂ

Definesc tipuri destinate pentru a reprezenta data calendaristică și ora.

- datetime**
- data și ora în intervalul de la 1 ianuarie 1753 la 31 decembrie 9999, cu acuratețea de 333 milisecunde (1/3 secundă). Nu este acceptată nici o dată calendaristică în afara intervalului 1753-9999.

**smalldatetime** - data și ora în intervalul de la 1 ianuarie 1900 la 6 iunie 2079, cu acuratețea de 1 minut.

#### Observație:

SQL Server reprezintă valorile **datetime** sub forma a doi întregi de 4 octeți fiecare. Primul întreg reprezintă numărul de zile dinainte sau după data de referință 1 ianuarie 1900. Al doilea întreg reprezintă ora din zi în milisecunde față de miezul nopții. Valorile **smalldatetime** sunt reprezentate cu o precizie mai mică, fiind folositi doi întregi de căte 2 octetă. Primul întreg reprezinta numărul de zile dinainte sau după data de referință 1 ianuarie 1900, iar al doilea numărul de minute față de miezul nopții.

Constantele de tip data calendaristică se reprezintă în formatele specifice, fiind încadrate între apostroafe.

#### Exemple:

'04/15/98', '15/04/98', '15 April, 1998'

### 7. TIPURI SIR DE CARACTERE

Definesc șiruri de caractere Unicod de lungime fixă sau variabilă.

**char[(n)]**

- sir de caractere non-Unicode cu lungimea fixă de *n* caractere (octeți).

Parametrul *n* poate lua valori de 1 la 8000. Dimensiunea reprezentării este de *n* octeți indiferent de mărimea actuală a șirului de caractere reprezentat. Spațiul rămas liber după ultimul caracter al șirului reprezentat se completează cu caractere spațiu.

**varchar[(n)]**

- sir de caractere non-Unicode cu lungimea variabilă de până la *n* caractere (octeți).

Parametrul *n* poate lua valori de 1 la 8000. Dimensiunea reprezentării este numărul de caractere din șirul reprezentat și nu *n* care indică doar dimensiunea maximă acceptată. Lungimea actuală a șirului de caractere poate fi 0.

#### Observații:

1. Dacă parametrul *n* nu este specificat, atunci se consideră implicit valoarea 1 în declaratiile de variabile sau de atrbute, respectiv valoarea 30 în funcția CAST.

2. Tipul **char** este folosit atunci când este de așteptat ca valorile datelor să aibă aproximativ aceeași dimensiune, în timp ce **varchar** se folosește ori de câte datele pot avea dimensiuni greu de estimat ori care variază în plajă largi.

Constantele sir de caractere se pot reprezenta fie încadrate între apostroafe, fie între ghilimele.

#### Exemple:

'abc', "abc'd"

### 8. TIPURI SIR DE CARACTERE UNICOD

Definesc șiruri de caractere Unicode de lungime fixă sau variabilă.

**nvarchar[(n)]**

- sir de caractere Unicode cu lungimea fixă de *n* caractere.

Parametrul *n* poate lua valori de 1 la 4000. Dimensiunea reprezentării este de 2\*i octeți indiferent de mărimea actuală a șirului de caractere reprezentat. Spațiul rămas liber

după ultimul caracter al sirului reprezentat se completează cu caractere spatiu.

**nvarchar[*n*]** - sir de caractere Unicode cu lungimea variabilă de până la *n* caractere.

Parametrul *n* poate lua valori de 1 la 4000. Dimensiunea reprezentării este dublul numărului de caractere din sirul reprezentat și nu  $2^*n$  care indică doar dimensiunea maximă acceptată. Lungimea actuală a sirului de caractere poate fi 0.

#### Observații:

1. Dacă parametrul *n* nu este specificat, atunci se consideră implicit valoarea 1 în declarăriile de variabile sau de attribute, respectiv valoarea 30 în funcția CAST.
2. Va fi preferat tipul nvarchar în loc de nchar ori de către datele pot avea dimensiuni greu de estimat ori care variază în plaje largi.

## 9. TIPURI SIR BINAR

Definesc siruri binare de lungime fixă sau variabilă.

**binary [ ( *n* ) ]** - dată binară cu lungimea fixă de *n* octeți.

Parametrul *n* poate lua valori de la 1 la 8000. Dimensiunea reprezentării este *n*+4 octeți.

**varbinary [ ( *n* ) ]** - dată binară cu lungimea variabilă de până la *n* octeți.

Parametrul *n* poate lua valori de la 1 la 8000. Dimensiunea reprezentării este lungimea actuală a datei binare+4 octeți. Lungimea actuală a datei poate fi 0.

**image** - date binare de lungime variabilă de maxim 2<sup>31</sup>-1 octeți, folosit pentru reprezentarea imaginilor.

#### Observații:

1. Dacă parametrul *n* nu este specificat, atunci se consideră implicit valoarea 1 în declarăriile de variabile sau de attribute, respectiv valoarea 30 în funcția CAST.

2. Va fi preferat tipul varbinary în loc de binary ori de către datele pot avea dimensiuni greu de estimat ori care variază în plaje largi.
- Constantele binare se reprezintă ca siruri de cifre hexa prefixate cu 0x.

## 10. ALTE TIPURI DE DATE

**cursor**

- reprezintă o referință la un cursor și poate fi asociat unei variabile sau unui parametru de ieșire al unei proceduri stocate.

Operatiile permise asupra variabilelor de tip cursor sunt cele definite pentru cursoarele propriu-zise. Tipul de date cursor nu poate fi folosit la definirea unei coloane într-un tabel.

**sql\_variant** - este un tip de date care poate stoca valori ale oricărui tip de date suportat de SQL Server exceptând text, ntext,

image, timestamp și sql\_variant. sql\_variant poate fi folosit pentru definirea de coloane, parametri, variabile și valori de return ale funcțiilor utilizator.

#### Observații:

1. O coloană de tip sql\_variant poate conține valori de tip diferit în tuple diferite.
2. O valoare sql\_variant poate avea până la cel mult 8016 octeți.
3. O valoare de tip sql\_variant trebuie convertită explicit la tipul dorit înainte de a fi folosita în operații.
4. În general, un obiect de tip sql\_variant are asociat un tip de bază care este dat de tipul valorii conținute. Atunci când această valoare este NULL tipul de date este nedefinit, exceptând situația în care se definește o valoare implicită pentru obiectul respectiv. Un obiect sql\_variant nu poate avea un alt sql\_variant ca tip de bază.
5. Coloanele de tip sql\_variant pot fi incluse în constrângeri UNIQUE, de cheie primară sau de cheie strânsă cu condiția ca dimensiunea cheii sau a indexului să nu depăsească 900 octeți.

**table** - este un tip de date special care poate fi folosit pentru stocarea relațiilor rezultat. Permite declararea variabilelor de tip **table** sub formă:

#### Sintaxă:

**DECLARE @variabila\_tabel TABLE (*definiție\_tip\_tabel*)**

unde:

*definiție\_tip\_tabel* - definește o structură de tabel în mod similar cu instrucțiunea **CREATE TABLE**.

#### Observații:

1. Atât funcțiile, cât și variabilele se pot declara cu tipul **table**. Variabilele **table** se pot folosi în funcții, proceduri stocate și batch-uri.
2. Începând cu versiunea SQL 2000, variabilele de tip **table** sunt introduse ca alternativă la tabelurile temporare și oferă o serie de avantaje:
  - o variabilă **table** are un domeniu de vizibilitate bine delimitat care este funcția, procedura sau batch-ul în care este definită;
  - în domeniul său de vizibilitate o variabilă **table** poate fi folosită la fel ca orice tabelă (în instrucțiuni **SELECT**, **INSERT**, **UPDATE** și **DELETE**) exceptând următoarele:  
**INSERT INTO @variabila\_tabel EXEC procedură\_stocată,**  
**SELECT lista\_select INTO @variabila\_tabel FROM ...**
3. Nu sunt permise operații de atribuire care să implice variabile de tip **table**.

**timestamp** - este un tip de date pentru care sistemul garantează generația automată de valori binare unice la nivelul bazei de date.

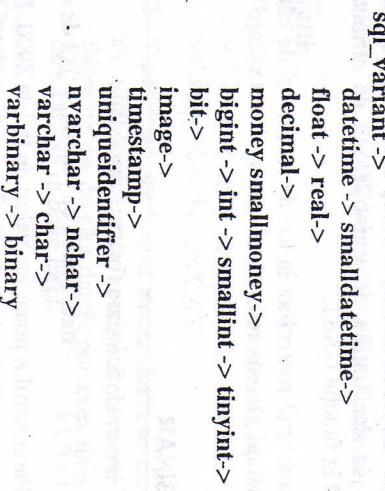
Tipul **timestamp** este folosit pentru a crea etichete unice pentru tuplele tabelelor. O valoare de tip **timestamp** este stocată pe 8 octeti. O tabelă poate avea o singură coloană de tip **timestamp**. O valoare de tip **timestamp** este modificată ori de către ori tupla care o conține este inserată într-un tabel sau este actualizată.

**uniqueidentifier** - este un tip de date care definește un identificator unic global (GUID).

Acesta permite generarea de valori unice la nivelul bazei de date, dar care nu sunt modificate automat de către sistem.

#### Precedență tipurilor de date

Când două expresii având tipuri diferențe sunt combinate printre-un operator sunt posibile conversii de tip implicite pentru a aduce operanții la același tip de date. Ierarhia de precedență a tipurilor de date este cea care stabilește care tip dată se convertește la celălalt și ce fel de conversie se execută. Regula de bază este aceea că tipul de prioritate mai mică în ierarhie se convertește la tipul de prioritate mai mare. Ierarhia de tipuri pentru SQL Server, în ordinea descrescătoare a priorității, este următoarea:



## 1.2. Variabile

O variabilă locală *Transact-SQL* este un obiect care conține o valoare de un anumit tip. Variabilele locale pot fi folosite în *fisierele de comenzi* (batch-uri) sau în procedurile stocate. Situațiile tipice de utilizare ale variabilor locale sunt:

- ca și contor pentru numărarea și controlul numărului de parcurgeri ale unei bucle;
- pentru a stoca o valoare ce va fi testată de o instrucțiune de control;
- pentru a transmite o valoare ca parametru la o procedură stocată, pentru a salva o valoare ce va fi returnată de o procedură stocată;

### 1.2.1. Declararea unei variabile

Instructiunea DECLARE initializează una sau mai multe variabile prin:

- Atribuirea către o singură variabilă numele variabilei și tipul său.
- Atribuirea unui tip de date (care poate fi un tip sistem sau un tip utilizator) și a unei lungimi. Pentru variabilele numerice se atribuie de asemenea precizia și scara.
- Se setează valoarea variabilei la NULL.

Sintaxă:

```
DECLARE {{ @variabila_locală_tip_dată }
| { @nume_variabilă_cursor_CURSOR }
| { TABLE(definiție_tip_tabel) } l, n}
```

unde:

@variabila\_locală - este numele variabilei care se declară;

tip\_dată - tipul de date al variabilei;

@nume\_variabilă\_cursor - numele unei variabile cursor, ce poate fi

privită ca un pointer la un cursor;

definiție\_tip\_tabel - definește o structură de tabel în mod similar cu

instructiunea CREATE TABLE.

Exemple:

1. Declarația unei singure variabile @x de tip int:

```
DECLARE @x int
```

Se pot declara mai multe variabile prin aceeași instructiune DECLARE separând prin virgulă definițiile variabilelor:

```
DECLARE @Nume varchar(30), @Prenume varchar(20),
        @oras varchar(25)
```

Observație:

Tipul de date și lungimea trebuie specificate pentru fiecare variabilă în parte chiar dacă se declară mai multe variabile de același tip. De exemplu declarația de mai jos este incorectă:

```
DECLARE @x, @y, @z int
```

2.

Domeniul de vizibilitate al unei variabile locale este reprezentat de mulțimea instructiunilor care pot referi variabila. Domeniul unei variabile locale începe din punctul în care este declarată până la sfârșitul fișierului de comenzi (batch) sau al procedurii stocate în care a fost declarată. Declarația unei variabile poate fi, în principiu, plasată oriunde în cadrul fișierului de comenzi sau a procedurii stocate.

Secvența de mai jos generează o eroare de sintaxă deoarece variabila @x este declarată în primul batch (terminat prin comanda GO) și referă în al doilea:

```
DECLARE @x INT
SET @x = 1
GO
SELECT @x
```

3. Declarația unei variabile de tip cursor presupune specificarea cuvântului cheie CURSOR:

```
DECLARE @x CURSOR
```

4. Declarația unei variabile tabel trebuie să contină o descriere de structură tabel precedată de cuvântul cheie TABLE (în cazul nostru structura tablei constă dintr-o singură coloană, Nume):

```
DECLARE @T TABLE (Nume varchar(20))
```

### 1.2.2. Atribuirea de valori unei variabile

La declararea unei variabile valoarea ei este implicit setată la NULL.

Pentru a atribui o valoare unei variabile se poate utiliza fie instructiunea SET, fie instructiunea SELECT. Instructiunea SET a fost introdusă începând cu versiunea 7.0 a SQL Server și este metoda preferată prin care se atribuie valori unei singure variabile. De asemenea, se pot atribui valori uneia sau mai multor variabile referindu-le într-o listă de atribuiri a unei instructiuni SELECT. Instructiunea SET nu poate fi folosită pentru a atribui valori la mai multe variabile, dar este mai eficientă decât instructiunea SELECT.

Exemple:

Secvența de mai jos declară trei variabile, le asignează valori, și apoi le returnează într-un rând printăr-o instructiune SELECT:

```
DECLARE @x int, @y int, @z int      --declarare variabile
SET @x=1                            --initializare @x
```

```
SELECT @y=2, @z=3  
SELECT @x, @y, @z
```

--initializare @y si @z  
--vizualizare

De regulă, dacă o variabilă este referită într-o listă SELECT, atunci fie i se atribuie variabilei o valoare scalară, fie instrucțiunea SELECT returnează un singur rând, din care rezultă o valoare unică care se atribuie variabilei:

```
DECLARE @numar_furnizori int  
SELECT @numar_furnizori=COUNT(*) FROM Furnizori
```

Dacă totuși o instrucție SELECT returnează mai mult de un rând și variabila referă o expresie nescalară, atunci variabila este setată la valoarea returnată de expresie prin ultimul rând al rezultatului. Secvența de mai jos atribuie variabilei @nume numele primului furnizor în ordine alfabetică (ultimul selectat în ordinea descrescătoare după atributul *Name*):

```
DECLARE @nume varchar(30)  
SELECT @nume = Nume  
FROM Furnizori  
ORDER BY Nume DESC  
SELECT @nume
```

## 1.3. Exerciții și probleme

1. Fiind date declarațiile de variabile:

```
DECLARE @b binary  
DECLARE @c char  
DECLARE @i int  
DECLARE @s sql_variant
```

Să se arate care dintre următoarele atribuiri este corectă și care nu:

```
SET @b=0x31  
SET @c='A'  
SET @i=-49  
SET @s='abc'  
SET @c=@b  
SET @c=@i  
SET @b=@c  
SET @s=@b  
SET @c=@s
```

2. Care este valoarea afisată pentru variabila @y în următoarele secvențe:

- a. DECLARE @x binary  
DECLARE @y int  
SET @x=0x11
- b. DECLARE @x binary(2)  
DECLARE @y smallint  
SET @x=0xFFE1

3. Explicați care sunt erorile în secvența următoare:

```
SET @y=@x  
SELECT @y
```

```
SET @y=@x  
SELECT @y
```

4. Corectați următoarea secvență de instrucțiuni:

```
DECLARE @a,@b int  
SET @a=1,@b=2  
SELECT @x
```

```
SET @y=@x  
SELECT @y
```

## 2. OPERATORI SI FUNCȚII

### Operatorul de atribuire

#### 2.1. Operatori

Un operator este un simbol care specifică o anumită operație executată asupra unei sau mai multe expresii operand. În SQL Server există următoarele categorii de operatori:

- operatori aritmeticici;
- operatori de atribuire;
- operatori pe biti;
- operatori de comparare;
- operatori logici;
- operatorul de concatenare siruri de caractere;
- operatori unari.

### Operatori aritmeticici

Operatorii aritmeticici execută operații matematice asupra a două expresii de tip numeric. Operatorii aritmeticici sunt:

Operator	Semnificație
+	Adunare.
-	Scădere.
*	Multiplicare.
/	Divizune.
%	Modulo - returnează restul unei împărțiri întregi.

### Observații:

1. Operatorul modulo se aplică numai unor operanzi de tip întreg și returnează ca rezultat tot un întreg. De exemplu:  $12 \% 5 = 2$  adică restul împărțirii întregi a lui 12 la 5.
2. Operatorii + și - se pot folosi și pentru operații asupra unor valori de tip datetime și smalldatetime.

### Operatori de comparare

Operatorii de comparare testează dacă două expresii sunt sau nu egale, respectiv dacă una dintre ele este mai mare sau nu decât cealaltă. Operatorii de comparare se pot aplica asupra tuturor tipurilor de expresii exceptând text, text sau image. Operatorii de comparare acceptați de SQL Server sunt:

Operator	Semnificație
=	Egalitate.
>	Mai mare.
<	Mai mic.
>=	Mai mare sau egal.
<=	Mai mic sau egal.
≠	Diferit.
!=	Diferit (nu este prevăzut în standardul SQL '92).
is	Nu mai mic decât (nu este prevăzut în standardul SQL '92).

### Observații:

1. Rezultatul oricarei operații de comparare este de tip boolean, având una din valorile: TRUE, FALSE sau UNKNOWN (în cazul comparațiilor cu valori NULL).
2. Spre deosebire de alte tipuri de date din SQL Server, tipul de date boolean nu poate fi specificat ca tipul de date al unei coloane sau variabile și nu se poate returna într-o relație rezultat. Expresiile booleene se pot folosi doar în următoarele contexte: clauzele WHERE sau HAVING, expresii CASE, instrucțiuni IF și WHILE.

În SQL Server există un singur operator de atribuire, reprezentat prin semnul de egalitate (=). Operatorul de atribuire poate fi folosit numai în cadrul unor instrucțiuni SET sau într-o clauză (instrucțiune) SELECT.

3. Dacă opțiunea SET ANSI\_NULLS este ON, atunci orice operator de comparare care are unul sau doi operanzi cu valoare NULL va returna valoarea UNKNOWN. Dacă opțiunea SET ANSI\_NULLS este OFF, atunci se aplică aceleși reguli, exceptând operatorul de egalitate (=) care returnează TRUE dacă ambi operaanzi sunt NULL.

- Astfel, NULL = NULL returnează TRUE dacă SET ANSI\_NULLS este OFF și UNKNOWN dacă SET ANSI\_NULLS este ON.

## Operatori logici

Operatorii logici testează valoarea de adever a unei condiții. La fel ca și operatorii de comparare, operatorii logici returnează a valoare de tip boolean care poate fi TRUE, FALSE sau UNKNOWN.

Operator	Semnificație
ALL	TRUE dacă toate comparațiile dintr-un set sunt TRUE.
AND	TRUE dacă ambele expresii booleene date ca operaanzi sunt TRUE, UNKNOWN dacă una din expresii este UNKNOWN și celalăt TRUE, respectiv FALSE dacă una din expresii este FALSE.
ANY	TRUE dacă cel puțin una dintre comparațiile dintr-un set este TRUE.
BETWEEN	TRUE dacă operandul este între două valori (inclusiv).
EXISTS	TRUE dacă o subintrebare returnează cel puțin o tuplă.
IN	TRUE dacă operandul este egal cu cel puțin una dintre valorile unei liste de expresii sănătate relației rezultat.
LIKE	TRUE dacă operandul respectă un anumit şablon.
NOT	TRUE dacă operandul este FALSE, FALSE dacă operandul este TRUE și UNKNOWN dacă operandul este UNKNOWN.
OR	TRUE dacă cel puțin una din expresiile booleene date ca operand este TRUE, FALSE dacă ambele expresii sunt FALSE și UNKNOWN în rest.
SOME	TRUE dacă cel puțin una dintre comparațiile dintr-un set este TRUE.

## Operatori Unari

Operatorii unari execută o operație dată doar asupra unei singure expresii de tip numeric. Operatorii unari din SQL SERVER sunt:

Operator	Semnificație
+	Valoarea numerică este pozitivă.
-	Valoarea negativă a operandului (Bitwise NOT).

## Precedența operatorilor

Precedența operatorilor determină ordinea de execuție a acestora în cazul unor expresii care conțin o secvență de operatori. Un operator cu precedență mai mare este executat înaintea altuia cu precedență mai mică. Precedența operatorilor din SQL Server este data în ordine descrescătoare astfel:

Operatori Unari:	+, -, ~
Operatori aritmetici multiplicative:	* , /, %
Operatori aritmici aditivi:	+ (Adunare), + (Concatenare), - (Scadere)
Operatori de comparare:	=, >, <, >=, <=, <>, !=, !=,
Operatori de biti:	^, &,
Operatorul de negație logică:	NOT
Operatorul SI logic:	AND
Alți operatori logici:	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
Operatorul de atribuire:	=

## Observație:

1. Implicit, sirul vid este interpretat ca atare în operațiile de concatenare. De exemplu: 'abc' + '' + 'def' produce ca rezultat 'abcdef'.

2. Sirul vid nu trebuie confundat cu valoarea NULL a unui sir. De exemplu: 'abc' + NULL + 'def' produce ca rezultat NULL, adică un sir cu valoare nedefinită.

## Operatorul de concatenare

Operatorul de concatenare a șirurilor de caractere este notat cu semnul de adunare (+). Aceasta este singura operație de manipulare a șirurilor de caractere care se realizează printă-un operator, toate celelalte fiind realizate prin diferite funcții.

Dacă doi operatori dintr-o expresie au aceeași precedență, evaluarea lor se face de la stânga la dreapta în ordinea apariției lor în expresie. Ordinea implicită de execuție poate fi modificată prin folosirea parantezelor.

## Operatorul BETWEEN

Sintaxa operatorului BETWEEN este următoarea:

*expresie\_test [NOT] BETWEEN*

și returnează valoarea TRUE dacă *expresie\_test* are valoarea între *expresie\_de\_la și expresie\_pîna\_la* (inclusiv) și FALSE în caz contrar (în varianta fără NOT). Prin folosirea operatorului NOT se reversează valoarea rezultatului returnat.

### Observații:

1. Operatorul BETWEEN realizează un test inclusiv. Pentru a face un test exclusiv se vor folosi operatorii > și <.
2. Dacă oricare dintre cele trei expresii operand ale operatorilor BETWEEN sau NOT BETWEEN au valoarea NULL, atunci rezultatul este UNKNOWN.

## Operatorul LIKE

Determină dacă un sir de caractere dat respectă sau nu un anumit şablon (pattern). Un şablon poate include atât caractere obişnuite, cât și caractere de înlocuire (wildcard). La compararea unui sir de caractere cu un şablon caracterelor obisnuite trebuie să se potrivească exact cu caracterele din şir. În schimb, caracterelor de înlocuire pot fi înlocuite cu fragmente arbitrarale ale şirului de caractere. Prin folosirea caracterelor de înlocuire operatorul LIKE devine mult mai flexibil decât operatorul = sau <>. Sintaxa operatorului LIKE este următoarea:

*expresie\_sir\_caractere [NOT] LIKE şablon*

unde:

*expresie\_sir\_caractere* – este orice expresie care se evaluatează la un sir de caractere valid în SQL Server.

*şablon*

- este şablonul cu care se compara şirul de caractere. Şablonul poate conține următoarele caractere de înlocuire:

[ESCAPE *caracter\_escape*]

**Exemplu:**

1. Furnizorii al căror nume începe cu litera 'F':

```
SELECT *
FROM Furnizor
WHERE Nume LIKE 'F%'
```

Caracter de înlocuire	Descriere
%	Orice fragment de şir conținând zero sau mai multe caractere.
-	Un singur caracter oricare.
[]	Orice caracter dintr-un set sau interval dat.
[^]	Orice caracter care nu face parte dintr-un set sau interval dat.

– este orice expresie validă de tip şir de caractere. *caracter\_escape* nu are valoare implicită și constă dintr-un singur caracter.

Operatorul LIKE returnează TRUE dacă *expresie\_sir\_caractere* se potrivește cu şablonul specificat.

### Observații:

1. La compararea unui şir de caractere prin LIKE toate caracterele din şablon sunt semnificative, inclusiv spațiile de la început sau sfârșit. De exemplu:

'abc' LIKE 'abc'

returnează FALSE. Totuși, aceste spații nu se iau în considerare la sirul ce comparare. De exemplu oricare dintre comparațiile:

'abc' LIKE 'abc', 'abc' LIKE 'abc', 'abc' LIKE 'abc'

returnează TRUE.

2. Regula de comparare a spațiilor se poate manifesta în mod neașteptat atunci când se fac comparații cu variabile sau câmpuri având tipul de date char sau varchar. Un char de o anumită dimensiune va conține spații de completare ori de câte ori sirul actual memorat este mai scurt decât dimensiunea declarată. Aceasta ar putea duce la eşuarea comparațiilor dacă se folosesc un şablon având tipul de date char. O expresie de tip varchar are întotdeauna dimensiunea şirului actual pe care îl conține fără a fi completat cu spații la sfârșit. Acest fapt ar putea cauza probleme dacă se folosesc un şir de caractere de comparat având tipul varchar.

2. Furnizorii al căror nume începe cu litera 'A' sau 'F':

```
SELECT *  
FROM Furnizor  
WHERE Nume LIKE '[AF]%'
```

3. Furnizorii al căror nume nu începe cu litera 'A' sau 'F':

*Varianta 1:*

```
SELECT *  
FROM Furnizor  
WHERE Nume NOT LIKE '[AF]%'
```

*Varianta 2:*

```
SELECT *  
FROM Furnizor  
WHERE Nume LIKE '^[AF]%'
```

Observație:

Pentru interogarea de mai sus cele două variante de rezolvare sunt echivalente și vor returna același rezultat. Acest lucru nu mai este adevărat în cazul săabloanelor care în partea fixă au specificat o secvență de mai multe caractere. Astfel, următoarea frază returnează furnizorii al căror nume nu începe cu secvența 'S.C.':

```
SELECT *  
FROM Furnizor  
WHERE Nume NOT LIKE 'S.C.%'
```

În timp ce următoarea frază elimină din rezultat și numele care încep cu 'S', 'S.' și 'S..'. s.a.m.d.

```
SELECT *  
FROM Furnizor  
WHERE Nume LIKE '^S[^.][^C] ^[^.]%'
```

Aceasta se datorează faptului că la comparațiile de șiruri cu săabloane care conțin caractere de înlocuire negative SQL Server face evaluarea pas cu pas, pentru că un singur caracter de înlocuire ia un moment dat.

4. Furnizorii al căror nume conține caracterul '\_' în a doua poziție:

```
SELECT *  
FROM Furnizor  
WHERE Nume LIKE '_[_]%'
```

Observație:

- Caracterele de înlocuire se pot folosi ca litere în săabloane prin încadrarea lor între paranteze drepte. În interogarea de mai sus simbolul '\_' este folosit ca și caracter de înlocuire în prima sa apariție și ca literal în a doua.

Tabelul de mai jos conține câteva exemple de folosire a operatorului LIKE și a caracterelor de înlocuire pe post de literele:

Sablon	Semnificație
LIKE 'S(%)'	5%
LIKE '_ln'	n
LIKE '[a-cdf]	a, b, c, d sau f
LIKE '[^acdf]'	~, a, c, d sau f
LIKE 'UU'	U
LIKE 'I'	I
LIKE 'abc[^d]%'	orice începe cu abc, d
LIKE 'abc(def)'	abcd, abce sau abcdef

2. Interogarea 4) se poate rezolva și prin folosirea clauzei ESCAPE și a caracterului de escape. Orice caracter de înlocuire care urmează după un caracter\_escape este interpretat ca literal. Dacă alegem să folosim simbolul '!' pe post de caracter de escape, atunci interogarea 4) se rezolvă astfel:

```
SELECT *  
FROM Furnizor  
WHERE Nume LIKE '_!_%' ESCAPE '!'
```

## 2.2. Funcții SQL

Limbajul de programare Transact-SQL oferă trei categorii de funcții:

- funcții care returneză un set de înregistrări (rowset functions)

Aceste funcții pot înlocui nume de tabele din frazele TSQL.

- funcții de agregare

Operăaza asupra unei colecții de valori, de regulă provenite din una sau mai multe coloane ale unor tabele și returneză o singură valoare aggregată. Funcțiile de agregare pot fi folosite exclusiv în cadrul frazelor SELECT și vor fi prezentate în contextul acestora.

- funcții scalare

Operază asupra unei singure valori și returnează tot o singură

valoare. O funcție scalară poate fi folosită oriunde este legală plasarea unei expresii.

## 2.2.1. Funcții scalare

Limbajul de programare Transact-SQL oferă mai multe categorii de funcții scalare dintre care cele mai comune sunt: funcțiile matematice, funcțiile care operează asupra sirurilor de caractere și funcțiile pentru manipularea valorilor de tip date calendaristică. De asemenea, de interes sunt funcțiile de sistem de tip scalar. Dintre acestea cele mai frecvent utilizate sunt: funcțiile de conversie explicită (CAST și CONVERT), funcțiile (expresii) CASE și funcțiile care tratează valorile de tip NULL (ISNULL).

### 2.2.1.1. Funcții matematice

Funcțiile matematice realizează calculul valorii unei funcții specificate pe baza a una sau mai multe valori numerice date ca argumente și returnează tot o valoare numerică. Cele mai importante funcții matematice împreună cu sintaxa acestora sunt prezentate mai jos:

Sintaxa	Semnificație
<code>ABS(expresie_numerică)</code>	valoarea absolută a argumentului;
<code>ACOS(expresie_floating)</code>	valoarea în radiani a unghiului al căruia cosinus este egal cu argumentul;
<code>ASIN(expresie_floating)</code>	valoarea în radiani a unghiului al căruia sinus este egal cu argumentul;
<code>ATAN(expresie_floating)</code>	valoarea în radiani a unghiului a căruia tangentă este egală cu argumentul;
<code>ATR2(expresie_floating)</code>	valoarea în radiani a unghiului a căruia tangentă este egală între cele două argumente;
<code>CEILING(expresie_numerică)</code>	cel mai mic întreg mai mare sau egal cu argumentul;
<code>COS(expresie_floating)</code>	cosinușul argumentului specificat în radiani;
<code>COT(expresie_floating)</code>	cotangenta argumentului specificat în radiani;
<code>DEGREES(expresie_numerică)</code>	valoarea în grade a argumentului specificat în radiani;
<code>EXP(expresie_floating)</code>	exponentiala argumentului;
<code>FLOOR(expresie_numerică)</code>	cel mai mare întreg mai mic sau egal cu argumentul;
<code>LOG(expresie_floating)</code>	logaritmul natural al argumentului;
<code>LOG10(expresie_floating)</code>	logaritmul zecimal al argumentului;
<code>PI()</code>	valoarea constantei PI;
<code>POWER(expresie_numerică, putere)</code>	ridicare la putere;
<code>RADIANS(expresie_numerică)</code>	valoarea în radiani a argumentului specificat în grade;
<code>RAND([origine])</code>	valoarea floating aleatorie în intervalul 0 la 1,

Observații:

1. `expresie_numerică` se evaluatează la orice valoare de tip numeric exceptând tipul `bit`;
2. `expresie_floating` se evaluatează la orice valoare de tip float (real sau `float`);
3. `putere` se evaluatează la orice valoare de tip numeric exceptând tipul `bit`;

4. În cazul funcțiilor ACOS și ASIN valoarea argumentului trebuie să fie în intervalul [-1, 1], în caz contrar funcțiile returnează valoarea NULL și se semnalizează o eroare de domeniu;
5. Similar, în cazul funcțiilor LOG, LOG10 și SQRT, argumentul trebuie să fie pozitiv, în caz contrar funcțiile returnează valoarea NULL și se semnalizează o eroare de domeniu;
6. *lungime* este precizia de rotunjire a lui `expresie_numerică` prin funcția ROUND. Dacă *lungime* este un număr pozitiv, atunci rotunjirea se face la numărul de zecimale indicate de acesta. Dacă *lungime* este un număr negativ atunci rotunjirea se face la numărul de poziții de la stânga punctului zecimal indicat de valoarea absolută a acestuia. Dacă numărul de cifre semnificative a lui `expresie_numerică` este mai mic decât valoarea absolută a parametrului *funcție*, atunci rezultatul returnat de funcție este zero. Parametrul *funcție* indică tipul de rotunjire care se face: valoarea 0 (implicită) semnifică rotunjire propriu-zisă, iar o valoare diferită de 0 indică truncare. Atât *lungime* cât și *funcție* pot fi de tip tinyint, smallint sau int.
7. Funcțiile aritmetiche cum ar fi ABS, CEILING, DEGREES, FLOOR, POWER, RADIANS și SIGN, returnează o valoare care are același tip de date ca argumentul. Funcțiile trigonometrice împreună cu EXP, LOG, LOG10, SQUARE și SQRT, returneză valori flotante.

### Exemplu:

```

SELECT ABS (-1)           - rezultat 1
SELECT ACOS (-1)          - rezultat 1.0
SELECT DEGREES(ACOS (-1)) - rezultat 3.1415926535897931
SELECT POWER (2.0, 0.5)    - rezultat 1
SELECT POWER (2.0, 0.5)    - rezultat 1.4
SELECT POWER (2.00000, 0.5) - rezultat 1.41421
SELECT POWER (2.0000000000000000, 0.5) - rezultat 1.4142135623730952
SELECT SQRT (2)            - rezultat 1.4142135623730951
SELECT ROUND (748.58, -1)   - rezultat 750.00
SELECT ROUND (748.58, -3)   - rezultat 1000.00
SELECT ROUND (748.58, -4)   - rezultat 0
SELECT ROUND (123.4565, 2) - rezultat 123.4600
SELECT ROUND (123.4565, 2, 0) - rezultat 123.4500
SELECT ROUND (123 .4565, 2, 1) - rezultat 123.4500
SELECT TAN (PI () / 4)      - rezultat 0.9999999999999989

```

### 2.2.1.2. Funcții care operează asupra șirurilor de caractere

Functiile scalare pe șiruri de caractere operează, de regulă, asupra unui șir de caractere dat ca argument și returnează fie un șir de caractere, fie o valoare numerică.

Cele mai importante funcții pe șiruri de caractere împreună cu sintaxa acestora și valoarea returnată sunt prezentate mai jos:

Sintaxa	Semnificație
ASCII( <i>expresie_caracter</i> )	codul ASCII al celui mai din stânga caracter al argumentului;
CHAR( <i>expresie_numerică_ASCII</i> )	caracterul ASCII cu al căruia cod este <i>expresie_numerică ASCII</i> ;
CHARINDEX( <i>expresie_subșir</i> , <i>expresie_caracter</i> , <i>pozitie_start</i> )	se caută <i>expresie_subșir</i> în <i>expresie_caracter</i> începând cu poziția <i>pozitie_start</i> și se returnează poziția de început a primei apariții găsite sau 0 în caz contrar. Dacă <i>pozitie_start</i> are valoare negativă sau zero căutarea începe de la prima poziție din <i>expresie_caracter</i> ;
DIFFERENCE( <i>expresie_caracter</i> , <i>expresie_caracter</i> )	diferența dintre valorile SOUNDDEX ale celor două argumente și indică gradul de similaritate între două șiruri de caractere pe o scăla de la 0 la 4 (similaritate maximă);
LEFT( <i>expresie_caracter</i> , <i>lungime</i> )	segmentul din stânga lui <i>expresie_caracter</i> având

#### Observații:

- expresie\_numerică ASCII* se evaluează la un întreg între 0 și 255, în caz contrar funcția CHAR returnează valoarea NULL;

Sintaxa	Semnificație
LEN( <i>expresie_caracter</i> )	lungimea în caractere a șirului <i>expresie_caracter</i> exclusiv eventualele caractere spațiu de la sfârșitul șirului;
LOWER( <i>expresie_caracter</i> )	expresia caracter returnată în urma transformării în litere mici a tuturor caracterelor din <i>expresie_caracter</i> ;
LTRIM( <i>expresie_caracter</i> )	șirul de caractere rezultat prin eliminarea spațiilor din stânga lui <i>expresie_caracter</i> ;
NCHAR( <i>expresie_numerică_UNICOD</i> )	caracterul UNICOD cu al căuui cod este <i>expresie_numerică UNICOD</i> ;
PATINDEX('%' <i>pattern%</i> ', <i>expresie_caracter</i> )	poziția de start a primei aparții a lui <i>pattern</i> în <i>expresie_caracter</i> sau 0 în caz contrar;
REPLACE('' <i>expresie_sir1</i> '', ' <i>expresie_sir2</i> ', ' <i>expresie_sir3</i> ')	înllocuiesc cu <i>expresie_sir3</i> aparțiiile lui <i>expresie_sir2</i> din <i>expresie_sir1</i> se încadrează șirul de caractere dat ca prim argument între caracterele de delimitare (implicit 'T' și 'T' ori specificat prin al doilea argument ca fiind una din: ',', '.', '_J') pentru a forma un șir UNICOD valid ca identificator în SQL Server;
REPLICATE( <i>expresie_caracter</i> , <i>întreg</i> )	repeta <i>expresie_caracter</i> de un număr de ori specificat prin <i>întreg</i> ;
REVERSE( <i>expresie_caracter</i> )	reversează șirul de caractere dat ca parametru;
RIGHT( <i>expresie_caracter</i> , <i>lungime</i> )	segmentul din dreapta lui <i>expresie_caracter</i> având dimensiunea egală cu valoarea parametrului <i>lungime</i> ;
RTRIM( <i>expresie_caracter</i> )	șirul de caractere rezultat prin eliminarea spațiilor din dreapta lui <i>expresie_caracter</i> ;
SOUNDEX( <i>expresie_caracter</i> )	returnează un cod SOUNDEX de 4 caractere folosit la evaluarea similarității dintre două șiruri;
SPACE( <i>lungime</i> )	șir de caractere format dintr-un număr de <i>lungime</i> spații;
STR( <i>expresie_floating</i> [, <i>lungime</i> [, <i>zecimală</i> ]])	șir de caractere rezultat din conversia parametrului <i>expresie_floating</i> ;
STUFF( <i>expresie_caracter1</i> , <i>start</i> , <i>lungime</i> , <i>expresie_caracter2</i> )	șirul rezultat prin înlăturarea segmentului din <i>expresie_caracter1</i> care începe din poziția <i>start</i> și are <i>lungime</i> caractere;
SUBSTRING( <i>expresie_caracter</i> , <i>start</i> , <i>lungime</i> )	subșirul din <i>expresie_caracter</i> care începe din poziția <i>start</i> și are <i>lungime</i> caractere;
UNICODE( <i>expresie_caracter</i> )	codul UNICOD al celui mai din stânga caracter al argumentului;
UPPER( <i>expresie_caracter</i> )	expresia caracter rezultată în urma transformării în litere mari a tuturor caracterelor din <i>expresie_caracter</i> ;

2. expresie numerică UNICOD se evaluează la un întreg între 0 și 65535, în caz contrar funcția NCHAR returnează valoarea NULL;

3. expresie sir(1,2,3) pot fi de tip caracter sau binar; funcția REPLACE întorce o valoare de tip binar dacă cel puțin una dintre argumentele sale este de tip binar.

#### Exemplu:

```

SELECT ASCII('A')
SELECT CHAR(65)
SELECT UNICODE('A')
SELECT NCHAR(65)
SELECT UNICODE('À')
SELECT DIFFERENCE('mere','pere')
SELECT DIFFERENCE('mere','ananas')
SELECT SOUNDEX('mere')
SELECT SOUNDEX('pere')
SELECT SOUNDEX('ananas')
SELECT LEFT('ananas',3)
SELECT RIGHT('ananas',3)
SELECT SUBSTRING('ananas',3,3)
SELECT STR(123,456,7,3)
SELECT STR(123,456,7,2)
SELECT STR(123,456,6,2)
SELECT STUFF('mere',1,1,'p')
SELECT STUFF('mere',2,1,'u')
SELECT LEN('struguri')
SELECT LOWER('ABCDEFGH')
SELECT UPPER('abcdefg')
SELECT LTRIM('abc')
SELECT RTRIM('abc')
SELECT PATINDEX('%abc%', 'abcbcabcb')
SELECT PATINDEX('abc%', 'abcbcabcb')
SELECT PATINDEX('%abc', 'abcbcabcb')

```

- rezultat 65  
- rezultat 'A'  
- rezultat 197  
- rezultat 'À'  
- rezultat 3  
- rezultat 0  
- rezultat M600  
- rezultat P600  
- rezultat A552  
- rezultat 'ana'  
- rezultat 'nas'  
- rezultat 'ana'  
- rezultat '123,456'  
- rezultat '123,46'  
- rezultat '123,46'  
- rezultat 'pere'  
- rezultat 'mure'  
- rezultat 8  
- rezultat 'abcdefg'  
- rezultat 'ABCDEFGH'  
- rezultat 1  
- rezultat 1  
- rezultat 7

### 2.2.1.3. Funcții pentru manipularea valorilor de tip dată calendaristică

Aceste funcții scalare actionează asupra unor valori de tip dată calendaristică și returnează o valoare care poate fi de tip: sir de caractere, numeric sau dată calendaristică.

Funcțiile pentru manipularea valorilor de tip dată calendaristică împreună cu sintaxa acestora și valoarea returnată sunt prezentate mai jos:

Sintaxă	Semnificație
DATEADD(element_dată, număr, dată)	data rezultată prin adăugarea la data calendaristică specificată a unui interval de timp
DATEDIFF(element_dată, dată_inicială, dată_finală)	calculează diferența dintre datele calendaristice dată_inicială și dată_finală exprimată în intervale de timp de tipul element_dată;
DATENAME(element_dată, dată)	returnează un sir de caractere reprezentând numele elementului de data specificat prin parametrul element_dată;
DATEPART(element_dată, dată)	returnează un întreg reprezentând valoarea elementului de data specificat prin parametrul element_dată;
DAY(dată)	returnează un întreg reprezentând valoarea zilei din luna a datei calendaristice furnizată ca parametru;
GETDATE()	returnează data curentă din sistem în formatul standard;
MONTH(dată)	returnează un întreg reprezentând valoarea lunii din an a datei calendaristice furnizată ca parametru;
YEAR(dată)	returnează un întreg reprezentând valoarea anului pentru data calendaristică furnizată ca parametru.

```

SELECT PATINDEX('abc', 'abcbcabcb') - rezultat 0
SELECT REPLACE('abcbcabcb', 'abc', 'a') - rezultat aaa
SELECT QUOTENAME('abc def') - rezultat [abc def]
SELECT REPLICATE('abc', 3) - rezultat 'abcabcabc'
SELECT SPACE(3) - rezultat ''

```

```

SELECT 'replicate('abc', 3) - rezultat 'abcabcabc'
SELECT REVERSE('abc') - rezultat 'cba'

```

## Observații:

1. *element\_dată* este un parametru care specifică o anumită parte a datei calendaristică. Tabelul de mai jos indică părțile unei date calendaristice împreună cu abrevierile ce le corespund:

Parte a datei	Abreviere
anul	yy sau yyyy
trimestrul	qq sau q
luna	mm sau m
ziua din an	dy sau Y
zīua din lună	dd sau d
zīua din săptămānă	wd
săptămāna	dw
ora	hh
minutul	mi sau n
secunda	ss sau s
millisecunda	ms

2. *dată\_inicială și dată\_finală* sunt expresii care se evaluatează la valori de tip date calendaristică (datetime sau smalldatetime);

## Exemple:

```

SELECT DATEADD(YY, 1, '01/21/2001') - rezultat 2002-01-21 00:00:00.000
SELECT DATEADD(ss, 1, '01/21/2001') - rezultat 2001-01-21 00:00:01.000

SELECT DATEDIFF(YY, '01/01/2000', '01/01/2001') - rezultat 1
SELECT DATEDIFF(qq, '01/01/2000', '01/01/2001') - rezultat 4
SELECT DATEDIFF(mm, '01/01/2000', '01/01/2001') - rezultat 12
SELECT DATEDIFF(day, '01/01/2000', '01/01/2001') - rezultat 366
SELECT DATEDIFF(hh, '01/01/2000', '01/01/2001') - rezultat 8784
...
SELECT DATEDIFF(day, '01/01/2000', getdate()) - numărul de zile
scurse de la 1 ianuarie 2000 și
până la data curentă;
```

SELECT DATENAME(mm, '01/22/2001') - rezultat 'January'  
 SELECT DATENAME(dw, '01/22/2001') - rezultat 'Monday'

SELECT DATEPART(mm, '01/22/2001') - rezultat 1

SELECT DATEPART(dw, '01/22/2001') - rezultat 2

SELECT DAY('01/22/2001') - rezultat 22

SELECT MONTH('01/22/2001') - rezultat 1

SELECT YEAR('01/22/2001') - rezultat 2001

## 2.2.1.4. Funcții sistem (conversii)

Dintre funcțiile sistem de tip scalar prezentă în acest paragraf doar funcțiile de conversie. Funcțiile CASE și cele pentru tratarea valorilor NULL vor fi prezentate ulterior.

