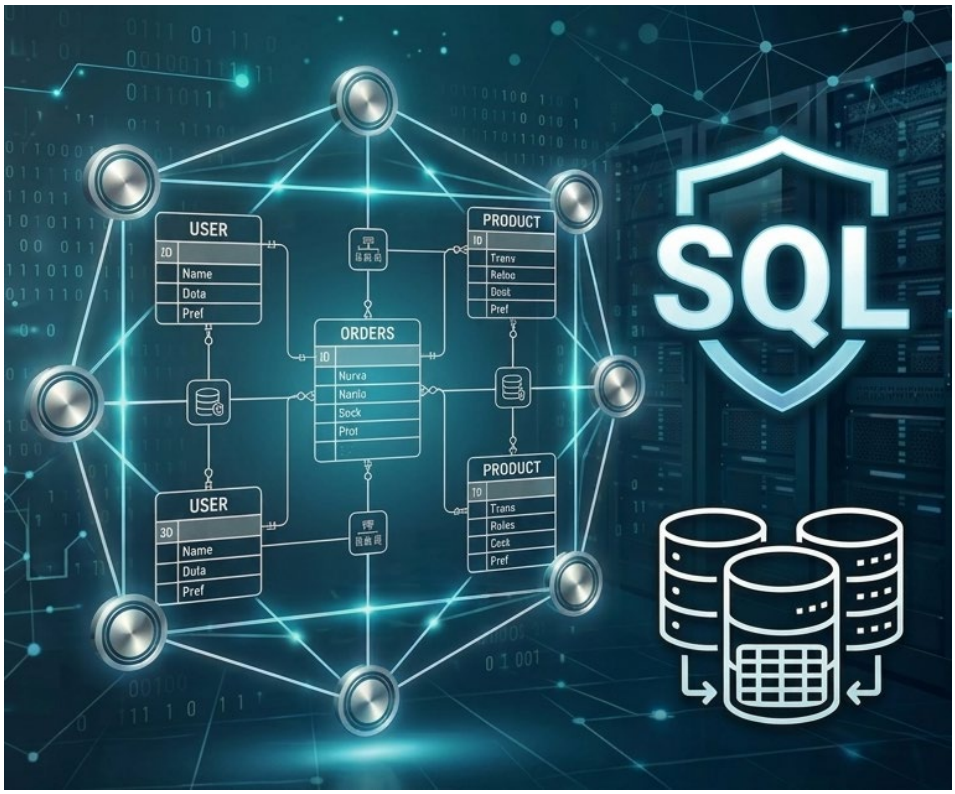
ACADEMIA DE STUDII ECONOMICE DIN MOLDOVA

Serghei OPREA

PROIECTAREA ȘI GESTIUNEA BAZELOR DE DATE

Note de curs



CHIȘINĂU, 2026

CZU 004.65(075.8)

O-64

Lucrarea ”Proiectarea și gestiunea bazelor de date. Note de curs” a fost examinată și aprobată pentru editare la ședința Departamentului ”Tehnologia Informației și Management Informațional” (proces verbal nr. 1 din 01.09.2026) și la ședința Comisiei metodice a facultății ”Tehnologii Informaționale și Statistică Economică” (proces verbal nr. 2 din 08.10.2026).

Referenți:
Lozan Victoria, dr., conf.univ., ASEM
Zgureanu Aureliu, dr., conf.univ., ASEM

DESCRIEREA CIP A CAMEREI NAŢIONALE A CĂRŢII DIN REPUBLICA MOLDOVA

Oprea, Serghei.

Proiectarea și gestiunea bazelor de date : Note de curs / Oprea Serghei ; Academia de Studii Economice din Moldova. – Chișinău : [S. n.], 2026 (SEP ASEM). – 175 p. : fig., tab. color.

Cerințe de sistem: PDF Reader.

Bibliogr.: p. 175 (12 tit.). – În red. aut.

ISBN 978-9975-182-47-8 (PDF).

Text : electronic.

004.65(075.8)

O-64

ISBN 978-9975-182-47-8 (PDF).

© Serghei OPREA, dr., conf. univ., 2026

© Serviciul Editorial-Poligrafic ASEM, 2026

CUPRINS

INTRODUCERE **6**

CAPITOLUL 1. CONCEPTE FUNDAMENTALE PRIVIND BAZELE DE DATE **7**

1.1. Evoluția organizării datelor în memoria calculatorului 7
1.2. Organizarea datelor în fișiere de date 8
1.3. Organizarea datelor în baze de date 9
1.4. Nivele de abstracție și independența datelor 10
1.5. Sistemul de gestiune a bazelor de date 12
1.6. Limbaje ale bazelor de date 14

CAPITOLUL 2. MODELE DE BAZE DE DATE **16**

2.1. Modele de date 16
2.2. Modelul relațional de date 19
2.3. Constrângeri de integritate 21
2.4. Modele logice orientate pe obiecte 22
2.5. Modele arhitecturale ale bazelor de date 23

CAPITOLUL 3. ELEMENTE DE ALGEBRĂ RELAȚIONALĂ **25**

3.1. Operații tradiționale pe mulțimi 25
3.2. Operațiile relaționale native 27

CAPITOLUL 4. FUNDAMENTELE LIMBAJULUI SQL **32**

4.1. Evoluția limbajului SQL 32
4.2. Tipuri de date, operatori și identificatori 33
4.3. Variabile SQL 36
4.4. Funcții SQL 37
4.4.1. Funcții matematice 37
4.4.2. Funcții de agregare 38
4.4.3. Funcții calendaristice (data&time) 38
4.4.4. Funcții șir de caractere 38
4.4.5. Funcții logice 39
4.5. Expresii SQL 39

CAPITOLUL 5. DEFINIREA DATELOR ÎN LIMBAJULUI SQL **41**

5.1. Definirea tabelelor inițiale 41
5.2. Definirea constrângerilor și reacțiilor 42
5.3. Indexarea tabelelor 43
5.4. Modificarea structurii tabelelor 45
5.5. Crearea vizualizărilor 47
5.6. Ștergerea obiectelor bazei de date 48

CAPITOLUL 6. INTEROGĂRI DE SELECȚIE **50**

6.1. Instrucțiunea SELECT 50
6.2. Sortarea tuplurilor în interogări 52
6.3. Filtrarea tuplurilor selectate 54
6.4. Utilizarea valorii Null în interogări 55

CAPITOLUL 7. INTEROGĂRI DE GRUPARE **57**

7.1. Utilizarea funcțiilor de agregare în interogări 57
7.2. Interogări de grupare 57

CAPITOLUL 8. JONCȚIUNI ȘI SUBCONSULTĂRI **61**

8.1. Joncțiuni interne și externe 61
8.2. Interogări cu operatori pe mulțimi 63
8.3. Subconsultări (subqueries) 64
8.4. Interogări corelate 66
8.5. Expresii tabelare comune (CTE) 67

CAPITOLUL 9. INSTRUCȚIUNI DE ACTUALIZARE A DATELOR 70

9.1. Instrucțiunea INSERT 70
9.2. Instrucțiunea DELETE 73
9.3. Instrucțiunea UPDATE 75

CAPITOLUL 10. STRUCTURI DE CONTROL ȘI PACHETE SQL 78

10.1. Structuri de control SQL 78
10.1.1. Instrucțiunea BEGIN…END 78
10.1.2. Instrucțiunea IF…ELSE 78
10.1.3. Instrucțiunea CASE 79
10.1.4. Instrucțiunea WHILE 80
10.1.5. Instrucțiunea RETURN 81
10.1.6. Instrucțiunea GOTO 81
10.1.7. Instrucțiunea WAITFOR 81
10.2. Pachete de instrucțiuni SQL 82

CAPITOLUL 11. SCHEME, PROCEDURI ȘI FUNCȚII UTILIZATOR 84

11.1. Administrarea schemelor 84
11.2. Administrarea sinonimilor 86
11.3. Proceduri stocate SQL 87
11.4. Funcții definite de utilizator 90

CAPITOLUL 12. TRANZACȚII ȘI DECLANȘATOARE SQL 95

12.1. Tranzacții SQL 95
12.2. Declanșatoare SQL 96
12.3. Cursoare SQL 99

CAPITOLUL 13. CICLUL DE VIAȚĂ AL UNEI BAZE DE DATE 104

13.1. Etapele de proiectare și implementare a bazei de date 104
13.2. Etapa de analiză și specificare a cerințelor 105
13.3. Procesul de proiectare a bazei de date 106
13.4. Implementarea și întreținerea bazei de date 109

CAPITOLUL 14. PROIECTAREA CONCEPTUALĂ A BAZEI DE DATE 112

14.1. Modelul Entitate-Relație 112
14.2. Entități, atribute, relații în diagrame E-R 113
14.3. Cardinalitatea relațiilor și atributelor 116
14.4. Identificatori interni și externi 120
14.5. Generalizări și ierarhii 121
14.6. Documentația pentru diagramele E-R 122

CAPITOLUL 15. PROIECTAREA LOGICĂ A BAZEI DE DATE 126

15.1. Reprezentarea entităților în dreptunghi 126
15.2. Reprezentarea relațiilor în notarea Crow's Foot 128
15.3. Capcane de conectare și rezolvarea relațiilor many-to-many 133
15.4. Modelarea datelor istorice 136
15.5. Modelarea generică a bazelor de date 137
15.6. Modelul logic al bazei de date relaționale 139

CAPITOLUL 16. DEPENDENȚE FUNCȚIOANALE 142

16.1. Descompunerea relațiilor cu pierderi de informații 142
16.2. Dependențe funcționale 142
16.3. Forma canonică a dependențelor funcționale 144
16.4. Dependențe multivalorice 145
16.5. Dependențe de joncțiune 146

CAPITOLUL 17. NORMALIZAREA BAZELOR DE DATE 149

17.1. Esența normalizării bazelor de date 149
17.2. Normalizarea prin descompunere a relațiilor 150
17.3. Forme normale 151
17.3.1. Forma normală 1 152