În SQL Server, sunt posibile două niveluri de conversii între tipurile de date:

- Când data dîntr-un obiect este mutată, comparată sau combinată cu o dată dintr-un alt obiect - data poate fi convertită din tipul de data al unui obiect în tipul de data al celuilalt obiect.
- Când data dîntr-o coloană rezultată dintr-o tranzacție SQL, codul returnat sau parametrul de ieșire al unei proceduri stocate este mutat într-o variabilă program, aceasta trebuie convertit la tipul de data al variabilei.

Există două categorii de conversii:

- Conversiile implicate care sunt transparente pentru utilizator. SQL Server convertește automat datele dîntr-un tip în altul ori de câte ori este posibil. De exemplu dacă smallint este comparat cu un int, atunci smallint este convertit implicit la int înaintea comparării.
- Conversiile explicate sunt realizate de către utilizator prin folosirea funcțiilor CAST sau CONVERT.

### Funcțiile CAST și CONVERT

Realizează conversia explicită a unei expresii de un anumit tip în alt tip. CAST și CONVERT sunt similare în funcționare, fiind diferite ca sintaxă. Funcția CAST este compatibilă cu standardul SQL'92 și este preferată (acolo unde este posibil!) față de CONVERT care este specifică SQL Server.

#### Sintaxa:

*Utilizând CAST:*

CAST(*expresie AS tip\_data*)

*Utilizând CONVERT:*

CONVERT(*tip\_data [ ( lungime ) ], expresie [, stil ]*)

unde:  
*expresie* - orice expresie SQL Server validă.

**tip\_data** - tipul de date întă furnizat de sistem. Tipurile de date utilizator nu pot fi folosite în conversii.

**lungime** - parametru optional al tipurilor de date **nchar**, **nvarchar**, **char**, **varchar**, **binary** sau **varbinary**.

**stil** - este stilul formatului de date calendaristic folosit la conversia **datetime** sau **smalldatetime** la un tip caracter (tipurile **nchar**, **nvarchar**, **char**, **varchar**, **nchar**, sau **nvarchar**), sau formatul unui sir când se convertește **float**, **real**, **money** sau **smalldmoney** la un tip caracter.

În tabela de mai jos primele două coloane din stânga reprezintă valorile pe care le poate avea parametrul **stil** în cazul conversiei de la **datetime** sau **smalldatetime** la un tip caracter.

Fără secol (yy)	Cu secol (yyyy)	Standard	Intrare/Iesire**
0 or 100 (*)	Implicit		mon dd yyyy hh:miAM (PM)
1	101	USA	mm/dd/yy
2	102	ANSI	yy-mm-dd
3	103	British/French	dd/mm/yy
4	104	German	dd.mm.yy
5	105	Italian	dd-mm-yy
6	106	-	dd mon yy
7	107	-	mon dd, yy
8	108	-	hh:mm:ss
9 or 109 (*)		Implicit + milisecunde	mon dd yyyy hh:mi:ss:mmmAM (PM)
10	110	USA	mm-dd-yy
11	111	JAPAN	yy/mm/dd
12	112	ISO	ymd
13, 113 (*)		Europa implicit + milisecunde	dd mon yyyy hh:mn:ss:mmm(24h)
14	114	-	hh:mi:ss:mmm(24h)
20, 120 (*)	ODBC canonice		yyyy-mm-dd hh:mi:ss(24h)
21, 121 (*)	+milisecunde		yyyy-mm-dd hh:mi:ss.mmm(24h)
*		Valoare implicită (stil 0 sau 100, 9 sau 109, 13 sau 113, 20 sau 120 și 21 sau 121)	returnează întotdeauna secolul (yyyy).
**			Intrare când se convertește la <b>datetime</b> , Iesire când se convertește la o date de tip caracter.

#### Observații:

- Implicit, SQL Server interpretează cele două cifre din an considerând anul 2049 ca an de referință. Aceasta înseamnă că anul 49 e interpretat ca 2049, iar anul 50 e interpretat ca 1950.

2. Când se convertește un **smalldatetime** la un tip caracter, stilurile ce includ secunde sau milisecunde vor avea zero în aceste poziții.

Tabelul următor indică natura conversiei din **float** sau **real** la un tip caracter în funcție de parametrul **stil**:

Valoare	Iesire
0 (implicit)	6 cifre maxim. Utilizată în notatiile stîrnifice, când corespunde.
1	Intotdeauna 8 cifre. Intotdeauna folosit în notatiile stîrnifice.
2	Intotdeauna 16 cifre. Intotdeauna folosit în notatiile stîrnifice

În tabelul următor coloana din stânga reprezintă valorile parametrului **stil** în cazul conversiei **money** sau **smalldmoney** la un tip caracter:

Valoare	Iesire
0 (implicit)	Fără virgulă la fiecare trei cifre din stânga punctului zecimal și cu două cifre la dreapta punctului zecimal. De ex., 4235.98.
1	Virgulă după fiecare trei cifre la stânga punctului zecimal și cu două cifre după punctul zecimal. De exemplu, 3,510.92.
2	Fără virgulă la fiecare trei cifre din stânga punctului zecimal și patru cifre la dreapta punctului zecimal. De ex., 4235.9819.

#### Observații:

- Conversiile implicate nu sunt suportate de către tipurile **text** și **image**. Se pot însă converti explicit date de tip text la tipul caracter și date de tip **image** la **binary** sau **varbinary**, dar lungimea maximă care poate fi specificată este 8000.
- Dacă se încearcă o conversie care nu este posibilă (de exemplu o expresie de tip caracter care include litere la **int**), SQL Server generează un mesaj de eroare.
- La conversia expresiilor de tip caracter sau binar (**char**, **nchar**, **nvarchar**, **varchar**, **binary** sau **varbinary**) la o expresie de un tip diferit, data poate fi trunchiată, tipărită parțial sau poate fi returnată eroare.

#### Exemple:

```

SELECT CAST(123 AS char(3))           - rezultat '123'
SELECT CONVERT(char(3), 123)          - rezultat '123'
SELECT CAST(123 AS char(2))          - rezultat *
SELECT CAST(123 AS char(4))          - rezultat '123'
SELECT CAST(123 AS varchar(4))        - rezultat '123'
SELECT CAST(GETDATE() AS varchar(12)) - rezultat 'Jul 13 2001'
SELECT CONVERT(varchar(12), GETDATE(), 3) - rezultat '13/07/2001'
SELECT CONVERT(varchar(12), GETDATE(), 103) - rezultat '13/07/2001'

```

## 2.4. Exerciții și probleme

1. Să se indice rezultatul returnat de următoarele operații SELECT:

```
SELECT POWER(11, 0.5)
SELECT SQRT(9)
```

```
SELECT ROUND(748, -1)
```

```
SELECT ROUND(48.58, 1)
```

```
SELECT ROUND(78.34, 1)
```

```
SELECT ROUND(701, -2)
```

```
SELECT ROUND(12.12, 2)
```

```
SELECT CHARINDEX('aa', 'aaaaaa')
```

```
SELECT CHARINDEX('aa', 'aaaaaa', 4)
```

```
SELECT LEFT('abcdef', 2)
```

```
SELECT RIGHT('abcdef', 4)
```

```
SELECT SUBSTRING('abcdef', 2, 3)
```

```
SELECT REPLACE('abcababc', 'a', 'abc')
```

```
SELECT CAST('123' AS char(2))
```

```
SELECT CAST('07/13/01' AS datetime)
```

```
SELECT CONVERT(varchar(12), CAST('07/06/01' AS datetime), 1)
```

```
SELECT CONVERT(varchar(12), CONVERT(datetime, '07/06/01', 1), 6)
```

```
SELECT CONVERT(varchar(12), CONVERT(datetime, '07/06/01', 3), 6)
```

Tabelele sunt cele mai importante obiecte dintr-o bază de date relatională. Toate datele sunt stocate în tabele organizate sub formă de lini și coloane. Fiecare tabelă corespunde unei relații din schema relațională a bazei de date. Linile tabelei corespund tupelor relației, iar coloanele corespund atribuitorilor acesteia.

Proiectarea unei baze de date presupune, în primul rând, stabilitarea colecției de tabele și a descrierii tabelelor împreună cu toate proprietățile asociate. Înainte de crearea unei tabele noi trebuie luate o serie de decizii de proiectare referitoare la:

- tipurile de date care vor fi stocate în tabelă;
- coloanele tabelei și tipul de date (eventual lungimea) pentru fiecare coloană;
- coloanele care (nu)acceptă valori NULL;
- constrângeri, reguli, valori implicate;
- indecsi necesari, coloanele care formează cheia primară ori care sunt parte a unei chei strâinăte.

Odată proiectată baza de date se poate trece la creația tabelelor în care vor fi stocate datele. Stocarea datelor se face în tabelele permanente care sunt grupate în una sau mai multe fișiere ale bazei de date. Pe lângă tabelele permanente, în SQL Server, există posibilitatea de crea tabele temporare. Acestea sunt similare tabelelor permanente, cu excepția faptului că sunt stocate în baza de date sistem *tempdb* de unde sunt stocate în mod automat de îndată ce nu mai sunt folosite. Tabelele temporare pot fi locale sau globale. Numele unei tabele temporare locale începe cu simbolul #, ea este vizibilă numai pentru utilizatorul curent și este distrusă când acesta se deconectează de la SQL Server. Numele tabelelor temporare globale încep cu ##, sunt vizibile pentru toți utilizatorii (care au drepturi corespunzătoare) și sunt păstrate în sistem până la deconectarea ultimului utilizator care le folosește.

## 3. TABELE SI INDECȘI

### 3.1. Instrucția CREATE TABLE

Creează o nouă tabelă.

Sintaxa (simplificată):

```
CREATE TABLE [nume_baza_date].[proprietar].[proprietar.] nume_tabel
( {<definite_coloana>} <construire_tabela> ) [,n ]
```

unde:

*nume\_baza\_date* – este numele bazei de date în care se creează tabela.

Dacă nu se specifică, atunci tabela va fi creată în baza de date curentă.

*proprietar*

– este numele utilizatorului care va fi proprietarul tabeliei.

Dacă nu se specifică, atunci proprietarul va fi chiar utilizatorul curent. Numai utilizatorii privilegiati, cum ar fi administratorul de sistem, pot crea tabele pentru alți utilizatori.

*nume\_tabel*

– este numele tabelei care se creează.

Combinarea *proprietar.nume\_tabel* trebuie să fie unică la nivelul bazei de date.

*definire\_coloana* – se referă la definitia completă a unei coloane împreună cu constrângările impuse asupra sa.

*construire\_tabela* – se referă la constrângeri exprimate la nivelul tabeliei și sunt, în general, constrângeri care nu pot fi exprimate la nivel de coloană sau se exprimă mai avantajos la nivel de tabelă.

### 3.1.1. Definirea unei coloane

Definirea unei coloane se face după următoarea sintaxă:

```
<definire_coloana> ::=  
{ nume_coloana tip_data }  
[ [ DEFAULT expresie_constanta ] | [ IDENTITY [ ( origine, increment ) ] ]  
[ ROWGUIDCOL ] [ < construire_coloana > ]  
| nume_coloana AS expresie_coloana_calculata [ ,n ]  
unde:  
nume_coloana – este numele unei coloane din tabelă.
```

Numele de coloană poate fi omis dacă tipul de date este timestamp, caz în care numele coloanei este chiar timestamp.

*tip\_data*

– specifică tipul de date al coloanei.

Poate fi un tip de date sistem sau un tip utilizator.

#### DEFAULT expresie\_constanta

– specifică o valoare implicită care este atribuită ori de către nu se specifică o valoare explicită pentru coloana corespunzătoare.

Această opțiune nu se poate aplica dacă tipul de date al coloanei este timestamp ori pentru coloane cu proprietatea IDENTITY. *expresie\_constanta* poate fi constantă, valoarea NULL sau o funcție sistem (de ex. GETDATE()).

**IDENTITY** – indică o coloană pentru care SQL Server generează în mod automat valori incrementale, unice la nivel de tabelă.

Proprietatea poate fi asociată tipurilor de date tinyint, smallint, int, bigint, decimal(p,0) sau numeric(p,0). Într-o tabelă poate exista cel mult o singură coloană cu proprietatea IDENTITY.

*origine* – este valoarea care se atribuie în coloana IDENTITY pentru prima tuplă inserată în tabelă.

Dacă parametrul *origine* lipsește, atunci se consideră implicit valoarea 1.

*increment* – este valoarea de increment față de valoarea ultimei tuple inserate.

Este implicit 1 în lipsa unei specificări explicate.

**ROWGUIDCOL** – indică faptul că noua coloană este de tip uniqueidentifier, pentru care sistemul acceptă numai valori generate prin funcția NEWID care garantează unicitatea valorilor la nivel de bază de date.

O tabelă poate avea cel mult o singură coloană cu proprietatea ROWGUIDCOL. Proprietatea ROWGUIDCOL în sine, nu garantează unicitatea valorilor și nici nu generează automat valorile unice. Pentru a genera valori unice se va folosi funcția NEWID la orice operație de inserare în această coloană.

*construire\_coloana*

– se referă la o constrângere exprimată la nivel de coloană.

*expresie\_coloana\_calculata* – este o expresie care definește valoarea pentru o coloană calculată.

- O coloană calculată este o coloană virtuală care nu este stocată fizic în baza de date, ci este calculată printre-expresie pe baza valorilor din alte coloane ale tabelui. Expressia poate fi numele unei coloane, o constantă, o funcție, o variabilă, respectiv orice combinație a acestora. Coloanele calculate se pot folosi în clauzele **SELECT**, **WHERE**, **ORDER BY** din interogări și în orice loc unde este legală plasarea unei expresii cu respectarea următoarelor restricții:

- O coloană calculată nu poate fi folosită la definirea unor contrângeri de tip **DEFAULT** sau **FOREIGN KEY**, respectiv nu poate fi definită cu opțiunea **NOT NULL**. Totuși, o coloană calculată poate fi folosită ca și cheie de indexare, ca parte a cheii primare și i se poate asocia o constrângere de tip **UNIQUE**, cu condiția ca valoarea calculată a coloanei să fie obținută printr-o expresie deterministă și tipul de date al rezultatului expresiei să fie unul acceptat pentru coloanele de indexare. De exemplu, dacă tabela are coloanele de tip întreg **a** și **b**, atunci coloana calculată **a+DATEPART(dd, GETDATE())** nu poate fi indexată deoarece valoarea ei se poate schimba în timp și va fi alta la fiecare invocare.
- Într-o coloană calculată nu se pot inseră valori prin instrucția **INSERT**, respectiv nu poate fi modificată prin **UPDATE**.

#### Exemplu:

- Tabela *Persoana*, creată prin instrucția de mai jos, are următoarele coloane:
  - ID*, coloană de identificare unică, cu proprietatea **IDENTITY**, pentru care se vor crea automat, în ordinea introducerii tupelor, valorile: 1,2,3 și a.m.d.
  - Nume*, său de caractere pentru stocarea numelui;
  - Anul\_nasterii*, coloană de tip întreg cu valoarea implicită 1950, nașterii, va indica la orice referire vârsta actuală a persoanei.

```
CREATE TABLE Persoana (
  ID int IDENTITY,
  Nume varchar(15),
  Anul_nasterii int DEFAULT 1950,
  Virsta AS DATEPART(yy, GETDATE()) - Anul_nasterii)
```

De remarcat că la inserarea unei tuple în tabela *Persoana* singurul câmp pentru care trebuie specificată o valoare explicită este *Nume*. Pentru toate celelalte câmpuri sistemul este capabil să furnizeze valori corespunzătoare dacă este nevoie. Astfel, dacă se execută secvența:

```
INSERT Persoana(Nume) VALUES ('Ion') - inserare tupla tabel
SELECT * FROM Persoana - afișare conținut tabel
```

immediat după crearea tabelului, atunci se obține următorul rezultat:

(1 row(s) affected)			
ID	Nume	Anul_nasterii	Virsta
1	Ion	1950	51

(1 row(s) affected)

### 3.1.2. Constrângeri

Constrângările asociate unei tabele se pot defini fie la nivel de coloană, fie la nivelul tabelui. Constrângările sunt proprietăți speciale prin care se asigură integritatea datelor. Anumite constrângeri au ca efect crearea de indecsi pentru tabele și coloanele acestora. În SQL Server se pot defini următoarele categorii de constrângeri:

- NOT NULL**, prin care se interzic valorile NULL pentru o anumită coloană;
- PRIMARY KEY** este constrângerea de integritate a relației care impune valori unice și nemile pentru coloanele care fac parte din cheia primară;
- UNIQUE** impune valori unice pentru una sau mai multe coloane, este de fapt mecanismul de definire al cheilor candidate din tabel;
- FOREIGN KEY** este constrângerea de integritate referențială, valorile cheii străine trebuie să fie o submulțime a valorilor cheii dintr-o tabelă referată;
- constrângeri **CHECK**, prin care se impune ca valorile dintr-o anumită coloană să satisfacă o expresie logică specificată, este o formă de realizare (partială) a integrării domeniului prin restricționarea valorilor care pot fi atribuite unei coloane.

### 3.1.2.1. Exprimarea constrângerilor la nivel de coloană

Sintaxa pentru definirea constrângerilor la nivel de coloană este următoarea:

```
<constrangere_coloana> ::= [ CONSTRAINT nume_constrangere ]
  [ [ NULL | NOT NULL ]
  [ { PRIMARY KEY | UNIQUE }
  [ CLUSTERED | NONCLUSTERED ]
  [ [ FOREIGN KEY ] REFERENCES tabela_referita ([coloana_referita])
    [ ON DELETE { CASCADE | NO ACTION }
    [ ON UPDATE { CASCADE | NO ACTION }
    [ NOT FOR REPLICATION ]
  ]
  [ CHECK [ NOT FOR REPLICATION ] (expresie_logica)
  ]
  unde:
```

CONSTRAINT

*nume\_constrangere* – cuvântul cheie optional

genii. *nume\_constrangere* este numele constrângerii. Numele trebuie să fie unic la nivelul bazei de date.

**NUL | NOT NULL** – specifică dacă coloana curentă acceptă sau nu

valori NULL. De fapt NULL nu este o constrângere în sens strict, dar poate fi specificat optional. O opțiunea implicită pentru acest tip de constrângere este influențată de o serie de setări de la nivelul bazei de date și al sesiunii de lucru. Din acest motiv este recomandabil ca la crearea unei tabele să se specifică explicit

**PRIMARY KEY** – definește o constrângere de cheie primară pentru o coloana curentă. Poate exista o singură cheie primară pentru o tabelă dată.

**UNIQUE** – constrângere de unicitate, impune valori unice pentru coloana curentă. Pot exista mai multe constrângerii de tip **UNIQUE** la nivelul unei tabele.

**CLUSTERED | NONCLUSTERED** – indică tipul de index prin care se

realizează o constrângere de tip **PRIMARY KEY** sau **UNIQUE**. Opțiunea implicită pentru **PRIMARY KEY** este **CLUSTERED**, iar pentru **UNIQUE** este **NONCLUSTERED**. Pentru un tabel se poate defini cel mult un index (constrângere) de tip **CLUSTERED**. Un index de tip **CLUSTERED** este mai eficient datorită faptului că se mapează peste ordonarea fizică a tuplelor

din baza de date (se sortează tuplele după cheia indexului **CLUSTERED**!).

**FOREIGN KEY ... REFERENCES** – marchează începutul definitiei unei coloane. Această constrângere impune condiția că orice valoare a coloanei curente să se regăsească printre valorile coloanei *coloana\_referita* din tabela *tabela\_referita*. Asupra coloanei *coloana\_referita* trebuie să existe definită o constrângere de tip **PRIMARY KEY** sau **UNIQUE** sau un index cu opțiunea **UNIQUE**.

(*coloana\_referita*) – este numele tabelei referite într-o constrângere de tip **FOREIGN KEY**.

**ON DELETE { CASCADE | NO ACTION }** – specifică tipul de acțiune prin care SQL Server va menține integritatea bazei de date în cazul unei operații **DELETE** care intră în conflict cu constrângerea de cheie străină. Opțiunea **CASCADE** are ca efect ștergerea în **ACTION** și are ca efect anularea operației **DELETE** și generarea unui mesaj de eroare, dacă aceasta intră în conflict cu constrângerea. Opțiunea **CASCADE** are ca efect ștergerea în cascădă a tuturor tuplelor care fac referire la o tuplă care se șterge din tabela referită. Tabela referită trebuie să fie în aceeași bază de referințe între baze de date diferite. Opțiunea **CASCADE** nu poate fi definită pentru operația **DELETE** dacă există definit un trigger **INSTEAD OF DELETE** pentru tabela în discuție.

**ON UPDATE { CASCADE | NO ACTION }** – specifică tipul de acțiune prin care SQL Server va menține integritatea bazei de date în cazul unei operații **UPDATE** care intră în conflict cu constrângerea de cheie străină. Opțiunea implicită este **NO ACTION** și are ca efect anularea operației **UPDATE** și generaarea unui mesaj de eroare, dacă aceasta intră în conflict cu constrângerea. Dacă se specifică opțiunea **CASCADE**, atunci modificările din tabela referită sunt propagate asupra tuturor acelor tuple din tabela de referință care referă tuple modificate în tabela referită. Tabela referită trebuie să fie în aceeași bază de date cu tabela curentă (de referință), nu sunt posibile legături referințiale între baze de date diferite. Opțiunea **CASCADE** nu

poate fi definită pentru operația UPDATE dacă există definit un trigger INSTEAD OF UPDATE pentru tabela în discuție.

Opțiunea CASCADE a fost introdusă începând cu versiunea SQL Server 2000. Opțiunea NO ACTION acoperă funcționalitatea constrângerii de tip FOREIGN KEY din versiunile până la SQL Server 7.0 și corespunde variantei de impunere a integrității referențiale prin restricționare.

**CHECK** – marchează începutul definiției unei constrângerii de integritate a domeniului.

**NOT FOR REPLICATION** – este o opțiune aplicabilă în cazul bazelor de date distribuite și indică faptul că constrângerea este aplicată numai modificărilor efectuate de către utilizatori, nu și procesului de replicare prin care SQL Server transferă date în tabela curentă..

*expresie\_logica* – este o expresie logică care returnează o valoare TRUE sau FALSE. Expresia logică poate face referire doar la coloana curentă.

#### Exemplu:

2. Redefinim tabela *Persoana* de la exemplul 1) astfel încât să o completăm cu următoarele constrângerii:

- coloana *ID* este cheie primară;
- coloana *Nume* să iabă valori unice și nenule;
- *Anul\_nasterii* să fie mai mare decât 1900, dar nu mai mare decât 1950.

Prin urmare:

```
IF EXISTS (SELECT * FROM sysobjects  
          WHERE id = object_id('Persoana'))  
    AND OBJECTPROPERTY(id, 'IsUserTable') = 1)  
GO  
CREATE TABLE Persoana(  
  ID int IDENTITY PRIMARY KEY,  
  Nume varchar(15) UNIQUE NOT NULL,  
  Anul_nasterii int DEFAULT 1950 CHECK  
                ((Anul_nasterii>1900 AND  
                 Anul_nasterii<DATEPART(YY, GETDATE())-Anul_nasterii))
```

În urma acestei redifiniri operația de inserare:

```
INSERT Persoana VALUES ('Ion', 1850)
```

rezultă eroarea: "Violation of CHECK constraint 'Anul\_nasterii'". De remarcat, că dacă se modifică valoarea DEFAULT pentru această coloană la 1850 (ori orice altă valoare din afara plajei permise!) tabela va fi creată fără nici o dificultate, dar nu va fi posibilă nici o inserare în tabelă altfel decât cu specificarea explicită a unei valori valide pentru coloana *Anul\_nasterii*.

### 3. Definim tabela *Imprumut* cu următoarele coloane:

- *ID*, identificator unic de împrumut, definit cu proprietatea IDENTITY;
- *PID*, referința la coloana *ID* din tabela *Persoana*;
- *Titlu*, sir de caractere pentru a stocca titlul cărții împrumutate (valori nenule).

Instrucțiunea de definire a acestei tabele este:

```
CREATE TABLE Imprumut(  
  ID int IDENTITY PRIMARY KEY,  
  PID int FOREIGN KEY REFERENCES Persoana (ID),  
  Titlu varchar (30) NOT NULL)
```

#### 3.1.2. Exprimarea constrângerilor la nivel de tabelă

Nu întotdeauna o constrângere poate fi exprimată la nivelul unei singure coloane. Atunci când definiția unei constrângerii implică mai multe coloane poate fi mai avantajoasă sau chiar necesară exprimarea acesteia la nivel de tabelă. Sintaxa pentru definirea constrângerilor la nivel de tabelă este următoarea:

```
<constrangere_tabela> ::= [CONSTRAINT nume_constrangere]  
{ { PRIMARY KEY | UNIQUE }  
[CLUSTERED | NONCLUSTERED ]  
(coloana [ASC | DESC] [, n])  
}  
[FOREIGN KEY (coloana [, n])  
REFERENCES tabela_referita [ (coloana_referita [, n])]  
[ ON DELETE { CASCADE | NO ACTION } ]  
[ ON UPDATE { CASCADE | NO ACTION } ]  
[ NOTFOR REPLICATION ]  
| CHECK [expresie_logica]
```

unde:

coloana

- este o coloană sau listă de coloane peste care se definește constrângerea.

[ASC | DESC]

- specifică ordinea în care se sorteză coloanele participante la definirea constrângerii. Opțiunea implicită este **ASC**.
- indică repetarea într-o listă a unui anumit tip de element.

Observație:

Toate celelalte elemente de sintaxă își păstrează semnificația de la definiția constrângerilor la nivel de coloană cu următoarele deosebiri:

1. Constrângere de tip **PRIMARY KEY** sau **UNIQUE** la nivel de tabelă se poate defini peste mai multe coloane și are ca argument o listă de coloane.
2. Contrângerea **FOREIGN KEY** specifică lista explicită a coloanelor care formează cheia străină. Această listă trebuie să corespundă unu la unu cu lista coloanelor referite din tabela referită.
3. Expresia logică atașată unei constrângeri de tip **CHECK** la nivel de tabelă poate face referire la orice coloană, dar numai din tabela curentă.

Exemplu:

4. Pentru a defini o cheie primară sau o cheie străină formată din mai multe coloane, constrângerile corespunzătoare trebuie exprimate la nivel de tabelă. Câmpurile *C1* și *C2* formează cheia primară în tabela *Referita*, cheie care este referată de perechea *RC1* și *RC2* care formează o cheie străină în tabela *Referinta*.

```
CREATE TABLE Referita (
    C1 int,
    C2 int,
    PRIMARY KEY (C1, C2)
)

CREATE TABLE Referinta (
    RC1 int,
    RC2 int,
    FOREIGN KEY (RC1, RC2) REFERENCES Referita(C1, C2)
```

Observație:

Constrângerile la nivel de coloană trebuie definite imediat după specificarea tipului de date pentru coloana respectivă, ele sunt parte a definiției coloanei. Ordinea de specificare a constrângerilor este arbitrară. Constrângerile la nivel de tabelă pot fi specificate oriunde în definiția tabeliei și trebuie despărțite prin virgulă pot fi specificate oriunde în definițiiile altor constrângerii. De exemplu, în varianta de mai jos, tabela *Referita* mai conține două constrângerii de tip **CHECK** (*C1 < C2* și *C1 > C2 + 10*).

```
CREATE TABLE Referita (
    C1 int,
    CHECK (C1>C2+10),
    C2 int,
    PRIMARY KEY (C1, C2),
    CHECK (C1<C2)
```

## 3.2. Instrucția **ALTER TABLE**

Modifică definiția unei tabele realizând orice combinație a următoarelor categorii de operații:

- modificarea definiției unei coloane sau constrângerii;
- adăugarea/ștergerea unei coloane sau constrângerii;
- activarea/dezactivarea unei constrângerii sau a unui trigger.

Sintaxa (simplificată):

```
ALTER TABLE name_table
  { [ALTER COLUMN name_colonană
      { [tip_data_nou | (precizia [, scala])]
      [NULL | NOT NULL]
      | [ADD | DROP] ROWGUIDCOL ]
    ]
    | ADD {<definire_colonă>
      [DEFAULT expresie_constanță [WITH VALUES]] [, n]
      [WITH CHECK | WITH NOCHECK]
      ADD {<constringere_tabelă>} [, n]
    }
    | ADD
      { DEFAULT expresie_constanță [FOR colonă]
        [WITH VALUES] [, n]
      }
    | DROP
      { [CONSTRAINT] nume_constringere | COLUMN colonă] [, n]
        [WITH CHECK | WITH NOCHECK] { CHECK | NOCHECK }
        CONSTRAINT { ALL | nume_constringere [, n] }
```

```
| {ENABLE | DISABLE} TRIGGER { ALL | nume_trigger [,n ] }
```

unde:

*nume\_tabel* – este numele tabelului care se modifică. Dacă tabela nu este

în baza de date curentă sau nu este proprietatea utilizatorului curent, atunci se va specifica explicit numele bazei de date și/sau proprietarul.

**ALTER COLUMN** *nume\_coloana* – specifică modificarea coloanei *nume\_coloana*. Coloana modificată nu poate fi:

- o coloană având tipul de dată **text**, **image**, **ntext** sau **timestamp**;
- coloana cu proprietatea **ROWGUIDCOL**;
- o coloană calculată sau o coloană folosită într-o coloană calculată;
- o coloană folosită într-un index, într-o cheie primară ori cheie străină;
- o coloană pentru care s-a definit o constrângere **CHECK** sau **UNIQUE**;
- o coloană pentru care s-a definit o valoare **DEFAULT**.

*tip\_data\_nou* [ *precizia* [ , *scala* ] ] – este tipul de data nou, asociat cu

coloana modificată, împreună cu precizia și scala definite pentru acest tip de date. *tip\_data\_nou* trebuie să respecte următoarele criterii:

- datele existente trebuie să poată fi convertite implicit la noul tip de date;
- *tip\_data\_nou* nu poate fi **timestamp**;
- dacă coloana modificată are proprietatea **IDENTITY**, atunci *tip\_data\_nou* trebuie să fie un tip de date care suportă această proprietate.

**NULL | NOT NULL** – specifică dacă coloana poate accepta sau nu

valori NULL. Orice coloană nou adăugată trebuie, fie să accepte valori NULL sau să aibă definită o valoare **DEFAULT**. Dacă coloana adăugată acceptă valori NULL și nu are definită o valoare **DEFAULT**, atunci coloana va fi initializată cu valori NULL pentru fiecare tupla a tabeliei. Dacă coloana adăugată acceptă valori NULL și-a definit și o valoare **DEFAULT**, atunci prin folosirea opțiunii **WITH VALUES** se poate forța initializarea coloanei cu valoarea **DEFAULT**. Dacă coloana nu acceptă valori NULL, atunci definirea valorii **DEFAULT** este

obligatorie și aceasta este valoarea cu care se realizează în mod automat inițializarea coloanei.

**[, (ADD | DROP) ROWGUIDCOL ]** – o coloană de tip **ROWGUIDCOL** nu poate fi modificată, ci numai adăugată (ADD) ori ștearsă (DROP).

**ADD** – specifică adăugarea a una sau mai multe coloane sau constrângeri.

*definire\_coloana* – are aceeași semnificație ca la instrucțiunea **CREATE TABLE**.

**[DEFAULT *expresie\_constanta* [WITH VALUES ]]** – opțiune care definește valoarea implicită, dată prin *expresie\_constanta*, pentru coloana adăugată. Nu se poate aplica coloanelor **IDENTITY** sau **ROWGUIDCOL**.

**WITH CHECK | WITH NOCHECK** – specifică dacă datele existente sunt sau nu validate în raport cu o constrângere nou adăugată sau reactivată. Opțiunea **WITH CHECK** este implicită în cazul unei constrângeri noi, iar **WITH NOCHECK** este implicită în cazul constrângерilor reactivate.

Opțiunile **WITH CHECK** și **WITH NOCHECK** nu pot fi folosite pentru constrângările de tip **PRIMARY KEY** și **UNIQUE**.

Constrângările definite cu opțiunea **WITH NOCHECK** nu sunt luate în seamă de sistem, fiind ignorate până în momentul reactivării lor printr-o instrucțiune **ALTER TABLE** cu clauza **CHECK CONSTRAINT ALL**.

*constrangere\_tabela* – are aceeași semnificație ca la instrucțiunea **CREATE TABLE**.

**ADD { DEFAULT *expresie\_constanta* [ FOR *coloana* ] | WITH VALUES}** – definirea unei valori **DEFAULT** pentru o coloană existentă.

**DROP { [CONSTRAINT] *nume\_constrangere* | COLUMN *coloana* }** – specifică ștergerea unei constrângeri ori a unei coloane. Se pot șterge mai multe coloane și/sau constrângeri printre singură clauză **DROP**. O coloană nu poate fi ștearsă dacă:

- este folosită într-un index;
- este folosită într-o constrângere **CHECK**, **FOREIGN KEY**, **UNIQUE** sau **PRIMARY KEY**.

{ CHECK | NOCHECK} CONSTRAINT – specifică activarea (cu opțiunea CHECK) ori dezactivarea (cu opțiunea NOCHECK) constrângerilor de tip FOREIGN KEY și CHECK.

**ALL** – specifică faptul că toate constrângerile sunt activate ori dezactivate.

**nume\_constrințe** – numele constrângerilor care sunt activate ori dezactivate în mod selectiv.

{ENABLE | DISABLE} TRIGGER - specifică activarea (cu opțiunea ENABLE) ori dezactivarea (cu opțiunea DISABLE) trigger-elor definite pentru tabela în discuție.

**ALL** - specifică faptul că toate trigger-ele sunt activate ori dezactivate.

**nume\_trigger** - numele trigger-elor care sunt activate ori dezactivate în mod selectiv.

#### Observații:

1. La folosirea instrucțiunii **ALTER TABLE** trebuie avut în vedere faptul că modificările în definiția unei tabele pot avea ca efect modificări implicate asupra datelor existente sau generări de date pentru coloanele noi. Aceste operații pot fi mai consumatoare de resurse și de lungă durată.
2. Dacă într-o tabelă sunt modificate coloane care sunt referite printre constănțe de tip FOREIGN KEY, atunci modificările se propagă și asupra tabelelor de referință în cazul opțiunii CASCADE ori modificarea inițială este revocată dacă opțiunea este **NO ACTION**.

#### Exemple:

5. În secvența de mai jos se creează inițial tabela **T** cu un singur câmp **ID** în care se inserează două tuple. Ulterior tabela este modificată prin adăugarea coloanelor **A**, **B** și **C**, după care se afișează conținutul tabelului:

```
CREATE TABLE T (ID int)
GO
INSERT T VALUES (1)
INSERT T VALUES (2)
GO
ALTER TABLE T ADD A int NULL,
                --completare implicit cu NULL
                B int NOT NULL DEFAULT 10,
                --completare cu valoarea 10
                --forțează completarea
                C int NULL DEFAULT 11
                --cu valoarea 11
SELECT * FROM T
```

Rezultatul obținut este:

ID	A	B	C
1		NULL	10
2		NULL	11

(2 row(s) affected)

6. Instrucțiunea următoare definește o valoare **DEFAULT** pentru coloana **A** a tabelui **T** din exemplul precedent. Tuplurile existente în tabelă nu vor fi modificate.

```
ALTER TABLE T ADD DEFAULT 111 FOR A WITH VALUES
```

7. Instrucțiunea de mai jos modifică tipul de dată al coloanei **ID** din tabela **T** la numeric cu scala 10 și precizia 5:

```
ALTER TABLE T ALTER COLUMN ID numeric(10,5)
După această modificare conținutul tabelui T este următorul:
ID          A          B          C
-----  -----  -----  -----
1.00000    NULL      10        11
2.00000    NULL      10        11
NULL       NULL      10        11
(2 row(s) affected)
```

8. Prin instrucțiunea de mai jos adăugăm o constranță de tip **UNIQUE** pentru coloana **ID** a tabelui **T**:

```
ALTER TABLE T ADD UNIQUE (ID)
```

9. Instrucțiunea de mai jos adăugă la tabela **T** o constranță care intră în conflict cu valorile existente în coloana **B**, motiv pentru care se semnalizează o eroare și constranța nu este adăugată:

```
ALTER TABLE T ADD CHECK (B<>10)
```

Totuși, dacă instrucțiunea este executată cu opțiunea **WITH NOCHECK**, atunci constranța este acceptată și ea poate fi apoi activată, fără a avea efect retroactiv asupra tuplelor existente în tabela **T**, ci numai asupra modificărilor ulterioare de date:

```
--definire constranță fără verificare
ALTER TABLE T WITH NOCHECK ADD CHECK (b<>10)
--OK, activare constranță
```

UPDATE T SET B=B

-eroare la înlocuire valori cu ele însele

#### Observație:

La reactivarea constrângерilor opțiunea implicită este WITH NOCHECK. Dacă în exemplul de mai sus reactivarea constrângерilor tablei T s-ar face cu specificarea explicită a opțiunii WITH CHECK, printre-o instrucțиune de forma:

ALTER TABLE T WITH CHECK CHECK CONSTRAINT ALL

atunci sistemul ar genera o eroare din cauza conflictelor dintre datele tablei și constrângерile definite.

### 3.3. Instrucțиunea DROP TABLE

Sterge din baza de date definiția unei table împreună cu toate datele continute, constrângерile, indecsii și triggerurile asociate. Toate vederile sau procedurile stocate care fac referire la tabela care se sterge vor trebui să fie sterse în mod explicit prin instrucții DROP VIEW sau DROP PROCEDURE.

Sintaxa:

DROP TABLE *nume\_table*

unde:

*nume\_table* - este numele tablei care se sterge.

Observații:

1. O tabelă nu poate fi stearsă atâtă timp cât este referită printre-o constângere de tip FOREIGN KEY. Înainte de stergerea tablei referite trebuie stearsă constângerea FOREIGN KEY sau tabela de referință.
2. Instrucția DROP TABLE nu se poate aplica tablelor sistem ale SQL Server.

### 3.4. Utilizarea instrucțiunilor

#### CREATE TABLE și ALTER TABLE

În acest paragraf ne propunem să prezentăm un exemplu de definire a tabelelor pentru o bază de date pe care o numim a *Agenți*. Schema relațională a acestei baze de date va fi folosită ca model de referință în capituloare următoare oriunde în exemple va fi nevoie să ne raportăm la o anumită structură de date.

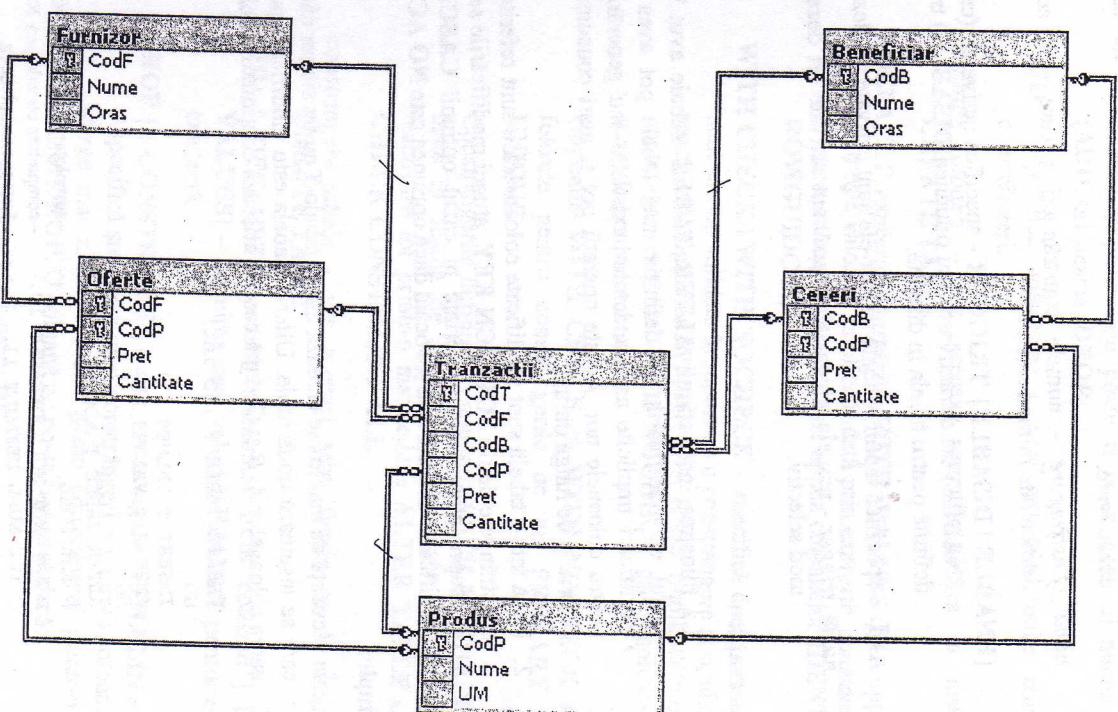


Fig. 3.1. Diagrama bazei de date *Agenți*

Modelul conceptual al bazei de date *Agentii* este definit astfel:

```
furnizor (CodF, Nume, Oras)
beneficiar (CodB, Nume, Oras)
produs (CodP, Nume, UM)
oferte (CodF, CodP, pret, Cantitate)
cereri (CodB, CodP, pret, Cantitate)
tranzactii (CodT, CodF, CodB, CodP, pret, Cantitate)
```

unde:

- atributele subliniate reprezintă cheile relațiilor;
- atributele *CodF* din *Oferte* și *Tranzactii*, *CodB* din *Cereri* și *Tranzactii*, respectiv *CodP* din *Oferte*, *Cereri* și *Tranzactii* sunt chei străine și referă fiecare tabelă în care apar ca și cheie primară;
- perechea *CodF-CodP* din *Tranzactii* este cheie străină și referă perechea corespunzătoare din relația *Oferte*;
- perechea *CodB-CodP* din *Tranzactii* este cheie străină și referă perechea corespunzătoare din relația *Cereri*.

Diagrama bazei de date *Agentii* este dată în figura 1.