17.3.2. Forma normală 2 ..... *153*
17.3.3. Forma normală 3 ..... *154*
17.3.4. Forma normală Boyce-Codd *156*
17.3.5. Forma normală 4 ..... *158*
17.3.6. Forma normală 5 ..... *159*
17.3.7. Generalizări pentru procesul de normalizare *162*
17.4. Normalizarea relațiilor prin metoda de sinteză 162

CAPITOLUL 18. PROIECTAREA FIZICĂ A BAZELOR DE DATE 164

18.1. Etapele proiectării fizice a bazei de date 164
18.2. Modelul fizic al bazei de date 165
18.3. Analiza CRUD 167

CAPITOLUL 19. GESTIONAREA TRANZACȚIILOR 169

19.1. Tranzacții în baza de date 169
19.2. Tehnici de control al concurenței 173

BIBLIOGRAFIE 175

INTRODUCERE

Aplicațiile informatice moderne gestionează volume imense de date, colectate din diferite domenii de activitate ale omului. Pentru prelucrarea operativă și eficientă a acestor masive de date în timp real sunt necesare metode specifice de organizare a datelor în memoria internă a calculatorului sau în cloud și tehnologii avansate de gestionare a datelor în cadrul rețelelor de calculatoare. Din aceste considerente bazele de date relaționale și obiect-relaționale, în simbioză cu tehnologia client-server sunt utilizate în diferite sisteme informatice ca instrumente eficiente și fiabile de gestionare a volumelor imense de date din diverse domenii de activitate.

În cadrul notelor de curs sunt explicate fundamentele algebrei relaționale și ale modelului relațional de organizare a bazelor de date. De asemenea, sunt prezentate fundamentele limbajului SQL ca instrument pentru crearea și interogarea bazelor de date relaționale. Sunt explicate abordările existente ale procesului de proiectare a unei baze de date relaționale pentru un domeniu de activitate umană și sunt atent analizate etapele ciclului de viață al unei baze de date relaționale. Fiecare etapă a procesului de proiectare a bazei de date este detaliat expusă și analizată în capitole separate ale lucrării. O atenție sporită se acordă studierii dependențelor funcționale existente în masivele de date reale și studierii principiilor și tehnicilor de normalizare a relațiilor. La final sunt examinate succint principiile de bază ale gestionării tranzacțiilor în mediul bazelor de date.

Lucrarea este dedicată studierii disciplinei ”Proiectarea și gestiunea bazelor de date”, inclusă în planurile de învățământ ale specialităților cu profil informatic ale Academiei de Studii Economice a Moldovei. Scopul de bază al lucrării constă în formarea la studenți a unor deprinderi practice de proiectare și gestionare eficientă a bazelor de date relaționale. Lucrarea poate fi utilizată ca suport metodic pentru studiul individual al studenților de la secția de zi și cu frecvență redusă de la ciclul I universitar (licență), care își fac studiile la specialitățile universitare din domeniul tehnologiei informației. În același timp, notele de curs pot servi ca sursă de informare pentru diverse categorii de utilizatori, care doresc să se inițializeze în activitățile de proiectare a bazelor de date relaționale.

Capitolul 1. CONCEPTE FUNDAMENTALE PRIVIND BAZELE DE DATE

1.1. Evoluția organizării datelor în memoria calculatorului

Societatea contemporană este caracterizată în prezent prin afluxul fără precedent de informații de diferit tip, ceea ce necesită strategii și instrumente din ce in ce mai complexe pentru stocare, procesare și, mai ales, interpretare corectă. În acest context, se pune problema transformării informației în date și organizarea acestora într-o asemenea manieră încât în orice moment ele să poată fi extrase cu promptitudine și exactitate.

Datele sunt fapte culese din lumea reală. Ele sunt preluate din măsurători și observații și constituie orice mesaj primit de un receptor sub o anumită formă. O percepție a lumii reale poate fi privită ca o serie de obiecte sau fenomene distincte sau interdependente.

Definim noțiunea de ”**date**” ca înregistrări realizate într-un anumit **cod,** care reflectă o observație, fenomen, imagine, sunet, text etc. Datele pot fi transmise prin canale de telecomunicații sub forma **semnalelor electromagnetice** și înregistrate în **codul binar** pe suporți magnetici. Pentru date este caracteristic că ele au o existență **obiectivă**, deoarece ele reflectă **lumea materială** și apar ca rezultat al interacțiunii cu ea. Datele au o existență **fizică** (se reprezintă in forma unor înregistrări fizice) și sunt **tangibile**, fiind un rezultat al unor măsurări, observații, experimente, etc.

”Informația” reprezintă **semnificația** ce poate fi desprinsă dintr-un ansamblu de date pe baza asociațiilor și corelărilor existente între aceste date. Informația trebuie tratată ca **partea utilizabilă** a unui ansamblu de date, iar datele trebuie considerate drept **materie primă** pentru informație. Fiind obținută în urma prelucrării unui set de date obiective, informația, totuși, are un caracter **subiectiv**, fiind **un produs al creierului uman** (sau al unei aplicații inteligente, proiectate de om). Oameni diferiți pot interpretat același set de date obiective **în mod diferit**, în funcție de nivelul de pregătire profesională, starea emoțională, starea fiziologică, mediul social din care fac parte etc., obținându-se ca rezultat al analizei datelor **informații diferite**. Ca produs al creierului uman informația este **intangibilă** și există doar în creierul omului. În sistemele informatice informația se păstrează sub forma unor date inteligibile pentru om.

Informația este obținută în urma procesului de prelucrare a datelor în cadrul unui **sistem informațional**. Prelucrarea datelor în cadrul sistemului informațional este un **proces creator de valoare.** Acest proces este compus din trei faze:

- ***regruparea, manipularea și organizarea datelor colectate;***
- ***analiza și evaluarea conținutului ansamblului de date;***
- ***prezentarea informației într-o formă necesară utilizatorului.***

Pentru a realiza prelucrarea automatizată a datelor colectate este necesară utilizarea unui model de organizare a datelor în memoria sistemului informatic. ***O mulțime formală și consistentă de reguli, aplicate datelor, definește un model de date***. În cadrul unui model de date un ***ansamblu de date***, organizat după anumite criterii, reprezintă ***o colecție de date***. Modelul de date este utilizat pentru descrierea ***obiectelor*** din lumea reală sau imaginară, precum și a ***asocierilor*** și ***dependențelor*** existente între ele. O ***colecție de obiecte*** care au identitate proprie și sunt caracterizate de o condiție de apartenență la un grup distinct de obiecte se numește ***mulțime***.

Procesul de definire și structurare a datelor în colecții, gruparea lor, stabilirea elementelor de legătură dintre componentele colecției și între colecții reprezintă **o metodă de organizare a datelor.** Evoluția în timp a metodelor de organizare a datelor e legată de soluțiile tehnice de înmagazinare a datelor în memoria internă și externă a calculatorului. Putem identifica în mod cronologic următoarele metode de organizare a datelor:

- ***organizarea datelor în fișiere secvențiale (*****pe benzi magnetice*****);***
- ***organizarea datelor în fișiere cu acces direct (*****pe discuri magnetice*****);***
- ***organizarea datelor în baze de date clasice (*** **sub forma unor fișiere integrate)*****;***
- ***organizarea datelor în baze de date relaționale;***
- ***organizarea datelor în baze de date distribuite.***

1.2. Organizarea datelor în fișiere de date

Fișierul este o **colecție de date** organizate pe criterii calitative, de prelucrare și de scop. Un fișier reprezintă o colecție de date aflate în asociere ce are o denumire și care este reprezentat, de obicei, cu ajutorul unei secvențe de bytes sub forma celor două vederi:

- ***Vederea logică***: *reprezintă felul în care* ***utilizatorul*** *vede fișierul;*
- ***Vederea fizică***: *reprezintă felul în care fișierul este stocat în* ***memoria externă*** *a calculatorului.*

În anii 50-60 ai secolului 20 în cadrul sistemelor informatice datele erau păstrate în fișiere secvențiale stocate pe benzi magnetice. Un exemplu de organizare a sistemului de fișiere în cadrul unei unități social-economice este prezentat în Figura 1.1.