O posibilă variantă pentru secvența de comenzi care creează tablelele bazei de date *Agentii* este următoarea:

```
CREATE TABLE dbo.Beneficiar (
    CodB int IDENTITY (1, 1) NOT NULL ,
    Nume varchar (50) NOT NULL ,
    Oras varchar (50) NULL,
    CONSTRAINT PK_Beneficiar PRIMARY KEY NONCLUSTERED
    (
        CodB
    )
)
GO

CREATE TABLE dbo.oferte (
    CodF int NOT NULL ,
    CodP int NOT NULL ,
    Pret money NULL ,
    Cantitate int NULL ,
    CONSTRAINT PK_Oferete PRIMARY KEY NONCLUSTERED
    (
        CodF,
        CodP
    )
)
GO

CREATE TABLE dbo.Produs (
    CodP int IDENTITY (1, 1) NOT NULL ,
    Nume varchar (50) NOT NULL ,
    UM varchar (50) NULL ,
    CONSTRAINT PK_Produs PRIMARY KEY NONCLUSTERED
    (
        CodP
    )
)
GO

CREATE TABLE dbo.Tranzactii (
    CodT int IDENTITY (1, 1) NOT NULL ,
    CodF int NULL ,
    CodB int NULL ,
    CodP int NULL ,
    Pret money NULL ,
    Cantitate int NULL ,
    CONSTRAINT PK_Tranzactii PRIMARY KEY NONCLUSTERED
    (
        CodT
    )
)
GO

ALTER TABLE dbo.Cerere ADD
CONSTRAINT FK_Cerere_Beneficiar FOREIGN KEY
(
    CodB
)
REFERENCES dbo.Beneficiar (
    CodB
)
CONSTRAINT FK_Cerere_Produs FOREIGN KEY
```

### 3.5. Indecși

```
(  
    CodP  
) REFERENCES dbo.Produs (  
    CodP  
)  
GO  
  
ALTER TABLE dbo.Oferte ADD  
(  
    CONSTRAINT FK_Oferete_Furnizor FOREIGN KEY  
        (  
            CodF  
        ) REFERENCES dbo.Furnizor (  
            CodF  
)  
,  
    CONSTRAINT FK_Oferete_Produs FOREIGN KEY  
        (  
            CodP  
        ) REFERENCES dbo.Produs (  
            CodP  
)  
)  
GO  
  
ALTER TABLE dbo.Tranzactii ADD  
(  
    CONSTRAINT FK_Tranzactii_Beneficiar FOREIGN KEY  
        (  
            CodB  
        ) REFERENCES dbo.Beneficiar (  
            CodB  
)  
,  
    CONSTRAINT FK_Tranzactii_Cerere FOREIGN KEY  
        (  
            CodB,  
            CodP  
        ) REFERENCES dbo.Cerere (  
            CodB,  
            CodP  
)  
,  
    CONSTRAINT FK_Tranzactii_Furnizor FOREIGN KEY  
        (  
            CodF  
        ) REFERENCES dbo.Furnizor (  
            CodF  
)  
,  
    CONSTRAINT FK_Tranzactii_Oferete FOREIGN KEY  
        (  
            CodF,  
            CodP  
        ) REFERENCES dbo.Oferte (  
            CodF,  
            CodP  
)  
)  
GO
```

În SQL Server se creează indecsi în special pentru a realiza un acces mai rapid la datele din tabele. Un index este o structură auxiliară care ordonează tuplele unui tabel după valorile a una sau mai multe coloane ale sale. Indexul conține referințe la tuplele tabelei și aceste referințe sunt ordonate după criteriile specificate la definirea indexului. SQL Server folosește indecsi pentru optimizarea interogărilor pe care le are de rezolvat atunci când construiește planul de execuție al fiecărei interogări. Acest proces este, în principiu, transparent utilizatorului care, deși nu poate decide direct modul de folosire al indecsilor, este cel care ia decizia referitoare la coloanele care se indeexează în momentul creației indexului. Această decizie este influențată de natura interogărilor care se anticipatează. De regulă, se vor indexa coloanele care figurează cel mai frecvent ca și criterii de căutare în interogări. Folosirea indecsilor se face, totuși, cu anumite costuri: consum de spațiu de memorie și încreșterea operatiilor de inserare, ștergere și actualizare. Numărul potrivit de indecsi pentru un tabel este, în mod clar, o decizie de proiectare influențată de raportul dintre frecvența interogărilor și cea a operatiilor de modificare date. În general este recomandabil să se evite excesul de indecsi, un număr de cel mult 5-7 indecsi per tabel fiind considerat un optim empiric pentru marea majoritate a aplicațiilor.

Dintre tipurile de indecsi care se pot defini în SQL Server, trei sunt cele care prezintă un interes special:

- **index unic** – are proprietatea că nu permite existența a două tuple cu aceeași valoare a indexului.
- **index de cheie primară** – este un tip special de index unic pentru care valorile coloanelor indexate nu acceptă valori NULL;
- **index de tip clustered** – se caracterizează prin faptul că determină reordonarea tuplelor tabelei la crearea indexului, astfel încât ordinea fizică să fie aceeași cu ordinea logică corespunzătoare indexului. Un astfel de index este mai rapid decât un index nonclustered. Pentru o tabelă se poate defini cel mult un singur index clustered la un moment dat.

Definirea unor constrângeri de tip PRIMARY KEY și UNIQUE se materializează, în mod automat, prin crearea de către SQL Server a indecsilor de cheie primară și unici corespunzători.

### 3.5.1. Instrucția CREATE INDEX

Permite definirea unui index pentru o tabelă sau o vedere.

Sintaxa(simplificată):

```
CREATE [ UNIQUE ] | CLUSTERED | NONCLUSTERED ]
INDEX name_index
    ON { tabela | vedere } ( coloana [ ASC | DESC ] [ ,n ] )
    [ WITH { IGNORE_DUP_KEY | DROP_EXISTING } [ ,n ] ]
```

unde:

**UNIQUE** – indică faptul că indexul va fi de tip unic. Înainte de crearea unui index unic, SQL Server verifică dacă datele existente satisfac condiția de unicitate a cheii de indexare și refuză crearea indexului dacă condiția nu este satisfăcută. Valorile NULL multiple se consideră duplicate la crearea unui index unic.

Dacă există un index **UNIQUE**, atunci operațiile **UPDATE** sau **INSERT** care ar genera valori dupicate ale cheii de indexare sunt rejectate integral și se generează un mesaj de eroare. Dacă este specificată opțiunea **IGNORE\_DUP\_KEY**, atunci la o operatie **INSERT** numai tuplurile care generează duplicate sunt rejectate, restul fiind inserate. În cazul unei operații **UPDATE** opțiunea **IGNORE\_DUP\_KEY** nu are nici un efect.

**CLUSTERED** – indică faptul că, pentru indexul creat, ordinea fizică coincide cu cea logică. Poate exista un singur index de acest fel pentru o tabelă sau vedere. Este recomandată creația indexului **CLUSTERED** înainte de creația celorlalți indexări, deoarece la crearea indexului **CLUSTERED** are loc reordonarea fizică a tuplelor tablei ceea ce implică recreearea tuturor indexelor deja definiți.

**NONCLUSTERED** – creează un index care specifică o ordine logică a tuplelor tablei sau vederii. **NONCLUSTERED** este opțiunea implicită la creația unui index. Se poate crea un index **NONCLUSTERED** pentru o vedere numai dacă există deja un index **CLUSTERED** definit pentru aceea vedere.

*name\_index* – este numele indexului; trebuie să fie unic la nivel de tabelă sau vedere.

*tabelă* – este numele tabelei pentru care se definește indexul.

*vedere* – este numele vederii pentru care se definește indexul. Facilitatea de indexare a vederilor apare pentru prima dată la versiunea SQL.

#### Server 2000.

**coloană** – se referă la coloana sau coloanele care se indexează. Dacă se specifică mai multe coloane, atunci se creează un index compozit a cărui cheie rezultă din combinarea coloanelor specificate.

**Coloanele** se vor specifica în ordinea priorității de sortare.

**[ASC | DESC]** – determină natura ordonării în index pentru valorile unei coloane date. Opțiunea implicită este **ASC**.

**IGNORE\_DUP\_KEY** – controlează funcționarea operațiilor de **INSERT** în cazul situațiilor de conflict cu un index **UNIQUE**. Vezi opțiunea **UNIQUE**.

**DROP\_EXISTING** - dacă există deja un index cu numele celui care se creează, atunci se va recrea indexul vechi conform definiției noi. Această clauză îmbunătățește performanța la recreearea indexului **CLUSTERED** pentru o tabelă sau vedere care are și alți indexări, fiind mai eficientă decât combinația **DROP INDEX** urmată de un nou **CREATE INDEX**.

### 3.5.2. Instrucția DROP INDEX

Șterge unul sau mai mulți indexări. Instrucția **DROP INDEX** nu se aplică asupra indexelor creați implicit prin definirea unor constrângeri de tip **PRIMARY KEY** sau **UNIQUE**.

Sintaxa:

```
DROP INDEX name_tabela.index | name_vedere.index [ ,n ]
```

unde:

*name\_tabela* | *name\_vedere* – specifică numele tabelei sau a vederii căreia iiii este asociat indexul.

*index* – este numele indexului care se șterge.

**Exemplu:**

- În secvența de mai jos se creează un index unic, de tip **clustered**, cu numele *IxName*, pește câmpul *Name* din tabela *Furnizor*:

```
IF EXISTS (SELECT name FROM sysindexes
            WHERE name = 'IxName')
    GO
    CREATE UNIQUE CLUSTERED INDEX IxName
        ON Furnizor (Name)
```

Dacă nu există nume duplicat în tabela *Furnizor*, atunci operația se realizează cu succes. De remarcat că pentru constrângerea PRIMARY KEY definită pe câmpul *CodF* s-a folosit opțiunea NONCLUSTERED, deci este posibil să definim indexul *IxName* cu opțiunea CLUSTERED.

După definirea acestui index se va semnaliza eroare la orice tentativă de inserare sau modificare date care ar produce nume duplicitat în tabela *Furnizor*. Astfel după a doua instrucție de mai jos:

```
INSERT Furnizor VALUES('IRIS', 'Cluj')
INSERT Furnizor VALUES('IRIS', 'Cluj')
```

sistemul va afișa următorul mesaj:

```
Cannot insert duplicate key row in object 'Furnizor' with unique index 'IxName'.
The statement has been terminated.
```

În acest exemplu vom crea un index unic pentru câmpul *Name* din tabela *Beneficiar*, dar cu opțiunea IGNORE\_DUP\_KEY:

```
IF EXISTS (SELECT name FROM sysindexes
            WHERE name = 'IxName')
    DROP INDEX Beneficiar.IxName
GO
```

```
CREATE UNIQUE CLUSTERED INDEX IxName ON Beneficiar(Name)
WITH IGNORE_DUP_KEY
```

De remarcat că indexul definit pentru tabela *Beneficiar* are același nume cu cel definit pentru tabela *Furnizor*. Numele de indecsi trebuie să fie unici doar la nivel de tabelă. Ca efect al opțiunii IGNORE\_DUP\_KEY la doua operație de inserare de mai jos nu se inserează nicio, ci doar se generează un mesaj de avertizare:

```
INSERT Beneficiar VALUES('IRIS', 'Cluj')
INSERT Beneficiar VALUES('IRIS', 'Cluj')
```

Mesajul de avertizare este:

```
Duplicate key was ignored.
```

3. Instrucțiunea de mai jos creează un index compozit pentru tabela *Tranzactii*:

```
CREATE INDEX IX_Coduri ON Tranzactii(CodF, CodB, CodP)
```

Indexul este creat peste coloanele *CodF*, *CodB* și *CodP* este implicit nonclustered și nu are proprietatea de unicitate.

### 3.6. Exerciții și probleme

- Să se indice care dintre exemplele de mai jos sunt corecte și care nu, iar dacă este cazul să se arate erorile din următoarele instrucții sau sevențe de instrucții:

- CREATE TABLE T(A int PRIMARY KEY, B int PRIMARY KEY)
- CREATE TABLE T(A int)
- ALTER TABLE T ADD PRIMARY KEY(A)
- CREATE TABLE T(CHECK (A<B), A int,B int)
- CREATE TABLE T (A int PRIMARY KEY FOREIGN KEY REFERENCES T(A))
- CREATE TABLE T(A int IDENTITY PRIMARY KEY)
- CREATE TABLE T(A int IDENTITY (0,0))
- CREATE TABLE T(A int IDENTITY, B int IDENTITY)

2. Fie tabela *T* definită prin instrucțiea:

```
CREATE TABLE T
    A int PRIMARY KEY FOREIGN KEY REFERENCES T(B),
    B int UNIQUE FOREIGN KEY REFERENCES T(A)
```

Este definiția tablei corectă? Dacă da explicăți ce se întâmplă la execuțarea operațiilor de inserare de mai jos:

```
INSERT T values (2,2)
INSERT T values (1,3)
```

De ce formă trebuie să fie tuplele acceptate de tabela *T*?

3. Fie tabela *T* definită astfel:

```
CREATE TABLE T(A tinyint IDENTITY (0,128),B int)
```

Să se execute sevența de instrucțuni de mai jos:

```
INSERT T values (1)
INSERT T values (2)
INSERT T values (3)
```

Explicați cele constatate!

3. Să se modifice cheile primare ale tabelelor din bază de date *Agenți* astfel încât să aibă la bază indecsi de tip clustered.

4. Să se definească indecsi de tip unic pentru câmpul *Nume* din tabelele *Furnizor*, *Beneficiar* și *Produs*.

5. Să se adauge constrângeri de tip CHECK pentru coloanele *Pret* și *Cantitate* din tabelele *Oferte*, *Cereri* și *Tranzacții*.

6. Să se adauge căte o coloană calculată tabelelor *Oferte*, *Cereri* și *Tranzacții* care să reprezinte valoarea pentru fiecare ofertă, cerere sau tranzacție.

7. Rescrieți exemplul 2) din paragraful 3.1.2.1. folosind instrucțiunea ALTER TABLE în locul combinației DROP-CREATE TABLE.

Instrucțiunile de control din limbajul Transact-SQL sunt cele din tabela de mai jos:

Instrucțiune	Descriere
BEGIN...END	Defineste un bloc de instrucțiuni.
BREAK	Determină ieșirea forțată din bucla WHILE curentă.
CONTINUE	Repuneste execuția unei bucle WHILE.
GOTO etichetă	Continuă prelucrarea cu instrucțiunea ce urmează după eticheta specificată prin GOTO.
IF...ELSE	Execuie condiționată, și optional, alternativă când condiția de testă a valorii FALSE.
RETURN	Ieșire necondiționată din procedura curentă.
WAITFOR	Stabileste o întârziete pentru execuția unei instrucțiuni.
WHILE	Repetă o secvență de instrucțiuni atâta timp cât condiția specificată este adevarată.

Următoarele instrucțiuni Transact-SQL sunt folosite frecvent în strânsă legătură cu instrucțiunile de control enumerate mai sus:

```
/*...*/ (Comentariu) EXECUTE
-- (Comentariu)
PRINT
CASE
DECLARE @local_variable
RAISERROR
```

## 4.1. Comentarii

Limbajul Transact-SQL acceptă două tipuri de comentarii:

--text\_comentariu  
/\*text\_comentariu\_multi-linie\*/

care indică un comentariu la nivel de linie; textul din linia curentă afiat după simbolurile "--" este ignorat de SQL Server;

care indică un comentariu multi-linie; textul afiat între perechile de simboluri /\* \*/ și /\* \*/ este ignorat de SQL Server.

## 4. INSTRUCȚIUNI DE CONTROL FİŞIERE DE COMENZI (BATCH-URI)

Observație:

SQL Server nu impune nici o restricție referitor la lungimea comentariilor, indiferent de tipul lor.

## 4.2. Instrucțiunile BEGIN și END

Instrucțiunile BEGIN și END sunt folosite pentru a grupa mai multe instrucțiuni într-un singur bloc logic. Perechea BEGIN...END delimită o secvență de instrucțiuni care formează un bloc. Blocurile BEGIN...END pot fi imbricate.

Sintaxa:

```
BEGIN  
  {instrucțiune_sql | bloc_instrucțiuni_sql}  
END
```

unde:

{instrucțiune\_sql | bloc\_instrucțiuni\_sql} - este orice instrucțiune sau grup de instrucțiuni definite ca un bloc.

Pentru a defini un bloc de instrucțiuni (*batch*), se utilizează cuvintele cheie BEGIN și END. În principiu, orice instrucțiune SQL Server poate să apară într-un bloc BEGIN...END, restricțiile care apar sunt cele care se referă și la batch-uri, întrucât orice bloc este parte a unui batch, iar anumite instrucțiuni nu pot fi grupate în același batch.

Instrucțiunile BEGIN și END se folosesc, întotdeauna, ca o pereche în cadrul aceluiasi batch și încadrează o secvență de instrucțiuni care formează un bloc în cadrul batch-ului. Perechea BEGIN...END se folosește ori de către ori este necesară gruparea logică a mai multe instrucțiuni. Câteva dintre situațiile tipice sunt:

- buclă WHILE care cuprindă mai mult de o instrucțiune;
- un element al unei funcții CASE care trebuie să includă un bloc de instrucțiuni;
- clauză IF sau ELSE care cuprindă mai mult de o instrucțiune.

Exemplu:

Dacă o instrucțiune IF controlează execuția doar a unei singure instrucțiuni, atunci nu este necesara folosirea perechii BEGIN...END:

```
IF (@@ERROR > 0)  
  SET @ErrorSaveVariable = @@ERROR
```

În secvență de mai sus se sare peste instrucțiunea SET dacă @@ERROR este 0. Dacă este nevoie să se sără peste mai multe instrucțiuni, atunci se folosesc BEGIN..END ca în secvență de mai jos:

```
IF (@@ERROR > 0)  
BEGIN  
  PRINT 'Error encountered, '+  
        CAST(@ErrorSaveVariable AS varchar(10))  
END
```

## 4.3. Instrucțiunea IF...ELSE

Impune condiții în executarea instrucțiunilor în funcție de rezultatul evaluării unei expresii booleene. Dacă condiția este satisfăcută se execută instrucțiunea sau blocul de instrucțiuni ce urmează cuvântului cheie IF și condiției. Cuvântul cheie optional ELSE introduce un bloc de instrucțiuni care se execută atunci când condiția IF nu este satisfăcută.

Sintaxa:

```
IF expresie_booleana  
  {instrucțiune_sql | bloc_instrucțiuni_sql}  
[ELSE]  
  {instrucțiune_sql | bloc_instrucțiuni_sql}]
```

unde:

expresie\_booleana - este o expresie ce returnează TRUE sau FALSE.

Dacă expresia booleană conține o instrucțiune SELECT aceasta trebuie să fie inclusă între paranteze.

{instrucțiune\_sql | bloc\_instrucțiuni\_sql} - este orice instrucțiune sau grup de instrucțiuni (bloc).

În lipsa unui bloc de instrucțiuni, condiția IF sau ELSE poate afecta execuția unei singure instrucțiuni. Pentru a defini un bloc de instrucțiuni se utilizează BEGIN...END.

Observații:

1. Construcția IF...ELSE poate fi folosită în batch-uri, în proceduri stocate și în interogări ad-hoc.
2. În anumite situații, expresie\_booleana se poate evalua la valoarea NULL, caz în care aceasta se interpretează ca și FALSE.

3. Testele IF pot fi imbricate, fie după alt IF sau după un ELSE. Nu există limite la numărul de niveluri imbricate. În cazul imbricărilor, se aplică regula implicită prin care fiecare ELSE se împerechează cu cel mai apropiat IF care nu are pereche ELSE.

**Exemplu:**

1. În secvența de mai jos se testează dacă există cel puțin un furnizor în tabela *Furnizori*, ca în caz se returnează numele tuturor furnizorilor, altfel se afișează un mesaj indicând absența furnizorilor:

```

IF EXISTS (SELECT * FROM Furnizori)
BEGIN
    PRINT 'Lista furnizorilor este:'
    SELECT Nume FROM Furnizori
ELSE
    PRINT 'Nu există furnizori!'

2. Orice procedură stocată care salvează codul de eroare returnat de către variabila globală @@ERROR pe timpul execuției unei tranzacții, trebuie să aibă la sfârșitul procedurii un IF similar cu următorul:

IF (@ErrorSaveVariable <> 0)
BEGIN
    PRINT 'Errors encountered, rolling back.'
    PRINT 'Last error encountered: ' + CAST(@ErrorSaveVariable AS VARCHAR(10))
    ROLLBACK
END
ELSE
BEGIN
    PRINT 'No Errors encountered, committing.'
    COMMIT
END
RETURN @ErrorSaveVariable

```

## 4.4. Instrucțiunea WHILE

Setează o condiție pentru execuția repetată a unei instrucțiuni sau bloc de instrucțiuni SQL. Instrucțiunile sunt executate repetat atât timp cât condiția specificată este adeverată. Execuția instrucțiunilor din buclă WHILE poate fi controlată din interiorul buclei folosind cuvintele cheie BREAK și CONTINUE.

**Sintaxa:**

```

WHILE expresie_booleană
{ instrucțiune_sql | bloc_instrucțiuni_sql }
[BREAK]
[CONTINUE]

```

unde:

*expresie\_booleană* - este o expresie booleană care returnează TRUE sau FALSE. Dacă expresia booleană conține o instrucțiune SELECT aceasta trebuie inclusă între paranteze.  
*{instrucțiune\_sql | bloc\_instrucțiuni\_sql}* - este orice instrucțiune Transact-SQL sau grup de instrucțiuni definite ca bloc cu ajutorul cuvintelor cheie BEGIN și END.

**BREAK** - este o instrucțiune care poate fi utilizată ca opțiune a instrucțiunii WHILE și are ca efect ieșirea din cea mai interioară buclă WHILE. Execuția se continuă cu instrucțiunea ce apare după cuvântul cheie END, ce marchează sfârșitul buclei curente. Instrucțiunea BREAK este, de cele mai multe ori, dar nu în mod necesar, activată de un test IF.

**CONTINUE** - este o instrucțiune care poate fi utilizată ca opțiune a instrucțiunii WHILE și are ca efect trecerea la iterarea următoare a buclei WHILE. Toate instrucțiunile de după cuvântul cheie CONTINUE până la cuvântul cheie END, ce marchează sfârșitul buclei curente, sunt ignorate, iar controlul este transferat la începutul buclei WHILE. Instrucțiunea CONTINUE este de cele mai multe ori, dar nu în mod necesar, activată de un test IF.

**Observație:**

În cazul a două sau mai multe bucle WHILE imbricate, o instrucțiune BREAK din bucla cea mai interioară determină ieșirea în următoarea buclă exterioară. Se executa toate instrucțiunile din buclă imediat exteroioră afișate după cuvântul cheie END, ce marchează sfârșitul buclei curente, după care se trece la începutul buclei exterioare.

**Exemplu:**

- În secvența de mai jos furnizorul cu codul *F1* își mărește prețurile la ofertele sale până când media prețurilor proprii ajunge la nivelul mediei prețurilor practicate de furnizorul cu codul *F2*. Procesul se întrerupe dacă prețul maxim al lui *F1* îl depășește pe cel al lui *F2* (al doilea IF cu

**BREAK**), dar numai dacă media prețurilor lui *F1* este cel puțin jumătate din media prețurilor lui *F2* (primul **IF** cu **CONTINUE**), în caz contrar continuându-se cu mărirea prețurilor.

```
WHILE (SELECT AVG(pret) FROM Oferte WHERE Codf='F1') <
      (SELECT AVG(pret) FROM Oferte WHERE Codf='F2')
BEGIN
    UPDATE Oferte
        SET pret = pret * 1.1
    WHERE Codf='F1'          -- mărește toate prețurile cu 10%
    IF (SELECT AVG(pret) FROM Oferte WHERE Codf='F1') <
       (SELECT AVG(pret) FROM Oferte WHERE Codf='F2') / 2
    CONTINUE
    IF (SELECT MAX(pret) FROM Oferte WHERE Codf='F1') >
       (SELECT MAX(pret) FROM Oferte WHERE Codf='F2')
    BREAK
END
```

**2.** Cazul tipic de utilizare al instrucțiunii **WHILE** este în contextul cursorelor. În acest caz instrucția **WHILE** testează valoarea variabilei globale @@*FETCH\_STATUS* care este setată de instrucția **FETCH**. Variabila @@*FETCH\_STATUS* are valoarea 0, dacă ultima execuție a instrucției **FETCH** s-a terminat prin încarcarea cu succes a unei noi linii în cursor. Secvența de mai jos definește un cursor peste tabela *Furnizor*, iar bucla **WHILE** conține o anumită prelucrare pentru fiecare furnizor încărcat în cursor.

```
DECLARE cursor_furnizor CURSOR FOR
    SELECT * FROM Furnizor
    OPEN cursor_furnizor
    FETCH NEXT FROM cursor_furnizor
    WHILE @@FETCH_STATUS = 0
        -- prelucrare furnizor current
    END
    CLOSE cursor_furnizor
    DEALLOCATE cursor_furnizor
```

## 4.5. Instrucția GOTO

Instrucția **GOTO** modifică ordinea de execuție a instrucțiunilor, provocând un salt la o etichetă. Instrucțiunile aflate după o instrucție **GOTO** nu sunt executate, în schimb execuția continua de la eticheta specificată în instrucția **GOTO**. Instrucția **GOTO** și etichetele asociate pot fi folosite

oriunde într-o procedură, batch sau bloc de instrucții. Instrucțiunile **GOTO** pot fi imbricate.

**Sintaxa:**

```
Definește eticheta:
    eticheta:
        GOTO eticheta
    unde:
```

**eticheta** – este punctul în care se transferă execuția dacă eticheta apare într-o instrucție **GOTO**.

Etichetele respectă regulile obișnuite pentru identificatori. Etichetele pot fi folosite ca și comentarii fie că sunt sau nu invocate într-o instrucție **GOTO**.

**Observație:**

Instrucția **GOTO** nu poate să specifice o etichetă din afara batch-ului curent, dar poate să conțină o etichetă definită fie înainte, fie după punctul în care se află instrucția **GOTO**.

Utilizarea instrucției **GOTO** trebuie făcută cu precauție. Utilizarea excesivă a instrucției **GOTO** dăunează claritatei unui batch. Secvențele de cod ce utilizează **GOTO** pot fi aproape întotdeauna înlăciute folosind alte instrucții de control. Cazul tipic de utilizare a instrucției **GOTO** este pentru a întrerupe iterările imbricate a mai multe bucle **WHILE**.

**Exemplu:**

Secvența de mai jos este echivalentă cu secvența din exemplul 2) de la instrucția **WHILE**, dar nu folosește această instrucție:

```
DECLARE cursor_furnizor CURSOR FOR
    SELECT * FROM Furnizor
    OPEN cursor_furnizor
    incarca_cursor:
        FETCH NEXT FROM cursor_furnizor
        IF (@@FETCH_STATUS = 0)
            -- prelucrare furnizor current
        END
        CLOSE cursor_furnizor
        DEALLOCATE cursor_furnizor
```

## 4.6. Instrucția WAITFOR

Specifică o anumită oră din zi ori interval de timp față de momentul curent după care se declanșează execuția unui bloc de instrucțuni, a unei proceduri stocate sau tranzacții.

Sintaxa:

```
WAITFOR {DELAY 'interval_time' | TIME 'moment_execuție'}  
unde:
```

**DELAY** - comunică serverului SQL să aștepte scurgerea unei durate de **tmp** (max 24 ore) înainte de execuția instrucției următoare.  
**'interval\_time'** - este durata timpului de așteptare care trebuie specificat într-unul din formatele acceptate de datele de tip **datetime** sau printre variabila locală de tip **datetime**. Instrucția **WAITFOR** nu permite specificarea datei calendaristice, portjuna referitoare la data dintr-o valoare de tip **datetime** este ignorată.

**TIME** - comunică serverului SQL să aștepte până în momentul specificat de parametrul **'moment\_execuție'**.

**'moment\_execuție'** - este ora din zi la care se lansează în execuție instrucționa ce urmează după **WAITFOR**; se specifică în același format ca și parametrul **'interval\_time'**.

Observație:

Un dezavantaj al instrucției **WAITFOR** este acela că după execuția unei instrucții **WAITFOR**, conexiunea la serverul SQL este inutilizabilă până la momentul specificat în instrucție.

Exemplu:

- În secvența de mai jos se lansează în execuție procedura stocată **update\_all\_stats** la ora fixă și anume 22:20:

```
BEGIN  
    WAITFOR TIME '22:20'  
    EXECUTE update_all_stats  
END
```

- Acet exemplu arată cum poate fi utilizată o variabilă de tip **datetime** în contextul opțiunii **DELAY**. Procedura stocată definită mai jos este proiectată pentru a aștepta o durată de timp variabilă (indicată prin

parametrul **@@DELAYLENGTH**), iar apoi să returneze utilizatorului

numărul de ore minute și secunde de așteptare:

```
CREATE PROCEDURE time_delay(@@DELAYLENGTH char(9)) AS  
DECLARE @RETURNINFO varchar(255)  
BEGIN
```

```
    WAITFOR DELAY @@DELAYLENGTH
```

```
    SELECT @@RETURNINFO = 'A total time of ' +  
        SUBSTRING(@@DELAYLENGTH, 1, 3) + ' hours, ' +  
        SUBSTRING(@@DELAYLENGTH, 5, 2) + ' minutes, and ' +  
        SUBSTRING(@@DELAYLENGTH, 8, 2) + ' seconds, ' +  
        'has elapsed! Your time is up.'
```

```
    PRINT @@RETURNINFO  
END
```

```
GO
```

Apelul de mai jos lansează procedura **time\_delay** specificând 10 secunde de așteptare:

```
EXEC time_delay '000:00:10'
```

iar mesajul rezultat este:

"A total time of 000 hours, 00 minutes, and 10 seconds, has elapsed! Your time is up."

## 4.7. Instrucția RETURN

Determină ieșirea necondiționată dintr-o interogare sau procedură. Execuția instrucției **RETURN** este imediată și completă, iar instrucția poate fi folosită în orice punct al unei proceduri, a unui batch sau a unui bloc de instrucții. Instrucțiunile care urmează după **RETURN** nu mai sunt executate.

Sintaxa:

```
RETURN [expresie_intreg]
```

unde:

**expresie\_intreg** - este o valoare de tip **intreg** returnată la ieșirea din procedură.

Procedurile stocate au posibilitatea de a returna o valoare întreagă către procedura sau aplicația apelantă.

Observație:

- În lipsa oricărei specificări explicate toate procedurile stocate ale sistemului **SQL** returnează implicit valoarea 0 ca indicație a executării cu succes a procedurii; returnarea unei valori diferite de zero indică apariția unei erori.

2. Dacă este utilizată într-o procedură stocată, instrucțiunea **RETURN** nu poate returna o valoare NULL. Dacă o procedură încearcă să întoarcă o valoare NULL se generează un mesaj de avvertiment.

Valoarea returnată de o procedură stocată poate fi atribuită unei variabile locale procedurii sau aplicației apelante printr-un apel de forma:

**EXECUTE @valoare\_retur = procedura\_apelată**

#### Exemple:

##### 1.

În procedura de mai jos se testează dacă s-a specificat o valoare pentru parametrul @nume\_furnizor și se trece la determinarea listei ofertelor furnizorului numai dacă valoarea parametrului nu este NULL, în caz contrar se afișează un mesaj și se ieșă din procedură:

```
CREATE PROCEDURE lista_oferte(
    @nume_furnizor VARCHAR(30) = NULL) AS
BEGIN
    PRINT 'Specificați un nume de furnizor!'
    RETURN
END
```

```
SELECT produs.Nume, Oferte.Pret, Oferte.Cant
FROM Furnizor,Oferte,Produs
WHERE Furnizor.Codep=Oferte.Codep AND
      Oferte.Codep=Produs.Codep AND
      Furnizor.Nume=@nume_furnizor
```

##### 2.

Procedura de mai jos testează dacă furnizorul dat ca parametru are sau nu oferte și returnează valoarea 0 în caz afirmativ, respectiv 1 în caz contrar.

```
CREATE PROCEDURE are_oferte(
    @nume_furnizor VARCHAR(30) = NULL) AS
IF EXISTS (SELECT * FROM Furnizor,Oferte
    WHERE Furnizor.Codep=Oferte.Codep AND
          Furnizor.Nume=@nume_furnizor)
    RETURN 1
ELSE
    RETURN 0
```

Se poate observa că procedura are\_oferte și interpretează rezultatul returnat de acesta prin afișarea unor mesaje adecvate și furnizarea listei de oferte dacă este cazul:

```
DECLARE @valoare_retur int
DECLARE @furnizor VARCHAR(30)
SET @furnizor="Terapia"
EXEC @valoare_retur = are_oferte @furnizor
```

```
IF @valoare_retur=1
    PRINT 'Furnizorul '+@furnizor+' nu are oferte!'
ELSE
    BEGIN
        PRINT 'Furnizorul '+@furnizor+' are următoarele oferte:'
        EXEC lista_oferte @furnizor
    END
```

## 4.8. Fișiere de comenzi (batch-uri)

Un fișier de comenzi sau batch este o instrucțiune sau un grup de instrucțiuni Transact-SQL trimise la un moment dat dintr-o aplicație către sistemul SQL Server pentru a fi executate. SQL Server compilează instrucțiunile dintr-un batch într-o singură unitate executabilă numit plan de execuție. Instrucțiunile din planul de execuție sunt apoi executate ca un tot unitar. Un fișier de comenzi constituie o unitate de compilare independentă având un context propriu care nu este influențat și nu influențează contextul altor fișiere de comenzi. Variabilele declarate într-un fișier de comenzi sunt vizibile doar în cadrul acestuia.

Dacă apar erori de sintaxă, care împiedică compilarea corectă a batch-ului și construirea planului de execuție, atunci nici una dintre instrucțiunile din batch nu va fi executată. În cazul unei erori de execuție ("run-time error"), cum ar fi cele de tipul depășirilor aritmetică sau violarea unei constrângeri din baza de date, se pot întâmpla următoarele:

- se oprește execuția instrucțiunii curente și se renunță la execuțarea instrucțiunilor care o urmează în "batch"; acesta este cazul cu cele mai multe erori de execuție;
- se oprește numai instrucțiunea curentă, toate instrucțiunile rămase în "batch" sunt executate; acesta este cazul violărilor de constrângeri și a altor câteva erori de execuție.

În oricare dintre situațiile de mai sus instrucțiunile executate până la întâlnirea unei erori de execuție nu sunt afectate. Singura excepție este dacă batch-ul este inclus într-o tranzacție și eroarea apăruta cauzează abortarea tranzacției ("rollback"). În acest caz toate modificările făcute în baza de date înainte de apariția erorii de execuție sunt anulate.

Asupra fișierelor de comenzi se impun o serie de restricții:

- Un fișier de comenzi poate conține doar una singură dintre următoarele instrucțiuni: **CREATE DEFAULT**, **CREATE PROCEDURE**, **CREATE RULE**, **CREATE TRIGGER**, sau **CREATE VIEW**; nici una dintre aceste instrucțiuni nu poate fi combinată cu o alta în același batch.

- Dacă o tabelă este modificată într-un batch, atunci coloanele noi ale acesteia nu pot să fie referite în același batch.
- Dacă o instrucție **EXECUTE** este prima într-un batch, atunci cuvântul cheie **EXECUTE** poate fi omis. Cuvântul cheie **EXECUTE** este necesar doar dacă instrucția **EXECUTE** nu este prima în batch.

#### 4.8.1. Comanda GO

Indică sfârșitul unui batch de instrucții pentru utilizarea SQL Server.

Sintaxa:

**GO**

Observații:

1. GO nu este o instrucție, ci este o comandă recunoscută de către **SQL Server Query Analyzer** și de alte utilitare. Batch-ul curent de instrucții constă din toate instrucțiunile de la ultimul GO sau de la începutul sesiunii ad-hoc dacă acesta este primul GO.
2. O instrucție nu poate ocupa aceeași linie cu o comandă **GO**. Totuși, linia cu o comandă **GO** poate conține comentarii.
3. Domeniul variabilelor locale (definite de către utilizator) este limitat la batch-ul curent, nici una dintre acestea nu poate fi referată după comanda GO.

Aplicațiile pot trimite mai multe instrucții la SQL Server pentru a fi executate ca un batch. Instrucțiunile dintr-un batch sunt compilate într-un singur plan de execuție. Utilizatorii on-line pot lansa în execuție instrucțiuni ad-hoc prin utilizarea **SQL Server** sau pot construi scripturi de instrucții care să fie rulate prin utilizarea **SQL Server**. În toate aceste cazuri se poate folosi comanda **GO** pentru a semnala sfârșitul unui batch. Comanda **GO** este adresată utilizatorilor **SQL Server** ea nu va fi trimisă niciodată mai departe către server-ul **SQL**.

Exemplu:

1. În același batch pot fi reunite mai multe instrucții **CREATE TABLE**, în schimb, pentru orice instrucție **CREATE VIEW** sau **CREATE PROCEDURE** este nevoie de câte un batch separat:

```
CREATE TABLE T1(x int)
CREATE TABLE T2(x int)
GO
CREATE PROCEDURE ShowT1 AS
SELECT * FROM T1
GO
CREATE VIEW VT2 AS
SELECT * FROM T2
GO
```

#### 4.9. Exerciții și probleme

1. Să se restrice sevența de mai jos să aibă posibilitatea de a folosi instrucția IF:

```
IF NOT EXISTS(SELECT * FROM Furnizor) RETURN
IF NOT EXISTS(SELECT * FROM Produs) RETURN
IF NOT EXISTS(SELECT * FROM Oferte) RETURN
SELECT Furnizor.Nume, Produs.Nume, Pret, Cantitate
FROM Furnizor, Oferte, Produs
WHERE Furnizor.CodF=Oferte.CodF
AND Oferte.CodP=Produs.CodP
```

2. Să se scrie sevența de la exercițiul 1), astfel încât să se afișeze căte un mesaj (prin instrucția PRINT 'Tabela este viață!'), pentru oricare dintre cele trei tabele *Furnizor*, *Oferte* sau *Produs* care nu conțin nici o tuplă.

3. În secvența de mai jos completează fiecare mesaj 'Tabela viață!' cu numele tabeliei la care se referă mesajul:

```
IF EXISTS (SELECT * FROM Furnizor)
IF EXISTS (SELECT * FROM Produs)
IF EXISTS (SELECT * FROM Oferte)
SELECT Furnizor.Nume, Produs.Nume, Pret, Cantitate
FROM Furnizor, Oferte, Produs
WHERE Furnizor.CodF=Oferte.CodF
AND Oferte.CodP=Produs.CodP
```

```

    ELSE
        PRINT 'Tabela vida!'
    ELSE
        PRINT 'Tabela vîdă!'
    END

```

- 4.** Să se scrie următoarea secvență folosind instrucțiunea WHILE și evitând instrucțiunea GOTO:

```

Test_secunda:
IF(DATEPART(ss, GETDATE()) > 0)
BEGIN
    WAITFOR DELAY '00:00:01'
    GOTO Test_secunda
END
PRINT DATEPART(nn, GETDATE())

```

- 5.** Care dintre secvențele de mai jos sunt corecte din punctul de vedere al grupării instrucțiunilor în batch-uri și care nu? Corectați secvențele greșite formând mai multe batch-uri dacă este nevoie!

- CREATE TABLE T1(a int)
- CREATE TABLE T2(a int)
- SELECT \* FROM T1
- CREATE TABLE T1(a int)
  - CREATE TABLE T1(a int)
    - CREATE TABLE T2(a int)
- CREATE TABLE T1(a int)
  - CREATE TABLE T1(a int)
    - CREATE TABLE T2(a int)
- CREATE VIEW V1 AS
  - CREATE TABLE T1 AS
    - CREATE TABLE T2(a int)
- CREATE TABLE T1(a int)
  - CREATE TABLE T1 AS
    - CREATE TABLE T2(a int)
- CREATE VIEW V1 AS
  - CREATE TABLE T1 AS
    - CREATE TABLE T2(a int)
- SELECT \* FROM V1

Fraza SELECT este principalul instrument pentru regăsirea datelor dintr-o bază de date relațională. Permite selectarea a una sau mai multe tuple și/sau attribute din una sau mai multe tabele după criterii din cele mai diverse și complexe. Sintaxa completă a frazei SELECT este destul de complexă, dar principalele clauze se pot sintetiza astfel:

```

SELECT lista_select [INTO tabela_nouă]
FROM [lista_relatări]
[WHERE condiție_de_căutare]
[GROUP BY expresie_de_grupare
[HAVING condiție_de_ordonare]
[ORDER BY expresie_de_ordonare [ASC | DESC]]]

```

În SQL Server se poate folosi operatorul de reuniune (UNION) pentru a reuni rezultatele a mai multe fraze SELECT într-o singură relație rezultat. Operatorii pentru diferență (MINUS) și intersecția (INTERSECT) a două mulțimi de tuple nu sunt implementați în SQL Server.

În cele de mai jos vom prezenta cele mai importante elemente de sintaxă pentru fiecare dintre clauzele frazei SELECT. Din cauza complexității sintaxei nu ne propunem o prezentare exhaustivă, ci vom lăsa la o parte elementele mai puțin utilizate sau cele care ar putea suferi modificări de la o versiune la alta a SQL Server. Pentru sintaxa completă recomandăm utilizarea documentațiilor de sistem.

## 5.1. Clauza SELECT

Specifică attributele relației rezultat.

Sintaxa:

```

SELECT [ ALL | DISTINCT ] [ TOP n [ PERCENT ] [ WITH TIES ] ]
      { *
      | { nome_label | nome_vedere | alias_tabel } *
      | { nome_coloana | expresie | IDENTITYCOL | ROWGUIDCOL }
      | [AS] alias_coloana
      | alias_coloana = expresie
      } [ ,n ]

```

unde semnificația diferitelor opțiuni este următoarea:

# 5. SINTAXA FRAZEI SELECT ÎN SQL SERVER

**ALL** - tuplele după care pot să apară în relația rezultat. **ALL** este opțiunea implicită.

**DISTINCT** - se vor elibera tuplele după care din relația rezultat. La eliminarea după care valorile NULL se consideră ca fiind egale între ele.

**TOP *n* [PERCENT]** - vor fi returnate numai primele *n* tuple din rezultat. Dacă se specifică și opțiunea **PERCENT**, atunci numai primele *n* procente din tuplele rezultatului sunt returnate. În acest caz *n* trebuie să fie între 0 și 100.

Dacă interogarea conține clauza **ORDER BY**, atunci sunt returnate primele *n* tuple (sau *n* procente de tuple) din rezultatul ordonat după criteriile clauzei **ORDER BY**. În lipsa clauzei **ORDER BY**, ordinea tuplelor este arbitrară.

**WITH TIES** - dacă există mai multe tuple cu aceeași valoare a criteriilor din clauza **ORDER BY** și sunt printre ultimele candidate pentru a face parte din cele **TOP n (PERCENT)** tuple, atunci ele vor fi incluse în rezultatul returnat. **TOP ... WITH TIES** se poate folosi numai în interogările care conțin clauza **ORDER BY**.

\* - vor fi incluse în rezultat toate attributele din toate tabelele și vederile menționate în clauza **FROM**. Attributele vor apărea ordinea specificării tabelelor și vederilor în clauza **FROM** și după ordinea în care se află attributele în acele tabele și vederi.

*nume\_table | nume\_vedere | alias\_table.\** - se includ în rezultat numai attributele din tabela sau vedere specificată.

*nume\_coloana* - este numele unei coloane. Numele de coloană trebuie prefixate cu numele tabelei sau vederii din care provin ori de către ori există ambiguități.

*expresie* - poate fi un nume de coloană, o constantă, o funcție, orice combinare a acestora realizată prin operatori, ori poate fi o frază **SELECT** îmbricată.

**IDENTITYCOL** - returnează coloana cu proprietatea **IDENTITY**.

Dacă în clauza **FROM** sunt referite mai multe tabele care au un atribut de tip **IDENTITY**, atunci **IDENTITYCOL** se va prefixa cu numele tabelei corespunzătoare, sub forma

### Tabelă.IDENTITYCOL.

**ROWGUIDCOL** - returnează coloana cu valorile identificatorului global unic de tupla pentru tabelele care au proprietatea **ROWGUIDCOL**. Dacă în clauza **FROM** sunt referite mai multe tabele care au această proprietate, atunci **ROWGUIDCOL** se prefixează cu numele tabelei.

*alias\_coloana* - este un nume alternativ cu care se poate înlocui numele unui atribut. *alias\_coloana* poate fi folosit în clauza **ORDER BY**, dar nu și în clauzele **WHERE**, **GROUP BY** sau **HAVING**.

## 5.2. Clauza INTO

Creează o nouă tabelă unde se înseră relația rezultat.

Sintaxa:

[INTO *tabela\_nouă*]

unde:

*tabela\_nouă* - este numele tabelei care se creează.

## 5.3. Clauza FROM

Specifică sursele din care se selectează tuplele (tabele, vederi, fraze SELECT îmbricate). Clauza **FROM** este obligatorie ori de căte ori în clauza **SELECT** se face referire la cel puțin un nume de atribut.

Sintaxa:

```
[ FROM {<sursa_tuple>} [,..,n] ]  
•  
<sursa_tuple> ::=  
  nume_table [ [AS] alias_table ]  
  view_name [ [AS] alias_table ]  
  funcție_relativ [ [AS] alias_table ]  
  frază_SELECT [AS] alias_table [ (alias_coloana [,..,n]) ]  
  | <tabelă_cuplate>  
  
  •  
  <tabelă_cuplate> ::=  
    <sursa_tuple> <tip_cuplare> <sursa_tuple> ON <condiție_cauzare>  
    <sursa_tuple> CROSS JOIN <sursa_tuple>  
  | <tabelă_cuplate>
```

$<tip\_cuplare> ::=$   
 [ INNER | { ( LEFT | RIGHT | FULL ) [ OUTER ] } JOIN

unde:

- $<sursa\_tuple>$  - specifică tablele, vederi, fraze SELECT și tablele cuplate care constituie sursa de tuple pentru fraza SELECT.
- $name\_tabel$  [ [ AS ] alias\_tabel ] - specifică numele unei table și un alias optional.