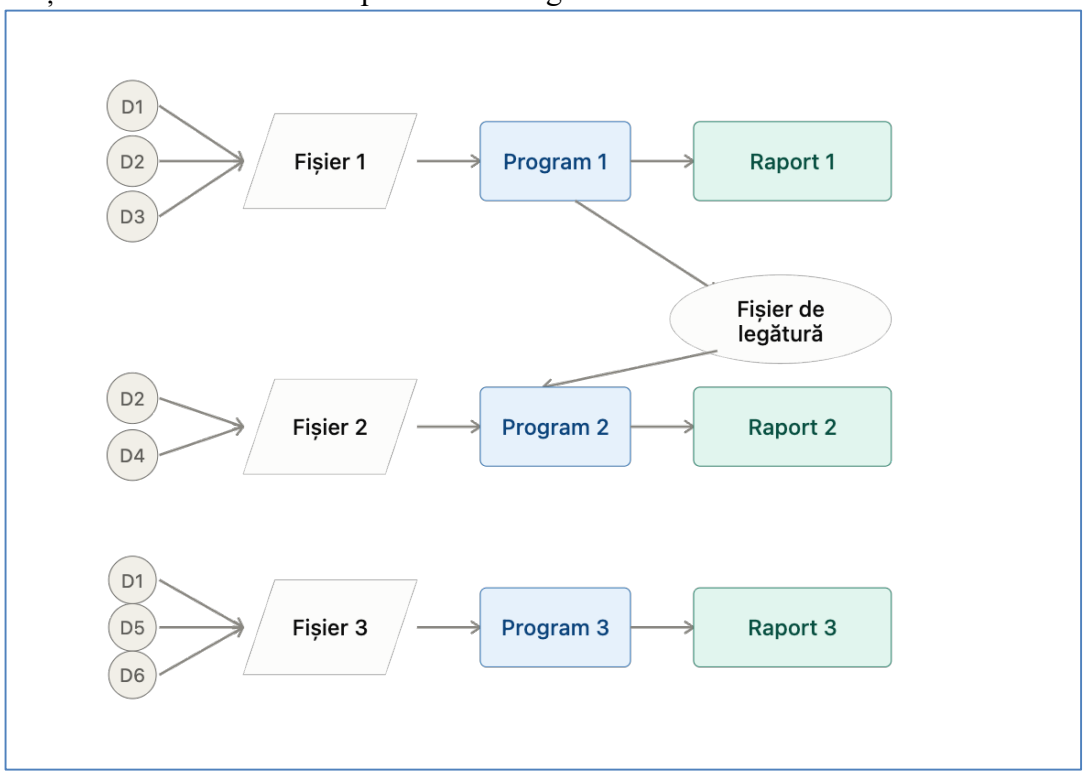


Figura 1.1. Organizarea datelor în fișiere de date

În cazul organizării datelor sub formă de fișiere de date fiecare aplicație-program conținea în codul său definirea structurii **fișierelor de intrare** (Fișier1, Fișier2, Fișier3, etc.) și a **rapoartelor sau formelor de ieșire** (Raport1, Raport2, Raport3 , etc.), precum și a procedurilor de prelucrare a datelor din aceste fișiere. Descrierea structurii datelor din fișiere (D1, D2, D3, etc.) era parte componentă a codului aplicației. Orice modificare în componența și structura fișierelor trebuia să fie reflectată în codul programului respectiv. Utilizarea rezultatelor obținute de un program de către alt program necesita descrierea în codul ambelor aplicații a structurii și procedurilor de prelucrare a datelor pentru un **fișier de legătură**.

Această metodă de organizare a datelor avea un șir de neajunsuri majore:

- *Redundanța datelor se manifesta prin înscrierea acelorași date în fișiere diferite (de exemplu, datele D1 și D2), fapt care genera condiții de apariție a erorilor la actualizarea acestor date.*
- *Dificultatea accesării datelor era determinată de necesitatea accesării lor prin intermediul aplicației ce a creat fișierul.*
- *Izolarea datelor determina apariția situațiilor când una și aceeași dată se reprezenta in formate diferite în fișiere diferite, generând erori logice sau de sintaxă în procesul prelucrării datelor.*

- *Complexitatea actualizării era determinată de redundanța datelor și necesita realizarea modificărilor în toate aplicațiile și fișierele în care apar datele supuse modificărilor.*
- *Nivel scăzut de securitate a datelor la nivel de fișiere, care puteau fi ușor șterse din sistemul de fișiere ale calculatorului.*
- *Costul ridicat de exploatare.*
- *Inflexibilitatea față de modificările ulterioare în cadrul întregului sistem informatic.*

1.3. Organizarea datelor în baze de date

- *Definiția 1. O bază de date reprezintă **un ansamblu structurat de fișiere**, ce reprezintă colecții de date aflate în interdependență, împreună **cu descrierea globală** a datelor și a relațiilor existente între ele.*
- *Definiția 2. O bază de date reprezintă **un ansamblu** de date structurate, coerente, neredundante, independente de orice program, direct accesibile după mai multe criterii*

Baza de date se caracterizează printr-o **structură (schemă)** și prin **conținut (instanță**). **Structura bazei de date** este relativ **constantă în timp** și reprezintă totalitatea obiectelor unei BD împreună cu relațiile existente între ele și constrângerile aplicate datelor. **Conținutul** unei baze de date reprezintă datele ce se păstrează în BD într-un moment dat de timp și el este **volatil**. Bazele de date sunt create cu ajutorul unor aplicații specializate numite **sisteme de gestiune a bazelor de date (SGBD).**

Pentru prima dată conceptul unei baze de date (Figura 1.2) a fost propus în anul 1969 de organizația **CODASYL** (*the Conference/Committee on Data Systems Languages* – un consorțiu format în 1959 pentru a ghida dezvoltarea standardelor pentru limbaje de programare. În octombrie 1969 sunt publicate primele specificații tehnice pentru **modelul bazei de date de rețea** (*network database model*), care devin cunoscute sub numele CODASYL Data Model. În aceste specificații sunt definite limbaje separate: **limbaj pentru definirea datelor** (Data Definition Language – DDL), utilizat pentru definirea schemei bazei de date și a viziunilor asupra ei, și **limbajul de manipulare cu datele** (Data Manipulation Language – DML), folosit pentru definirea instrucțiunilor limbajului de programare COBOL pentru interogarea și actualizarea bazei de date.