- $view\_name$  [ [ AS ] alias\_tabel ] - specifică numele unei vederi și un alias optional.

- $funcție\_relație$  [ [ AS ] alias\_tabel ] - specifică numele unei funcții care returnează o referință la un set de tuple (relație) și un alias optional.

- $fraza\_SELECT$  [ [ AS ] alias\_tabel ] - este o frază SELECT imbricată.

- $alias\_coloana$  - este un alias optional pentru a redenumi coloanele relației returnate de o frază SELECT imbricată.

- $<tabele\_cuplate>$  - este o relație obținută prin cuplarea a două sau mai multe tabele.

- CROSS JOIN** - specifică produsul cartezian a două tabele.

- $<tip\_cuplare>$  - specifică tipul operației de cuplare. Acesta poate fi:

- INNER** - specifică faptul că toate perechile care se leagă prin condiția de cuplare și numai acestea fac parte din relația rezultat.

- LEFT [OUTER]** - specifică faptul că toate tuplele din tabela stânga care nu satisfac condiția de cuplare se includ în relația rezultat alături de cele care satisfac această condiție. Coloanele corespunzătoare tablelei din dreapta se setează la valoarea NULL în toate aceste tuple.

- RIGHT [OUTER]** - specifică faptul că toate tuplele din tabela dreapta care nu satisfac condiția de cuplare se includ în relația rezultat alături de cele care satisfac această condiție. Coloanele corespunzătoare tableei din stânga se setează la valoarea NULL în toate aceste tuple.

- FULL [OUTER]**] - specifică faptul că toate tuplele din ambele tabele care nu satisfac condiția de cuplare se includ în relația rezultat alături de cele care satisfac această condiție. Coloanele corespunzătoare celeilalte tabele se setează la valoarea NULL în toate aceste tuple.

- JOIN** - este operatorul generic de cuplare.

**ON**  $<condiție\_cautare>$  - specifică condiția de cuplare. În general, condiția de cuplare poate specifica orice predicat, cel mai adesea exprimat sub forma unor comparații între attribute. Compararea atributelor se poate face între attribute de același tip sau între attribute având tipuri compatibile, caz în care SQL Server face o conversie implicită. Dacă nu este posibilă conversia implicită, atunci se pot folosi funcțiile de conversie explicită (CAST sau CONVERT) pentru a aduce attributele comparate la același tip de date.

O condiție de căutare este o combinație de două sau mai multe predicate prin operatorii logici AND, OR și NOT. Sintaxa unei condiții de căutare este:

$<condiție\_cautare> ::=$

```

{ [ NOT ] <predicat> | ( <condiție_cautare> )
  [ ( AND | OR ) [ NOT ] { <predicat> | ( <condiție_cautare> ) } ]
  } [,..n]
  
```

unde predicat este o expresie care returnează una din valorile TRUE, FALSE ori UNKNOWN. Sintaxa simplificată a unui predicat este următoarea:

$<predicat> ::=$

```

expresie { = | < | > | >= | > | < | <= | <= | ! < } expresie
| expresie _str [ NOT ] LIKE expresie _str
| expresie [ NOT ] BETWEEN expresie AND expresie
| expresie IS [ NOT ] NULL
| expresie [ NOT ] IN ( subinterrogare | expresie [,..n] )
| expresie { = | < | > | >= | > | < | <= | <= | ! < } { ALL | SOME | ANY }
          ( subinterrogare )
| EXISTS ( subinterrogare )
  
```

expresie - este un nume de coloană, o constantă, o funcție, o expresie CASE, o variabilă, o subinterrogare care returnează un scalar sau o combinare a acestora.

expresie \_str - este un sir de caractere.

[NOT] LIKE - este operatorul de comparare inexactă pentru şiruri de caractere.

[NOT] BETWEEN - specifică un interval închis de valori (inclusiv valoile extreme).

IS [NOT] NULL - specifică o căutare de valori NULL (sau diferite de NULL, dacă se specifică cuvântul cheie NOT).

[NOT] IN - specifică o căutare a unei expresii (constantă sau nume de coloană) într-o listă de valori constante sau în relația rezultat returnată de o subinterrogare.

subinterrogare - este o frază SELECT restricționată care nu poate conține clauzele ORDER BY sau INTO.

ALL - poate fi folosit exclusiv ca prefix al unei subinterrogări în operații de comparare. Predicatul în care este inclus returnează TRUE dacă toate valorile din rezultatul returnat de subinterrogare satisfac condiția de comparare și FALSE în caz contrar (rezultatul este FALSE și în cazul în care subinterrogarea returnează ca rezultat o relație vidă - NULL).

{SOME | ANY} - pot fi folositi exclusiv ca prefix al unei subinterrogări în operații de comparare. Predicatul în care este inclus returnează TRUE dacă cel puțin una dintre valorile din rezultatul returnat de subinterrogare satisfac condiția de comparare și FALSE dacă nici o valoare nu satisfac condiția de comparare sau dacă subinterrogarea returnează ca rezultat o relație vidă. Altfel, predicatul va lua valoarea UNKNOWN.

EXISTS - poate fi folosit exclusiv ca prefix al unei subinterrogări și returnează TRUE dacă rezultatul returnat de subinterrogare conține cel puțin o tuplă și FALSE în caz contrar.

## 5.4. Clauza WHERE

Specifica o condiție de căutare prin care se restricționează setul tuplelor din relația rezultat.

Sintaxă:

[ WHERE < condiție\_căutare > ]

unde:

< condiție\_căutare > - are aceeași semnificație ca și în cazul clauzei FROM.

## 5.5. Clauza GROUP BY

Specifică grupurile care se formează din tuplele selectate prin clauzele anterioare. Elementele din clauza SELECT trebuie să producă o valoare unică la nivel de grup. Orice funcție de agregare care apare în fraza SELECT acționează la nivelul fiecărui grup în parte calculând aggregarea corespunzătoare pentru tuplele care fac parte din grup.

Sintaxă:

[ GROUP BY [ALL] expresie\_group\_by [, n] ]

unde:

ALL

- indică faptul că toate grupurile vor fi incluse în relația rezultat, inclusiv cele care nu conțin nici o tuplă care să satisfacă condițiile specificate în clauza WHERE.

Dacă este specificată opțiunea ALL, atunci funcțiile de agregare vor returna NULL pentru grupurile ale căror tuple nu satisfac condițiile din WHERE (funcția COUNT() va returna 0 pentru aceste grupuri!).

expresie\_group\_by - este o expresie după care se face gruparea.

Poate fi un nume de coloană sau o expresie care face referire la un nume de coloană fără funcții de agregare.

## 5.6. Clauza HAVING

Specifica o condiție de căutare la nivelul grupurilor. Clauza HAVING poate fi folosită numai în prezența clauzei GROUP BY.

Sintaxă:

[ HAVING < condiție\_căutare > ]

unde:

< condiție\_căutare > - specifică condiția de căutare pentru grupurile de tuple.

Condiția de căutare este similară ca sintaxă cu cele de la clauzele **FROM** și **WHERE** cu mențiunea că este exprimată doar în termenii acelor elemente care au o valoare unică la nivel de grup.

Dacă se folosește **HAVING** împreună cu **GROUP BY ALL**, atunci clauza **HAVING** are prioritate și anulează efectul opțiunii **ALL**. Tipurile de date **text**, **image** și **ntext** nu se pot folosi într-o clauză **HAVING**.

## 5.7. Operatorul UNION

Combină rezultatele a două sau mai multe interogări într-o singură relație rezultat care conține toate tuplele din interogările inițiale. Relațiile combinate prin **UNION** trebuie să satisfacă condiția de compatibilitate la reuniune, ceea ce înseamnă că:

- numărul și ordinea coloanelor trebuie să fie aceeași în toate relațiile;
- tipurile de date ale coloanelor corespondente trebuie să fie compatibile.

Sintaxa:

```
{<interrogare>}  
UNION [ALL]  
<interrogare>  
[UNION [ALL]  
<interrogare>  
[...n]
```

unde:

**<interrogare>**

- este o interogare care returnează o relație rezultat pentru a fi combinată cu o alta relație rezultat având aceeași structură. Coloanele corespondente din relațiile reunite trebuie să fie compatibile prin conversiile implicate ale SQL Server.

**UNION**

- specifică combinarea prin reuniune a două sau mai multe relații.

**ALL**

- încorporează toate tuplele în rezultat, inclusiv dupli-catele. Dacă nu se specifică opțiunea **ALL** dupli-catele sunt automat eliminate.

## 5.8. Clauza ORDER BY

Specifică operația de sortare a relației rezultat. Clauza **ORDER BY** nu poate să apăre în subinterogări sau în definițiile de vederi.

Sintaxa:

```
[ORDER BY {expresie_ordonare [ ASC | DESC ] } [,...n] ]
```

unde:

**expresie\_ordonare** - specifică o coloană pe baza căreia se face sortarea.

Poate fi un nume sau alias de coloană, o expresie sau un întreg pozitiv reprezentând poziția criteriului de sortare în lista clauzei **SELECT**.

Se pot specifica mai multe coloane în aceeași clauză **ORDER BY** pentru a forma o secvență ierarhică cu criteriu de sortare primar, secundar, terțiar și.c.m.d.

Clausa **ORDER BY** poate să includă și criterii de sortare care nu apar în lista clauzei **SELECT**; totuși, dacă se folosesc **SELECT DISTINCT** sau dacă fraza **SELECT** conține operatorul **UNION**, atunci toate criteriile de sortare trebuie să apară în lista **SELECT**. Mai mult, dacă fraza **SELECT** include operatorul **UNION**, criteriile de sortare trebuie să aibă aceeași nume cu cele din lista clauzei **SELECT** a primei interogări. Tipurile de date **ntext**, **text** și **image** nu se pot folosi ca și criterii de sortare.

**ASC** - specifică ordonare crescătoare.

**DESC** - specifică ordonare descreșcătoare.

Valoare **NULL** sunt considerate ca cele mai mici valori posibile și vor apărea primele într-o ordonare crescătoare.

Observații:

1. Ordinea clauzelor într-o frază **SELECT** este semnificativă. Oricare dintre clauzele optionale poate să fie omisă, dar atunci când apare, trebuie plasată în poziția sa corectă din cadrul frazei **SELECT**.
2. O frază **SELECT** care returnează o singură valoare poate să apară oriunde este legală folosirea unei expresii: în lista clauzei **SELECT**, în predicaticele din condițiile de căutare ce apar în clauzele **FROM**, **WHERE** sau **HAVING**, în clauza **ORDER BY** §.a.m.d.
3. O frază **SELECT** care conține clauzele **WHERE**, **GROUP BY** și **HAVING** este prelucrată în ordinea apariției acestor clauze: întâi se

elimină tuplele care nu satisfac condiția din WHERE, apoi se formează grupuri și, în final, se elimină grupurile care nu satisfac condiția din clauza HAVING.

4. În general, la selectarea unei coloane de tip IDENTITY într-o tabelă nouă aceasta își păstrează proprietatea IDENTITY cu excepția următoarelor situații:

- fraza SELECT conține o cuplare, o clauză GROUP BY sau o funcție de agregare;
- interogarea este formată din mai multe fraze SELECT reunite prin operatorul UNION;
- coloana cu proprietatea IDENTITY apare de mai multe ori în tabela nouă;
- coloana cu proprietatea IDENTITY este parte a unei expresii.

În oricare din situațiile de mai sus, noua coloană este creată cu proprietatea NOT NULL în loc să moștenească proprietatea IDENTITY.

#### Exemple:

1. În lipsă unei condiții WHERE, de legătură între tabele, o frază SELECT cu mai multe tabele în clauza FROM va returna produsul cartezian al acestor tabele. Astfel fraza:

```
SELECT *  
FROM Furnizor, Beneficiar
```

produce ca rezultat toate perechile furnizor-beneficiar. Dacă ne interesează numai perechile de nume, atunci rescriem fraza de mai sus astfel:

```
SELECT Furnizor.Nume, Beneficiar.Nume  
FROM Furnizor, Beneficiar
```

În SQL Server există două posibilități de a redenumi attributele din rezultatul interogării de mai sus:

Varianta 1 (conformă cu standardul SQL'92):

```
SELECT Furnizor.Nume AS "Nume furnizor",  
Beneficiar.Nume AS "Nume beneficiar"  
FROM Furnizor, Beneficiar
```

Varianta 2 (specifică SQL Server):

```
SELECT "Nume furnizor"=Furnizor.Nume,  
"Nume beneficiar"=Beneficiar.Nume  
FROM Furnizor, Beneficiar
```

#### 2.

Clauza FROM poate conține o frază SELECT imbricată în lista surselor de tuple. În SQL Server este obligatoriu ca fraza SELECT imbricată să primească un nume (alias) prin care poate fi, eventual, referit ulterior. Fiecare dintre următoarele fraze SELECT:

```
SELECT * FROM (SELECT * FROM Furnizor) F  
(SELECT * FROM Furnizor) F F  
SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM Furnizor)) FF) FFF
```

sunt echivalente cu:

```
SELECT * FROM Furnizor
```

#### 3. Interogarea:

```
SELECT Oras, COUNT(CodF)  
FROM Furnizor  
GROUP BY Oras
```

returnnează numele fiecărui oraș în care există furnizori împreună cu numărul de furnizori din orașul respectiv. Pe de altă parte, interogarea:

```
SELECT Oras, COUNT(CodF)  
FROM Furnizor  
WHERE Oras IN ('cluj', 'Bucuresti')  
GROUP BY Oras
```

returnnează datele de la prima interogare numai pentru orașele Cluj și București. Dacă se adaugă opțiunea ALL la clauza GROUP BY din ultima interogare se obține același rezultat ca la prima interogare. Astfel interogarea:

```
SELECT Oras, COUNT(CodF)  
FROM Furnizor  
WHERE Oras IN ('cluj', 'Bucuresti')  
GROUP BY ALL Oras
```

returnnează numele tuturor orașelor împreună cu numărul de furnizori din fiecare oraș.

De remarcat că Oras este atribut de grupare, ceea ce înseamnă că se poate muta condiția din clauza WHERE în clauza HAVING fără efect asupra rezultatului returnat:

```
SELECT Oras, COUNT(CodF)  
FROM Furnizor  
GROUP BY Oras  
HAVING Oras IN ('cluj', 'Bucuresti')
```

De reținut, totuși, că variantele în care condiția este exprimată în clauza WHERE sunt mai eficiente și este de dorit ca, ori de câte ori este posibil, să se treacă cât mai multe condiții din clauza HAVING în clauza WHERE, astfel încât să se obțină o interogare echivalentă logic cu cea initială, dar mai eficientă.

#### 4.

Pentru a obține lista tuturor furnizorilor împreună cu numărul de produse oferite de fiecare, listă care să includă și furnizorii fără oferte, putem folosi un operator de cuplare externă ca în interogarea de mai jos:

```
SELECT Nume, "Numar produse"=COUNT(CodP)
      FROM Furnizor LEFT JOIN Oferte
      ON Furnizor.CodF=Oferte.CodF
      GROUP BY Nume
```

Este interesant de remarcat faptul că același rezultat se obține prin interogarea:

```
SELECT Nume, "Numar produse"=COUNT(CodP)
      FROM Furnizor,Oferte
      WHERE Furnizor.CodF=Oferte.CodF
      GROUP BY ALL Nume
```

#### 5.

Interrogarea de mai jos:

```
SELECT * FROM Furnizor
UNION
SELECT * FROM Furnizor
```

este echivalentă cu:

```
SELECT * FROM Furnizor
```

în timp ce interogarea:

```
SELECT * FROM Furnizor
UNION ALL
SELECT * FROM Furnizor
```

returnează de două ori succesiv lista furnizorilor, iar varianta

```
SELECT * FROM Furnizor
UNION ALL
SELECT * FROM Furnizor
ORDER BY Nume
```

returnează o listă în care fiecare furnizor apare de două ori consecutiv în listă.

## 5.9. Funcții de agregare

Funcțiile de agregare efectuează un anumit calcul asupra unui set de valori și returnează ca rezultat o singură valoare. Cu excepția funcției COUNT, toate funcțiile de agregare ignoră valorile NULL. Funcțiile de agregare sunt adesea folosite împreună cu clauza GROUP BY pentru a calcula valori agregate la nivelul grupurilor de tuple. Folosirea funcțiilor de agregare este permisă numai în lista SELECT a unei fraze SELECT principala sau imbricată sau într-o clauză HAVING.

Principalele funcții de agregare disponibile în SQL Server sunt:

Sintaxă	Semnificație
AVG([ALL   DISTINCT] [expresie numerică])	media valorilor unui grup;
COUNT([ALL   DISTINCT] [expresie] [*])	numărul de elemente ale unui grup;
MAX([ALL   DISTINCT] [expresie])	maximul dintr-un grup de valori.
MIN([ALL   DISTINCT] [expresie])	minimul dintr-un grup de valori.
SUM([ALL   DISTINCT] [expresie numerică])	suma valorilor unui grup.
STDDEV([expresie numerică])	returnează deviația standard pentru valorile exprimării date ca argument.
STDDEVSP([expresie numerică])	returnează deviația standard pentru populația tuturor valorilor expresiei date ca argument.
VAR([expresie numerică])	returnează varianța statistică pentru valoările expresiei date ca argument.
VARP([expresie numerică])	returnează varianța statistică pentru populația tuturor valorilor expresiei date ca argument.

unde:

**ALL** - indică faptul că agregarea se aplică tuturor valorilor. ALL este opțiunea implicită.

**DISTINCT** - specifică faptul că agregarea se aplică doar asupra unei singure instanțe a fiecărei valori din grup, după care fiind ignorate.

**expresie** - este o expresie de orice tip, mai puțin **uniqueidentifier**, **text**, **image** sau **ntext**. Expressia nu poate conține funcții de agregare sau fraze SELECT imbricate.

**expresie\_numerică** - este o expresie de tip numeric, exceptând tipul bit. Expressia nu poate conține funcții de agregare sau fraze SELECT imbricate.

\* - specifică faptul că toate tuplele vor fi numărate. Varianta COUNT(\*) nu poate fi folosită cu opțiunea DISTINCT și returnează numărul de tuple dintr-o relație rezultat numărând toate dupicatele și tuplele cu valori NULL.

#### Observații:

1. Distingem trei variante de folosire a funcției de agregare COUNT:

- COUNT(\*) - include toate dupicatele și valorile NULL;
- COUNT(ALL expresie) - include toate dupicatele, dar fără valorile NULL;
- COUNT(DISTINCT expresie) - fără dupicate și fără valori NULL.

2. Opțiunea DISTINCT nu are nici un efect în cazul funcțiilor MAX și MIN, fiind păstrată doar pentru compatibilitatea cu standardul SQL'92.

#### Exemple:

1. Numărul furnizorilor înregistrati în tabela Furnizor?

```
SELECT COUNT(*)  
FROM Furnizor
```

2. Numărul orașelor în care sunt furnizori?

```
SELECT COUNT(DISTINCT Oras)  
FROM Furnizor
```

3. Numărul și media prețurilor ofertelor de calculatoare?

```
SELECT COUNT(*), AVG(Pret)  
FROM Oferte_Produs  
WHERE Oferte_Produs.CodP=Produs.CodP  
AND Produs.Nume='Calculatoare'
```

4. Cel mai mic și cel mai mare preț pentru ofertele de calculatoare?

```
SELECT MIN(Pret), Max(Pret)  
FROM Oferte_Produs  
WHERE Oferte_Produs.CodP=Produs.CodP  
AND Produs.Nume='Calculatoare'
```

## 5.10. Exerciții și probleme

1. Arătați ce este greșit în cazul următoarelor fraze SQL:

- a. SELECT Oras, COUNT(CodF)  
FROM Furnizor  
GROUP BY Oras  
HAVING Nume="Furnizor1"
- b. SELECT Furnizor, \*, COUNT(CodF)  
FROM Furnizor  
GROUP BY Oras  
HAVING Nume="Furnizor1"
- c. SELECT Oras, COUNT(CodF)  
FROM Furnizor
- d. SELECT ALL DISTINCT Nume  
FROM Furnizor
- e. SELECT Oras  
FROM Furnizor  
ORDER BY Oras
- f. SELECT Oras  
FROM Furnizor  
ORDER BY Oras
- g. SELECT COUNT(DISTINCT \*)  
FROM Furnizor
- h. (SELECT Oras  
FROM Furnizor)  
UNION  
(SELECT Oras  
FROM Beneficiar)
- i. (SELECT DISTINCT Oras  
FROM Furnizor)  
UNION ALL  
(SELECT Oras  
FROM Beneficiar)
- j. (SELECT DISTINCT Oras  
FROM Furnizor)  
UNION

```
(SELECT DISTINCT Oras
FROM Beneficiar)          (SELECT DISTINCT Oras
                           FROM Beneficiar)
```

### 3. Sunt următoarele fraze SQL echivalente? Explicați de ce!

a.

```
SELECT Nume
FROM Furnizor
WHERE CodF IN
      (SELECT CodF
       FROM Oferte)
```

b.

```
SELECT Nume
FROM Furnizor, Oferte
WHERE Furnizor.CodF
      = Oferte.CodF
```

4.

**Explicați deosebirea dintre rezultatele returnate de următoarele două interogări:**

a.

```
SELECT Nume, COUNT(CodP)
FROM Furnizor LEFT JOIN Oferte
ON Furnizor.CodF=Oferte.CodF
GROUP BY Nume
```

b.

```
SELECT Nume, COUNT(*)
FROM Furnizor LEFT JOIN Oferte
ON Furnizor.CodF=Oferte.CodF
GROUP BY Nume
```

## 6. FORMULARAREA INTEROGĂRIILOR ÎN LIMBAJUL SQL (PARTEA I)

În cele ce urmăză prezentăm modalități de rezolvare în limbajul SQL pentru un set cât mai larg și acoperitor de interogări. Pentru unele interogări sunt prezентate mai multe variante de rezolvare. Soluțiile sunt, în general, compatibile SQL Server, iar pentru rezolvările care nu satisfac această condiție se specifică în mod explicit acest lucru, eventual cauzele incompatibilității care se semnalizează. Sunt puse în evidență și unele limitări ale dialectului TSQL (cum ar fi lipsa operatorilor MINUS și INTERSECT), precum și modalitățile de folosire a acestora. Nu în ultimul rând, sunt prezентate și unele tehnici de construire a unor fraze SQL cât mai eficiente, cum ar fi modalitățile de înlocuire a operatorului IN. Aceasta permite programatorului să aleagă varianta mai eficientă de rezolvare a unei interogări în situațiile în care există mai multe soluții echivalente din punct de vedere logic. Toate interogările fac referire la baza de date *Agenți* prezentată în capitolul 3.

### 6.1. Interogări simple

#### 1. "Numele furnizorilor din Cluj?"

```
SELECT Nume
FROM Furnizor
WHERE Oras='Cluj'
```

#### 2. "Lista numelor furnizorilor din orașele Cluj și București?"

```
SELECT Nume
FROM Furnizor
WHERE Oras='Cluj' OR Oras='Bucuresti'
```

Se returnează toate numele de furnizori din tuplele pentru care proprietatea *Oras* are valoarea Cluj sau București.

#### 3. "Lista oraselor în care există furnizori?"

```
SELECT DISTINCT Oras
FROM Furnizor
```

Întrucât, de regulă, pot exista mai mulți furnizori în același oraș, pot exista mai multe tuple în relația *Furnizor* pentru un oraș anume. Din

acest motiv pentru a obține o listă a orașelor în care fiecare oră apare cel mult odată este nevoie de operatorul DISTINCT care elimină orașele după ce sunt adăugate.

4.

"Lista, în ordine alfabetica, a orașelor în care există furnizori?"

```
SELECT DISTINCT Oras
FROM Furnizor
ORDER BY Oras
```

Se ordonează rezultatul după criteriul Oras, ordonarea fiind implicită în ordine crescătoare. Ca și criteriu de ordonare poate să apară orice câmp din relația/relațiile referite în interogare sau expresii construite din acestea. Astfel interogarea:

"Lista furnizorilor ordonată după prima literă a numelor?"

se rezolvă în felul următor:

```
SELECT Nume
FROM Furnizor
ORDER BY Left(Nume, 1)
```

Dacă apare opțiunea DISTINCT în clauza SELECT, atunci orice criteriu de ordonare care apare în clauza ORDER BY trebuie, de asemenea, să fie prezent și în lista clauzei SELECT. Astfel dacă încercăm să obținem lista orașelor în care există furnizori, ordonată după prima literă a numelor de orașe, printre-o interogare de forma:

```
SELECT DISTINCT Oras
FROM Furnizor
ORDER BY Left(Oras, 1)
```

atunci sistemul va afișa un mesaj de eroare de forma:

```
Server: Msg 145, Level 15, State 1, Line 1
        ORDER BY items must appear in the select list if
        SELECT DISTINCT is specified.
```

Varianta următoare returnează lista numerelor de orașe împreună cu inițiala fiecărui oraș:

```
SELECT DISTINCT Oras, Left(Oras, 1)
FROM Furnizor
ORDER BY Left(Oras, 1)
```

"Numerele și orașul beneficiarilor care nu sunt din București?"

```
SELECT Nume, Oras
FROM Beneficiar
WHERE Oras <> 'Bucuresti'
```

## 6.2. Cuplarea a mai multe relații

6. "Numerele furnizorilor care oferă cel puțin un produs?"

```
SELECT DISTINCT Nume
FROM Furnizor, Oferte
WHERE Furnizor.CodF=oferte.CodF
```

Observație:

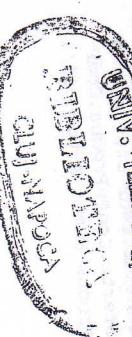
În cazul interogărilor care fac referire la mai multe tabele devine actuală problema prefixării numelor de attribute cu numele tabelelor din care provin. Compilatorul SQL poate determina în mod automat numele tabelei corespunzătoare unui atribut ori de căte ori nu există ambiguitate în acest sens. Programatorul poate să omită precizarea numelui de tabel în toate aceste cazuri, dar poate să opteze și pentru specificarea explicită a numelui de tabel pentru a da mai multă claritate frazei SQL. În interogarea se mai sus atributul Nume din clauza SELECT nu trebuie prefixat prin numele tabelei din care provine, dat fiind că aceasta nu poate fi alta decât tabela Furnizor. În schimb în clauza WHERE este necesar să precizăm în mod explicit apartenența celor două attribute CodF care provin din tabelele Furnizor, respectiv Oferte.

7. "Numerele furnizorilor din Cluj care oferă calculatoare?"

```
SELECT DISTINCT Furnizor.Nume
FROM Furnizor, Oferte, Produs
WHERE Furnizor.CodF=Oferte.CodF
AND Oferte.CodP=Produs.CodP
AND Furnizor.Oras='Cluj',
AND Produs.Nume='Calculator'
```

8. "Numerele furnizorilor din Cluj care au vândut calculatoare unor beneficiari din București?"

```
SELECT DISTINCT Furnizor.Nume
FROM Furnizor, Beneficiar, Produs, Tranzactii
WHERE Furnizor.CodF=Tranzactii.CodF
AND Tranzactii.CodB=Beneficiar.CodB
AND Tranzactii.CodP=Produs.CodP
AND Furnizor.Oras='Cluj',
AND Beneficiar.Oras='Bucuresti',
AND Produs.Nume='Calculator'
```



### 6.3. Folosirea operatorului IN

În SQL Server singura formă acceptată a operatorului IN este:

*expresie* [NOT] IN *set\_de\_valori*

unde:

*expresie*

- se evaluatează la o singură valoare (nu poate fi o listă de valori);

*set\_de\_valori*

- este o mulțime de valori având elemente de același tip ca și *expresie*. *set\_de\_valori* specifică fie o listă explicită de valori constante, fie se obține prin evaluarea unei fraze SELECT.

"*Numele furnizorilor care oferă calculatoare sau automobile?*"

```
SELECT Furnizor.Nume
      FROM Furnizor, Oferte, Produs
      WHERE Furnizor.CodF = Oferte.CodF
            AND Oferte.CodP=Produs.CodP
            AND Produs.Nume IN ('Calculator', 'Automobil')
```

Observație:

Ultima condiție din fraza SELECT de mai sus:

```
Produs.Nume IN ('Calculator', 'Automobil')
```

este echivalentă cu:

```
(Produs.Nume='Calculator' OR Produs.Nume='Automobil')
```

10.

"*Numele furnizorilor care oferă calculatoare și automobile?*"

Soluția la această interogare se poate exprima în mod natural ca intersecția dintre furnizorii de calculatoare și furnizorii de automobile:

```
(SELECT DISTINCT Furnizor.Nume
      FROM Furnizor, Oferte, Produs
      WHERE Furnizor.CodF=Oferte.CodF
            AND Produs.Nume='Calculator')
INTERSECT
(SELECT DISTINCT Furnizor.Nume
      FROM Furnizor, Oferte, Produs
      WHERE Furnizor.CodF=Oferte.CodF
            AND Produs.Nume='Automobil')
```

11.

"*Numele furnizorilor care nu au nici o ofertă?*"

```
SELECT Nume
      FROM Furnizor
      WHERE CodF NOT IN
            (SELECT CodF
              FROM Oferte)
```

Interrogarea de mai sus returnează numele furnizorilor ai căror coduri nu apar cel puțin odată în relația *Oferte*.

Folosirea în varianta negată a operatorului IN (NOT IN) este cea mai comună, atunci când membrul drept (*set\_de\_valori*) se obține dintr-o frază SELECT. Varianta fără negație având ca membru drept o relație (ce provine dintr-o frază SELECT) se poate, de regulă, substitui prin operații de cuplare.

sau într-o variantă mai eficientă:

```
SELECT Nume
      FROM Furnizor
      WHERE CodF IN
            ((SELECT CodF
              FROM Oferte, Produs
              WHERE Oferte.CodP=Produs.CodP
                    AND Nume='Calculator'))
INTERSECT
      (SELECT CodF
        FROM Oferte, Produs
        WHERE Oferte.CodP=Produs.CodP
              AND Nume='Automobil'))
```

Deși ambele soluții sunt compatibile SQL'92 ele nu sunt acceptate de SQL Server care nu are implementat operatorul INTERSECT. Folosirea intersecției poate fi evitată, în acest caz, după cum se observă din următoarea soluție care este compatibilă nu numai cu standardul SQL 92, ci și cu implementarea SQL Server:

```
SELECT Nume
      FROM Furnizor
      WHERE CodF IN
            ((SELECT CodF
              FROM Oferte, Produs
              WHERE Oferte.CodP=Produs.CodP
                    AND Nume='Calculator'))
INTERSECT
      (SELECT CodF
        FROM Oferte, Produs
        WHERE Oferte.CodP=Produs.CodP
              AND Nume='Automobil'))
```



12. "Numele furnizorilor din Cluj fără beneficiari în București?"

```
SELECT Nume
FROM Furnizor
WHERE Oras='Cluj'
AND CodF NOT IN
    (SELECT CodF
     FROM Tranzactii, Beneficiar
      WHERE Tranzactii.CodB=Beneficiar.CodB
        AND Oras='Bucuresti' )
```

fără negație, se poate înlocui prin operații de cuplare, atunci când membrul

drept (*set\_de\_valori*) se obține dintr-o frază SELECT.

13. "Numele beneficiarilor care nu cumpără calculatoare?"

```
SELECT Nume
FROM Beneficiar
WHERE CodB NOT IN
    (SELECT CodB
     FROM Tranzactii, Produs
      WHERE Tranzactii.CodP=Produs.CodP
        AND Nume='Calculatoare' )
```

14. "Numele furnizorilor care nu oferă nici calculatoare, nici automobile?"

```
SELECT DISTINCT Nume
FROM Furnizor
WHERE CodF NOT IN
    (SELECT CodF
     FROM Oferta, Produs
      WHERE Oferta.CodP=Produs.CodP
        AND (Produs.Nume='Calculatoare',
             OR Produs.Nume='Automobile' ))
```

15. "Numele furnizorilor din Cluj care au beneficiari numai în București?"

```
SELECT DISTINCT Furnizor.Nume
FROM Furnizor,Tranzactii,Beneficiar
WHERE Furnizor.CodF=Tranzactii.CodF
AND Tranzactii.CodB=Beneficiar.CodB
AND Furnizor.Oras='Cluj'
AND Beneficiar.Oras='Bucuresti'
AND Furnizor.CodF NOT IN
    (SELECT DISTINCT CodF
     FROM Tranzactii,Beneficiar
      WHERE Tranzactii.CodB=Beneficiar.CodB
        AND Oras=>'Bucuresti' )
```

În interogarea de mai sus, fraza **SELECT** imbricată returnează codurile furnizorilor care au cel puțin un beneficiar în alt oraș decât București. Rezultatul dorit se obține prin selecția acelor furnizori din Cluj care au

cel puțin un beneficiar în București și nu fac parte din multimea furnizorilor cu beneficiari din alt oraș decât București.

Urmașoarele două exemple ilustrează faptul că operatorul **IN**, în varianta

fără negație, se poate înlocui prin operații de cuplare, atunci când membrul

drept (*set\_de\_valori*) se obține dintr-o frază SELECT.

16. "Numele furnizorilor care livrăză mai mult de un singur produs?"

*Varianta1:*

```
SELECT Nume
FROM Furnizor
WHERE CodF IN (SELECT CodF
    FROM Tranzactii T
     WHERE CodF IN (SELECT CodF
        FROM Tranzactii
         WHERE CodP >> T.codP))
```

*Varianta2:*

```
SELECT Nume
```

```
FROM Furnizor
WHERE CodF IN (SELECT T1.codF
    FROM Tranzactii T1,Tranzactii T2
     WHERE T1.CodF=T2.CodF
       AND T1.CodP>>T2.CodP)
```

*Varianta3:*

```
SELECT DISTINCT Nume
FROM Furnizor, Tranzactii T1,Tranzactii T2
WHERE Furnizor.CodF= T1.CodF
    AND T1.CodP=T2.CodF
      AND T1.CodP<>T2.CodP
```

17. "Numele furnizorilor care au cel puțin doi beneficiari?"

*Varianta1:*

```
SELECT Nume
FROM Furnizor
WHERE CodF IN (SELECT CodF
    FROM Tranzactii T
     WHERE CodF IN
        (SELECT CodF
          FROM Tranzactii
           WHERE CodB=>T.CodB))
```

*Varianta 2:*

```
SELECT Nume
      FROM Furnizor
            WHERE CodF IN (SELECT T1.CodF
                           FROM Tranzactii T1,Tranzactii T2
                                 WHERE T1.CodF=T2.CodF
                                       AND T1.CodB>>T2.CodB)
```

*Varianta 3:*

```
SELECT DISTINCT Nume
      FROM Furnizor, Tranzactii T1,Tranzactii T2
            WHERE Furnizor.CodF= T1.CodF
                  AND T1.CodB=T2.CodF
                        AND T1.CodB>>T2.CodB
```

## 6.4. Folosirea operatorului EXISTS

Operatorul EXISTS se aplică unei fraze SELECT pentru a construi o expresie de tip boolean care poate fi folosită în clauza WHERE a unei alte fraze SELECT, dar și în cadrul unor instrucțiuni IF ori în expresii CASE.

Sintaxa este de forma:

[NOT] EXISTS(*fraza\_SELECT*)

rezultatul evaluării expresiei fiind TRUE dacă fraza SELECT, dată ca argument, returnează cel puțin un tuplu și FALSE în caz contrar.

În multe cazuri, o interogare care folosește forma negată a operatorului IN (NOT IN) se poate întocui printre-o alta, asemănătoare, dar care folosește operatorul EXISTS tot în forma negată (NOT EXISTS). Următoarele două exemple sunt rezcrieri ale interogărilor 14) și 15) care ilustrează acest aspect.

18. "Numerele furnizorilor care nu oferă nici calculatoare, nici automobile?"

```
SELECT Nume
      FROM Furnizor F
            WHERE NOT EXISTS(SELECT *
                           FROM Oferte_Produs
                                 WHERE Oferte_Produs.CodP=Produs.CodP
                                       AND (Produs.Nume='Calculatoare'
                                             OR Produs.Nume='Automobile')
                                         AND F.CodF=Oferte.CodF)
```

care returnează numele furnizorilor cu proprietatea că mulțimea tuturor beneficiarilor din București este inclusă în mulțimea beneficiarilor fiecărui furnizor din rezultat (vezi variabila de tuplu F, referita în al doilea membru al operatorului IN, prin care se returnează codurile de beneficiar pentru furnizorul curent din rezultat). Din păcate soluția de mai sus nu este acceptată în standardul SQL '92 care cere ca membrul stâng al operatorului IN să fie un scalar ori o tuplu, dar nu mulțime de tuple. În aceste condiții interogarea se poate rezolva folosind operatorul MINUS în combinație cu NOT EXISTS; se vor returna numele acelor furnizori cu proprietatea că nu există nici un beneficiar în București care să nu fie printre beneficiarii oricărui furnizor din rezultat.

19. "Numerele furnizorilor din Cluj care au beneficiari numai în București?"

```
SELECT DISTINCT F.Nume
      FROM Furnizor F,Tranzactii,Beneficiar
            WHERE F.CodF=Tranzactii.CodF
                  AND Tranzactii.CodB=Beneficiar.CodB
                        AND Furnizor.Oras='Cluj'
                              AND Beneficiar.Oras='București'
```

```
AND NOT EXISTS (SELECT *
                  FROM Tranzactii,Beneficiar
                        WHERE Tranzactii.CodB=Beneficiar.CodB
                              AND F.CodF=CodF
                                AND Beneficiar.Oras>>'București')
```

### Observație:

În general, operatorul IN este considerat ca fiind lent în varianța când membrul său drept este o frază SELECT. Din acest motiv se preferă înlocuirea operatorului IN prin operații de cuplare, iar a variantei negate, NOT IN, prin operatorul NOT EXISTS, ori de câte ori acest lucru este posibil.

20.

"Numerele furnizorilor care au ca beneficiari pe toți beneficiarii din București?"

În sistemele în care operatorul IN funcționează și ca operator de inclusiune între două mulțimi de tuple, soluția comodă pentru interogarea de mai sus este:

```
SELECT Nume
      FROM Furnizor F
            WHERE
                  (SELECT CodB
                     FROM Beneficiar
                           WHERE Oras = 'București')
                         IN
                               (SELECT CodB
                                  FROM Tranzactii
                                        WHERE F.CodF=CodF)
```

SELECT Nume  
FROM Furnizor F  
WHERE NOT EXISTS  
(SELECT CodB

FROM Beneficiar  
WHERE Oras = 'Bucuresti' )

$\cap \cup (\text{MINUS})$

(SELECT CodB  
FROM Tranzactii  
WHERE F.CodF=Tranzactii.CodF )

Nici această din urmă soluție nu este acceptată în SQL Server care nu are implementat operatorul MINUS. Acest neajuns poate fi eliminat prin folosirea repetată a operatorului NOT EXISTS, logica rezolvării fiind următoarea: returnează numele furnizorului dacă nu există nici un beneficiar din București care să nu fie și beneficiar al furnizorului curent.

```
SELECT Nume  
FROM Furnizor F  
WHERE NOT EXISTS(SELECT *  
                  FROM Beneficiar B  
                  WHERE Oras = 'Bucuresti'  
                  AND NOT EXISTS(SELECT *  
                                 FROM Tranzactii  
                                 WHERE F.CodF=Tranzactii.CodF  
                                 AND Tranzactii.CodB=B.CodB))
```

Soluția de mai sus este compatibilă SQL Server și este acceptată de orice sistem compatibil cu standardul SQL '92. Soluția precedentă, bazată pe folosirea operatorului MINUS este acceptată doar de sistemele care implementează complet standardul SQL'92 cum sunt cele din familia ORACLE.

22. "Numele furnizorilor care au beneficiari în toate orașele?"

```
SELECT Nume  
FROM Furnizor F  
WHERE NOT EXISTS (SELECT *  
                  FROM Produs P  
                  WHERE NOT EXISTS (SELECT *  
                                    FROM Tranzactii  
                                    WHERE CodF=F.CodF  
                                    AND CodP=P.CodP))  
  
AND  
NOT EXISTS (SELECT *  
                  FROM Beneficiar B  
                  WHERE NOT EXISTS (SELECT *  
                                    FROM Tranzactii  
                                    WHERE CodF=F.CodF  
                                    AND CodB=B.CodB ))
```

23. "Numele furnizorilor care au livrat toate tipurile de produse și au tranzacții cu toți beneficiarii?"

```
SELECT Nume  
FROM Furnizor F  
WHERE NOT EXISTS (SELECT *  
                  FROM Produs P  
                  WHERE NOT EXISTS (SELECT *  
                                    FROM Tranzactii  
                                    WHERE CodF=F.CodF  
                                    AND CodP=P.CodP))  
  
AND  
NOT EXISTS (SELECT *  
                  FROM Beneficiar B  
                  WHERE NOT EXISTS (SELECT *  
                                    FROM Tranzactii  
                                    WHERE CodF=F.CodF  
                                    AND CodB=B.CodB ))
```

24. "Numele furnizorilor care au livrat toate tipurile de produse fiecărui beneficiar?"

```
SELECT Nume  
FROM Furnizor F  
WHERE NOT EXISTS (SELECT *  
                  FROM Produs P  
                  WHERE NOT EXISTS (SELECT *  
                                    FROM Tranzactii  
                                    WHERE CodF=F.CodF  
                                    AND CodP=P.CodP  
                                    AND CodB=B.CodB ))
```

21. "Numele furnizorilor care oferă toate tipurile de produse?"

```
SELECT Nume  
FROM Furnizor F  
WHERE NOT EXISTS (SELECT *  
                  FROM Produs P  
                  WHERE NOT EXISTS (SELECT *  
                                    FROM Oferte  
                                    WHERE F.CodF=Oferte.CodF  
                                    AND CodP=P.CodP))
```

Observație:

La acest din urmă exemplu condiția care se pune este ca pentru fiecare pereche posibilă de produs-beneficiar (vezi produsul cartezian din fraza SELECT imbricată pe primul nivel!) să existe o tranzacție corespunzătoare a furnizorului selectat.

## 6.5. Exerciții și probleme

1. Să se rezolve în limbajul SQL următoarele interogări:

- a. "Numele beneficiarilor din București?"
- b. "Numele beneficiariilor care nu sunt din Cluj sau București?"
- c. "Numele produselor pentru care s-a încheiat cel puțin o tranzacție?"
- d. "Numele beneficiarilor din București care au cumpărat calculatoare?"
- e. "Numele beneficiariilor care au cereri de calculatoare?"
- f. "Numele beneficiarilor care au cereri de calculatoare și automobile?"
- g. "Numele beneficiariilor care nu au nici o cerere?"
- h. "Numele beneficiariilor care nu au cereri de calculatoare și nici de automobile?"
- i. "Numele beneficiariilor care au cumpărat cel puțin două tipuri de produse?"
- j. "Numele beneficiarilor care au cereri pentru toate tipurile de produse?"
- k. "Numele beneficiariilor care au cereri pentru toate tipurile de calculatoare și autoturisme?"

- l. "Numele beneficiariilor care nu au cereri pentru toate tipurile de produse?"
  - m. "Numele beneficiarilor care au cumpărat toate tipurile de produse pentru care au cerer?"
2. Să se scrie în variantă compatibilă SQL Server următoarele interogări:

a. (SELECT Furnizor.Nume  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.CodB=Beneficiar.CodB  
AND Tranzactii.CodB=Beneficiar.CodB  
INTERSECT  
(SELECT Furnizor.Nume  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.Oras='Bucuresti')  
(SELECT Furnizor.Nume  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.Oras='Cluj')

WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.CodB=Beneficiar.CodB  
AND Beneficiar.Oras='Cluj')

b. (SELECT Furnizor.Nume  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.Oras='Bucuresti')  
MINUS  
(SELECT Furnizor.Nume  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.Oras='Cluj')

c. SELECT Nume  
FROM Furnizor F  
WHERE  
(SELECT CodP  
FROM Produs)  
IN  
(SELECT CodP  
FROM Oferte  
WHERE F.CodF=CodP)

## 7. FORMULAREA INTEROGĂRIILOR ÎN LIMBAJUL SQL (PARTEA A II-A)

3.

"Numele orașelor și numărul de furnizori pentru primele 10 orașe în ordinea descrescătoare a numărului de furnizori?"

```
SELECT TOP 10 Oras, COUNT(CodF)
FROM Furnizor
GROUP BY Oras
ORDER BY COUNT(CodF) DESC
```

### 7.1. Folosirea clauzelor GROUP BY și HAVING

Interrogările ce urmăză ilustrează modul de folosire al clauzelor GROUP BY și HAVING.

De reținut că într-o clauza SELECT a unei interogări cu GROUP BY pot să apară doar caracteristici unice de grup, ceea ce în SQL Server înseamnă strict numai atributurile invocate în clauza GROUP BY, funcții de agregare (care în acest caz se calculează la nivel de grup) și combinații ale acestora. Sunt situații când, datorită unei corespondențe 1 la 1 între două attribute, s-ar putea considera, din punct de vedere logic, ca și caracteristici unice de grup și alte attribute decât cele folosite efectiv la grupare; de exemplu la gruparea după cod furnizor, numele furnizor este tot caracteristică unică de grup. Dacă se dorește includerea acestora în lista clauzei SELECT, atunci aceste attribute trebuie incluse și printre criteriile efective de grupare, chiar dacă aceasta nu va modifica configurația grupurilor. Același lucru trebuie făcut dacă se dorește folosirea unui asemenea atribut în clauza HAVING.

Desi gruparea este, în principiu, o operație costisitoare, ea se dovedește a fi, de multe ori, o soluție mai eficientă decât eventualele alternative care nu folosesc gruparea, dar recurg la fraze SELECT imbricate. La unele dintre exemplele ce urmează sunt arătate soluțiile posibile care nu folosesc gruparea și sunt mai lente decât soluțiile echivalente bazate pe grupare.

1. "Numele orașelor și numărul de furnizori din fiecare oraș?"  

```
SELECT Oras, COUNT(CodF)
FROM Furnizor
GROUP BY Oras
```
2. "Lista orașelor în care sunt cel puțin cinci furnizori?"  

```
SELECT Oras
FROM Furnizor
GROUP BY Oras
HAVING COUNT(CodF)>=5
```

4.

"Numărul de oferte săcute de fiecare furnizor (se va reține numele furnizorului și numărul ofertelor)?"

```
SELECT Nume, COUNT(CodP)
FROM Furnizor, Oferte
WHERE Furnizor.CodF=Oferte.CodF
GROUP BY Nume
```

5.

"Numele beneficiarilor care au încheiat tranzacții în valoare totală de peste un milion?"

```
SELECT Nume
FROM Beneficiar, Tranzactii
WHERE Tranzactii.CodB=Beneficiar.CodB
GROUP BY Nume
HAVING SUM(Cantitate*Pret)>1000000
```

La soluția de mai sus gruparea este folosită doar pentru a oferi un criteriu de selecție al beneficiarilor din relația rezultat. În acest caz se poate rezolva interogarea fără grupare, folosind o frază SELECT imbricată care calculează valoarea totală a tranzacțiilor pentru fiecare beneficiar în parte:

```
SELECT Nume
FROM Beneficiar B
WHERE (SELECT SUM(Cantitate*Pret)
      FROM Tranzactii
      WHERE Tranzactii.CodB=B.CodB)>1000000
```

De reținut că această din urmă soluție se dovedește a fi mult mai lentă decât cea în care se folosește GROUP BY.

6.

"Numele beneficiarilor din Cluj care au achiziționat calculatoare în valoare de peste un milion de la furnizori din București?"

```
SELECT Beneficiar.Nume
FROM Furnizor, Tranzactii, Beneficiar, Produs
WHERE Furnizor.CodF=Tranzactii.CodF
AND Tranzactii.CodB=Beneficiar.CodB
AND Beneficiar.Oras='Cluj'
AND Produs.CodP=Furnizor.CodF
AND Produs.Oras='Bucuresti'
```

AND Beneficiar.Oras='Cluj'  
AND Produs.Nume='Calculator'

HAVING SUM(Cantitate\*Pret)>1000000

7. "Lista perechilor furnizor-beneficiar care au încheiat cel puțin 10 tranzacții?"

```
SELECT Furnizor.Nume, Beneficiar.Nume  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Tranzactii.CodB=Beneficiar.CodB  
GROUP BY Furnizor.Nume, Beneficiar.Nume  
HAVING COUNT(Tranzactii.CodF)>=10
```

8. "Lista perechilor furnizor-beneficiar care nu au încheiat între ei nici o tranzacție?"

Pornind de la interogarea precedentă, din punct de vedere logic, această interogare se poate reformula astfel:

- "Lista perechilor furnizor-beneficiar care au încheiat 0 tranzacții?"

ceea ce ar sugera o soluție derivată din soluția interogării precedente în care condiția:

COUNT(Tranzactii.CodF)>=10

COUNT(Tranzactii.CodF)=0

Rezultatul returnat de fraza **SELECT** astfel obținută va returna invariabil o relație fără nici o tuplă deoarece în relația *Tranzactii* apar perechile de coduri furnizor-beneficiar numai pentru cei care au încheiat cel puțin o tranzacție între ei, iar cei fără tranzacții (cu 0 tranzacții) nu apar în tabela *Tranzactii*. Variantele de mai jos rezolvă interogarea fără a folosi gruparea. Poate fi dezvoltată și o soluție care folosește gruparea, pornind de la ideea de mai sus, dar presupune folosirea operatorilor de cuplare **RIGHT JOIN** și **CROSS JOIN** (vezi capitolul 8, operatori de cuplare externă).

- Varianta 1 (fără GROUP BY și fără operatorul COUNT):

```
SELECT F.Nume, B.Nume  
FROM Furnizor F,Beneficiar B  
WHERE NOT EXISTS(SELECT *
```

*Varianța 2 (fără GROUP BY, folosind operatorul COUNT):*

```
SELECT F.Nume, B.Nume  
FROM Furnizor F,Beneficiar B  
WHERE (SELECT COUNT(*)  
FROM Tranzactii  
WHERE CodF=F.CodF  
AND CodB=B.CodB)=0
```

*Varianța 3 (soluție compatibilă SQL '92, dar necompatibilă SQL Server):*

```
SELECT Furnizor.Nume, Beneficiar.Nume  
FROM Furnizor, Beneficiar  
WHERE (CodB,CodF) NOT IN (SELECT DISTINCT CodB, CodF  
FROM Tranzactii)
```

#### Observație:

În mod clar, *Varianta 1* de mai sus este mai eficientă decât *Varianta 2*.

Aceasta se datorează faptului că operatorul **EXISTS** este bine optimizat în SQL Server și în toate cazurile este de preferat unei construcții de forma:

(SELECT COUNT (...) )=0

*Varianta 3* nu este acceptată de SQL Server deoarece operatorul **IN (NOT IN)** nu acceptă ca membru stâng o listă de valori, ci numai expresii care returnează o singură valoare.

9. "Lista perechilor furnizor-beneficiar, împreună cu numărul de tranzacții încheiate între cei doi?"

```
SELECT Furnizor.Nume, Beneficiar.Nume, COUNT(*)  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Tranzactii.CodB=Beneficiar.CodB  
GROUP BY Furnizor.Nume, Beneficiar.Nume
```

10. "Lista perechilor furnizor-beneficiar, împreună cu media valorii tranzacțiilor încheiate între cei doi?"

```
SELECT Furnizor.Nume, Beneficiar.Nume, AVG(Cantitate*Pret)  
FROM Furnizor,Tranzactii,Beneficiar  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Tranzactii.CodB=Beneficiar.CodB  
GROUP BY Furnizor.Nume, Beneficiar.Nume
```

11.

"Lista perechilor furnizor-beneficiar cu furnizor din Cluj și beneficiar din București cu media valorii tranzacțiilor pește 10000?"

```
SELECT Furnizor.Nume, Beneficiar.Nume
FROM Furnizor,Tranzactii,Beneficiar
WHERE Furnizor.CodF=Tranzactii.CodF
AND Tranzactii.CodB=Beneficiar.CodB
AND Beneficiar.Oras='Cluj',
AND Beneficiar.Oras='Bucuresti'
GROUP BY Furnizor.Nume, Beneficiar.Nume
HAVING AVG(Cantitate*Pret)>10000
```

12. "Lista perechilor de orașe astfel încât în al doilea oraș să existe cel puțin 10 beneficiari care au cel puțin un furnizor în primul oraș?"

```
SELECT DISTINCT Furnizor.Oras, Beneficiar.Oras
FROM Furnizor,Tranzactii,Beneficiar
WHERE Furnizor.CodF=Tranzactii.CodF
AND Tranzactii.CodB=Beneficiar.CodB
GROUP BY Furnizor.Oras, Beneficiar.Oras
HAVING COUNT(DISTINCT Beneficiar.CodB)>=10
```

13. "Lista perechilor de orașe astfel încât în primul oraș să existe cel puțin un furnizor care să aibă cel puțin 10 beneficiari din al doilea oraș?"

```
SELECT DISTINCT Furnizor.Beneficiar.Oras
FROM Furnizor,Tranzactii,Beneficiar
WHERE Furnizor.CodF=Tranzactii.CodF
AND Tranzactii.CodB=Beneficiar.CodB
GROUP BY Furnizor.CodF,Furnizor.Oras
HAVING COUNT(DISTINCT Beneficiar.CodB)>=10
```

Observație:

Pentru a obține rezultatul dorit este necesar să formăm grupuri astfel încât fiecare grup să conțină beneficiarii căreia unui furnizor și dintr-un singur oraș beneficiar. De aici rezultă că și criterii de grupare atribuibile: *Furnizor.CodF* și *Beneficiar.Oras*. Totuși, rezultatul trebuie să conțină valorile atributului *Furnizor.Oras* ceea ce înseamnă că și acesta trebuie inclus printre atribuibile de grupare. De remarcat că acest atribut nu influențează modul de formare al grupurilor deoarece într-un grup format după *Furnizor.CodF* nu pot exista două valori diferite pentru *Furnizor.Oras*. Opțiunea *DISTINCT* din clauza *SELECT* este necesară pentru că o pereche de orașe care satisfac condițiile cerute va fi selectată pentru fiecare furnizor existent în orașul furnizor corespunzător. În funcția de agregate *COUNT*, opțiunea *DISTINCT* este

14.

"Cea mai bună ofertă pentru fiecare tip de produs (se va reține numele tranzacției care au furnizorul curent) pentru același beneficiar."

```
SELECT Nume, MIN(Pret)
FROM Oferte, Produs
WHERE Oferte.CodP=Produs.CodP
GROUP BY Nume
```

15. "Cea mai bună ofertă pentru fiecare produs și furnizorul care a făcut oferta (se va reține numele furnizorului, numele produsului și prețul ofertei)?"

Această întrebare, în ciuda asemănării cu precedenta, nu se rezolvă la fel de ușor din cauza cerinței de a avea în rezultat numele furnizorului care a făcut cea mai bună ofertă pentru un produs dat. Aceasta elimină posibilitatea unei grupări directe în vederea aplicării funcției de agregare *MIN* asupra atributului preț la nivelul fiecărui grup, deoarece ar impune gruparea și după numele furnizor ceea ce nu corespunde cu rezultatul dorit. Prezentăm 3 variante diferite de rezolvare dintre care doar ultima apelăază la grupare.

Varianta1:

```
SELECT P.Nume, Oferte.Pret, Furnizor.Nume
FROM Furnizor,Oferte,Produs P
WHERE Furnizor.CodF=Oferte.CodF
AND Oferte.CodP=P.CodP
```

```
AND Oferte.Pret <=ALL(SELECT Pret
FROM Oferte
WHERE CodP=P.CodP)
```

Varianta2:

```
SELECT P.Nume, Oferte.Pret, Furnizor.Nume
FROM Furnizor,Oferte,Produs P
WHERE Furnizor.CodF=Oferte.CodF
AND Oferte.CodP=P.CodP
AND Oferte.Pret = (SELECT MIN(Pret)
FROM Oferte
WHERE CodP=P.CodP)
```

Varianta3:

```
SELECT Produs.Nume, OM.Pret_minim, Furnizor.Nume
FROM Furnizor,Oferte, (SELECT CodP, 'Pret_minim'=MIN(Pret)
FROM Oferte
```

necesară pentru că un grup poate conține mai multe tuple (după căte tranzacții are cu furnizorul curent!) pentru același beneficiar.

```

        GROUP BY CodP) OM_Produs
WHERE Furnizor.CodF=Oferte.CodF
AND Oferte.CodP=OM.CodP
AND OM.CodP=Produs.CodP
AND OM.Pret_minim=Oferte.Pret

```

#### Observație:

Prin cele două variante sunt asemănătoare: se selectează acele oferte care au proprietatea că prețul corespunzător este mai mic sau egal decât toate celelalte prețuri pentru același produs (vezi operatorul **ALL** în *Varianta1*), respectiv dacă prețul este egal cu minimul prețului pentru acel produs (*Varianta2*). *Varianta3* se bazează pe contruirea relației *OM* care conține prețul minim pentru fiecare produs, urmată de determinarea furnizorului care a făcut oferta la acel preț și a numelor furnizor-produs prin operatii de cuplare.

**16.** "Lista orașelor împreună cu numele și valoarea tranzacțiilor încheiate de către furnizorul din acel oraș care a realizat cea mai mare cifră de venituri în săptămâna prezentată?"

```

SELECT Oras.Nume, SUM(Cantitate*Pret)
FROM Furnizor F,Tranzactii
WHERE F.Code=F.Tranzactii.CodF
GROUP BY Oras,Nume

```

HAVING

```

SUM(Cantitate*Pret)>=ALL(SELECT SUM(Cantitate*Pret)
FROM Furnizor,Tranzactii
WHERE Furnizor.CodF=Tranzactii.CodF
GROUP BY Furnizor.Oras)

```

#### Observație:

Dintre grupurile de tranzacții corespunzătoare fiecărui furnizor se selectează, prin clauza **HAVING**, doar grupurile care au proprietatea că suma valorilor tranzacțiilor este mai mare sau egală decât oricare din sumele similare pe acel oraș. Incluzarea numelui de oraș ca și criteriu de grupare, în fraza **SELECT** principală, este impusă doar de cerința că acest atribut să apară în rezultat și nu influențează formarea grupurilor. Ca și la interogarea precedentă, se poate formula o variantă care în locul operatorului **ALL** folosește funcția de agregare **MAX**.

**17.** "Lista orașelor și numele beneficiarilor din fiecare oraș cu cea mai mică valoare a mărfurilor achiziționate?"

```

SELECT Oras.Nume,SUM(Cantitate*Pret)
FROM Beneficiar,B,Tranzactii
WHERE B.CodB=Tranzactii.CodB
GROUP BY Oras,Nume

```

HAVING

```

SUM(Cantitate*Pret)<=ALL(SELECT SUM(Cantitate*Pret)
FROM Beneficiar,Tranzactii
WHERE Beneficiar.CodB=Tranzactii.CodB
AND Beneficiar.Oras=B.Oras
GROUP BY Beneficiar.CodB)

```

## 7.2. Exerciții și probleme

**1.** Să se rezolve în limbajul SQL următoarele interogaři:

- "Numărul de beneficiari din fiecare oraș?"
- "Numărul de beneficiari din fiecare oraș cu proprietatea că au încheiat cel puțin o tranzacție?"
- "Numele produsului și numărul de oferte pentru fiecare produs?"
- "Numele produsului împreună cu prețul mediu și cantitatea totală oferită din acel produs?"
- "Lista perechilor furnizor-beneficiar care au încheiat între ei tranzacții și au sediul în același oraș?"
- "Lista orașelor împreună cu perechile furnizor-beneficiar care au încheiat între ei tranzacții și au sediul în acel oraș?"
- "Lista perechilor de orașe împreună cu numărul de perechi furnizor-beneficiar care au încheiat între ei tranzacții, furnizorul fiind din primul oraș și beneficiarul din al doilea oraș?"

## 8. VALORI NULL SI OPERATII DE CUPLARE EXTERNA

Valorile NULL și operațiile de cuplare externă sunt două subiecte aflate în strânsă legătură. SQL Server, ca de altfel orice SGBD care acceptă valori NULL, definește o serie reguli, operatori și funcții prin care se asigură tratamentul valorilor NULL. În SQL Server sunt prevăzute două variante diferite de tratare a valorilor NULL:

- *varianta compatibilă SQL'92*

Indicatorul de sistem ANSI\_NULLS este setat pe opțiunea ON, acesta este modul de lucru implicit și de asemenea cel care asigură compatibilitatea cu alte SGBD-uri.

- *varianta specifică SQL Server*

Indicatorul de sistem ANSI\_NULLS este setat pe opțiunea OFF; acest mod de lucru are anumite particularități care vor fi evidențiate în cele ce urmează, dar nu este compatibil cu standardul SQL'92, ceea ce înseamnă că programele dezvoltate în acest mod de lucru ar putea funcționa incorrect dacă sunt migrate pe alt tip de SGBD.

Operațiile de cuplare externă (FULL JOIN, RIGHT JOIN, LEFT JOIN) produc, de regulă, rezultate care, prin definiție, pot conține valori NULL. Din acest motiv prezentarea și înțelegerea lor este condiționată de cunoașterea modului în care sistemul tratează valorile NULL.

### 8.1. Valori Null

- valoare NULL indică faptul că valoarea respectivă este necunoscută.
- valoare NULL este diferită de un sir vid sau de o valoare 0. Două valori NULL nu sunt niciodată egale între ele. Compararea dintre două valori NULL și orice altă valoare returnează o valoare necunoscută (UNKNOWN).
- La operarea cu valori NULL trebuie să se țină seama de următoarele reguli:

AND	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

În prezența valorilor NULL operatorii logici și de comparație acionează în conformitate cu o logică trivalentă și ca atare pot returna o sau trei valoare de tip UNKNOWN pe lângă valourile logice TRUE și FALSE.

Tabele de mai jos indică efectul operatorilor logici în prezența valorilor NULL:

- Pentru a testa în clauza WHERE a unei interogări dacă o valoare este sau nu NULL se folosesc operatorii IS NULL și IS NOT NULL.
- La afisarea rezultatului unei interogări valorile NULL sunt afișate sub forma (null).
- Valorile NULL pot fi inserate în tabele într-o din următoarele moduri:
  - specificând explicit valoarea NULL într-o comandă INSERT sau UPDATE;
  - prin omisiunea unei coloane într-o comandă INSERT;
  - prin adăugarea unei coloane noi la o tabelă existentă prin comanda ALTER TABLE.
- Valorile NULL nu pot fi folosite pentru a face distincție între două rânduri ale unei tabele.
- Toate funcțiile de agregare ignoră valorile NULL existente în setul de valori asupra căreia se aplică.
- Operatorii aritmeticici, pe bit și cei pe șiruri de caractere returnează NULL dacă cel puțin unul din operanzi este NULL.
- Dacă există una sau mai multe valori NULL printre valorile unui atribut specificat într-o clauza GROUP BY, acestea formează toate un singur grup.
- La cuplarea tabelelor valorile NULL nu se vor lega la alte valori NULL și la nici o altă valoare. Pentru a avea în rezultat tuplele care au valori NULL în coloanele de legătură se va folosi una din variantele de cuplare externă (LEFT/RIGHT/FULL JOIN).

De remarcat că în varianta:

OR	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE
NOT		Resultat	
TRUE		FALSE	
UNKNOWN		UNKNOWN	
FALSE		TRUE	

Standardul SQL'92 introduce operatorii postfix **IS NULL** și **IS NOT NULL** pentru a testa dacă o valoare este sau nu **NULL**.

IS NULL	Rezultat	IS NOT NULL	Rezultat
TRUE	FALSE	TRUE	TRUE
NULL	TRUE	NULL	FALSE
FALSE	FALSE	FALSE	TRUE

Observație:

SQL Server oferă o extensie proprie în tratamentul valorilor **NULL**. Dacă opțiunea **ANSI\_NULLS** este setată pe **OFF**, atunci comparațiile între valori **NULL**, cum ar fi **NULL = NULL**, se evaluatează la **TRUE**, iar comparațiile între **NULL** și orice altă valoare produc **FALSE**.

### 8.1.1. Funcția ISNULL

Înlocuiește o valoare **NULL** cu o valoare de înlocuire specificată.

Sintaxa:

**ISNULL(expresie, valoare\_de\_inlocuire)**

Dacă *expresie* se evaluatează la **NULL**, atunci se returnează *valoare\_de\_inlocuire*. În caz contrar rezultatul este cel obținut din evaluarea parametrului *expresie*.

Exemple:

- "Listă ofertelor cu cantitate neprecizată (nume furnizor și nume produs)?"

```
SELECT Furnizor.Nume, Produs.Nume
      FROM Furnizor, Oferte, Produs
     WHERE Furnizor.CodP=Oferte.CodP
       AND Oferte.CodP=Produs.CodP
     AND Cantitate IS NULL
```

```
SELECT Furnizor.Nume, Produs.Nume
      FROM Furnizor, Oferte, Produs
     WHERE Furnizor.CodP=Oferte.CodP
       AND Oferte.CodP=Produs.CodP
     AND Cantitate=NULL
```

De remarcat că în varianta resultatul returnat este o listă vidă chiar dacă există oferte cu valoarea **NULL** pentru cantitate.

Observație:

Dacă opțiunea **ANSI\_NULLS** este setată pe **OFF**, atunci ambele variante de interogare returnează rezultatul corect. Cu toate acestea varianta a două este de evitat deoarece nu este acceptată de standardul SQL'92 și ca atare nu va funcționa corect pe alte tipuri de SGBD.

### 2.

"Numărul de oferte cu cantitate neprecizată (cu valoare **NULL** pentru atributul *Cantitate*)?"

Interogarea:

```
SELECT COUNT(Cantitate)
      FROM Oferte
```

returnează numărul de oferte cu cantitate nenuă deoarece celelalte sunt ignorate de funcția de agregare **COUNT**. Numărul total de oferte se poate obține apelând la funcția **ISNULL** astfel:

```
SELECT COUNT(ISNULL(Cantitate, 1))
      FROM Oferte
```

iar numărul de oferte cu valoare **NULL** a atributului *Cantitate* este dat de:

```
SELECT COUNT(ISNULL(Cantitate, 1)) - COUNT(Cantitate)
      FROM Oferte
```

Același rezultat se poate obține mai direct prin interogarea:

```
SELECT COUNT(*)
      FROM Oferte
     WHERE Cantitate IS NULL
```

Atenție:

Varianță:

```
SELECT COUNT(Cantitate)
      FROM Oferte
     WHERE Cantitate IS NULL
```

nu returnează rezultatul dorit, ci valoarea 0, deoarece numai valorile NULL sunt selectate și ele nu sunt luate în considerare la numărare. Următoarea interogare conduce și ea la rezultatul corect:

```
SELECT COUNT(IIFNULL(Cantitate, 1))  
FROM Oferte  
WHERE Cantitate IS NULL
```

### 3. "Media cantităților oferite?"

Să considerăm două variante de rezolvare:

```
SELECT AVG(Cantitate)  
FROM Oferte
```

```
SELECT AVG(IIFNULL(Cantitate, 0))  
FROM Oferte
```

De regulă, prima soluție returnează o medie mai mare decât a doua care ia în considerare și valorile NULL în calculul mediei, înlocuindu-le cu 0. Care dintre variante este corectă constituie mai degrabă o problemă de interpretare a prezenței valorilor NULL. De exemplu a doua variantă ar putea fi mai corectă decât prima dacă în expresia:

```
AVG(IIFNULL(Cantitate, 0))
```

valoarea 0 se înlocuiește cu o valoare medie estimată a cantităților neprecizate încă în oferte (presupunând că această estimare se poate face!).

Observație:

Comparativ:

```
Cantitate=Cantitate
```

returnează valoarea UNKNOWN pentru acele oferte care au NULL ca valoare a atributului *Cantitate*. Valoarea UNKNOWN se comportă ca și valoarea logică FALSE în operațiile de tip AND și OR, dar nu se poate nicidecum substitui cu aceasta. Deosebita apără la folosirea operatorului logic de negație NOT. Astfel: NOT UNKNOWN are ca rezultat valoarea UNKNOWN, față de NOT FALSE care returnează TRUE.

În consecință, interogarea:

```
SELECT *  
FROM Oferte  
WHERE Cantitate=Cantitate
```

returnează ofertele pentru care cantitatea nu este NULL (deoarece pentru tuplele care au valoarea NULL a atributului *Cantitate*, testul *Cantitate=Cantitate*

returnează valoarea UNKNOWN, iar WHERE UNKNOWN se interpretează ca WHERE FALSE!), în timp ce interogarea:

```
SELECT *  
FROM Oferte  
WHERE NOT Cantitate=Cantitate
```

returnează o relație vidă, deoarece WHERE NOT UNKNOWN este WHERE UNKNOWN care se interpretează ca WHERE FALSE.

De remarcat că interogarea:

```
SELECT *  
FROM Oferte  
WHERE Cantitate IS NULL OR Cantitate=Cantitate
```

returnează întreaga relație *Oferte*, fiind echivalentă cu:

```
SELECT * FROM Oferte
```

## 8.2. Operații de cuplare externă - interogări

1. "Lista furnizorilor, împreună cu numărul de oferte al fiecărui furnizor (pentru furnizorii fără oferte se va trece 0 la numărul de oferte)?"

```
SELECT Nume, 'Numar produse' = COUNT(CodP)  
FROM Furnizor LEFT JOIN Oferte ON  
Furnizor.CodF=Oferte.CodF  
GROUP BY Nume
```

Observație:

Operația de cuplare externă:

Furnizor LEFT JOIN Oferte ON Furnizor.CodF=Oferte.CodF produce o relație care include și furnizorii fără nici o ofertă. Va exista câte o tuplă pentru fiecare asemenea furnizor și aceasta va avea valori NULL în câmpurile corespunzătoare relației *Oferte*. Pentru furnizorii fără oferte operatorul COUNT(*CodP*) acționează asupra unei singure tuple care are valoarea NULL în câmpul *CodP*. Prin definiție, valorile NULL sunt ignorate de funcțiile de agregare, ceea ce înseamnă că pentru grupurile de furnizori fără oferte COUNT(*CodP*) va fi 0.

2.

"Numele produsului și numărul de tranzacții încheiate pentru fiecare produs (pentru produsele neterminante se va trece 0 la numărul de tranzacții)"

```
SELECT Nume, 'Numar tranzactii'=COUNT(CodT)
FROM Produs LEFT JOIN Tranzactii ON
Produs.CodP=Tranzactii.CodP
GROUP BY Nume
```

3.

"Lista perechilor furnizor-beneficiar care au încheiat între ei tranzacții pentru beneficiarii fără furnizor se va trece "fără furnizor"?"

```
SELECT 'Nume furnizor'=
ISNULL(Furnizor.Nume, 'Fara furnizor'),
'Nume beneficiar'=
ISNULL(Beneficiar.Nume, 'Fara beneficiar')
FROM Furnizor FULL JOIN (Tranzactii RIGHT JOIN Beneficiar ON
ON Furnizor.CodF=Tranzactii.CodF
AND Beneficiar.CodB=Tranzactii.CodB)
GROUP BY Furnizor.Nume, Beneficiar.Nume
```

Observație:

Operația de cuplare externă:

```
Tranzactii RIGHT JOIN Beneficiar
ON Tranzactii.CodB=Beneficiar.CodB
```

produce tuple cu valoarea **NUL**a atributului *CodF* pentru beneficiarii fără tranzacții. Dacă dorim ca aceștia să se regăsească în rezultatul final, cuplarea cu relația *Furnizori* trebuie făcută printr-o operație **FULL JOIN**. Dacă această ultimă cuplare s-ar face prin **LEFT JOIN**, atunci beneficiarii fără tranzacții s-ar pierde, deoarece la cuplarea tabelelor valoile **NUL** nu se leagă la alte valori **NUL** și nici la vreo altă valoare.

4.

"Lista perechilor furnizor-beneficiar între care nu s-a încheiat beneficiar și 0 tranzacții și invers?"

```
SELECT 'Nume furnizor'=
ISNULL(Furnizor.Nume, 'Fara furnizor'),
'Nume beneficiar'=
ISNULL(Beneficiar.Nume, 'Fara beneficiar')
```

```
'Numar tranzactii'=COUNT(CodT)
FROM Furnizor FULL JOIN (Tranzactii RIGHT JOIN Beneficiar ON
Tranzactii.CodF=Furnizor.CodP
AND Beneficiar.CodB=Tranzactii.CodB)
```

5.

"Lista tuturor perechilor furnizor-beneficiar și numărul de tranzacții încheiate între cei doi?"

```
SELECT 'Numar tranzactii'=COUNT(CodT)
FROM Tranzactii RIGHT JOIN (Furnizor CROSS JOIN Beneficiar
ON Furnizor.CodF=Tranzactii.CodF
AND Beneficiar.CodB=Tranzactii.CodB)
GROUP BY Furnizor.Nume, Beneficiar.Nume
```

Observație:

Să remarcăm deosebirea dintre interogările 4) și 5) precum și soluțiile corespunzătoare. Relația rezultat returnată la interogarea 4) este mai compactă. De exemplu, un furnizor care nu are nici un beneficiar apare o singură dată, spre deosebire de rezultatul de la interogarea 5) unde furnizorul respectiv apare pentru fiecare beneficiar (cu care nu are încheiate tranzacții).

În schimb, soluția interogării 5) se poate adapta ușor pentru a rezolva următoarea interogare:

"Lista perechilor furnizor-beneficiar între care nu s-a încheiat nici o tranzacție?"

Pornind de la interogarea precedență, soluția se obține simplu prin alegerea doar a acelor perechi furnizor-beneficiar pentru care numărul tranzacțiilor încheiate între cei doi este zero.

```
SELECT Furnizor.Nume, Beneficiar.Nume
FROM Tranzactii
RIGHT JOIN (Furnizor CROSS JOIN Beneficiar)
Beneficiar.CodB=Tranzactii.CodB
GROUP BY Furnizor.Nume, Beneficiar.Nume
HAVING COUNT(CodT)=0
```

O variantă mai directă de rezolvare a acestei interogări presupune eliminarea din mulțimea tuturor perechilor furnizor-beneficiar a celor între care s-a încheiat cel puțin o tranzacție:

```
SELECT F.Nume, B.Nume
FROM Furnizor F, Beneficiar B
WHERE NOT EXISTS
(SELECT CodF, CodB
FROM Tranzactii T
WHERE F.CodF=T.CodF
AND B.CodB=T.CodB)
```

Această variantă se dovedește a fi mai rapidă decât varianta bazată pe grupare.

Reamintim că varianta:

```
SELECT Furnizor..Nume, Beneficiar..Nume
FROM Furnizor, Beneficiar
WHERE (CodF, CodB) NOT IN
(SELECT CodF, CodB
FROM Tranzactii)
```

desi perfect legală conform standardului SQL'92, nu este acceptată în SQL Server.

6.

"Lista oraselor în care există furnizori sau beneficiari împreună cu numărul de furnizori, respectiv numărul de beneficiari din fiecare oraș (se va trece 0 în coloana corespunzătoare dacă într-unul din oraș lipsește furnizorii ori beneficiarii)?"

Varianta1:

```
SELECT
'Nume oras' = ISNULL(Furnizor..Oras, Beneficiar..Oras),
'Numar furnizori' = COUNT(DISTINCT CodF),
'Numar beneficiari' = COUNT(DISTINCT CodB)
FROM Furnizor FULL JOIN Beneficiar ON
Furnizor..Oras=Beneficiar..Oras
GROUP BY Furnizor..Oras, Beneficiar..Oras
```

Varianta2:

```
SELECT
'Nume oras' =
ISNULL(Orase_Furnizor..Oras, Orase_Beneficiar..Oras),
'Numar beneficiari' = ISNULL([Numar furnizori], 0),
'Numar furnizori' = COUNT(DISTINCT CodF)
FROM
(SELECT Oras, 'Numar furnizori' = COUNT(DISTINCT CodF)
FROM Furnizor
GROUP BY Oras) AS Orase_Furnizor
FULL JOIN
(SELECT Oras, 'Numar beneficiari' =
COUNT(DISTINCT CodB)
FROM Beneficiar
GROUP BY Oras) AS Orase_Beneficiar
ON Orase_Furnizor..Oras=Orase_Beneficiar..Oras
```

Observație:

Varianța2, de rezolvare a interogării de mai sus, exploatează facilitatea existentă în SQL Server, dar și alte SGBD-uri, prin care se permite ca relația rezultată dintr-o frază SELECT și deci, din punct de vedere

Varianta2, de rezolvare a interogării de mai sus, exploatează facilitatea existentă în SQL Server, dar și alte SGBD-uri, prin care se permite ca relația rezultată dintr-o frază SELECT și deci, din punct de vedere

sintactic, chiar fraza SELECT care o produce să fie plasată oriunde este legală plasarea unui nume de tabel, deci inclusiv în lista unei clauze FROM. În cazul nostru, relațiile Orase\_Furnizor și Orase\_Beneficiar conțin liste de orașe în care există furnizori, respectiv beneficiari împreună cu numărul acestora.

7.

"Lista furnizorilor împreună cu numărul de tipuri de calculatoare și tipuri de automobile pe care le oferă (se va trece 0 pentru cei care nu oferă unul sau nici unul dintre aceste produse)?"

```
SELECT Furnizor..Nume,
'Numar tipuri calculatoare' = ISNULL(Tipuri_calculatoare, 0),
'Numar tipuri auto' = ISNULL(Tipuri_auto, 0)
FROM Furnizor
LEFT JOIN (
(SELECT Oferte..CodF, Tipuri_calculatoare=COUNT(Produs..CodP)
FROM Oferte, Produs
WHERE Oferte..CodP=Produs..CodP
AND Produs..Nume='Calculator'
GROUP BY Oferte..CodF) AS Furnizor_Calculatoare
FULL JOIN
(SELECT Oferte..CodF, Tipuri_auto=COUNT(Produs..CodP)
FROM Oferte, Produs
WHERE Oferte..CodP=Produs..CodP
AND Produs..Nume='Auto',
GROUP BY Oferte..CodF) AS Furnizor_Auto
ON Furnizor..Calculatoare.CodF=Furnizor_Auto..CodF
ON Furnizor..CodF=ISNULL(Furnizor_Calculatoare..CodF,
Furnizor_Auto..CodF))
```

Observație:

1. În relațiile Furnizor\_Calculatoare și Furnizor\_Auto se determină codurile furnizorilor de calculatoare, respectiv de auto, împreună cu numărul de tipuri diferite de produs din fiecare categorie. Cuplarea prin FULL JOIN a celor două rezultate intermediere produce o relație care conține atât furnizorii care oferă numai calculatoare, cât și pe cei care oferă numai auto. Prima categorie va avea valoarea NULL în câmpul Furnizor\_Auto..CodF, iar a doua va avea valoarea NULL în câmpul Furnizor\_Calculatoare..CodF. Pentru a obține totuși rezultatul dorit, inclusiv cei care nu oferă nici calculatoare și nici auto, se face LEFT JOIN cu lista tuturor furnizorilor, criteriul de cuplare fiind egalitatea codului furnizor (Furnizor..CodF) cu oricare dintre codurile de furnizor de calculatoare (Furnizor\_Calculatoare..CodF) sau furnizor de auto (Furnizor\_Auto..CodF), care are o valoare diferita de NULL. De fapt condiția:

ON Furnizor.CodF=ISNULL(Furnizor\_Calculatoare.CodF,  
Furnizor\_Auto.CodF)

se poate scrie echivalent sub forma:

ON Furnizor.CodF=Furnizor\_Calculatoare.CodF OR  
Furnizor.CodF=Furnizor\_Auto.CodF

2.

Pentru această ultimă interogare se poate formula și o soluție care nu face apel la operații de cuplare externă. Soluția se bazează pe includerea în lista SELECT a două fraze SELECT imbicate care returnează câte o singură valoare și anume numărul tipurilor de calculatoare, respectiv auto pentru furnizorul curent:

"Numar

tipuri calculatoare"

(SELECT COUNT(Oferte.CodP)

FROM Oferte, Produs

WHERE Oferte.CodP=Produs.CodP

AND Oferte.CodF=F.CodF

"Numar tipuri auto"

(SELECT COUNT(Oferte.CodP)

FROM Oferte, Produs

WHERE Oferte.CodP=Produs.CodP

AND Oferte.CodF=F.CodF

AND Produs.Nume='calculator')

(SELECT COUNT(Oferte.CodP)

FROM Oferte, Produs

WHERE Oferte.CodP=Produs.CodP

AND Oferte.CodF=F.CodF

AND Produs.Nume='Auto')

Acaceași idee se poate aplica și în cazul interogării 6) astfel:

SELECT 'Nume oras'=0.Oras,  
'Numar furnizori'=(SELECT COUNT(CodF)  
FROM Furnizor  
WHERE Furnizor.Oras=0.Oras),  
'Numar beneficiari'=(SELECT COUNT(CodB)  
FROM Beneficiar  
WHERE Beneficiar.Oras=0.Oras)

UNION  
SELECT Oras FROM Furnizor  
JOIN Oras ON  
Oras.CodB=Beneficiar.CodB  
WHERE Beneficiar.Oras=0.Oras)

SELECT Oras FROM Beneficiar  
JOIN Oras ON  
Oras.CodB=Beneficiar.CodB  
WHERE Beneficiar.Oras=0.Oras)

Acetă gen de soluții sunt utile în sistemele care nu au implementat operatorii de cuplare externă, dar principal, sunt mai puțin eficiente din cauza frazelor SELECT imbicate care produc câte un rezultat pentru fiecare iterație din fraza SELECT principală.

Prin urmare, este recomandat să se folosească operatorii de cuplare internă sau să se folosească rezultatele obținute în urma unei interogări precedente.

### 8.3. Exerciții și probleme

1. Să se rezolve în limbajul SQL următoarele interogări:

a. "Lista beneficiarilor împreună cu numărul de cereri al fiecărui beneficiar (pentru beneficiarii fără oferte se va trece 0 la numărul de cereri)?"

b. "Numerele produsului și numărul de oferte, respectiv cereri pentru fiecare produs (pentru produsele fără oferte se va trece 0 la numărul de oferte, iar la cele fără cereri se va trece 0 la numărul de cereri)?"

c. "Numerele produsului și media valorilor ofertelor, cererilor și tranzacțiilor pentru fiecare produs (în lipsă de oferte, cereri sau tranzacții se va trece 0 pentru media respectivă)?"

d. "Lista perechilor oraș\_furnizor, oraș\_beneficiar și numărul de perechi furnizor-beneficiar din orașele respective și care au încheiat între ei cel puțin o tranzacție (se va trece 0 în lipsă)?"

2. Să se rezice interogarea de mai jos fără a folosi operatorul CROSS JOIN:

SELECT \*  
FROM Tranzactii CROSS JOIN  
(Furnizor CROSS JOIN  
(Beneficiar CROSS JOIN Produs))  
WHERE Furnizor.CodF=Tranzactii.CodF  
AND Beneficiar.CodB=Tranzactii.CodB  
AND Produs.CodP=Tranzactii.CodP

## 9. FUNCȚII CASE. OPERAȚII DE INSERARE, STERGERE și ACTUALIZARE

**WHEN** *expresie\_when* - este o expresie simplă cu care se compară *expresie\_intrare* în cazul funcțiilor CASE simple.  
*expresie\_intrare* și *expresie\_when* trebuie să fie de același tip sau să poată fi aduse la același tip printr-o conversie implicită.

*n* - indică faptul că expresia CASE poate conține mai multe clauze WHEN ...

### 9.1. Funcții (expresii) CASE

Evaluează o listă de condiții și returnează una din mai multe expresii rezultat posibile.

Exprăsie CASE au două formate:

1. Funcția CASE simplă compară o expresie cu un set de expresii simple pentru a determina rezultatul.
2. Funcția CASE cu căutare evaluează un set de expresii booleene pentru a determina rezultatul.

Ambele formate suportă o clauză ELSE optională.

Sintaxa:

Functia CASE simplă:

```
CASE expresie_intrare
      WHEN expresie_when THEN expresie_rezultat [...n]
```

Functia CASE simplă:

```
CASE expresie_intrare
      WHEN expresie_when THEN expresie_rezultat [...n]
      [ ]
      ELSE expresie_rezultat_else
END
```

Functia CASE cu căutare:

```
CASE
      WHEN expresie_booleana THEN expresie_rezultat[...n]
      [ ]
      ELSE expresie_rezultat_else
END
```

unde:

*expresie\_intrare* - este expresia de intrare care se compară cu expresiile *expresie\_when* în cazul funcțiilor CASE simple.

Functia CASE cu căutare:

1. Se evaluează *expresie\_intrare* și apoi, în ordinea specificată, se evaluează comparația *expresie\_intrare* = *expresie\_when* pentru fiecare clauză WHEN.
2. Returnează *expresie\_rezultat* pentru prima comparație (*expresie\_intrare* = *expresie\_when*) care se evaluează la TRUE.
3. Dacă nici o comparație *expresie\_intrare* = *expresie\_when* nu se evaluează la TRUE, atunci se returnează *expresie\_rezultat\_else* dacă există clauza ELSE ori NULL în caz contrar.

2. Se returnează *expresie\_rezultat* pentru prima *expresie\_booleana* care se evaluează la TRUE.

3. Dacă nici o *expresie\_booleana* nu se evaluează la TRUE, atunci se returnează *expresie\_rezultat\_else* dacă există clauza ELSE ori NULL în caz contrar.

**Exemplu:**

1. Atributul *UM* din tabela *Produs* conține prescurtări ale unităților de măsură folosite pentru fiecare produs. Fraza SELECT de mai jos folosește o funcție CASE simplă pentru a afișa o listă a produselor în care apare explicitată unitatea de măsură:

```
SELECT Nume,  
       CASE UM  
        WHEN 'KG' THEN 'Kilograme'  
        WHEN 'L' THEN 'Litri'  
        WHEN 'BUC' THEN 'Bucati'  
        WHEN 'T' THEN 'Tone'  
        ELSE ISNULL(UM, 'Unitate de masura nespecificata!')  
       END  
FROM Produs
```

**Observație:**

În cazul unui produs pentru care nu s-a specificat o explicitare a atributului *UM*, valoarea acestuia va fi afișată ca atare (vezi ramura ELSE!), exceptând cazul în care valoarea este NULL și se afișează un mesaj indicând faptul că unitatea de măsură nu este specificată. Astfel expresia CASE va returna întotdeauna o valoare diferită de NULL.

2. Interogarea de mai jos folosește o funcție CASE cu căutare pentru a afișa lista furnizorilor de calculatoare și categoria de preț pe care o practică fiecare:

```
SELECT 'Nume furnizor'=Furnizor.Nume,  
       'Categoria Preț'=_  
CASE  
    WHEN Pret IS NULL THEN 'Preț neprecizat!'  
    WHEN Pret < 500 THEN 'Preț foarte scăzut!'  
    WHEN Pret BETWEEN 501 AND 1000 THEN 'Preț mediu!'  
    ELSE 'Preț ridicat!'  
END  
FROM Furnizor,Oferete,Produs  
WHERE Furnizor.CodF=Oferete.CodF  
AND Oferete.CodP=Produs.CodP  
AND Produs.Nume='Calculator'
```

3. Interrogarea următoare returnează valoarea tranzacțiilor referitoare la calculatoare și auto pentru fiecare furnizor:

```
SELECT Furnizor.Nume  
      Calculetoare=SUM(CASE  
                           WHEN Nume='Calculator' THEN Pret*Cant  
                           ELSE 0 END),  
      Auto=SUM(CASE  
                           WHEN Nume='Auto' THEN Pret*Cant  
                           ELSE 0 END)  
      FROM Furnizor,Tranzactii,Produs  
      WHERE Furnizor.CodF=Tranzactii.CodF  
            AND Tranzactii.CodP=Produs.CodP  
      GROUP BY Furnizor.Nume
```

## 9.2. Instrucțiunea INSERT

### 9.2.1. Instrucțiunea INSERT

Adaugă una sau mai multe tuple noi la o relație sau vedere.

**Sintaxa (simplificată):**

```
INSERT [INTO] { nume_tabela | nume_vedere }  
{ ( lista_coloane ) }  
{ VALUES ( DEFAULT | NULL | expresie ) [,..,n] }  
| tabela_derivata  
| instructiune_execute
```

| DEFAULT VALUES

unde:

[INTO] - este un cuvânt cheie optional.  
*nume\_tabela* - este numele tabelei în care se face inserarea.