În cadrul CODASYL Data Model baza de date **este independentă** de aplicațiile-program, care interacționează cu ea prin intermediul unei structuri, denumite **dicționar de date** sau **catalogul bazei de date**. În dicționarul de date se păstrează informațiile referitoare la **obiectele** bazei de date. Aceste informații poartă denumirea de **metadate ale bazei de date** și ele descriu:

- ***denumirile tabelelor;***
- ***denumirile câmpurilor tabelelor;***
- ***domeniile de valori și lungimea câmpurilor din tabele;***
- ***denumirile și definițiile vederilor utilizatorilor;***
- ***date referitoare la indecșii folosiți în cadrul fiecărui tabel;***
- ***constrângerile de integritate (informații despre cheia primară și cheia externă);***
- ***date despre utilizatorii sistemului (numele și condițiile de acces ale acestora);***
- ***date descriptive și statistice (numărul de înregistrări din fiecare tabel, metoda de stocare folosită în cazul fiecărui tabel (de exemplu, în grup).***

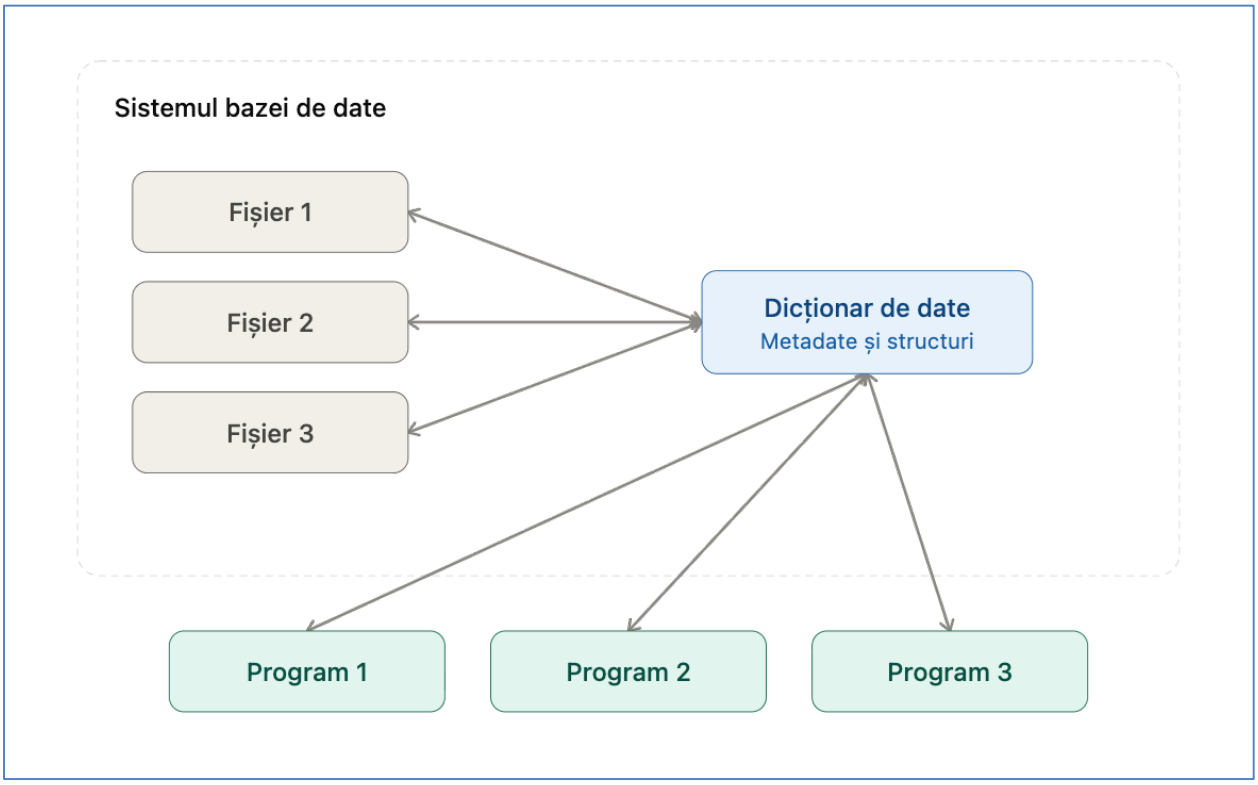


Figura 1.2. Structura funcțională a unei baze de date (CODASYL, 1969)

1.4. Nivele de abstracție și independența datelor

Bazele de date se modifică des în decursul exploatării. Datele aflate într-o bază de date la un anumit moment dat alcătuiesc o **instanță** a acelei baze de date. Proiectul general al bazei de date este denumit **schema** bazei de date.

O **schemă** reprezintă o descriere a datelor conform **modelului** de date propus. **Schema bazei de date** reprezintă descrierea generală a bazei de date și conține:

- ✓ *informațiile fizice de proiectare;*
- ✓ *informații referitoare la utilizator;*
- ✓ *descrieri de nivel înalt ale tranzacțiilor și aplicațiilor, și a legăturilor utilizatorilor cu ele;*
- ✓ *relațiile dintre date și tranzacții;*
- ✓ *statistici de utilizare.*

Pot fi evidențiate **trei nivele** de abstractizare a schemelor unei baze de date: **intern, logic și extern** (Figura 1.3).

Schema externă se află la **nivel superior** și corespunde **unei valori a datelor**. Ea descrie **interpelările utilizatorilor** (queries) bazei de date, ce se folosesc într-o anumită aplicație și ea corespunde **schemei conceptuale**. **Schema externă** reprezintă **vederea utilizatorilor** asupra datelor.