*nume\_vedere* - este numele unei vederi actualizabile în care se face inserarea, înținând cont de restricțiile referitoare la actualizările vederilor.

(*lista\_coloane*) - o listă de una sau mai multe coloane. Dacă o coloană lipsește din această listă atunci ea trebuie să poată fi completată automat pe baza definiției coloanei (aceptă NULL, valoare implicită definită sau are proprietatea IDENTITY), în caz contrar inserarea eșuează.

**VALUES** - precede lista de valori de inserat. Trebuie să existe câte o valoare pentru fiecare coloană specificată în *listă\_coloane* (daca există) ori în tabela (vedere) în care se face inserarea. Dacă coloanele din lista **VALUES** nu sunt în aceeași ordine ca și corespondentă pentru fiecare coloană, atunci se folosește *listă\_coloane* pentru a specifica explicit care valoare cărei coloane îi corespunde.

**DEFAULT** - forțează încărcarea valorii implicate definită pentru coloana corespunzătoare.

*expresie* - este o constantă, variabila sau o expresie. Expressia nu poate conține o instrucțiune **SELECT** sau **EXECUTE**.

*tabelă\_derivată* - este orice instrucțiune **SELECT** validă care returnează un set de tuple.

*instrucțiune\_execute* - este orice instrucțiune **EXECUTE** validă care returnează date printr-o instrucțiune **SELECT** sau **READTEXT**.

Dacă se folosește o *instrucțiune\_execute* într-un **INSERT**, atunci fiecare dintre seturile de tuple rezultat trebuie să fie compatibile cu coloanele tabelei în care se face inserarea sau cu cele specificate în *listă\_coloane*. O *instrucțiune\_execute* poate lansa în execuție o procedură stocată care poate returna una sau mai multe seturi rezultat.

**DEFAULT VALUES** - se vor insera în fiecare coloană valorile definite pentru tabelul în care se face inserarea.

#### Observații:

- O variabilă de tip **table** poate fi accesată la fel ca orice tabelă și poate fi invocată ca tabelă destinație într-o instrucțiune **INSERT**.
- Dacă o instrucțiune **INSERT** violează o constrângere sau regula ori furnizează o valoare incompatibilă pentru una din coloane, atunci întreaga operație eșuează și se afișează un mesaj de eroare.

#### Exemple:

- Inserare produs calculator (nu se specifică nici o valoare pentru câmpul *CodP* care este de tip *IDENTITY*):  
`INSERT Produs VALUES ('Calculator', 'Buc.')`

## 2.

Inserare produs portocală fără a specifica unitatea de măsură (se va insera **NULL** în câmpul **UM**):

```
INSERT Produs (Nume) VALUES ('Portocale')
```

## 3.

Inserarea tuturor beneficiarilor în tabela de furnizori:

```
INSERT Furnizor SELECT Nume, Oras FROM Beneficiar
```

### 9.2.2. Instrucțiunea **DELETE**

Realizează ștergerea de tuple dintr-o tabelă.

#### Sintaxa (simplificată):

```
DELETE [ FROM ] { nume_tabela | nume_vedere }
[ FROM { <subx_tabela> } [ ,...n ] ]
[ WHERE { <condiție_căutare>
| [ CURRENT OF
{ [ GLOBAL ] nume_cursor } | nume_variabilă_cursor }
] ]
```

unde:

**FROM** - cuvânt cheie optional care precede numele tabelei din care se face ștergerea.

*nume\_tabela* - este numele tabelei din care se face ștergerea.

*nume\_vedere* - este numele unei vederi actualizabile din care se face ștergerea îninând cont de restricțiile referitoare la actualizările vederilor.

**FROM** <*subx\_tabela*> - este o clauză **FROM** adițională. Aceasta este o extensie specifică SQL Server și înexistă în standardul SQL'92. Prin această extensie se poate specifica o cuplare de mai multe tabele care se poate folosi în locul unei fraze **SELECT** care să fie imbricată în clauza **WHERE** a instrucțiunii **DELETE**. Această extensie poate conține tabele, vederi, seturi rezultat obținute din fraze **SELECT** imbicate, cât și toate tipurile acceptate de operații de cuplare (**LEFT JOIN**, **RIGHT JOIN**, **FULL JOIN** etc.)

**WHERE** - specifică condițiile prin care se identifică tuplele de șters. În lipsa unei clauze **WHERE** se șterg toate tuplele din tabelă.

Există două forme de operații de stergere:

- ***Stergere cu căutare*** - se specifică o condiție de căutare pentru tuplele de șters.

datele din tabela de referință. (De exemplu nu se poate șterge tabela *Furnizor* prin **TRUNCATE TABLE** nici măcar atunci când tabelele *Oferte* și *Tranzacții* sunt vîde.)

**Example:**

- ## **1. Stergerea tuturor tranzacțiilor**

DELETE Furnizor WHERE Oras = 'Cluj'

Stergereea ofertelor de calculatoare.

Variation in community richness in subtropical forests, 1: 93

```
DELETE Oferte  
WHERE CodP IN (SELECT CodP
```

```
WHERE Nume='Calculator')
```

Varianta cu extensia SQL Server

DELETE Oferte

WHERE oferte.CodP=Produs.CodProd

Structiunea UPDATE

M-252-2

Modifica date existente într-o tabelă.

EDDIE (

SET

```
{  
    name_colon = { expresie | DEFAULT | NULL }  
}
```

| @variabila = coloana = expresie } [ ...n ]

**FROM** { *<sub\_tabela>* } [ ,...n ]

CURRENT OF

```
    } {[some] name_cursor} | name_variabla_cursor
```

1

### **Observații:**

2. Instrucția **DELETE** eșuează dacă se încearcă ștergerea unei tuple care este referită dintr-o altă tabelă printre constrângere de tip **FOREIGN KEY**. Dacă operația **DELETE** se referă la mai multe tuple și una dintre ele violatează o constrângere, atunci întreaga operație este anulată, se afișează un mesaj de eroare și nu se șterge nici o tuplă.

1  
3  
8

unde:

*nume\_tabela* - este numele tabelei în care se face modificarea.

*nume\_vedere* - este numele unei vederi actualizabile în care se face modificarea ținând cont de restricțiile referitoare la actualizările vederilor.

**SET** - specifică lista de coloane sau nume de variabilă ce se modifică.

*nume\_coloana* - este numele unei coloane care se modifică. Coloanele de tip **IDENTITY** nu se pot modifica.

**expresie** - este o variabilă, literal, expresie sau instrucțiune **SELECT** închisă între paranteze returnând o valoare unică.

**DEFAULT** - valoarea implicită definită pentru coloana specificată va înlocui valoarea existentă. Dacă coloana acceptă valori **NULL** și nu are definită o valoare implicită, atunci valoarea câmpului @variabila - este o variabilă declarată în prealabil care va lua valoarea rezultată din evaluarea lui *expresie*.

**SET @variabila = coloana = expresie** - setează variabila specificată la noua valoare a coloanei. Acesta diferă de construcția

**SET @variabila = coloana, coloana = expresie,**

prin care variabila este setată la valoarea veche a coloanei.

**FROM < sursa\_tabela >** - este o clauză **FROM** adițională. Este o extensie specifică SQL Server și permite aceleiasi facilități ca și clauza **FROM** din instrucțiunea **DELETE**.

**WHERE** - specifică condițiile prin care se identifică tuplele de modificat. În lipsa unei clauze **WHERE** se modifică toate tuplele din tabelă. Există două forme de operații de actualizare:

- *Actualizare cu căutare* - se specifică o condiție de căutare pentru tuplele de modificat.

- *Actualizare pozitionată* - se folosește clauza **CURRENT OF** pentru a specifica un cursor, iar actualizarea se referă la tupla afișată la poziția curentă a cursorului. Acest tip de actualizare permite schimbarea valorilor doar pentru una dintre mai multe tuple identice.

< conditie\_căutare > - orice condiție de căutare acceptată într-o clauză **WHERE**.

**CURRENT OF** - specifică actualizarea tuplei de la poziția curentă a cursorului specificat.

**GLOBAL** - specifică faptul că *nume\_cursor* se referă la un cursor global.

*nume\_cursor* - este numele unui cursor deschis care trebuie să permită operații de actualizare. Dacă există atât un cursor global, cât și unul local cu același nume, atunci *nume\_cursor* se referă la cursorul global dacă se specifică opțiunea **GLOBAL** și la cel local în caz contrar.

*nume\_variabila\_cursor* - este numele unei variabile cursor, care se referă la un cursor deschis și actualizabil.

#### Observații:

1. Dacă o operație de actualizare violează o constrângere, o regulă, încercă să schimbe la **NULL** o coloană care nu acceptă **NULL** ori valoarea nouă este incompatibilă cu tipul de date al coloanei, atunci operația este anulată, se afișează un mesaj de eroare și nu se face nici o actualizare.

2. În anumite situații operația **UPDATE** poate fi nedeterministă, ceea ce se manifestă prin faptul că noua valoare a unui atribut sau variabilă nu este unic determinată. În aceste cazuri rezultatul operației de actualizare este nedeterminat. De exemplu:

```
UPDATE Oferte
SET Pret=Tranzactii.Pret
FROM Oferte,Tranzactii
WHERE Oferte.CodP=Tranzactii.CodP
```

Prin operația **UPDATE** de mai sus se dorește aducerea prețurilor de ofertă la nivelul celor de tranzacționare. Deși corecta sintactic, operația de mai sus este ambiguă și cu rezultat imprevizibil ori de către ori pentru un produs există mai mult de o tranzacție.

3. Se pot folosi nume de variabile în instrucțiunile **UPDATE** pentru a stoca fie valoarea veche, fie cea nouă a unui atribut. Folosirea acestei facilități este recomandată pentru situațiile când operația de actualizare se referă la o singură tuplă, în cazul actualizărilor multiple variabila va conține valoarea corespunzătoare ultimei tuple modificate.

**Exemple:**

**1. Reducere cu 10% a prețurilor pentru toate ofertele:**

```
UPDATE Oferte  
SET Pret=0.9*Pret
```

**2.**

*Reducere cu 10% a prețurilor pentru ofertele cu prețul peste 1000:*

```
UPDATE Oferte  
SET Pret=0.9*Pret  
WHERE Pret>1000
```

**3.**

*Reducere cu 10% a prețurilor la calculatoare:*

*Varianța compatibilă cu standarul SQL'92:*

```
UPDATE Oferte  
SET Pret=0.9*Pret  
WHERE CodP IN (SELECT CodP  
FROM Produs  
WHERE Nume='Calculator')
```

*Varianța cu extensia SQL Server*

UPDATE Oferte

```
SET Pret=0.9*Pret  
FROM Oferte, Produs  
WHERE Oferte.CodP=Produs.CodP  
AND Nume='Calculator'
```

**4.**

*Reducere diferențială a prețurilor (5% pentru preț mai mic de 1000,*

*10% pentru de la 1000 la 1499 și 20% de la 1500 în sus):*

```
UPDATE Oferte  
SET Pret=  
CASE  
WHEN Pret<1000 THEN 0.95*Pret  
WHEN Pret>=1000 AND Pret<1500 THEN 0.80*Pret  
END  
FROM Oferte
```

**5.**

*Reducere diferențială a prețurilor (în condițiile de la exemplul 4), dar numai pentru calculatoare și auto:*

```
UPDATE Oferte  
SET Pret=  
CASE  
WHEN Pret<1000 THEN 0.95*Pret  
WHEN Pret>=1000 AND Pret<1500 THEN 0.90*Pret  
ELSE 0.80*Pret  
END  
FROM Oferte
```

### 9.3. Exerciții și probleme

- Să se rezarcă frazele UPDATE de la exemplele 5 și 6 fără a folosi expresii care conțin operatorul OR.

### Observație:

De remarcat la exemplele 5 și 6 modul de folosire al expresiilor CASE în clauza WHERE. Aceste construcții pot fi un înlocuitor pentru

- Să se rezarcă frazele UPDATE de la exemplele 5 și 6 fără a folosi expresii CASE în clauza WHERE.
- Dați un exemplu de posibilă operație de actualizare nedeterministă.
- Să se reducă cu 10% prețul tuturor produselor pentru care există cel puțin 2 oferte.
- Să se șteargă ofertele pentru care nu s-a încheiat nici o tranzacție.
- Să se reducă cu 20% prețul ofertelor pentru care cantitatea depășește 1000000 bucăți ori 100 tone și prețul unitar este peste 1000.

Care sunt valorile atributelor a și b pentru unica tuplă a tabeliei x după execuția sevenței următoare:

```
CREATE TABLE x(a int, b int)
INSERT x VALUES(1,2)
UPDATE x SET a=b, b=a
```

## 10. VEDERI. SQL DINAMIC

### 10.1. Vederi

O vedere este o tabelă virtuală al cărei conținut este definit printre interogare. La fel ca orice tabelă reală, o vedere constă dintr-un set de attribute și se materializează printr-un set de tuple. Datele corespunzătoare unei vederi nu sunt memorate într-un obiect al bazei de date, ceea ce se memorează este o frază SELECT pe baza căreia datele sunt determinate în momentul invocării vederii. Din această perspectivă o vedere poate fi considerată ca o interogare prememorată. Interogarea care definește o vedere poate face referire la una sau mai multe tabele de bază sau chiar vederi. Vederile se comportă și pot fi folosite la fel ca orice tabelă de bază în interogări, dar trebuie să respecte anumite restricții în cazul operațiilor de ștergere, inserare sau actualizare.

#### – simplificarea și specializarea vizionii utilizatorilor

Vederile pot fi văzute ca o suprastructură, peste structura de date a unei baze de date, care oferă utilizatorilor un model al datelor mai apropiat de întrebările lor și de necesitățile aplicatiilor pe care le dezvoltă; grupuri diferențiate de utilizatori pot avea vizuni diferite asupra acelorași colecții de date prin intermediul unor seturi de vederi diferite.

#### – oferă un mecanism suplimentar de securitate

Drepturile de acces și manipulare pot fi acordate la nivelul vederilor, în mod independent de cele de la nivelul tabelelor de bază; astfel anumite grupuri de utilizatori pot avea asupra vederilor drepturi specifice, în timp ce la nivelul tabelelor de bază drepturile acestora pot fi, mai mult sau mai puțin restricționate.

#### – simplificarea manipulării datelor

Prin-o proiectare adecvată a vederilor, interogările și operațiile de manipulare date pot fi exprimate mai simplu în termenii vederilor decât în termenii tabelelor de bază.

### - simplificarea operațiilor de import și export date

Prin vederi se poate realiza adaptarea dintre structurile de tabele diferite aparținând unor baze de date între care se face un transfer de date.

## 10.1.1. Instrucțiunea CREATE VIEW

Creează o nouă vedere:

Sintaxa:

```
CREATE VIEW nume_vedere [(coloană [, n]]]
[WITH ENCRYPTION]
AS
    instrucțiune_SELECT
[WITH CHECK OPTION]
```

unde:

*nume\_vedere* - este numele vederii.

*coloană* - este numele sub care apare o anumită coloană în vedere.

Numirea explicită a unei coloane este necesară doar dacă aceasta este obținută ca rezultat al unei expresii ori dacă se dorește un nume diferit al coloanei fătă de numele avut în tabela de bază din care provine. Numele de coloană pot fi atribuibile și prin fraza SELECT asociată vederii. Dacă nu se specifică o listă de nume din fraza SELECT care definește vederea.

**WITH ENCRYPTION** - determină stocarea în formă criptată a instrucțiunei SELECT care definește vederea.

Aceasta poate face referire la una sau mai multe tabele de bază și/sau vederi și poate fi de orice complexitate cu condiția respectării următoarelor restricții:

- nu poate folosi clauza ORDER BY.
- nu se poate folosi opțiunea INTO.
- nu se poate face referire la o tabelă temporară.

**WITH CHECK OPTION** - forțează toate operațiile de modificare date, referitoare la vedere, să adere la criteriile stabilite prin

### Observații:

1. Pentru orice interogare referitoare la o vedere, sistemul SQL Server verifică faptul că toate obiectele referite în vedere există și sunt valide în contextul bazei de date, iar în cazul operațiilor de modificare se verifică să nu se violeze nici una dintre constrângările de integritate de la nivelul tabelelor de bază. Dacă toate aceste condiții sunt satisfăcute, atunci operația exprimată în termenii vederii este tradusă în una sau mai multe operații echivalente la nivelul tabelelor de bază.

2. Dacă o vedere face referire la o tabelă care între timp a fost ştersă, atunci se generează o eroare la orice tentativă de utilizare a vederii. Chiar dacă tabela lipsă este recreată cu aceeași structură vedere rămâne invalidă. De aceea, ori de câte ori se modifică structura unei tabele sau vederi care este referată de o altă vedere, aceasta din urmă trebuie ştersă și recreată.

3. Definiția unei vederi este stocată în sistem în mod similar cu definiția procedurilor stocate. Spre deosebire de aceasta definiția unei vederi este recomplată la fiecare accesare a acesteia.

### Exemplu:

1. Instrucțiunea CREATE VIEW de mai jos produce o vedere numită *Tranzactii\_Desfasurat* care conține toate tranzacțiile împreună cu datele corespunzătoare (nume și oraș pentru furnizor și beneficiar, numele produsului, cantitatea tranzacționată și prețul de tranzacție):

```
IF EXISTS (SELECT Table_Name FROM Information_Schema.VIEWS
            WHERE Table_Name = 'Tranzactii_Desfasurat')
DROP VIEW Tranzactii_Desfasurat
GO
CREATE VIEW Tranzactii_Desfasurat AS
SELECT Numar_Furnizor=Furnizor.Oras,
       Nume_Beneficiar=Beneficiar.Nume,
       Oras_Beneficiar=Beneficiar.Oras,
       Nume_Produs=Produs.Nume,
       Cantitate,Pret
FROM Furnizor, Beneficiar, Produs, Tranzactii
WHERE Furnizor.CodeF=Tranzactii.CodeF
      AND Beneficiar.CodeB=Tranzactii.CodeB
      AND Produs.CodeP=Tranzactii.CodeP
```

**instrucțiune\_SELECT.** Dacă are loc modificarea unei tuple din vedere, clauza **WITH CHECK OPTION** garantează faptul că datele modificate rămân vizibile în vedere.

2.

Vedereea *Tranzactii\_Cluj\_Bucuresti* este definită pe baza vederii *Tranzactii\_Desfasurat* și se referă doar la tranzacțiile dintre furnizori din Cluj și beneficiari din București. Atributele acestei vederi vor fi a instrucțiunii **CREATE VIEW** care are prioritate asupra listei din fraza **SELECT** asociată.

```
IF EXISTS (SELECT Table_Name FROM Information_Schema.Views
           WHERE Table_Name = 'Tranzactii_Cluj_Bucuresti')
  GO
CREATE VIEW Tranzactii_Cluj_Bucuresti (Furnizor, Beneficiar, Produs)
  AS
SELECT Nume_Furnizor, Nume_Beneficiar, Nume_Produs
  FROM Tranzactii_Desfasurat
 WHERE Oras_Furnizor='Cluj' AND
       Oras_Beneficiar='Bucuresti'
GO
```

3.

Vedereea *Agentii\_Comerciali* permite referirea unitară atât la furnizori, cât și la beneficiari.

```
IF EXISTS (SELECT Table_Name FROM Information_Schema.Views
           WHERE Table_Name = 'Agentii_Comerciali')
  GO
DROP VIEW Agentii_Comerciali
CREATE VIEW Agentii_Comerciali (Cod, Nume, Oras) AS
SELECT * FROM Furnizor
UNION
SELECT * FROM Beneficiar
GO
```

Oricare dintre vederile de mai sus poate fi invocată în interogări la fel ca o tabelă obișnuită și, în general, contribuie la exprimarea simplificată a unora dintre interogări:

*Numele beneficiarilor care au cumpărat calculatoare de la furnizori din Cluj?*

```
SELECT Nume_Beneficiar
  FROM Tranzactii_Desfasurat
 WHERE Oras_Furnizor='Cluj' AND
       Nume_Produs='Calculatoare'
```

*Numele beneficiarilor din București care au cumpărat calculatoare de la furnizori din Cluj?*

```
SELECT Beneficiar
  FROM Tranzactii_Cluj_Bucuresti
 WHERE Produs='Calculatoare'
```

### 10.1.2. Instrucțiunea ALTER VIEW

Modifică definiția unei vederi creată anterior.

Sintaxa:

```
ALTER VIEW nume_vedere [(coloană [ , n ])]
  [WITH ENCRYPTION]
  AS
  instrucție_SELECT
  [WITH CHECK OPTION]
  unde:
    nume_vedere, coloană și instrucție_SELECT - au aceeași semnificație
    ca și în cazul instrucțiunii CREATE VIEW.
```

Observație:

Dacă vederea inițială a fost creată cu opțiunile **WITH ENCRYPTION** și/sau **CHECK OPTION** acestea sunt menținute numai dacă sunt specificate și în instrucțiunea **ALTER VIEW**.

Exemplu:

Modificăm definiția vederii *Tranzactii\_Cluj\_Bucuresti* astfel încât să conțină inclusiv valoarea tranzacțiilor:

```
ALTER VIEW Tranzactii_Cluj_Bucuresti (Furnizor, Beneficiar,
                                         Produs, Valoare) AS
SELECT Nume_Furnizor, Nume_Beneficiar,
       Produs, Valoare
  FROM Tranzactii_Desfasurat
 WHERE Oras_Furnizor='Cluj' AND
       Oras_Beneficiar='Bucuresti'
GO
```

#### 10.1.3. Instrucțiunea DROP VIEW

Sterge una sau mai multe vederi din baza de date.

Sintaxa:

```
DROP VIEW {nume_vedere} [, n]
```

*nume\_vedere* - este numele unei vederi care se stergă.

#### Observații:

- La ștergerea unui tabel printr-o instrucțiune **DROP TABLE**, toate vederile care referă tabelul devin invalide, fără a fi automat șterse. Ele trebuie șterse explicit (dacă e cazul!) prin căte o comandă **DROP VIEW**.

2. La ștergerea unei vederi prin **DROP VIEW** definiția și toate celelalte informații despre vedere sunt șterse din baza de date.
3. Încercarea de a șterge o vedere care nu există produce un mesaj de eroare.

#### Exemplu:

După ștergerea vederii *Tranzactii\_Desfasurat* vedere definită pe baza acesteia, *Tranzactii\_Cluj\_Bucuresti*, devine invalidă, astfel că orice referire la ea produce o eroare.

```
IF EXISTS (SELECT Table_Name FROM Information_Schema.Views  
           WHERE Table_Name = 'Tranzactii_Desfasurat')  
DROP VIEW Tranzactii_Desfasurat  
GO  
SELECT * FROM Tranzactii_Cluj_Bucuresti --eroare vedere invalidă
```

#### 10.1.4. Modificarea datelor din vedere

Efectuarea de modificări asupra datelor din vedere presupune respectarea unor restricții care se referă atât la vedere, cât și la instrucțiunile de modificare a datelor care acționează asupra vederilor.

În SQL Server o vedere este considerată modificabilă dacă:

- vedere referă (direct sau indirect) cel puțin o tabelă de bază, deci nu este bazată exclusiv pe una sau mai multe expresii.
- lista **SELECT** a interogării care definește vedere nu conține o funcție de agregare sau una din opțiunile **DISTINCT** sau **TOP**. Interogările imbinate din clauza **FROM** pot conține funcții de agregare cu condiția ca valorile deriveate din acestea să nu constituie obiectul modificărilor.

- fraza **SELECT** nu conține clauzele **GROUP BY** sau **UNION**.
- dacă operația este de inserare, atunci vedere nu poate conține coloane deriveate (obținute dintr-o expresie, funcție și.a.) în lista **SELECT**.

#### Observație:

- dacă operația este de ștergere date (**DELETE**), atunci fraza **SELECT** care definește vedere poate referi o singură tabelă de bază în clauza **FROM**.
- numele unei coloane nu trebuie să apară de mai multe ori în lista **SELECT** a interogării care definește vedere.

#### Exemplu:

În sistemul SQL Server 2000 o vedere care nu se încadrează în criteriile de mai sus poate fi totuși modificată prin triggere **INSTEAD OF** care permit explicitarea operațiilor **INSERT**, **UPDATE** și **DELETE**. O altă categorie de vederi care pot fi modificate direct în SQL Server 2000 sunt **vederile particionate modificabile**.

Pe lângă criteriile pe care trebuie să le satisfacă vederile pentru a fi modificabile, se mai impun o serie de restricții și asupra operațiilor de modificare a vederilor, și anume:

1. Coloanele referite într-o instrucțiune **UPDATE** sau **INSERT** trebuie să aparțină unei singure tabele din cele referite în definiția vederii, astfel încât să se poată determina fără ambiguități tabela de bază la care se referă operația.

2. Instrucțiunea **UPDATE** sau **INSERT** se traduce într-o operație legală la nivelul tabelului de bază din punctul de vedere al constrângерilor existente la nivelul tabelului (cheie primară, cheie străină, acceptare valori **NULL**, valori implicate etc.)

3. Dacă vedere este definită cu opțiunea **WITH CHECK OPTION**, atunci operația de modificare trebuie să adere la criteriile stabilite prin fraza **SELECT** care definește vedere. Astfel vor genera erori operațiile **UPDATE** care produc modificări prin care tuplele se exclud din vedere sau operațiile de **INSERT** ale unor tuple care nu pot face parte din vedere.

#### Exemplu:

1. Vederea *Data\_Currentă* creată mai jos nu este modificabilă, deoarece nu referă nici o tabelă de bază (conține o singură valoare rezultată din evaluarea unei funcții):

```
CREATE VIEW Data_Currentă AS  
SELECT GETDATE() AS Azi
```

2.

Următoarea operație de stergere:

```
DELETE FROM Tranzactii_Desfasurat
WHERE Oras_Furnizor='Cluj',
este ilegală deoarece vederea Tranzactii_Desfasurat este definită prin
cuplarea a mai multe tabele de bază.
```

3.

Operația de actualizare de mai jos este corectă și are ca efect modificarea atributului *Oras* în tabela *Furnizori*, dar numai pentru furnizorii care au cel puțin o tranzacție (care este de fapt condiția pentru a face parte din vederea *Tranzactii\_Desfasurat*):

```
UPDATE Tranzactii_Desfasurat
SET Oras_Furnizor='Timisoara'
```

4.

Următoarea operație determină modificări în tabela *Produs*

```
UPDATE Tranzactii_Desfasurat
SET Num_Produs='Televizor'
WHERE Num_Produs='Auto'
```

5.

Operația de inserare de mai jos este, de asemenea, legală și are ca efect inserarea unei tuple în tabela *beneficiar*:

```
INSERT Tranzactii_Desfasurat(Num_Beneficiar,
                               Oras_Beneficiar)
VALUES ('SANEX', 'Cluj')
```

6.

Inserarea următoare este ilegală deoarece referă attribute din două tabele (*Furnizor* și *Beneficiar*):

```
INSERT Tranzactii_Desfasurat(Num_Beneficiar,
                               Oras_Beneficiar, Num_Furnizor)
VALUES ('SANEX', 'Cluj', 'IRIS')
```

7.

Cele trei operații de mai jos sunt ilegale deoarece vederea *Agenti\_Comerciali* conține operatorul **UNION**:

```
INSERT Agenti_Comerciali(Num, Oras)
VALUES ('SHELL', 'Timisoara')
DELETE FROM Agenti_Comerciali WHERE Num='SANEX'
UPDATE Agenti_Comerciali
SET Num='IRIS' WHERE Num='SANEX'
```

8.

Fie vederea *Furnizori\_Cluj* definită astfel:

```
CREATE VIEW Furnizori_Cluj AS
SELECT * FROM Furnizor
WHERE Oras='Cluj'
GO
```

Operația:

```
INSERT Furnizori_Cluj VALUES ('SHELL', 'Timisoara')
```

Are ca efect inserarea unei tuple în tabela *Furnizor*, dar fără ca aceasta să fie vizibilă în vedere. După instrucțiunea de modificare:

```
UPDATE Furnizori_Cluj SET Oras='Timisoara'
tuplele din tabela Furnizor care aparțin vederii se modifică, și ... dispar din vedere.
```

Dacă se adaugă opțiunea **WITH CHECK** la definiția vederii *Furnizori\_Cluj*, de exemplu prin operația **ALTER VIEW** de mai jos:

```
ALTER VIEW Furnizori_Cluj AS
SELECT * FROM Furnizor
WHERE Oras='Cluj',
WITH CHECK OPTION
GO
```

atunci operațiile de mai sus produc eroare ori de către există cel puțin o tuplă care ar fi afectată de operație și prin care s-ar viola definiția vederii. Indiferent de prezența sau absența opțiunii **WITH CHECK** operațiile de mai jos sunt acceptate, dar nu au nici un efect:

```
UPDATE Furnizori_Cluj SET Oras='Cluj'
WHERE Oras>>'Cluj'
```

```
DELETE Furnizori_Cluj WHERE Oras>>'Cluj'
```

## 10.2. SQL dinamic

Sistemul SQL Server oferă posibilitatea de a lansa în execuție un sir de caractere. Acest lucru se poate realiza prin instrucțiunea **EXECUTE**, iar începând cu versiunea SQL Server 7.0 și prin procedura stocată sistem **sp\_executesql** care devine varianta preferată pentru execuția dinamica. Fie că se face cu **EXECUTE**, ori cu **sp\_executesql** un sir de caractere este introdusă și executat ca un batch de sine stătător. SQL Server compilează sirul de caractere și produce un plan de execuție care este separat de planul de execuție al batch-ului din care s-a lansat execuția instrucțiunii **EXECUTE** ori a

procedurii `sp_executesql`. La folosirea facilității de execuție dinamică trebuie avută în vedere o serie de particularități, și anume:

- Complierea instrucțiunilor SQL din șirul de caractere are loc abia în momentul lansării în execuție a instrucțiunii `EXECUTE` ori a procedurii `sp_executesql`. În acest moment are loc și verificarea corectitudinii acestor instrucțiuni și rezolvarea referințelor (către variabile) existente în șir.

- Contextul de execuție al șirului de caractere este separat de contextul batch-ului din care se lansează execuția. Astfel șirul executat nu are acces la variabilele din batch-ul care conține instrucțiunea `EXECUTE` sau apelul procedurii `sp_executesql`. Reciproc, batch-ul din care se lansează execuția nu are acces la variabilele sau cursoarele locale declarate în șirul executat.
- Dacă șirul executat conține o instrucție `USE` care schimbă contextul bazei de date, atunci această schimbare are efect doar pe durata execuției șirului și nu afectează batch-ul exterior.

Sintaxa pentru lansarea în execuție a unui șir prin instrucțiunea `EXECUTE` este:

```
EXECUTE ( { @variabilă_șir | [ N ] 'șir_SQL' } [ + ...n ] )
```

unde:

`@variabilă_șir` - este numele unei variabile locale de tip `char`, `varchar`, `nchar` sau `nvarchar` a cărei dimensiune este limitată de spațiul de memorie disponibil. Dacă șirul de executat depășește 4000 de caractere Unicod sau 8000 caractere ASCII, atunci se pot concatena mai multe variabile șir pentru a forma șirul de executat.

`[N]șir_SQL` - este un șir constant de tip `nvarchar` sau `varchar`. Dacă se specifică indicatorul `N`, atunci șirul se consideră de tip `nvarchar` (caractere Unicod) cu dimensiunea limitată de spațiul de memorie disponibil. Dacă șirul de executat depășește 4000 de caractere Unicod, atunci se pot concatena mai multe variabile șir pentru a forma șirul de executat.

Lansarea în execuție a unui șir prin procedura sistem `sp_executesql` se face după următoarea sintaxă:

```
sp_executesql [ @stmt = ] instrucție  
[ {, @params = } N'@nume_parametru tip_data [ ,n' ]  
{,[@nume_parametru = ] 'valoare' [ ,n ] } ]
```

unde:

`[@stmt = ] instrucție` - este un șir Unicod constant sau o variabilă șir Unicod. Nu sunt permise concatenări de șiruri și nici șirurile de caractere ASCII (o constantă șir va fi precedată de indicatorul `N`). Șirul poate conține parametrii având aceeași formă ca și un nume de variabilă. Fiecare parametru trebuie să aibă un corespondent în lista de parametri și în lista de valori.

`[@params = ] N'@nume_parametru tip_data [ ,n ]'` - este un șir Unicod constant sau o variabilă șir care conține definițiile tuturor parametrilor din șirul executat. Definiția fiecărui parametru constă dintr-un nume de parametru și un tip de dată.  
`[@nume_parametru = ] 'valoare' [ ,n ]` - este o listă de perechi de forma nume-valoare prin care se atribuie valori parametrelor. Valoarea atribuită poate fi o constantă sau o variabilă (nu sunt permise nume-valoare prin care se atribuie valori parametrelor). Valoarea poate fi o constantă sau o variabilă (nu sunt permise nume-valoare prin care se atribuie valori parametrelor).

#### Observații:

1. Apelul procedurii `sp_executesql` returnează 0 dacă șirul s-a executat cu succes și 1 în caz contrar.
2. Sunt returnate toate relațiile rezultat produse de frazele `SELECT` din șirul executat.
3. Procedura `sp_executesql` asigură execuția unei secvențe de instrucțiuni dintr-un șir de caractere parametrizat care se poate construi în mod dinamic și se poate reutiliza de mai multe ori.
4. `sp_executesql` se poate folosi în locul procedurilor stocate pentru execuția repetată a unei secvențe de instrucții, atunci când modificarea valorilor parametrilor este singura variație de la o execuție la alta.

Execuția dinamică a instrucțiunilor SQL oferă o flexibilitate care se dovedește a fi utilă, uneori chiar indispensabilă, în aplicații. Această flexibilitate aduce cu sine o oarecare penalizare la timpul de prelucrare, datorită faptului că are loc o recomplilare și construirea unui plan de execuție nou pentru fiecare lansare în execuție a secvenței dinamice. Acest lucru este inevitabil deoarece o secvență dinamică este, în principiu alta, la fiecare execuție, fiind dependentă de valorile actuale din contextul batch-ului din care se lansează execuția. Totuși, aceste penalizări sunt, de cele mai multe ori neglijabile și pe deplin compensează de avantajele execuției dinamice. Execuția oricărui șir de caractere se face într-un context propriu, izolat de contextul batch-ului din care

Se execută instrucțiunea **EXECUTE** sau procedura **sp\_executesql** și de contextul altor șiruri executate dinamic. Din acest motiv variabilele declarate local în cadrul șirului sau instrucțiunile **USE** au efect numai pe durata execuției și îlui în care apar. Astfel următoarele două secvențe sunt corecte (unde Agenti este numele bazei de date):

--corect!!!

USE Agenti

```
SELECT * FROM Furnizor
```

--corect!!!

EXECUTE ("USE Agenti

```
SELECT * FROM Furnizor")
```

în cele de mai jos sunt greșite:

```
--gresit!!!
EXECUTE ("SELECT * FROM Furnizor")
```

--gresit!!!
USE Agenti

```
EXECUTE ("SELECT * FROM Furnizor")
--gresit!!!
EXECUTE ("USE Agenti")
SELECT * FROM Furnizor
```

#### Exemple:

- Cele două instrucțiuni **EXECUTE** din secvența de mai jos au același efect:

```
DECLARE @nume_tabel varchar(10)
DECLARE @sir_SQL varchar(1000)
SET @nume_tabel='Furnizor'
SET @sir_SQL='SELECT * FROM '+@nume_tabel
EXECUTE ('SELECT * FROM '+@nume_tabel)
EXECUTE(@sir_SQL)
```

#### Observație:

La prima instrucțiune **EXECUTE** de mai sus apare variabila **@nume\_tabel** a cărei valoare din momentul execuției se concatenează cu partea constantă a șirului executat. A nu se confunda acest lucru cu folosirea variabilelor în șirul executat. Astfel instrucțiunea:

```
EXECUTE ('SELECT @nume_cimp FROM Furnizor')
produce o eroare de tip variabilă nedeclarată.
```

## 2.

Pentru a putea insera, în șirul de executat, valoarea unei variabile reprezentând un șir de caractere, acesta trebuie să fie încadrat între "" iar opțiunea **QUOTED\_IDENTIFIER** trebuie să fie setată pe **OFF**, deoarece, în caz contrar, valoarea variabilei se va interpreta ca un identificator:

```
DECLARE @nume_oras varchar(10)
SET @nume_oras='Cluj'
SET QUOTED_IDENTIFIER OFF
EXECUTE (' SELECT * FROM Furnizor
WHERE Oras="'+@nume_oras+' ')
```

## 3.

Acest exemplu pune în evidență modul de folosire a procedurii sistem **sp\_executesql** comparativ cu instrucțiunea **EXECUTE** (de remarcat că și apelul procedurii stocate **sp\_executesql** se face tot prin **EXECUTE**!). În cazul execuției prin apelul procedurii sistem **sp\_executesql** șirul de executat nu este reconstruit după fiecare modificare a numelui de oraș, ci se face apelul doar cu valoarea modificată a parametrului **@par\_nume\_oras**.

```
DECLARE @par_nume_oras varchar(10)
DECLARE @sir_SQL nvarchar(1000)
--utilizeaza instructiunea EXECUTE
SET QUOTED_IDENTIFIER OFF
SET @sir_SQL=
' SELECT * FROM Furnizor WHERE Oras="'+@par_nume_oras+' '
```

#### Exemple:

- Cele două instrucțiuni **EXECUTE** din secvența de mai jos au același efect:

```
DECLARE @nume_tabel varchar(10)
DECLARE @sir_SQL varchar(1000)
SET @nume_tabel='Furnizor'
SET @sir_SQL='SELECT * FROM '+@nume_tabel
EXECUTE ('SELECT * FROM '+@nume_tabel)
EXECUTE(@sir_SQL)

DECLARE @sir_SQL=
' SELECT * FROM Furnizor WHERE Oras="'+@par_nume_oras+' '
```

## Observații:

1. Procedura `sp_executesql` are mai multe avantaje față de execuția directă prin instrucțiunea `EXECUTE`:

- Parametrii care apar în sirul executat se transmit în formă nativă fără a fi nevoie de conversii explicite în sururi de caractere.
- De multe ori execuția repetată prin `sp_executesql` poate reduce considerabil penalitățile inherente în cazul folosirii instrucțiunii `EXECUTE`. Acest lucru se datorază faptului că SQL Server va încerca să reutilizeze planul de execuție generat la prima lansare în execuție ori de câte ori se reexecută un sir cu alte valori ale parametrilor.

2. De remarcat că nu se pot transmite numele de tabele din clauza `FROM` ca parametri la un sir executat prin `sp_executesql`. În aceste cazuri este necesară reconstruirea sirului de executat într-o manieră similară execuției prin instrucțiunea `EXECUTE`.

3. Secvențele de mai jos actualizează cantitățile de produse oferite din tabela *Oferte* ori cerute în tabela *Cereri*. A se remarcă faptul că valorile întregi pentru atributul cantitate trebuie convertite la un sir de caractere atunci când se folosește instrucțiunea `EXECUTE`. La schimbarea numelui de tabelă sirul de executat trebuie reconstruit chiar și atunci când se folosește procedura `sp_executesql`.
- ```
DECLARE @cantitate int
DECLARE @tabel varchar (20)
DECLARE @sir_SQL nvarchar(1000)

----utilizare instrucțiune EXECUTE
SET @tabel='Oferte'
SET @sir_SQL='UPDATE '+@tabel+' SET Cantitate=Cantitate+
SET @cantitate=100
SET @sir_cantitate=CONVERT(varchar(20),@cantitate)
EXECUTE(@sir_SQL+@sir_cantitate)

--utilizare procedura sp_executesql
DECLARE @sir_parametri nvarchar(1000)
SET @sir_SQL=N'UPDATE '+@tabel+' SET Cantitate=Cantitate+@par_cantitate
SET @sir_parametri=N'@par_cantitate int'

--actualizari Oferte fara refacerea sirului de executat @sir_SQL
EXECUTE sp_executesql
    @sir_SQL, @sir_parametri, @par_cantitate=@cantitate
SET @cantitate=200
```

`EXECUTE sp_executesql  
 @sir_SQL, @sir_parametri, @par_cantitate=@cantitate`

--refacerea simbului de executat @sir\_SQL pentru a actualiza tabela cereri

```
SET @tabel='Cereri'
SET @sir_SQL=N'UPDATE '+@tabel+' SET Cantitate=Cantitate+@par_cantitate'
EXECUTE sp_executesql
    @sir_SQL, @sir_parametri, @par_cantitate=@cantitate
```

## 10.3. Exerciții și probleme

1. Fie vedere *Oferte\_Desfasurat* definită prin instrucțiunea:

```
CREATE VIEW Oferte_Desfasurat AS
SELECT
    Nume_Furnizor=Furnizor.Nume,
    Oras_Furnizor=Furnizor.Oras,
    Nume_Produs= Produs.Nume,
    Cantitate,Pret,
    Valoare=Cantitate*Pret
FROM Furnizor, Oferte, Produs
WHERE Furnizor.CodF= Oferte.CodF
    AND Produs.CodP= Oferte.CodP
```

Să se arate care dintre instrucțiunile de mai jos este legală și care nu:

```
UPDATE Oferte_Desfasurat SET Oras_Furnizor='Timisoara'
UPDATE Oferte_Desfasurat SET Nume_Produs='Televizor'
INSERT Oferte_Desfasurat (Nume_Furnizor, Oras_Furnizor)
VALUES ('SANEX', 'Cluj')
```

2. Fie vedere *O* definită prin instrucțiunea:

```
CREATE VIEW O AS
SELECT * FROM Oferte
UNION
SELECT * FROM Oferte
```

Este vedere *O* modificabilă? Ce ambiguități apar în cazul unei operații de forma:

```
UPDATE O SET Cantitate=Cantitate+100
```

```
CREATE VIEW FF (F1,01_02) AS
SELECT F1.Nume,F1.Oras F2.Oras
FROM Furnizor F1, Furnizor F2
WHERE F1.Code=F2.Code
```

Explicați ce se întâmplă în cazul următoarei operații de modificare:

UPDATE FF SET 01='Bucuresti'

Este permisă următoarea operare:

UPDATE FF SET 01='Bucuresti', 02='Cluj' ?

Ce ambiguități pot să apară?

#### 4. Ce erori apar la instrucțiunile de mai jos:

```
EXECUTE('SELECT * FROM Furnizor WHERE Oras=Cluj')
EXECUTE('SET QUOTED_IDENTIFIER ON
        SELECT * FROM Furnizor WHERE Oras="Cluj"')
EXECUTE('SELECT * FROM @tabel')
```

#### 5. Să se scrie secvența de mai jos astfel încât operația de actualizare să fie executată sub controlul procedurii sp\_executesql.

```
DECLARE @table_name varchar(50)
DECLARE @field varchar(50)
DECLARE @value varchar(50)
DECLARE @newvalue varchar(50)

SET @table_name='Furnizor'
SET @field='Oras'
SET @value='Cluj'
SET @newvalue='Timisoara'

EXECUTE('UPDATE '+@table_name+' SET '+@field+'=@newvalue
        WHERE '+@field+' = "' +@value+ '"')
```

Atunci când se folosește SQL Server la dezvoltarea aplicațiilor (de exemplu într-o configurație client-server) se poate vorbi, în principiu, despre fi memorate local la nivelul aplicațiilor care trimit comenzi către SQL Server și prelucrarea rezultatelor returnate de acesta. A doua opțiune presupune dezvoltarea și înregistrarea programelor ca **proceduri stocate** în SQL Server și crearea de aplicații care apelează aceste proceduri și prelucrarea rezultatelor returnate de acestea.

Procedurile stocate din SQL Server sunt similare procedurilor din alte limbi de programare în sensul că:

- acceptă parametrii de intrare de la și returnează valori prin parametri de ieșire către un program apelant;
- conțin instrucțiuni de programare care efectuează operații în baza de date și pot apela, la rândul lor, alte proceduri stocate;
- returnează către apelant o valoare care indică succesul sau eșecul execuției procedurii, eventual cauză eșecului.

O procedură stocată se poate lansa în execuție prin comanda EXECUTE. Procedurile stocate au un comportament diferit față de funcțiile din SQL Server în sensul că nu returnează o valoare în locul numelui procedurii și, deci, acesta nu poate fi folosit direct în expresii.

Utilizarea procedurilor stocate prezintă mai multe avantaje față de varianta programelor stocate la nivelul aplicațiilor utilizator:

#### • Programare modulară

O procedură stocată poate fi creată o singură dată și apelată de

mai multe ori din mai multe aplicații. Mai mult, procedura stocată poate fi creată de către o persoană specializată în baze de date și poate fi modificată independent de aplicațiile care o apelează.

#### • Performanță îmbunătățită

În cazul programelor cu volum mare de cod sau a celor executate în mod repetat procedurile stocate sunt mai eficiente decăreace compilarea și optimizarea lor se face o singură dată la crearea procedurii și sunt memorate într-o formă direct executabilă prin care se evită repetarea fazelor de compilare și optimizare la fiecare apel al procedurii.

## 11. PROCEDURI STOCATE

Explicați ce se întâmplă în cazul următoarei operații de modificare:

UPDATE FF SET 01='Bucuresti',

Este permisă următoarea operație:

UPDATE FF SET 01='Bucuresti', 02='Cluj', ?

Ce ambiguități pot să apară?

4. Ce erori apar la instrucțiunile de mai jos:

```
EXECUTE (' SELECT * FROM Furnizor WHERE Oras=Cluj ')
EXECUTE (' SET QUOTED_IDENTIFIER ON
          SELECT * FROM Furnizor WHERE Oras="Cluj" ')
EXECUTE (' SELECT * FROM @table ')
```

5. Să se scrie secvența de mai jos astfel încât operația de actualizare să fie executată sub controlul procedurii sp\_executesql.

```
DECLARE @table_name varchar(50)
DECLARE @field varchar(50)
DECLARE @value varchar(50)
DECLARE @newvalue varchar(50)

SET @table_name='Furnizor'
SET @field='Oras'
SET @value='Cluj'
SET @newvalue='Timisoara'
EXECUTE('UPDATE '+@table_name+' SET '+
@field+'="'+
@newvalue+'" WHERE '+@field+'="'+@value+'"')
```

Atunci când se folosește **SQL Server** la dezvoltarea aplicațiilor (de exemplu într-o configurație client-server) se poate vorbi, în principiu, despre două opțiuni de bază pentru stocarea și executarea programelor. Programele pot fi memorate local la nivelul aplicatiilor care trimit comenzi către **SQL Server** și prelucrează rezultatele returnate de acesta. A doua opțiune presupune dezvoltarea și înregistrarea programelor ca **proceduri stocate** în **SQL Server** și crearea de aplicații care apelează aceste proceduri și prelucrează rezultatele returnate de acestea.

Procedurile stocate din **SQL Server** sunt similară procedurilor din alte limbi de programare în sensul că:

- acceptă parametrii de intrare de la și returnează valori prin parametri de ieșire către un program apelant;
- conțin instrucțiuni de programare care efectuează operații în baza de date și pot apela, la rândul lor, alte proceduri stocate;
- returnează către apelant o valoare care indică succesul sau eșecul execuției procedurii, eventual cauză eșecului.

O procedură stocată se poate lansa în execuție prin comanda **EXECUTE**. Procedurile stocate au un comportament diferit față de funcțiile din **SQL Server** în sensul că nu returnează o valoare în locul numelui procedurii și, deci, acesta nu poate fi folosit direct în expresii.

Utilizarea procedurilor stocate prezintă mai multe avantaje față de varianta programelor stocate la nivelul aplicatiilor utilizator:

#### • Programare modulară

O procedură stocată poate fi creată o singură dată și apelată de mai multe ori din mai multe aplicații. Mai mult, procedura stocată poate fi creată de către o persoană specializată în baze de date și poate fi modificată independent de aplicațiile care o apeleză.

#### • Performanță îmbunătățită

În cazul programelor cu volum mare de cod sau a celor execute în mod repetat procedurile stocate sunt mai eficiente deoarece compilarea și optimizarea lor se face o singură dată la crearea procedurii și sunt memorate într-o formă direct executabilă prin care se evită repetarea fazelor de compilare și optimizare la fiecare apel al procedurii.

## 11. PROCEDURI STOCATE

- Reducerea traficului de rețea

O prelucrare care presupune execuția a sute de liniilor de cod poate fi realizată printr-o singură linie de comandă care apelăază procedura stocată prin care este implementată acea prelucrare. Se evită astfel transmiterea prin rețea a codului respectiv la fiecare execuțare a acelei prelucrări.

- Oferă un mecanism suplimentar de securitate

Utilizatorii nu au acces direct la codul procedurilor stocate, iar dreptul de execuție al unei proceduri poate fi acordat sau nu în funcție de statutul fiecărui utilizator de către administratorul bazei de date.

În SQL Server o procedură stocată este creată prin comanda CREATE PROCEDURE și poate fi modificată ulterior printr-o comandă ALTER PROCEDURE. Definiția unei proceduri stocate conține următoarele elemente:

- specificarea *nume\_lui\_procedurii* și a *parametrilor*;
- *corpus\_procedurii*, care conține instrucțiunile SQL ce realizează prelucrarea pentru care a fost creată procedura.

## 11.1. Instrucțiunea CREATE PROCEDURE

Creează o procedură stocată cu caracter permanent ori temporar pentru utilizare în sesiunea utilizatorului curent (procedură temporară locală) sau în toate sesiunile utilizator deschise la un moment dat (procedură temporară globală). Procedurile stocate pot, de asemenea, să fie create pentru lansare automată în execuție la pornirea SQL Server.

Sintaxa:

```

CREATE PROC[EDURE] nume_procedura [;numar]
    { @parametru tip_de_data } [VARYING] [= valoare_implicită]
    [b, n]
    [WITH
        { RECOMPILE
        | ENCRYPTION
        | RECOMPILE, ENCRYPTION}
    ]
    [FOR REPLICATION]
    AS
    insracutuni_sql [...]n]

```

**OUTPUT** - indică faptul că parametrul curent este unul de ieșire și poate fi folosit pentru a returna valori dinspre procedură. Tipul de date **text** (sau echivalent) este interzis pentru parametri de tip **OUTPUT**.

n - indică faptul că o procedură poate avea mai mulți parametri, până la 1024.

**RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION**

- opțiunea RECOMPILE indică faptul că procedura va fi recompilată înainte de fiecare execuție a sa. Opțiunea

unde:

**nume\_procedura** - este numele procedurii stocate care se creează. Prin convenție, numele unei proceduri temporare locale începe cu simbolul #, iar al unei proceduri temporare globale cu ##.

Numele unei proceduri poate avea cel mult 128 de caractere.

**;numar** - întreg opțional folosit pentru a reuni mai multe proceduri cu același nume într-un grup. Toate procedurile dintr-un grup pot fi șterse cu o singură comandă **DROP PROCEDURE** având ca parametru numele comun al procedurilor din grup.

**@parametru** - este un parametru al procedurii stocate. O procedură stocată poate avea până la 1024 parametri care se comportă în cadrul procedurii ca orice variabilă locală. La execuție un parametru poate lua doar o valoare constantă și nu poate fi folosit în locul numelor de tabele, coloane sau alte obiecte din baza de date.

**tip\_de\_data** - este tipul de date al parametrului. Toate tipurile de date, inclusiv **text** și **image**, pot fi folosite ca parametru. Totuși, tipul de date **cursor** poate fi folosit doar pentru parametrii de tip **OUTPUT** (de ieșire). La specificarea unui tip de date **cursor**, cuvintele cheie **VARYING** și **OUTPUT** trebuie, de asemenea, specificate.

**VARYING** - specifică relația rezultat suportată ca parametru de ieșire. Se aplică numai la parametrii de tip cursor.

**valoare\_implicită** - este valoarea implicită a parametrului. Dacă o asemenea valoare este definită, atunci la apelul procedurii se poate omite specificarea unei valori pentru acest parametru. Valoarea implicită poate fi o constantă (care poate include caracterele speciale %, \_, [], și [^]) atunci când parametrul este utilizat cu operatorul **LIKE** sau **NULL**.

**OUTPUT** - indică faptul că parametrul curent este unul de ieșire și poate fi folosit pentru a returna valori dinspre procedură. Tipul de date **text** (sau echivalent) este interzis pentru parametri de tip **OUTPUT**.

n - indică faptul că o procedură poate avea mai mulți parametri, până la 1024.

**RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION**

- opțiunea RECOMPILE indică faptul că procedura va fi recompilată înainte de fiecare execuție a sa. Opțiunea

**ENCRYPTION** indică faptul că textul procedurii va fi memorat sub forma încriptată.

#### FOR

#### REPLICATION

- acastă opțiune se exclude reciproc cu opțiunea **WITH RECOMPILE** și specifică faptul că procedura curentă este creata pentru procesul de replicare și nu poate fi executată decât pe serverul curent.

#### AS

- marchează începutul corpului procedurii.

**instructioni.sql [...]n]** - instrucțiunile SQL care formează corpul procedurii. Într-o procedură, pot exista mai multe instrucțiuni cu respectarea următoarelor restricții:

- se respectă toate regulile specifice unui batch SQL (o procedură stocată este o unitate independentă de compilate);
- se poate include orice instrucțiune de tip SET cu excepția: **SET SHOWPLAN\_TEXT** și **SET SHOWPLAN\_ALL**; opțiunile SET acceptate rămân în vigoare până la ieșirea din procedură;
- dacă procedura stocată urmează să fie folosită și de alți utilizatori decât cel care creează procedura, atunci numele de obiecte utilizate în următoarele instrucțiuni:

- **ALTER TABLE;**
- **CREATE INDEX;**
- **CREATE TABLE;**
- toate instrucțiunile DBCC;
- **DROP TABLE;**
- **DROP INDEX;**
- **TRUNCATE TABLE;**
- **UPDATE STATISTICS**

trebuie prefixate prin numele proprietarului obiectului respectiv.

- următoarele instrucțiuni de tip CREATE nu pot să apară într-o procedură stocată:
  - **CREATE DEFAULT;**
  - **CREATE PROCEDURE;**
  - **CREATE RULE;**
  - **CREATE TRIGGER;**
  - **CREATE VIEW.**

pentru obiectele care se pot crea într-o procedură stocată trebuie respectată condiția ca acestea să fie referite doar după crearea lor;

- o procedură stocată poate accesa tabele temporare, dar trebuie avut în vedere faptul că o tabelă temporară creată într-o procedură dispără la ieșirea din procedură;
- procedură apelată de o altă procedură are acces la toate obiectele create de procedura apelantă (inclusiv tabele temporare!);

## 11.2. Instrucția EXECUTE

În urma executării cu succes a instrucțiunii **CREATE PROCEDURE** numele și textul procedurii stocate corespunzătoare sunt înregistrate în tabelele directoriale ale SQL Server. La primul apel de execuție a procedurii are loc compilarea acesteia și construirea unui plan de execuție optimizat. Lansarea în execuție a unei proceduri stocate se face cu comanda **EXECUTE**.

Sintaxa:

```
[[EXECUTE]
{
    [ @valoare_retur = ] { nume_procedura [;numar] } @variabila_nume_procedura
    [[@parametru =] {valoare | @variabila [OUTPUT] | [DEFAULT] } [,n]
    [WITH RECOMPILE]
```

unde:

**@valoare\_retur** - este o variabilă optională de tip întreg în care se memorează valoarea de return a procedurii stocate.

**nume\_procedura** - numele procedurii apelate. Dacă se apelează o procedură dintr-o altă bază de date (și, eventual, alt server) decât cea curentă, atunci numele procedurii trebuie prefixat cu numele bazei de date (și al serverului) în care se găsește și unde se execută. Numele unei proceduri stocate este case-sensitive!

**;numar** - are aceeași semnificație ca și în cazul comenzii **CREATE PROCEDURE**.

**@variabila\_nume\_procedura** - este numele unei variabile ce conține numele unei proceduri stocate.

**@parametru** - este numele unui parametru al procedurii aşa cum a fost definit prin instrucția CREATE PROCEDURE. În varianta de apel de forma **@parametru = valoare**, perechile de forma nume-constantă pot să apară și într-o ordine diferită față de cea dată la crearea procedurii. Totuși, dacă se folosește forma **@parametru = valoare** pentru un parametru, atunci ea trebuie folosită pentru toți parametrii procedurii.

**valoare** - este valoarea parametrului. Dacă nu se specifică numele parametrului corespunzător, atunci valorile trebuie date în ordinea definirii parametrilor corespunzători.

**@variabila** - este numele unei variabile în care se va returna valoarea unui parametru de tip OUTPUT.

**OUTPUT** - indică faptul că parametrul este de tip ieșire. Un asemenea parametru trebuie asociat la apel cu o variabilă de tipul valorii săd de tipuri @parametru = **@variabila sau simplu @variabila**. Variabila trebuie să fie asociată unui parametru de tip OUTPUT poate fi inițializată cu o valoare care este accesibilă în interiorul procedurii, ceea ce înseamnă că parametrii de tip OUTPUT pot foarte bine să funcționeze și ca parametrii de intrare. Dacă la apel se omite cuvântul cheie OUTPUT pentru un parametru declarat la creare ca fiind de ieșire, atunci acesta nu va putea fi folosit pentru a transmite valori dinspre procedura apelată către apelant, desigur că nu poate funcționa ca parametru de intrare. Dacă se associază la apel opțiunea OUTPUT unui parametru care nu a fost declarat cu OUTPUT la crearea procedurii, atunci se generează o eroare.

**DEFAULT** - indică faptul că se va lăsa în considerare valoarea implicită a parametrului aşa cum a fost definită la crearea procedurii (în forma, **@parametru = DEFAULT sau simplu DEFAULT**). Dacă parametrul nu are definită o valoare implicită, atunci se va genera o eroare.

**WITH RECOMPILE** - forțează recomplirea procedurii și crearea

unei nou plan de execuție. Se recomandă folosirea acestei opțiuni dacă au avut loc modificări semnificative ale datelor față de momentul creerii procedurii. În acest caz, este, de asemenea, utilă execuțarea unei comenzi UPDATE STATISTICS îmânte de apelul (cu recomplirea) a procedurii.

### 11.3. Instrucția ALTER PROCEDURE

Modifică o procedură stocată creată anterior fără a afecta în vreun fel obiectele dependente (proceduri stocate sau trigger care apelează procedura). Modificarea prin instrucția ALTER PROCEDURE este preferată variantei care presupune stegerea și recrearea procedurii deoarece, în acest caz, procedurile sau triggerele dependente nu mai trebuie recompileate.

Sintaxa:

```
ALTER PROC[EDURE] nume_procedura [;numar]
[ { @parametru tip_de_data } [VARYING] [= valoare_implicită]
  [OUTPUT]
```

```
[@parametru] [,n]
[WITH
  {RECOMPILE
  |ENCRYPTION
  |RECOMPILE, ENCRYPTION}
  ]
[FOR REPLICATION]
AS
  instructioni_sql[,n]
```

unde toate elemente de sintaxă au aceeași semnificație ca și în cazul instrucției CREATE PROCEDURE.

### 11.4. Instrucția DROP PROCEDURE

Sterge una sau mai multe proceduri sau grupuri de proceduri din baza de date curentă. Nu se pot sterge doar anumite proceduri dintr-un grup, ci doar grupuri întregi de proceduri care au același nume. Stergerea unei proceduri înseamnă eliminarea definiției sale din baza de date.

Sintaxa:

```
DROP PROC[EDURE] nume_procedura [,n]
```

unde:

**nume\_procedura** - este numele procedurii sau al grupului de proceduri care se stergă.

## 11.5. Exemple de utilizare a procedurilor stocate

### Exemplu 1:

Procedura stocată definită mai jos conține o singură frază SELECT care returnează date despre tranzacțiile existente:

```
IF EXISTS (SELECT name FROM sysobjects  
          WHERE name = 'Tranzactii_Desfasurat' AND type = 'P')  
DROP PROCEDURE Tranzactii_Desfasurat  
GO  
CREATE PROCEDURE Tranzactii_Desfasurat  
AS  
SELECT Furnizor.Nume, Beneficiar.Nume,  
       Produs.Nume, Cantitate.Pret  
  FROM Furnizor, Beneficiar, Produs, Tranzactii  
 WHERE Furnizor.CodF=Tranzactii.CodF  
   AND Beneficiar.CodB=Tranzactii.CodB  
   AND Produs.CodP=Tranzactii.CodP  
   AND Furnizor.Oras=@Oras_Furnizor  
   AND Beneficiar.Oras=@Oras_Beneficiar  
GO
```

### Observație:

Numele unei proceduri stocate este înregistrat în tabela director sysobjects ca obiect având tipul *P*. Prin secvența care precede crearea procedurii stocate se testează dacă nu cunosc există deja o procedură cu același nume și în caz afirmativ aceasta se distruge. SQL Server face distincția dintre două proceduri stocate exclusiv prin nume (care include și numărul procedurii în cadrul unui grup), motiv pentru care în aceeași bază de date nu sunt acceptate două proceduri cu același nume.

Lansarea în execuție a procedurii *Tranzactii\_Desfasurat*, creată mai sus, se poate face cu instrucțiunea EXECUTE urmată de numele procedurii:

```
EXECUTE Tranzactii_Desfasurat
```

### Exemplu 2:

Să înlocuim procedura de mai sus printr-o variantă parametrizată care returnează doar tranzacțiile dintre furnizori și beneficiari din două orașe date ca parametru:

```
IF EXISTS (SELECT name FROM sysobjects  
          WHERE name = 'Tranzactii_Desfasurat' AND type = 'P')  
DROP PROCEDURE Tranzactii_Desfasurat  
GO  
CREATE PROCEDURE Tranzactii_Desfasurat  
AS  
SELECT Furnizor.Nume, Beneficiar.Nume,  
       Produs.Nume, Cantitate.Pret  
  FROM Furnizor, Beneficiar, Produs, Tranzactii  
 WHERE Furnizor.CodF=Tranzactii.CodF  
   AND Beneficiar.CodB=Tranzactii.CodB  
   AND Produs.CodP=Tranzactii.CodP  
   AND Furnizor.Oras LIKE @Oras_Furnizor  
   AND Beneficiar.Oras LIKE @Oras_Beneficiar  
GO
```

În acest caz apelurile:

```
EXECUTE Tranzactii_Desfasurat 'Cluj', 'Bucuresti'  
EXECUTE Tranzactii_Desfasurat 'Alba', 'Bucuresti'
```

au același efect ca la exemplul 2), în timp ce apelul:

```
SELECT Furnizor.Nume, Beneficiar.Nume,  
       Produs.Nume, Cantitate.Pret  
  FROM Furnizor, Beneficiar, Produs, Tranzactii  
 WHERE Furnizor.CodF=Tranzactii.CodF  
   AND Beneficiar.CodB=Tranzactii.CodB  
   AND Produs.CodP=Tranzactii.CodP  
   AND Furnizor.Oras=@Oras_Furnizor  
   AND Beneficiar.Oras=@Oras_Beneficiar
```

Tranzacțiile furnizorilor din Cluj cu beneficiari din București se obțin prin oricare din apelurile:

```
EXECUTE Tranzactii_Desfasurat 'Cluj', 'Bucuresti'  
EXECUTE Tranzactii_Desfasurat  
      @Oras_Furnizor='Cluj', @Oras_Beneficiar='Bucuresti'  
EXECUTE Tranzactii_Desfasurat  
      @Oras_Beneficiar='Bucuresti', @Oras_Furnizor='Cluj'  
iar a furnizorilor din Alba cu beneficiari din București se obține prin:  
EXECUTE Tranzactii_Desfasurat 'Alba', 'Bucuresti'
```

sau orice alt apel echivalent.

### Exemplu 3:

Varianta parametrizată a procedurii se poate adapta să returneze toate tranzacțiile atunci când apelul se face fără parametri, păstrând nemodificat comportamentul în celelalte cazuri:

```
IF EXISTS (SELECT name FROM sysobjects  
          WHERE name = 'Tranzactii_Desfasurat' AND type = 'P')  
DROP PROCEDURE Tranzactii_Desfasurat  
GO  
CREATE PROCEDURE Tranzactii_Desfasurat(  
      @Oras_Furnizor varchar(15)='%',  
      @Oras_Beneficiar varchar(15)='%')
```

### Exemplu 2:

Să înlocuim procedura de mai sus printr-o varianta parametrizată care returnează doar tranzacțiile dintre furnizori și beneficiari din două orașe date ca parametru:

```
IF EXISTS (SELECT name FROM sysobjects  
          WHERE name = 'Tranzactii_Desfasurat' AND type = 'P')  
DROP PROCEDURE Tranzactii_Desfasurat  
GO  
CREATE PROCEDURE Tranzactii_Desfasurat  
AS  
SELECT Furnizor.Nume, Beneficiar.Nume,  
       Produs.Nume, Cantitate.Pret  
  FROM Furnizor, Beneficiar, Produs, Tranzactii  
 WHERE Furnizor.CodF=Tranzactii.CodF  
   AND Beneficiar.CodB=Tranzactii.CodB  
   AND Produs.CodP=Tranzactii.CodP  
   AND Furnizor.Oras LIKE @Oras_Furnizor  
   AND Beneficiar.Oras LIKE @Oras_Beneficiar  
GO
```

EXECUTE Tranzactii\_Desfasurat

returnează toate tranzacțiile deoarece expresiile:

Furnizor.Oras LIKE '%'

și

Beneficiar.Oras LIKE '%'

au valoarea TRUE indiferent de valoarea atributelor oraș.

**Exemplu 4:** Varianta următoare a procedurii returnează, prin parametrul @suma de tip OUTPUT, valoarea totală a tranzacțiilor din relația rezultat. Procedura returnează, de asemenea, o valoare care indică starea prelucrării: 0, dacă există cel puțin o tranzacție în rezultat și parametrul suma are o valoare validă, respectiv 1 în caz contrar.

```
IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'Tranzactii_Desfasurat' AND type = 'P')
  GO
CREATE PROCEDURE Tranzactii_Desfasurat(
           @suma money
           @Oras_Furnizor varchar(15)='%', --@Oras_Beneficiar varchar(15)='%', --@total money OUTPUT)
AS
IF EXISTS (SELECT *
           FROM Furnizor, Beneficiar, Tranzactii
           WHERE Furnizor.CodF=Tranzactii.CodF
                 AND Beneficiar.CodB=Tranzactii.CodB
                 AND Furnizor.Oras LIKE @Oras_Furnizor
                 AND Beneficiar.Oras LIKE @Oras_Beneficiar)
BEGIN
  SET @total=(SELECT SUM(Cantitate*Pret)
              FROM Furnizor, Beneficiar, Tranzactii
              WHERE Furnizor.CodF=Tranzactii.CodF
                    AND Beneficiar.CodB=Tranzactii.CodB
                    AND Furnizor.Oras LIKE @Oras_Furnizor
                    AND Beneficiar.Oras LIKE @Oras_Beneficiar)
SELECT Furnizor.Nume, Beneficiar.Nume,
      Produs.Nume, Cantitate.Pret
     FROM Furnizor, Beneficiar, Produs, Tranzactii
    WHERE Furnizor.CodF=Tranzactii.CodF
          AND Beneficiar.CodB=Tranzactii.CodB
          AND Produs.CodP=Tranzactii.CodP
          AND Furnizor.Oras LIKE @Oras_Furnizor
          AND Beneficiar.Oras LIKE @Oras_Beneficiar
  RETURN 0
END
ELSE
```

RETURN 1

GO

Secvența de cod care urmează apelează de două ori procedura *Tranzactii\_Desfasurat*: odată pentru toate tranzacțiile și a doua oară pentru tranzacțiile dintre furnizori din Cluj și beneficiari din București. Funcție de valoarea variabilei de return @status se afișează un mesaj adekvat.

```
DECLARE @suma money
DECLARE @suma_Cluj_Bucuresti money
DECLARE @status int
EXECUTE @status=
        Tranzactii_Desfasurat DEFAULT, DEFAULT, @suma OUTPUT
IF @status=0
  PRINT 'Suma totală a tranzacțiilor este: '
        +convert(varchar(30), @suma)
ELSE
  PRINT 'Nu sunt tranzacții!'
EXECUTE @status=
        Tranzactii_Desfasurat 'cluj', 'Bucuresti', @suma_Cluj_Bucuresti OUTPUT
IF @status=0
  PRINT 'Suma totală a tranzacțiilor efectuate între'
        , 'Cluj și Bucuresti este:'
        +convert(varchar(30), @suma_Cluj_Bucuresti)
ELSE
  PRINT 'Nu sunt tranzacții între furnizori din'
        , 'Cluj și beneficiari din Bucuresti!'
```

#### Observație:

1. La primul apel al procedurii:

```
EXECUTE @status=
        Tranzactii_Desfasurat DEFAULT, DEFAULT, @suma OUTPUT
este necesară precizarea cuvântului cheie DEFAULT pentru primii doi parametrii, nefind acceptată o variantă de formă:
```

```
EXECUTE @status=Tranzactii_Desfasurat, @suma OUTPUT
```

O altă variantă de apel, acceptată în acest caz este:

```
EXECUTE @status=Tranzactii_Desfasurat
        @total=@suma OUTPUT
2. De remarcat că atât parametrul de tip OUTPUT, cât și valoarea de return pot fi complet ignorate la apelul procedurii Tranzactii_Desfasurat. Astfel apelurile:
```

```
EXECUTE @status=Tranzactii_Desfasurat  
EXECUTE Tranzactii_Desfasurat
```

sunt valide și întorc ca rezultat lista tuturor tranzacțiilor (în plus, primul apel returnează și starea prelucrării presupunând că variabila @status a fost corect declarată anterior).

## 11.6. Exerciții și probleme

1.
  - a. Să se scrie o procedură stocată care are ca parametru un nume de produs și produce ca rezultat lista ofertelor pentru acel produs.
  - b. Să se adauge la procedura de mai sus un parametru de tip OUTPUT prin care se va returna media valorilor ofertelor pentru produsul dat.
2. Să se scrie o procedură stocată având ca parametrii numele unui produs, un preț minim și un preț maxim. Procedura va șterge acele oferte pentru produsul dat care nu se încadrează între prețul minim și prețul maxim. Daca la apel nu se precizează numele produsului, atunci se vor lua în considerare pentru ștergere ofertele pentru toate produsele. Dacă lipsește unul dintre limitele de preț ori ambele, atunci se va ține cont doar de limitele rămase. Apelul fără parametrii va șterge toate ofertele! Procedura va întoarce ca valoare de return numărul de oferte care au fost șterse.

### 3. Fie procedura:

```
CREATE PROCEDURE P (@x int=0, @y int OUTPUT, @z int=0) AS  
RETURN ISNULL(@x=@z, 111)
```

și declaratiile de variabile:

```
DECLARE @a int  
DECLARE @b int  
DECLARE @c int  
DECLARE @retur int
```

Să se arate care dintre următoarele apeluri este corect și care nu.

- a. EXECUTE P @y=@a
- b. EXECUTE P DEFAULT, @a, DEFAULT
- c. EXECUTE P DEFAULT, @y=@a, DEFAULT

d. EXECUTE @retur=P DEFAULT, @y=@a, DEFAULT

e. EXECUTE P

f. EXECUTE @retur=P

g. EXECUTE @retur=P , @b

h. EXECUTE @retur=P NULL, @b, NULL

i. EXECUTE @retur=P NULL, NULL,NULL

j. EXECUTE @retur=P @a, @b, @c

k. EXECUTE @retur=P @a+1, @b, @c

l. EXECUTE @retur=P @a, @b+1, @c

4. Pentru procedura de la problema 3) să se arate care este valoarea de return și valoarea parametrului OUTPUT în cazul următoarelor apeluri:
  - a. EXECUTE @retur=P 1, @b, 1
  - b. EXECUTE @retur=P 1, @b OUTPUT
  - c. EXECUTE @retur=P @y=@b OUTPUT
  - d. EXECUTE @retur=P 1, @b OUTPUT, NULL
  - e. EXECUTE @retur=P NULL, @a OUTPUT, NULL
  - f. EXECUTE @retur=P @a, @b, @c
  - g. EXECUTE @retur=P @a, @a, @a
  - h. EXECUTE @retur=P @a, @a OUTPUT, @a

## 12. CURSOARE SQL SERVER

### 12.1. Instrucțiunea DECLARE CURSOR

Specificul bazelor de date relationale, și al limbajului SQL în particular, este modul de operate asupra relațiilor ca un tot unitar. Orice operație, se face asupra setului complet de tuple care satisfac anumite condiții. Cursearele constituie un mod complementar de lucru prin care se permite accesul la tuple, una către una, și prelucrarea independentă a fiecărei tuple în parte. Prin introducerea cursearelor se aduce o extensie utilă limbajului SQL care este astfel întregit cu toate facilitățile specifice limbajelor navigaționale:

- poziționare pe o anumită tuplă;

- modificări ale tuplei de la poziția curentă;

- deplasare cursor înainte-înapoi;

- prelucrarea subsetului de tuple începând de la poziția curentă și.a.m.d.

De remarcat că nu este utilă folosirea cursearelor în operațiile obișnuite care se pot rezolva prin fraze SQL. Cursearele au fost introduse ca o extensie și nu ca alternativă la frazele SELECT, UPDATE și.a.m.d. care sunt, în general, mult mai rapide.

O prelucrare bazată pe cursoare se desfășoară în mai multe faze după cum urmează:

1. Declararea unui cursor - unui nume de cursor i se asociază setul de tuple rezultat corespunzător unei fraze SELECT; de asemenea se specifică o serie de caracteristici ale cursorului.
2. Deschiderea cursorului - se execută fraza SELECT asociată, realizându-se ceea ce se numește *popularea cursorului*.
3. Încărcarea cursorului - se poziționează cursorul în dreptul unei tuple și se realizează accesul la conținutul acestora.
4. Prelucrare - se execută operațiile specifice aplicației.
5. Închiderea cursorului - se șterge setul de tuple cu care s-a populat cursorul, dar se menține cursorul în sine împreună cu definiția sa.
6. Dealocarea cursorului - se șterge cursorul împreună cu definiția sa.

Defineste attributele unui cursor și setul de tuple cu care se populează. În prezent instrucțiunea DECLARE CURSOR din SQL Server acceptă atât sintaxa bazată pe standardul SQL'92, cât și extinsă cu o serie de facilități specifice sistemului SQL Server. În cele de mai jos ne vom limita la sintaxa SQL'92.

#### Sintaxa SQL'92:

```
DECLARE nume_cursor [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR fraza_select
    [ FOR { READ ONLY | UPDATE [ OF nume_coloană [, n] ] } ]
    unde:
```

*nume\_cursor* – este numele cursorului;

#### INSENSITIVE

– definește un cursor care creează o copie temporară a setului de tuple corespunzător. Modificările ulterioare asupra datelor originale nu sunt reflectate în datele din cursor. Sunt interzise operațiile de actualizare. Dacă este omisă opțiunea INSENSITIVE, atunci modificările din tabelele de bază se vor reflecta la nivelul cursorului.

#### SCROLL

– indică faptul că toate opțiunile de navigare-accesare date (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) sunt permise. În lipsa acestei opțiuni, NEXT este singura opțiune de navigare permisă.

#### *fraza\_select*

– definește setul de tuple corespunzător cursorului;

#### READ ONLY

– interzice operațiile de actualizare date la nivelul cursorului;

UPDATE [ OF *nume\_coloană* [, *n*] ] – indică coloanele care pot fi modificate în cursor. Numai coloanele specificate în lista OF *nume\_coloană* [, *n*] pot fi modificate. Dacă aceasta listă lipsește, atunci, implicit, toate coloanele sunt modifiable.

### 12.2. Instrucțiunea OPEN

Deschide un cursor și populează cursorul prin executarea frazei SELECT specificată în declarația cursorului.

Sintaxă:

**OPEN** { { *nume\_cursor* } | *nume\_variabilă\_cursor* }

unde:

- *nume\_cursor* - este numele unui cursor declarat anterior;

- *nume\_variabilă\_cursor* - este numele unei variabile de tip cursor care referă cursorul.

Observație:

După deschiderea unui cursor variabilă sistem @@CURSOR\_ROWS conține numărul de tuple încărcate de ultima operație OPEN.

## 12.3. Instrucțiunea **FETCH**

Accesează o tuplă dintr-un cursor și încarcă conținutul acesteia într-un set de variabile.

Sintaxă:

```
FETCH [ [NEXT | PRIOR | FIRST | LAST  
| ABSOLUTE { n | @nvar }  
| RELATIVE { n | @nvar } ]  
FROM  
[ { { nume_cursor } | nume_variabilă_cursor }  
[ INTO @nume_variabilă [, n ] ] ]
```

unde:

**NEXT** - mută poziția curentă la tupla următoare și încarcă conținutul acesteia. Dacă FETCH NEXT este prima operație de încărcare a unui cursor, atunci va încarcă prima tuplă a cursorului. **NEXT** este opțiunea implicită de încărcare.

**PRIOR** - mută poziția curentă la tupla precedentă și încarcă conținutul acesteia. Dacă FETCH PRIOR este prima operație de încărcare a unui cursor, atunci nu se încarcă nimic și cursorul rămâne pozitionat pe prima tuplă.

**FIRST** - mută poziția curentă la prima tuplă și încarcă conținutul acesteia.

**LAST** - mută poziția curentă la ultima tuplă și încarcă conținutul acesteia.

**ABSOLUTE** { *n* | @*nvar* } - pozitionare absolută pe tupla din poziția *n* sau @*nvar* de la început dacă *n* sau @*nvar* este pozitiv. Dacă *n* sau @*nvar* este negativ pozitionarea se face față de sfârșitul cursorului (valoarea -1 pozitionează pe ultima tuplă din cursor!).

Dacă *n* sau @*nvar* este 0 nu se încarcă nimic.

**RELATIVE** { *n* | @*nvar* } - pozitionare relativă față de tupla curentă, valoarea *n* sau @*nvar* având valoarea 0 se încarcă tupla curentă, valoarea 1 corespunde opțiunii **NEXT**, iar -1 corespunde opțiunii **PRIOR**.

*nume\_cursor* - este numele unui cursor declarat anterior.

*nume\_variabilă\_cursor* - este numele unei variabile de tip cursor care va referi cursorul.

**INTO** @*nume\_variabilă* [ , *n* ] - permite încărcarea conținutului tupletelor curente într-un set de variabile locale. Fiecare variabilă din listă este asociată, în ordine, către unui atribut din relația cu care s-a încărcat cursorul. Corespondența trebuie să fie unu la unu ca număr și ca tip de date până la nivelul conversiilor implicate suportate de SQL Server.

Observație:

Variabila de sistem @@FETCH\_STATUS indică modul în care s-a desfășurat ultima operație FETCH executată.

## 12.4. Variabilă sistem @@FETCH\_STATUS

Returnează stareea ultimei operații FETCH executată asupra unui cursor. Valorile posibile ale variabilei @@FETCH\_STATUS și semnificația acestora este dată în următorul tabel:

| @@FETCH_STATUS | Descriere                                                         |
|----------------|-------------------------------------------------------------------|
| 0              | Înstrucțiunea FETCH s-a executat cu succes.                       |
| -1             | Înstrucțiunea FETCH a eșuat sau cursor în afara setului rezultat. |
| -2             | Lipsă tuplă încărcată.                                            |

Observații:

- Deoarece @@FETCH\_STATUS este o variabilă globală testarea sa trebuie făcută imediat după operația a cărei stare vrem să o aflăm,

înaintea oricărei alte operații FETCH care modifică la rândul ei variabila @@FETCH\_STATUS.

2. Variabila @@FETCH\_STATUS poate avea valoarea -2 în cazul unui cursor fără opțiunea INSENSITIVE, dacă între timp un utilizator concurent a șters tupla pe care s-a încercat poziționarea prin ultima operație FETCH.

## 12.5. Instrucția CLOSE

Închide un cursor deschis și eliberează setul rezultat asociat. Un cursor închis poate fi redeschis din nou printr-o instrucție OPEN care calculează o nouă valoare actualizată a setului rezultat. Instrucția CLOSE se poate executa numai asupra unui cursor deschis.

Sintaxa:

```
CLOSE { { nume_cursor } | nume_variabilă_cursor }
```

unde:

nume\_cursor - este numele unui cursor declarat anterior;

nume\_variabilă\_cursor - este numele unei variabile de tip cursor care referă cursorul.

## 12.6. Instrucția DEALLOCATE

Sterge o referință la un cursor. La stergerea ultimei referințe către un cursor toate structurile de date și resursele asociate cursorului vor fi eliberate.

Sintaxa:

```
DEALLOCATE { { nume_cursor } | nume_variabilă_cursor }
```

unde:

nume\_cursor - este numele unui cursor declarat anterior;

nume\_variabilă\_cursor - este numele unei variabile de tip cursor care referă cursorul.

### Observații:

1. Instrucția DEALLOCATE elimină asocierea dintre un cursor și numele său sau variabila cursor care referă cursorul. Dacă numele

cursor sau variabila este singura care referă cursorul, atunci cursorul este dealocat și toate resursele folosite de acesta sunt eliberate.

2. O variabilă cursor poate fi asociată cu un cursor în două moduri:

a. Prin atribuirea unui nume cursor unei variabile cursor. De exemplu:

```
DECLARE @cursor_furnizor CURSOR  
DECLARE cursor_furnizor CURSOR FOR  
    SELECT * FROM Furnizor
```

- b. Prin asocierea unei definiții de cursor direct unei variabile cursor. De exemplu:

```
DECLARE @cursor_furnizor CURSOR  
SET @cursor_furnizor= CURSOR FOR  
    SELECT * FROM Furnizor
```

## 12.7. Exemple de utilizare a cursorilor

1. Secvența de mai jos folosește un cursor pentru a accesa, pe rând, fiecare furnizor cu scopul de a trimite acestora câte un email:

```
DECLARE @Nume varchar (30)  
DECLARE @Oras varchar (30)  
DECLARE cursor_furnizor CURSOR FOR  
OPEN cursor_furnizor  
    SELECT Nume, Oras FROM Furnizor  
    FETCH NEXT FROM cursor_furnizor INTO @Nume, @Oras  
    WHILE @@FETCH_STATUS=0  
        BEGIN  
            EXEC Send_email @Nume, @Oras  
                --procedura stocata care realizeaza  
                --trimierea unui email  
        END  
    CLOSE cursor_furnizor  
    DEALLOCATE cursor_furnizor  
GO
```

2. Următoarea secvență afișează numele furnizorilor în ordinea inversă a apariției lor fizice în tabela furnizor. De remarcat că setul rezultat

asociat cursorului este parcurs de la coadă la cap, ceea ce impune folosirea unui cursor cu opțiunea **SCROLL**.

```
DECLARE @Nume varchar(30)
DECLARE cursor_furnizor CURSOR SCROLL FOR
OPEN cursor_furnizor
  SELECT Nume FROM Furnizor
  FETCH LAST FROM cursor_furnizor INTO @Nume
  WHILE @@FETCH_STATUS=0
    BEGIN
      PRINT 'Numele furnizorului este: '+@Nume
      FETCH PRIOR FROM cursor_furnizor INTO @Nume
    CLOSE cursor_furnizor
    DEALLOCATE cursor_furnizor
  GO
```

3.

O operație implementată de sevența de cod de la exemplul 1 ar putea fi una de lungă durată, mai ales dacă numărul furnizorilor este mare. Între timp ar putea să apară noi furnizori și se dorește ca și aceștia să primească mesajele cuvenite, dar fără a mai retrimit mesaje celor care deja le-au primit. Procedura de mai jos folosește pozitionarea absolută a cursorului cursor\_furnizor pentru a începe trimiterea de mesaje începând de la furnizorul indicat prin parametrul *@primul*. Acest parametru este folosit și pentru a determina câți dintre furnizori au primit mesaje și este întors ca parametru de ieșire la terminarea procedurii.

```
CREATE PROCEDURE Mesaje (@primul int output) AS
DECLARE @Nume varchar(30)
DECLARE cursor_furnizor CURSOR SCROLL FOR
OPEN cursor_furnizor
  SELECT Nume,Oras FROM Furnizor
  FETCH ABSOLUTE @primul FROM cursor_furnizor
  WHILE @@FETCH_STATUS=0
    BEGIN
      EXEC Send_email @Nume, @Oras
        --procedura stocata care realizeaza
        --trimierea unui email
      SET @primul=@primul+1
      FETCH NEXT FROM cursor_furnizor INTO @Nume, @Oras
    END
  CLOSE cursor_furnizor
  DEALLOCATE cursor_furnizor
GO
```

**Observație:**

Procedura de mai sus ar poate fi lansată și menită permanent în execuție print-o secvență de forma:

```
DECLARE @dela int
SET @dela=1
WHILE 1=1
  BEGIN
    EXEC Mesaj @dela OUTPUT
  END
```

În acest mod, cel puțin în principiu, orice nou furnizor poate primi mesajul său odată cu înregistrarea sa în tabela *Furnizor* (deși nu este o modalitate prea eficientă de rezolvare a acestei probleme!).

4.

Procedura de mai jos trimite un email de atenționare fiecărui furnizor pentru fiecare produs pe care acesta nu îl oferă.

```
CREATE PROCEDURE AtentieProdus AS
DECLARE @codP int
DECLARE @codP int
DECLARE @Nume varchar(30)
DECLARE @Oras varchar(30)
DECLARE furnizor CURSOR FOR
  SELECT * FROM Furnizor
DECLARE produs SCROLL CURSOR FOR
  SELECT CodP FROM Produs
OPEN furnizor
  SELECT CodP FROM Produs
OPEN produs
  FETCH FROM furnizor INTO @codF, @Nume, @Oras
  WHILE @@FETCH_STATUS = 0
    BEGIN
      FETCH FIRST FROM produs INTO @codP
      WHILE @@FETCH_STATUS = 0
        BEGIN
          IF NOT EXISTS (SELECT * FROM Oferte
            WHERE CodP=@codF AND CodP=@codP)
            EXEC Send_email @Nume, @Oras
          FETCH NEXT FROM produs INTO @codP
        END
      END
      FETCH FROM furnizor INTO @codF, @Nume, @Oras
    END
  CLOSE furnizor
  CLOSE produs
  DEALLOCATE furnizor
  DEALLOCATE produs
GO
```

**Observație:**

Procedura *Atenționare* ia în considerare fiecare pereche posibilă de furnizor-produs. Pentru acesta cursorul *produs* trebuie readus la prima poziție pentru fiecare furnizor, ceea ce impune ca acesta să fie declarat cu opțiunea **SCROLL**.

5.

Procedura de mai jos realizează popularea cu date de test a oricărăia dintre tabelele *Oferte* ori *Cereri* în funcție de valoarea parametrului *@Type*. Popularea trebuie să țină cont de contrângările de cheie străină existente între tabela *Oferte* și *Furnizor*, respectiv *Produs*. Se generează câte o ofertă (cerere) pentru fiecare pereche posibilă de furnizor (beneficiar)-produs, folosindu-se aceleși valori pentru atributele cantitate și preț. De remarcat că procedura poate fi ușor extinsă pentru a implementa scenarii mai realiste de populare a tabelelor (de exemplu prin prelucrarea interactivă a valorilor pentru cantitate și preț pentru fiecare ofertă în parte!).

```

CREATE PROCEDURE Populare_Oferete_Cereri(
    @Type as varchar(15),
    @Cant int,
    @Pret money) AS
DECLARE @coda int
DECLARE @codp int
IF @Type='Oferte'
    DECLARE agent CURSOR FOR
        SELECT CodF FROM Furnizor
ELSE IF @Type='Cereri'
    DECLARE agent CURSOR FOR
        SELECT CodB FROM Beneficiar
ELSE
    BEGIN
        PRINT 'Tip agent ilegal!'
        RETURN
    END
DECLARE produs SCROLL CURSOR FOR
    SELECT CodP FROM Produs
OPEN agent
OPEN produs
FETCH FROM agent INTO @coda
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH FIRST FROM produs INTO @codp
    WHILE @@FETCH_STATUS = 0
        BEGIN
            IF @Type='Oferte'
                INSERT oferte VALUES (@coda, @codp, @cant, @Pret)
            ELSE
                INSERT Cereri VALUES (@coda, @codp, @cant, @Pret)
        END
        FETCH NEXT FROM produs INTO @codp
    END
    FETCH FROM agent INTO @coda
END
CLOSE agent
CLOSE produs
DEALLOCATE agent
DEALLOCATE produs
GO

```

## 12.8. Exerciții și probleme

1.