Schema conceptuală (logică) corespunde **nivelului conceptual** și descrie articolele, relațiile și constrângerile dintre ele. Ea este o **descriere abstractă și integrată a tuturor datelor**, independentă de sistemul de gestiune a bazelor de date folosit, și trebuie să corespundă **schemei interne**. Schema reprezintă perspectiva selectării sistemului de gestiune a bazelor de date.

Schema internă se află la **nivel inferior** și conține **definițiile** tuturor înregistrărilor stocate în baza de date, **metodele** de reprezentare, **câmpurile** și **indecșii** datelor. Ea descrie modul de stocare fizică a datelor, precum și structurile de acces la date, și reprezintă perspectiva realizării bazei de date.

Sistemul de gestiune al bazelor de date (SGBD) efectuează corespondențe între cele trei tipuri de scheme pentru a le corela. Dacă se produce o modificare la nivel fizic, **schema internă** trebuie modificată, dar **schema conceptuală** poate rămâne **neatinsă**.

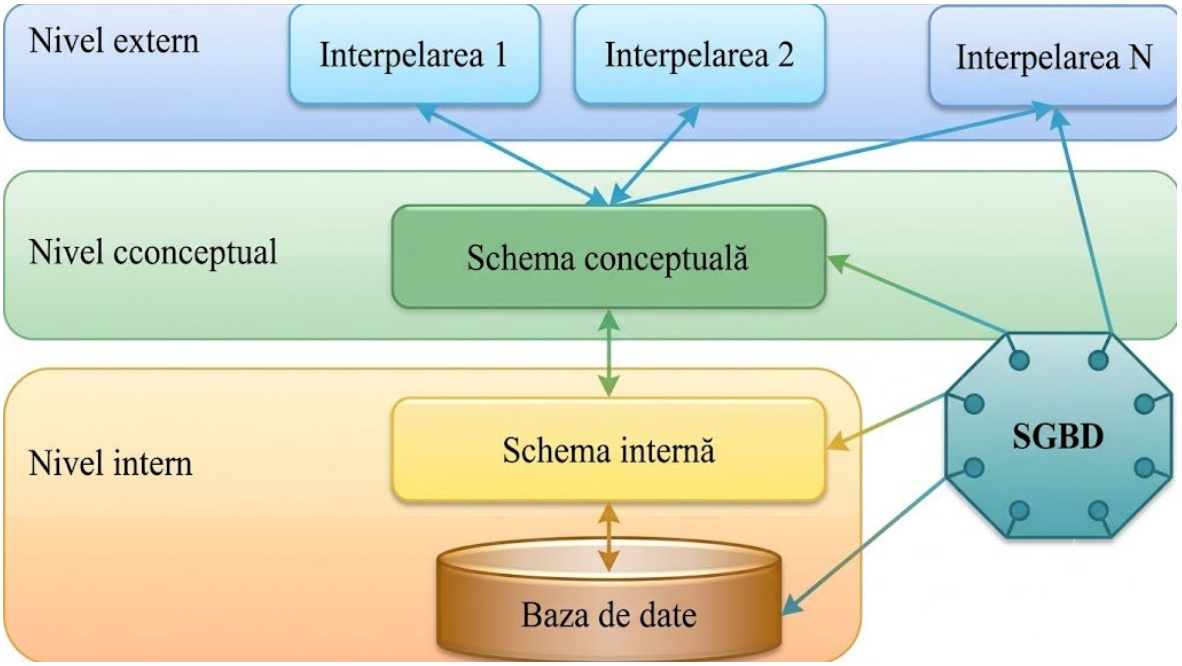


Figura 1.3. Nivelele de abstractizare a schemelor unei baze de date

Corespondențele efectuate între **interpelările utilizatorilor** (nivelul extern) și **stocarea fizică a datelor** (nivelul intern) ajută la **ascunderea complexității** nivelului fizic, ceea ce face să crească **flexibilitatea** și **posibilitățile** de adaptare a bazei de date. SGBD trebuie să verifice dacă **fiecare schemă externă** poate fi derivată din **schema conceptuală**. Pentru a stabili corespondența dintre schema **externă** și cea **internă**, sistemul trebuie să folosească informațiile din **schema conceptuală**.

Independența datelor reprezintă faptul că nivelele superioare **nu sunt afectate** de modificările făcute la nivelele inferioare (Figura 1.3). Aceasta înseamnă și faptul că interpelarea **unui utilizator** (schema externă) este **complet independentă** de interpelarea **altui utilizator**, ceea ce are un efect extrem de favorabil în cazul modificărilor efectuate de către administratorul bazei de date (ce pot apare ca urmare a cererilor venite din partea utilizatorului respectiv, ca urmare a necesităților de adaptare a sistemului la noile condiții sau ca urmare a dorinței de optimizare a funcționării sistemului), care poate face modificări ale interpelării unui singur utilizator fără a afecta interpelările celorlalți utilizatori.