Cum ar trebui modificată procedura *Mesaje* de la exemplul 3 pentru a răspunde situațiilor în care are loc o cădere a sistemului în timpul execuției procedurii? Se dorește ca trimitera mesajelor să poată fi reluată începând cu primul furnizor care nu a primit un mesaj.

2.

Să se modifice procedura *Populare\_Oferete\_Cereri* de la exemplul 5 astfel încât să nu se folosească opțiunea *SCROLL* pentru cursorul produs.

3.

Să se scrie o procedură pentru popularea tablei *Tranzactii*.

4.

Să se scrie o procedură și o secvență de apel prin care să se realizeze monitorizarea tranzacțiilor din tabela *Tranzactii* (se va afișa câte un mesaj la înregistrarea unei noi tranzacții).

5.

Fie o procedură definită astfel:

```

CREATE PROCEDURE Test AS
    DECLARE cursor_furnizor CURSOR FOR
        SELECT * FROM Furnizor
    OPEN cursor_furnizor
    FETCH FROM cursor_furnizor
    GO

```

și următarea secvență de instrucțuni:

```

EXEC Test
    FETCH FROM cursor_furnizor
    FETCH FROM cursor_furnizor
    ...

```

Explicați ce se întâmplă la execuția secvenței de instrucțuni de mai sus.

# 13. TRIGGERE

Triggerurile sunt o clasă specială de proceduri stocate, asociate unei tabele, definite pentru a fi lansate în execuție automat la inițierea unei operații de tip UPDATE, INSERT sau DELETE asupra tabelei în cauză. Triggerurile sunt un instrument puternic pentru implementarea a ceea ce în aplicațiile de baze de date poartă numele de **business rules**. Termenul se referă la acele reguli, în general constrângeri, care jin în mod inherent de structura bazei de date și sunt induse de semantică unei colecții de date, fiind comune tuturor aplicațiilor care folosesc aceea bază de date și relativ independente de fiecare aplicație în parte. Triggerurile extind posibilitățile altor instrumente de verificare a integrității din SQL Server cum ar fi constrainti, valori implicate și reguli. Totuși, aceste instrumente declarative vor fi preferate în locul triggerelor ori de câte ori este posibil.

Exemple tipice de utilizare a triggerelor sunt:

- Cascadarea operațiilor de modificare de la tabela curentă la alte tabele din baza de date. De exemplu, ștergerea unui furnizor implică ștergerea ofertelor sale din tabela *Oferte*.
- Anularea modificărilor care ar duce la violarea integrității referențiale. Se pot folosi triggerurile ca mijloace alternative, eventual mai sofisticate, la instrumentele de tip declarativ cum ar fi declarațiile de cheie strânsă.
- Exprimarea unor constrângeri mai complexe care nu pot fi exprimate sub forma unor constrângeri de tip CHECK. De exemplu, pentru a garanta unicitatea numelui furnizor în tabela Furnizor nu se poate folosi o simplă declarație de constrângere de tip CHECK. (în acest caz s-ar putea folosi un index de tip UNIQUE, dar acesta nu rezolvă problema unor mesaje specializate care să ofere detalii despre natura erorii care a apărut.) Alt exemplu de constrângere complexă, implementabilă prin triggere ar putea fi aceea ca în fiecare oraș să existe cel mult 10 furnizori.

Un trigger este inițiat ori de căte ori se încarcă operația de modificare corespunzătoare asupra tabelei căreia îl este atașat. Un trigger poate conține instrucțiuni SQL complexe și poate accesa datele din alte tabele. Triggerul și operația care îl declanșeză sunt considerate ca un tot unitar (tranzacție) ceea ce înseamnă că dacă execuția triggerului eșuează, dintr-un motiv sau altul, atunci și operația care a declanșat triggerul se anulează. O tabelă poate avea asociate mai multe trigger. Instrucțiunea CREATE TRIGGER poate fi definită cu

oricare dintre clauzele FOR UPDATE, FOR INSERT sau FOR DELETE

pentru a crea triggere specializate pentru fiecare tip de modificare a unei tabele. Dacă se folosește clauza FOR UPDATE, atunci se poate de asemenea folosi clauza IF UPDATE (*nume\_coloana*) prin care se poate crea un trigger specializat pentru situația modificării unei coloane anume. Clauza IF UPDATE (*nume\_coloana*) este de fapt un test care returnează valoarea logică TRUE dacă coloana dată ca parametru a fost modificată de operația care a declanșat triggerul și FALSE în caz contrar.

SQL Server permite asocierea la o singură tabelă a mai multe triggere de același tip (UPDATE, INSERT sau DELETE). Triggerurile existente în SQL Server (până la versiunea 7.0 inclusiv) sunt de tip post (AFTER), ceea ce înseamnă că instrucțiunile din corpul triggerului ajung să fie executate la terminarea operației care a declanșat triggerul. În SQL Server 2000 sau introdus triggerele INSTEAD OF al căror cod se execută în locul operației care a inițiat triggerul.

## 13.1. Instrucțiunea CREATE TRIGGER

Creează un trigger, atașat unei anumite tabele pentru a fi executat la inițierea unui anumit tip de operație de modificare a tabelei.

Sintaxa:

```
CREATE TRIGGER nume_trigger
ON {tabelă|vedere} [WITH ENCRYPTION]
```

```
{ { FOR | AFTER | INSTEAD OF }
{ [DELETE] [,] [INSERT] [,] [UPDATE] }
[NOT FOR REPLICATION]
```

AS

```
    instrucțiune_sql[...n]
```

```
} { FOR { [INSERT] [,] [UPDATE] }
[NOT FOR REPLICATION]
```

AS

```
{
```

```
    IF UPDATE (coloana)
        [{AND | OR} UPDATE (coloana) [...]n]
    | IF (COLUMNS_UPDATED() {operator_bit})
        { operator_comparare } masca_modificare
    }
```

```
        {operator_comparare} masca_test [...]n]
        instrucțiune_sql[...n]
    }
```

unde:

*trigger\_name* - este numele triggerului. Optional se poate specifica numele proprietarului triggerului.

*tabelal vedere* - este numele tablei sau a vederii căreia i se atașează triggerul (nu poate fi un nume de vedere în cazul opțiunii **AFTER**). Opțional se poate specifica numele proprietarului tablei.

**WITH ENCRYPTION** - triggerul este înregistrat cu textul în formă încriptată.

**AFTER** - specifică faptul că triggerul se execută la terminarea operației care a activat triggerul. Opțiunea a fost introdusă începând cu versiunea SQL Server 2000 pentru a face distincție față de triggerele **INSTEAD OF**, **AFTER** este opțiunea implicită atunci când **FOR** este singurul cuvânt cheie specificat.

Triggerele **AFTER** nu se pot ataşa vederilor.

**INSTEAD OF** - specifică faptul că triggerul se execută în locul instrucțiunii SQL care a activat triggerul, substituindu-se acesteia. Cel mult un singur trigger **INSTEAD OF** pentru fiecare tip de operație **INSERT**, **UPDATE** sau **DELETE** poate fi atașat unei tabele sau vederi. Totuși, este posibil să definim vederi pe baza autor vederi și fiecare vedere poate avea propriile trigger **INSTEAD OF**.

{ [ **DELETE** [,] [**INSERT**] [,] [**UPDATE**] } | { [**INSERT**] [,] [**UPDATE**] } - specifică tipul operației (operațiilor) prin care se activează triggerul. Trebuie specificat cel puțin un tip de operație sau mai multe operații separate prin virgule.

**NOT FOR REPLICATION** - indică faptul că triggerul nu se va activa dacă modificarea tablei are loc ca urmare al unui proces de replicare.

**AS** - marchează începutul corpului triggerului.

*instructiune\_sql* - specifică condițiile și acțiunile triggerului. Un trigger poate conține condiții prin care se specifică criterii suplimentare pentru executarea acțiunilor din corpul triggerului. Triggerurile pot conține orice număr și orice tip de instrucțiuni SQL cu excepția frazei **SELECT**. Un trigger are rolul de a face verificări și eventual modificări asupra unei baze de date și în nici un caz nu trebuie să returneze date utilizatorului care l-a activat. Instrucțiunile din corpul unui trigger pot face referire la

două tabele speciale ale SQL Server numite **deleted** și **inserted**.

- **deleted** și **inserted** sunt tabele logice având aceeași structură cu tabela căreia îi este atașat triggerul. Tabela **deleted** conține valorile vechi ale tupelor asupra cărora actionează operația care a declanșat triggerul, iar tabela **inserted** conține valorile noi ale acelorași tuple. În cazul unui trigger **INSTEAD OF** atașat unei vederi structura tabelelor **deleted** și **inserted** este aceeași cu a vederii căreia îi este atașat triggerul.

**IF UPDATE** (*coloana*) - testează dacă coloana specificată a fost modificată în urma unei operații **INSERT** sau **UPDATE**. Nu este folosită în cazul operațiilor **DELETE**. Se pot testa mai multe coloane prin aceeași instrucțiune IF construind o expresie logică care conține mai multe clauze **UPDATE**(*coloana*).

**UPDATE**(*coloana*) se poate folosi oriunde în corpul unui trigger.

*coloana* - este numele coloanei pentru care se testează dacă a fost sau nu modificată.

**IF (COLUMNS\_UPDATED())** - testează dacă coloanele specificate au fost modificate în urma unei operații **INSERT** sau **UPDATE**. **COLUMNS\_UPDATED** returnează o mască de biți prin care se indică coloanele modificate. **COLUMNS\_UPDATED** se poate folosi oriunde în corpul unui trigger.

**operator\_bit** - este operatorul pe biți folosit în comparare.

*masca\_modificare* - este masca de biți folosită în comparație. Coloanele se testează.

**operator\_comparare** - este un operator de comparare. Operatorul = va testa dacă toate coloanele specificate prin *masca\_modificare* au fost modificate, iar operatorul > va testa dacă cel puțin una dintre coloanele specificate s-a modificat.

**masca\_test** - este masca de biți folosită ca referință în operația de comparație.

### Observații:

1. Triggerele sunt adesea folosite ca alternativă pentru a implementa condiții de integritate referențială în baza de date. Deși SQL Server ofera facilități declarative pentru exprimarea integrității referențiale, acestea sunt limitate la cazul restricționat (până la versiunile SQL Server 7.0 inclusiv nu se poate opta pentru ștergeri cascadați) și, mai mult, sunt limitate la tabelele bazei de date curente, fără a fi posibilă exprimarea de legături referențiale între tabele din baze de date diferite.
2. Orice constrângere referitoare la tabela căreia îi este atașat un trigger este verificată înainte de execuția triggerului. De aceea orice operație care ar viola o constrângere de cheie primară sau cheie străină este respinsă și implicit triggerul asociat nu se va activa. Din acest motiv, dacă se dorește implementarea unei constrângeri de integritate referențială în varianta cascadă prin folosirea unui trigger, este necesar să se stearga ori să se dezactiveze declarația de cheie străină pe care o înlocuește.

## 13.2. Instrucția DROP TRIGGER

Sterge una sau mai multe trigger din baza de date curentă.

### Sintaxa:

DROP TRIGGER {*nume\_trigger*} [, *n*]

unde:

*nume\_trigger* - este numele triggerului care se sterge.

### Observații:

Un trigger poate fi sters fie printr-o operație explicită de stergere trigger care referă triggerul în cauză, fie implicit prin stergerea tabelei căreia îi este asociat acesta. Pentru a redenumi un trigger acesta trebuie întâi sters și apoi recreat cu noul nume (combinatie **DROP TRIGGER** și **CREATE TRIGGER**). Pentru a modifica definiția unui trigger se poate folosi instrucția **ALTER TRIGGER** a cărei sintaxă este similară instrucției **CREATE TRIGGER**.

## 13.3. Proprietăți ale triggerelor

1. Instrucția **CREATE TRIGGER** trebuie să fie prima într-un batch și se aplică unei singure tabele.
2. Deși un trigger poate să facă referire la obiecte din afara bazei de date curente, el poate fi creat numai în baza de date curentă.

Același trigger poate fi asociat cu mai multe tipuri de acțiuni asupra unei tabele; practic orice combinație dintre **INSERT**, **DELETE** și **UPDATE** este permisă.

4. Orice operație de tip **SET** poate fi specificată în corpul unui trigger. Setările rămân valabile până la ieșirea din trigger după care revin la valorile inițiale.
5. La execuțarea unui trigger se pot returna rezultate la fel ca în cazul procedurilor stocate. Acest lucru este, de regulă, nedorit în aplicații de acesta trebuie evitate frazele **SELECT** sau atribuirile de variabile prin instrucțiea **SET**. Orice trigger care conține una dintre aceste tipuri de instrucții necesită un tratament special în fiecare aplicație care ar începeputul triggerului se poate evita returnarea de către trigger a oricărui relațional rezultat.

6. Un trigger **AFTER** nu se poate asocia unei vederi.
7. O instrucție **TRUNCATE TABLE** nu activează eventualele trigger de tip **DELETE** asociate tabelei curente. Similar instrucția **WRITETEXT** nu activează triggerele.
8. Următoarele instrucții sunt interzise în corpul unui trigger:

|                |                  |                   |
|----------------|------------------|-------------------|
| ALTER DATABASE | ALTER PROCEDURE  | ALTER TABLE       |
| ALTER TRIGGER  | ALTER VIEW       | CREATE DATABASE   |
| CREATE DEFAULT | CREATE INDEX     | CREATE PROCEDURE  |
| CREATE RULE    | CREATE SCHEMA    | CREATE TABLE      |
| CREATE TRIGGER | CREATE VIEW      | DENY              |
| DISK INIT      | DISK RESIZE      | DROP DATABASE     |
| DROP DEFAULT   | DROP INDEX       | DROP PROCEDURE    |
| DROP RULE      | DROP TABLE       | DROP TRIGGER      |
| DROP VIEW      | GRANT            | LOAD DATABASE     |
| LOAD LOG       | RESTORE DATABASE | RESTORE LOG       |
| REVOKE         | RECONFIGURE      | UPDATE STATISTICS |
| TRUNCATE TABLE |                  |                   |

## 13.4. Triggere multiple

SQL Server permite crearea a mai multe triggere, cu nume diferite, pentru același tip de operatie (DELETE, INSERT sau UPDATE) și același tabel. Orice trigger creat cu un nume diferit de cele existente (pentru un tip de operatie și o tabelă) se adaugă la acestea. Dacă se încarcă crearea unui trigger cu un nume care există deja, atunci se produce o eroare.

## 13.5. Triggere recursive

SQL Server permite apelul recursiv al triggerelor cu condiția că opțiunea de sistem corespunzătoare să fie activată. În acest caz pot să apară două tipuri de recursivitate:

- Recursivitate indirectă

Este cazul în care reactivarea triggerului curent se face prin intermediu altui trigger.

- Recursivitate directă

Este cazul în care reactivarea triggerului curent se datoră modificărilor pe care el însuși le face în baza de date.

În cazul recursivității indirecte, o aplicație care modifică tabela T1 activează triggerul TR1, aceasta modifică tabela T2 și activează triggerul TR2. La rândul lui triggerul TR2 actualizează tabela T1 și activează îărăși triggerul TR1.

Pentru recursivitatea directă scenariul tipic este următorul: o aplicație modifică tabela T1 și activează triggerul TR1, care modifică chiar ei tabela T1, ceea ce reactivăza triggerul TR1 și.a.m.d.

### Observații:

1. Tabelele inserted și deleted pentru un anumit trigger conțin, chiar și în cazul apelurilor recursive, numai acele tuple care corespund condițiilor din instrucțiunea UPDATE care a inițiat apelurile triggerelor.
2. Nu există nici o ordine predefinită pentru execuția triggerelor multiple corespunzătoare aceluiași eveniment. Din acest motiv fiecare trigger trebuie să fie independent din punct de vedere logic, iar acțiunile cumulate ale triggerelor multiple nu trebuie să depindă de ordinea în care sunt executate. În versiunea SQL Server 2000 există posibilitatea de a influența parțial ordinea de execuție a triggerelor.

## 13.7. Exemple de utilizare a triggerelor

**Exemplu 1:** Triggerul de mai jos afișează un mesaj la terminarea cu succes a unei operații de inserare furnizor. Triggerul se va activa doar dacă operația de inserare nu violează nici una dintre condițiile de integritate ale bazei de date.

```
IF EXISTS (SELECT name FROM sysobjects WHERE name = 'inserat_furnizor' AND type = 'TR')  
DROP TRIGGER inserat_furnizor  
GO  
CREATE TRIGGER inserat_furnizor
```

multiple prin specificarea triggerului care să se execute primul și/sau a celui care se execută ultimul (vezi procedura stocată sp\_settriggerorder).

## 13.6. Triggere imbricate

Implicit, SQL Server permite apelurile imbricate de trigger (recursiv sau nu) până la 32 nivele de adâncime. Prin această limitare se garantează faptul că, în cazul unui lanț, potențial infinit, de apeluri recursive acesta va fi terminat prin intervenția sistemului în momentul depășirii nivelului de imbricare maxim admis. Prin setarea corespunzătoare a opțiunilor de sistem se poate bloca propagarea activărilor de trigger prin intermediul altor trigger.

### Exemplu:

Presupunem că am definit trigger pentru cascadarea ștergerilor pe tabelele *Furnizor* și *Oferte*, astfel încât ștergerea unui furnizor determină ștergerea ofertelor, iar ștergerea unei oferte determină ștergerea tranzacțiilor corespunzătoare. În modul de lucru implicit, cu facilitatea de imbricare a triggerelor activată, prin cele două trigger de mai sus se obține efectul ștergerii tuturor ofertelor și tranzacțiilor unui furnizor atunci când acesta este sters din tabela *Furnizor*. Dacă se dezactivează opțiunea de imbricare a triggerelor, atunci la ștergerea unui furnizor se va activa numai triggerul de ștergere a ofertelor, iar tranzacțiile rămân neafectate. O variantă care rezolvă problema în acest caz, este aceea de a proiecta triggerul atașat tabeliei *Furnizor* astfel ca la ștergerea unui furnizor să șteargă atât ofertele, cât și tranzacțiile acestuia. Bineînțeles că se va păstra și triggerul atașat tabeliei *Oferte* prin care se asigură ștergerea tranzacțiilor atunci când are loc ștergerea unei oferte.

```

ON Furnizor
FOR INSERT AS
PRINT 'Furnizor inserat cu succes!'
GO

```

### Exemplu 2:

Câmpul *Name* din tabela *Furnizor* nu este definit ca și cheie primară. Totuși, dorim ca acesta să fie cheie candidată în tabela *Furnizor*, adică numele de furnizor trebuie să fie unic. Acest lucru poate fi obținut prin crearea unui index unic peste atributul *Name* din tabela *Furnizor*. Dacă nu dorim acest lucru și, eventual, vrem să implementăm un comportament specializat al sistemului la orice tentativă de violare a acestor condiții, putem să folosim un trigger atașat tabelei *Furnizor*, activat prin instrucția **INSERT** sau **UPDATE**.

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'furnizor_unic' AND type = 'TR')
    DROP TRIGGER furnizor_unic
GO
CREATE TRIGGER furnizor_unic
ON Furnizor
FOR INSERT, UPDATE AS
    IF EXISTS (SELECT * FROM Furnizor, inserted
               WHERE Furnizor.CodF=>inserted.CodP
                 AND Furnizor.Nume=inserted.Nume)
        BEGIN
            RAISERROR ('Există deja furnizor cu același nume
                       16, 1)
            ROLLBACK TRANSACTION
        END
    ELSE
        PRINT 'Furnizor inserat cu succes!'
GO

```

### Observații:

- În momentul executării triggerului, furnizorii inserăți prin instrucția **INSERT** sau **UPDATE** care a activat triggerul se găsesc deja în tabela *Furnizor*.
- Prin instrucția **ROLLBACK TRANSACTION** se anulează atât efectele triggerului, cât și ale instrucției **INSERT** sau **UPDATE** care a activat triggerul.
- De remarcat că triggerul de mai sus funcționează corect atât în cazul operațiilor de inserare sau modificare simplă, cât și în cazul celor multiple.

**Exemplu 3:** O constrângere de tip **CHECK** poate face referire numai la coloana sau tabela pentru care este definită. Din acest motiv declarațiile **CHECK** nu se pot folosi pentru a exprima constrângeri între mai multe coloane sau între tabele diferite. În aceste cazuri se pot folosi triggerurile.

Triggerul de mai jos verifică pentru fiecare tranzacție inserată sau modificată în tabela *Tranzactii* dacă prețul de tranzacționare este între prețul ofertei și cel al cererii. Tranzacțiile care nu respectă această condiție nu vor fi inserate, ori modificarea va fi refuzată, și se va afișa un mesaj corespunzător cu valorile celor trei prețuri.

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'pret_tranzactie' AND type = 'TR')
    DROP TRIGGER pret_tranzactie
GO
CREATE TRIGGER pret_tranzactie
ON Tranzactii
FOR INSERT, UPDATE AS
    DECLARE @rows int
    DECLARE @pret_tranzactie money
    DECLARE @pret_oferta money
    DECLARE @pret_cerere money
    DECLARE @mesaj varchar(255)
    SET @rows=@@rowcount
    IF (@rows=0) RETURN
    IF UPDATE(Pret)
        BEGIN
            IF (@rows!=1)
                --mai multe tuple modificate?
                ROLLBACK TRANSACTION
                RETURN
        END
    SELECT @pret_tranzactie=Tranzactii.Pret
    WHERE Tranzactii.CodT=inserted.CodT
    SELECT @pret_oferta=Oferte.Pret
    FROM Oferte, inserted
    WHERE Oferte.CodO=inserted.CodO
          AND Oferte.CodP=inserted.CodP
    SELECT @pret_cerere=Cereri.Pret
    FROM Cereri, inserted
    WHERE Cereri.CodB=inserted.CodB
          AND Cereri.CodP=inserted.CodP
    IF NOT (@pret_tranzactie BETWEEN @pret_cerere
           AND @pret_oferta)
        BEGIN
            SET @mesaj='Prețul de tranzacție este'
            CONVERT(varchar(10), @pret_tranzactie)

```

```

SET @mesaj=@mesaj+' nu este intre pret_cerere='+
      CONVERT(varchar(10),@pret_cerere)
SET @mesaj=@mesaj+' si pret_oferta='+
      CONVERT(varchar(10),@pret_oferta)
RAISERROR (@mesaj,16,1)
ROLLBACK TRANSACTION
END
GO

```

#### Observații:

- Variabila sistem `@rowcount` indică numărul de tuple afectate de operația `INSERT` sau `UPDATE` care a activat triggerul. Dacă acest număr este zero (în cazul unui `UPDATE` cu condiție `WHERE` evaluată la `FALSE`) triggerul nu face nimic. În caz contrar se testează prin clauza `UPDATE(Pret)` dacă operația a modificat atributul `Pret`. De remarcat că triggerul funcționează corect numai dacă numărul de tuple modificat prin operația inițială este cel mult 1. De aceea operațiile `INSERT` sau `UPDATE` multiple sunt rejetate din start. În cazul unei modificări simple se trece la verificarea constrângerii propriu-zise, și anume prețul de tranzacție să fie între cel de cerere și cel de ofertă. În variabila `@mesaj` se construiește mesajul de avizare care este afișat în cazul violării condiției menționate.
- Testul `IF UPDATE(Pret)` se poate înlocui prin:

```

IF (COLUMNNS_UPDATED() & 32) = 32

```

deoarece `Pret` este al şaselea câmp din tabela `Tranzactii`, iar masca cu valoarea 32 are 1 pe al şaselea bit (de la dreapta la stânga). De remarcat că `COLUMNNS_UPDATED()` poate returna o mască formată din mai mulți octeți: primul octet corespunde cu primele 8 câmpuri ale tabelei, al doilea cu următoarele 8 și.m.d.

Clauza `COLUMNNS_UPDATED()` este utilă atunci când se dorește testarea simultană a mai multe coloane. Iată câteva exemple:

```

IF (COLUMNNS_UPDATED() & 7) = 7 --test modificare coloanele 1,2 și 3
IF (COLUMNNS_UPDATED() | 7) = 7 --test modificare cel puțin una din
--coloanele 1,2 sau 3

```

--test modificare coloanele 9,10 și 11 (adică 1,2 și 3 din octetul 2)
IF (SUBSTRING(COLUMNNS\_UPDATED(), 2, 1) & 7) = 7

--test modificare coloanele 1,2,3,9,10, și 11
IF ((COLUMNNS\_UPDATED() & 7) = 7 AND
 (SUBSTRING(COLUMNNS\_UPDATED(), 2, 1) & 7) = 7)

#### Exemplu 4:

Triggerul de mai jos este atașat tablei `Oferte` și este activat de o operație de tip `INSERT` simplă sau multiplă. Rolul său este de a afișa un mesaj prin care se comunică valoarea totală a ofertelor inserate prin operația care a inițiat triggerul.

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'valoare_oferte' AND type = 'TR')
DROP TRIGGER valoare_oferte
GO
CREATE TRIGGER valoare_oferte
ON Oferte
FOR INSERT AS
DECLARE @rows int
SET @rows=@rowcount
IF (@rows=0) RETURN
SELECT @total_oferte=SUM(Cantitate*Pret)
FROM inserted
PRINT 'Valoarea totală a ofertelor introduse este: '+
      CONVERT(varchar(10),@total_oferte)
GO

```

#### Observație:

Triggerul valoare\_oferte funcționează corect și în cazul unei operații de inserare multiplă de forma: `INSERT Oferte SELECT ...!` De reținut că activarea și execuția triggerului are loc o singură dată pentru întreaga operație de inserare și nu pentru fiecare tuplă în parte.

#### Exemplu 5:

Orice trigger din SQL Server are acces la două tabele temporare, rezidente în memorie numite: `deleted` și `inserted`. Datele din aceste tabele nu pot fi modificate, dar pot fi folosite pentru a testa efectul operațiilor care au inițiat triggerel, pentru condiții de executare a diverse operații din cadrul triggerelor și pentru a face discriminare între diferențele tipuri de operații (`DELETE`, `INSERT` sau `UPDATE`) care pot declansa un trigger.

Tabela `deleted` conține copiile tuplelor afectate de o instrucție `DELETE` sau `UPDATE`. În timpul execuției unei asemenea operații tuplele afectate sunt ștersse din tabela asociată și transferate în tabela `deleted`. În mod normal, tabela asociată și tabela `deleted` nu vor avea nici o tuplă în comun. În cazul unei operații `INSERT` tabela `deleted` va fi vidă.

Tabela `inserted` conține copile tuplelor afectate de o instrucție `INSERT` sau `UPDATE`. În timpul execuției unei asemenea operațiuni tuplele noi sunt adăugate simultan atât în tabela `inserted`,

```

ON Furnizor
FOR INSERT AS
PRINT 'Furnizor inserat cu succes!'
GO

```

**Exemplu 2:**

Câmpul *Nume* din tabela *Furnizor* nu este definit ca să fie cheie primară. Totuși, dorim ca acesta să fie cheie candidată în tabela *Furnizor*, adică numele de furnizor trebuie să fie unice. Acest lucru poate fi obținut prin crearea unui index unic peste atributul *Nume* din tabela *Furnizor*. Dacă nu dorim acest lucru și, eventual, vrem să implementăm un comportament specializat al sistemului la orice tentativă de violare a acestei condiții, putem să folosim un trigger atașat tabelei *Furnizor*, activat prin instrucția **INSERT** sau **UPDATE**.

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'furnizor_unic' AND type = 'TR')
DROP TRIGGER furnizor_unic
GO
CREATE TRIGGER furnizor_unic
ON Furnizor
FOR INSERT, UPDATE AS
IF EXISTS (SELECT * FROM Furnizor, inserted
           WHERE Furnizor.CodF<>inserted.CodF
           AND Furnizor.Nume=inserted.Nume)
    RAISERROR ('Există deja furnizor cu același nume!', 16, 1)
ROLLBACK TRANSACTION
END
ELSE
    PRINT 'Furnizor inserat cu succes!'
GO

```

**Observații:**

- În momentul executării triggerului, furnizorii inserați prin instrucția **INSERT** sau **UPDATE** care a activat triggerul se găsesc deja în tabela *Furnizor*.
- Prin instrucția **ROLLBACK TRANSACTION** se anulează atât efectele triggerului, cât și ale instrucțiunii **INSERT** sau **UPDATE** care a activat triggerul.
- De remarcat că triggerul de mai sus funcționează corect atât în cazul operațiilor de inserare sau modificare simplă, cât și în cazul celor multiple.

**Exemplu 3:** O constrângere de tip **CHECK** poate face referire numai la coloana sau tabela pentru care este definită. Din acest motiv declaratiile **CHECK** nu se pot folosi pentru a exprima constrângeri între mai multe coloane sau între tabele diferite. În aceste cazuri se pot folosi triggerele.

Triggerul de mai jos verifică pentru fiecare tranzacție inserată sau modificată în tabela *Tranzactii* dacă prețul de tranzacționare este între prețul ofertei și cel al cererii. Tranzacțiile care nu respectă această condiție nu vor fi inserate, ori modificarea va fi refuzată, și se va afișa un mesaj corespunzător cu valorile celor trei prețuri.

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'pret_tranzactie' AND type = 'TR')
DROP TRIGGER pret_tranzactie
GO
CREATE TRIGGER pret_tranzactie
ON Tranzactii
FOR INSERT, UPDATE AS
DECLARE @rows int
DECLARE @pret_tranzactie money
DECLARE @pret_oferta money
DECLARE @pret_cerere money
DECLARE @mesaj varchar(255)
SET @rows=@rowcount
IF (@rows=0) RETURN --daca nu s-a modificat nimic se revine
IF UPDATE(Pret)
BEGIN
    IF (@rows!=1) --mergi mai departe numai daca s-a
        --modificat preț
    BEGIN
        ROLLBACK TRANSACTION
        RETURN
    END
    ELSE
        SELECT @pret_tranzactie=tranzactii.Pret
        FROM Tranzactii,inserted
        WHERE Tranzactii.CodIn=inserted.CodIn
        SELECT @pret_oferta=Cereri.Pret
        FROM Oferte,inserted
        WHERE Oferte.CodB=inserted.CodB
        AND Oferte.CodP=inserted.CodP
        AND Cereri.CodB=inserted.CodB
        AND Cereri.CodP=inserted.CodP
        IF NOT (@pret_tranzactie BETWEEN @pret_cerere
               AND @pret_oferta)
        BEGIN
            SET @mesaj='Prețul de tranzacție='+
                      CONVERT(varchar(10),@pret_tranzactie)
            PRINT @mesaj
        END
    END
END

```

```

SET @mesaj=@mesaj+' nu este intre pret cerere='+
      CONVERT(varchar(10),@pret_cerere)
SET @mesaj=@mesaj+' si pret oferta='+
      CONVERT(varchar(10),@pret_oferta)
RAISERROR (@mesaj,16,1)
ROLLBACK TRANSACTION
END
GO

```

#### Observații:

- Variabila sistem `@rowcount` indică numărul de tuple afectate de operația `INSERT` sau `UPDATE` care a activat triggerul. Dacă acest număr este zero (în cazul unui `UPDATE` cu condiție `WHERE` evaluată la FALSE!) triggerul nu face nimic. În caz contrar se testează prin clauza `UPDATE(Pret)` dacă operația a modificat atributul `Pret`. De remarcat că triggerul funcționează corect numai dacă numărul de tuple modificat prin operația inițială este cel mult 1. De aceea operațiile `INSERT` sau `UPDATE` multiple sunt rejetate din start.
- In cazul unei modificări simple se trece la verificarea constrângерii propriu-zise, și anume prețul de tranzacție să fie între cel de cerere și cel de ofertă. În variabila `@mesaj` se constituiește mesajul de avertizare care este afișat în cazul violării condițiilor menționate.

#### 2. Testul IF UPDATE(Pret) se poate înlocui prin:

```

IF (COLUMNNS_UPDATED() & 32) = 32

```

din carece `Pret` este al şaselea câmp din tabela `Tranzactii`, iar masca cu valoarea 32 are 1 pe al şaselea bit (de la dreapta la stanga). De remarcat că `COLUMNNS_UPDATED()` poate returna 0 mască formată din mai mulți octeți: primul octet corespunde cu primele 8 câmpuri ale tabelei, al doilea cu următoarele 8 s.a.m.d.

Clauza `COLUMNNS_UPDATED()` este utilă atunci când se dorește testarea simultană a mai multe coloane. Iată câteva exemple:

```

IF (COLUMNNS_UPDATED() & 1) = 1 --test modificare coloanele 1,2 și 3
IF (COLUMNNS_UPDATED() & 7) = 7 --test modificare cel puțin una din
--coloanele 1,2 sau 3
--test modificare coloanele 9,10 și 11 (adica 1,2 și 3 din octetul 2)
IF (SUBSTRING(COLUMNNS_UPDATED(), 2, 1) & 7) = 7
--test modificare coloanele 1,2,3,9,10 și 11
IF ((COLUMNNS_UPDATED() & 7) = 7 AND
    (SUBSTRING(COLUMNNS_UPDATED(), 2, 1) & 7) = 7)

```

#### Exemplu 4:

Triggerul de mai jos este atașat tabelui `Oferte` și este activat de o operație de tip `INSERT` simplă sau multiplă. Rolul său este de a afișa un mesaj prin care se comunică valoarea totală a ofertelor inserate prin operația care a inițiat triggerul.

```

CREATE TRIGGER valoare_oferte
ON Oferte
FOR INSERT AS
BEGIN
    DECLARE @rows int
    SELECT @total_oferte=SUM(Cantitate*Pret)
    IF (@rows=0) RETURN
    PRINT 'Valoarea totala a ofertelor introduse este: '+
          CONVERT(varchar(10),@total_oferte)
    GO

```

#### Observație:

Triggerul `valoare_oferte` funcționează corect și în cazul unei operații de inserare multiplă de forma: `INSERT Oferte SELECT ...!` De reținut că activarea unei operații de inserare a unei tabele se face pe totul, nu pe singură linie.

#### Exemplu 5:

Orice trigger din SQL Server are acces la două tabele temporare, rezidente în memorie numite: `deleted` și `inserted`. Datele din aceste tabele nu pot fi modificate, dar pot fi folosite pentru a testa efectul operațiilor care au inițiat triggerle, pentru condiții de executare a diverse operații din cadrul triggerelor și pentru a face discriminare între diferențele tipurii de operații (`DELETE`, `INSERT` sau `UPDATE`) care pot declanșa un trigger.

Tabela `deleted` conține copiile tupelor afectate de o instrucție `DELETE` sau `UPDATE`. În timpul execuției unei asemenea operații tuplele afectate sunt șterse din tabela asociată și transferate în tabela `deleted`. În mod normal, tabela asociată și tabela `deleted` nu vor avea nici o tuplă în comun. În cazul unei operații `INSERT` tabela `deleted` va fi vidă.

Tabela `inserted` conține copiile tupelor afectate de o instrucție `INSERT` sau `UPDATE`. În timpul execuției unei asemenea operații tuplele noi sunt adăugate simultan atât în tabela `inserted`,

cât și în tabela asociată. În cazul unei operații DELETE tabela **inserted** va fi vidă.

O operație UPDATE poate fi privită ca o ștergere urmată de o inserare. Tuplele vecni sunt copiate în tabela **deleted**, iar cele noi în tabela asociată și tabela **inserted**.

În trigger se poate determina tipul instrucțiunii declanșatoare astfel:

- dacă atât **inserted**, cât și **deleted** sunt nevide, atunci

instrucțiunea este **UPDATE**;

- dacă **inserted** este vidă și **deleted** nevidă, atunci instrucțiunea este **DELETE**;

- dacă **inserted** este nevidă și **deleted** vidă, atunci instrucțiunea este **INSERT**.

Triggerul de la exemplul precedent poate fi extins pentru a informa asupra modificărilor de oferte nu numai la apariția unor oferte noi, ci și atunci când, prin închiderea unor tranzacții, are loc modificarea sau ștergerea ofertelor afectate de tranzacție. În urma unei operații UPDATE se va afișa nu numai valoarea tranzacționată, ci și valoarea rămasă în cadrul ofertelor afectate.

```

IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'bilant_oferte' AND type = 'TR')
    DROP TRIGGER bilant_oferte
GO
CREATE TRIGGER bilant_oferte
ON Oferte
FOR INSERT, DELETE, UPDATE AS
DECLARE @rows int
DECLARE @total_oferte_inserare money
DECLARE @total_oferte_sterse money
SET @rows=@@rowcount
IF (@rows=0) RETURN
IF NOT EXISTS (SELECT * FROM deleted)      -- operația este INSERT
BEGIN
    SELECT @total_oferte_inserare=SUM(Cantitate*Pret)
    FROM inserted
    PRINT 'Valoarea totală a ofertelor introduse este:'+
          CONVERT(varchar(10), @total_oferte_inserare)
    RETURN
END
IF NOT EXISTS (SELECT * FROM inserted)      -- operația este DELETE
BEGIN
    SELECT @total_oferte_sterse=SUM(Cantitate*Pret)
    FROM deleted
    PRINT 'Valoarea totală a ofertelor sterse este:'+
          CONVERT(varchar(10), @total_oferte_sterse)
    RETURN

```

END

--dacă să ajuș aici, atunci operația este UPDATE  
SELECT @total\_oferte\_sterse=SUM(Cantitate\*Pret)  
FROM deleted

```

PRINT 'Valoarea tranzacționată este:'+
      CONVERT(varchar(10), @total_oferte_sterse)
SELECT @total_oferte_inserare=SUM(Cantitate*Pret)
FROM inserted
PRINT 'Valoarea rămasă este:'+
      CONVERT(varchar(10), @total_oferte_inserare)
GO

```

#### Exemplu 6:

O instrucțiune INSERT, UPDATE sau DELETE care referă o vedere având atașat un trigger INSTEAD OF declanșeză execuțarea instrucțiunilor din corpul triggerului în locul operației inițiale. În acest caz tablele **inserted** și **deleted** au aceeași câmpuri ca și vederea referită.

Un trigger INSTEAD OF INSERT se atașează unei vederi pentru a înlocui o operație de tip INSERT care face referire la acea vedere. În acest mod se pot defini operații de inserare într-o vedere care se traduc prin inserări în mai multe tabele de bază. Fie vederea *Tranzactii\_Desfasurat* definită astfel:

```

CREATE VIEW Tranzactii_Desfasurat AS
SELECT Nume_Furnizor=Furnizor.Nume,
       Oras_Furnizor=Furnizor.Oras,
       Nume_Beneficiar=Beneficiar.Nume,
       Oras_Beneficiar=Beneficiar.Oras,
       Nume_Produs=Produs.Nume,
       Cantitate,Pret
  FROM Furnizor, Beneficiar, Produs, Tranzactii
 WHERE Furnizor.CodF=Tranzactii.CodF
   AND Beneficiar.CodB=Tranzactii.CodB
   AND Produs.CodP=Tranzactii.CodP

```

O instrucțiune INSERT care face referire la vedere *Tranzactii\_Desfasurat* poate fi interpretată în diverse moduri:

- se inserează câte o tuplă în fiecare dintre tabele *Furnizor*, *Beneficiar*, *Produs* și *Tranzactii*;
- se inserază o tuplă doar în tabela *Tranzactii* cu codurile obținute din tabelele *Furnizor*, *Beneficiar* și *Produs* pe baza valorilor date pentru attributele *Nume*, *Oras* și *Cantitate*;
- orice varianta intermedieră între cele două de sus.

În oricare dintre variante apar probleme suplimentare de rezolvat:

- Ce se întâmplă dacă există deja un furnizor cu același nume? (la prima variantă);

- Ce se întâmplă dacă furnizorul menționat nu există? (la a doua variantă) §.a.m.d.

Un trigger **INSTEAD OF INSERT** atât vederii

*Tranzactii\_Desfasurat* poate da un sens bine determinat unei operații de inserare în această vedere, operație care altfel nu ar fi permisă.

Triggerul de mai jos implementează un scenariu intermediar primelor două variante prezentate mai sus, și anume se inserează tuple în tabelele *Furnizor*, *Beneficiar* sau *Produs* numai dacă nu există furnizorul, beneficiarul sau produsul cu numele menționat.

IF EXISTS (SELECT name FROM sysobjects

WHERE name = 'inserare\_tranzactie'

GO

DROP TRIGGER inserare\_tranzactie

CREATE TRIGGER inserare\_tranzactie

ON Tranzactii

FOR INSTEAD OF INSERT AS

DECLARE @rows int

DECLARE @codf int

DECLARE @codb int

DECLARE @codp int

SET @rows=@@rowcount

IF (@rows=0) RETURN

IF (@rows!=1)

BEGIN

ROLLBACK TRANSACTION --se acceptă numai inserări simple

END

IF NOT EXISTS (SELECT \* FROM Furnizor, inserted  
WHERE Nume=Nume\_Furnizor)

INSERT Furnizor SELECT Nume\_Furnizor, Oras\_Furnizor  
FROM inserted

SELECT @codf=CodeF

FROM Furnizor, inserted

WHERE Nume=Nume\_Furnizor

IF NOT EXISTS (SELECT \* FROM Beneficiar, inserted  
WHERE Nume=Nume\_Beneficiar)

INSERT Beneficiar SELECT Nume\_Nume\_Beneficiar  
FROM inserted

SELECT @codb=CodeB

FROM Beneficiar, inserted

WHERE Nume=Nume\_Beneficiar

IF NOT EXISTS (SELECT \* FROM Produs, inserted  
WHERE Nume=Nume\_Produs)

INSERT Produs SELECT Nume\_Produs FROM inserted  
FROM Produs, inserted  
WHERE Nume=Nume\_Produs

```
INSERT Tranzactii SELECT @codf, @codb, @codp, Cantitate, Pret
FROM inserted
GO
```

## 13.8. Exerciții și probleme

1. În cazul unei operații de inserare sau modificare multiplă triggerul de la exemplul 2 anulează toate modificările în cazul în care cel puțin una dintre ele produce un duplicat pentru câmpul nume. Sa se modifice acest trigger astfel încât să permită reiectarea selectivă a modificărilor, doar pentru tuplele care conduc la violarea condiției menționate. Pentru fiecare dintre acestea se va afișa un mesaj

specializat care să conțină și numele pentru care apare duplicit.

2. Să se scrie un trigger după modelul celui de la exemplul 3 prin care să se verifice faptul că într-o tranzacție cantitatea de produse este mai mică decât cantitatea din ofertă și decât cea din cerere.

3. Să se rescrie triggerul *pret\_tranzactie* de la exemplul 3 astfel încât să funcționeze corect și în cazul operațiilor INSERT sau UPDATE multiple. Se vor rejecta doar tuplele care violează condiția menționată, cele corecte fiind păstrate.

4. Să se scrie un trigger INSTEAD OF care să se activeze la o operărie de stergere (DELETE) din vedere *Tranzactii\_Desfasurat*.

5. Să se scrie un trigger INSTEAD OF care să se activeze la o operărie de modificare (UPDATE) a vederii *Tranzactii\_Desfasurat*.

## BIBLIOGRAFIE

1. Ageloff, R. - "A Primer on SQL", Times Mirror/Mosby College Publishing
2. Bjeletich, S., Mable,G. - "Microsoft SQL Server 7.0 Unleashed", Sams Publishing, 1999, USA
3. Date, C. J., Darwen H. - "A Guide to The SQL Standard", Addison-Wesley Publishing Company, 1994, USA
4. Date, C. J. - "An Introduction to Database Systems", Addison-Wesley Publishing Company, 1995, USA
5. Dollinger, R. - "Baze de date și gestiunea tranzacțiilor", Editura Albastră, Cluj-Napoca, 2001
6. Leverenz, L., Rehfield, D. - "Oracle8i Concepts, Release8.1.5.", Oracle Corporation, 1999, USA
7. Panttaja, J., Panttaja, M., Prendergast, B. - "The Microsoft SQL Server Survival Guide", John Wiley & Sons, Inc., 1996, USA
8. Papa, J., Shekter, M. - "Microsoft SQL Server 7.0 Unleashed, Programming", Sams Publishing, 1999, USA
9. Spenik, M., Sledge, O. - "Microsoft SQL Server 7.0 DBA Survival Guide", Sams Publishing, 1999, USA
10. Soukup, R., Delaney, K., - "Inside Microsoft SQL Server 7.0", Microsoft Press, Redmond, Washington, 1999, USA
11. \*\*\* - Microsoft SQL Server 7.0, Books Online
12. \*\*\* - Microsoft SQL Server 2000, Books Online