Conform arhitecturii propuse de ANSI (American National Standards Institute) se identifică două nivele de independență a datelor:

- ✓ ***Independența logică de date;***
- ✓ ***Independența fizică de date.***

Independența logică de date se referă la **imunitatea** schemelor **externe** față de modificările efectuate în **schema conceptuală**. Modificările din schema conceptuală pot fi:

- ***adăugarea sau eliminarea de noi entități;***
- ***adăugarea sau eliminarea de atribute;***
- ***adăugarea sau eliminarea de relații.***

Independența logică de date înseamnă că se pot face modificări în schema conceptuală fără a fi necesară schimbarea schemei externe existente sau rescrierea programelor de aplicație. Modificările nu trebuie să afecteze toți utilizatorii, ci doar pe acei pentru care s-au făcut. Aceștia trebuie informați de acest lucru.

Independența fizică de date se referă la **imunitatea** schemei **conceptuale** față de modificările efectuate în **schema internă**. Modificările din schema internă pot fi:

- ***organizare diferită a fișierelor;***
- ***structuri de stocare diferite;***
- ***dispozitive de stocare diferite;***

- ***indecşii sau algoritmi de calcul diferiţi.***

Modificările făcute în schema internă se fac cu scopul **îmbunătăţirii performanţelor** bazei de date.

Bazele de date relaţionale, datorită limbajelor relaţionale pe care le folosesc, oferă un înalt grad de independenţă faţă de date.

1.5. Sistemul de gestiune a bazelor de date

Sistemul de gestiunea a bazelor de date (SGBD**)** este un pachet de programe specializate, care permit utilizatorului să interacţioneze cu o **bază de date** (BD) în vederea creării, actualizării şi interogării acesteia (Figura 1.4).



Figura 1.4. Interacţiunea utilizatorilor cu baza de date

Un sistem de gestiune a bazelor de date are următoarele **funcţii de bază** :

- *Manipularea cu datele stocate în baza de date;*
- *Asigurarea tranzacţiilor cu datele stocate;*
- *Controlul integrităţii bazei de date în procesul exploatării ei;*
- *Asigurarea serviciilor de reconstituire a bazei de date în cazul deteriorării ei;*
- *Asigurarea confidenţialităţii datelor stocate şi prelucrate în sistem;*
- *Asigurarea accesului simultan la date a utilizatorilor autorizaţi;*
- *Asigurarea autorizării utilizatorilor şi securităţii în funcţionare a sistemului;*
- *Asigurarea serviciilor de comunicare şi utilitare în cadrul sistemului.*

Avantajele sistemelor de gestiune a bazelor de date în comparaţie cu sistemele informatice clasice bazate pe organizarea datelor sub formă de fişiere sunt următoarele:

- **Controlul redundanţei datelor** reprezintă existenţa unor mecanisme interne ale SGBD care asigură lipsa repetărilor inutile de date în tabelele bazei de date;
- **Coerenţa datelor** înseamnă că valoarea articolului e aceeaşi pentru toate copiile sale din fişiere diferite;
- **Partajarea datelor** înseamnă că datele pot fi utilizate în timpul unei sesiuni de lucru cu baza de date de către mai mulţi utilizatori în acelaşi timp;
- **Integritatea crescută a datelor** înmagazinate în baza de date este asigurată prin utilizarea constrângerilor (regulilor de coerenţă). Constrângerile se pot aplica articolelor de date din cadrul unei singure înregistrări sau relaţiilor dintre înregistrări;
- **Securitatea crescută a bazei de date** se realizează prin atribuirea unor nume de utilizatori şi parole ce permit identificarea persoanelor autorizate să folosească baza de date şi care impun modalitatea de utilizare a datelor;
- **Aplicarea standardelor internaţionale** referitoare la utilizarea formatului pentru date, convenţiilor privind denumirile, documentarea, procedurile de reactualizare, regulile de acces etc. asigură compatibilitate bazei de date cu programele de aplicaţii;
- **Reducerea costurilor de elaborare şi exploatare a BD;**
- **Rezolvarea conflictelor -** administratorul bazei de date poate lua decizii ce duc la utilizarea optimă a resurselor sistemului;
- **Creşterea accesibilităţii datelor şi a capacităţii de răspuns** se asigură prin utilizarea limbajelor de programare din generaţia a IV-a (SQL, QBE, etc.);

- **Creşterea productivităţii sistemului** este determinată de existenţa unor funcţii, ce permit manipularea eficientă a fişierelor, şi de utilizarea limbajelor de programare din generaţia a IV-a, ce reduc mult timpul de programare a aplicaţiilor;
- **Independenţa datelor** este un rezultat firesc al faptul că descrierile datelor sunt separate de codul aplicaţiilor care le utilizează;
- **Controlul concurenţei** este îmbunătăţit considerabil şi el garantează că dacă doi sau mai mulţi utilizatori accesează simultan aceleaşi date, atunci ele nu se pierd sau nu se alterează integritatea acestora;
- **Asigurarea salvării de siguranţă şi a refacerii** asigură recuperarea automată a ultimei stări coerente a bazei de date în cazul apariţiei unei defecţiuni hardware sau software.

Sistemele de gestiune a bazelor de date sunt alcătuite dintr-o serie de module (Figura 1.5) ce îndeplinesc diverse funcţionalităţi. Anumite funcţionalităţi sunt îndeplinite împreună cu sistemul de operare pe care este folosit sistemul respectiv.

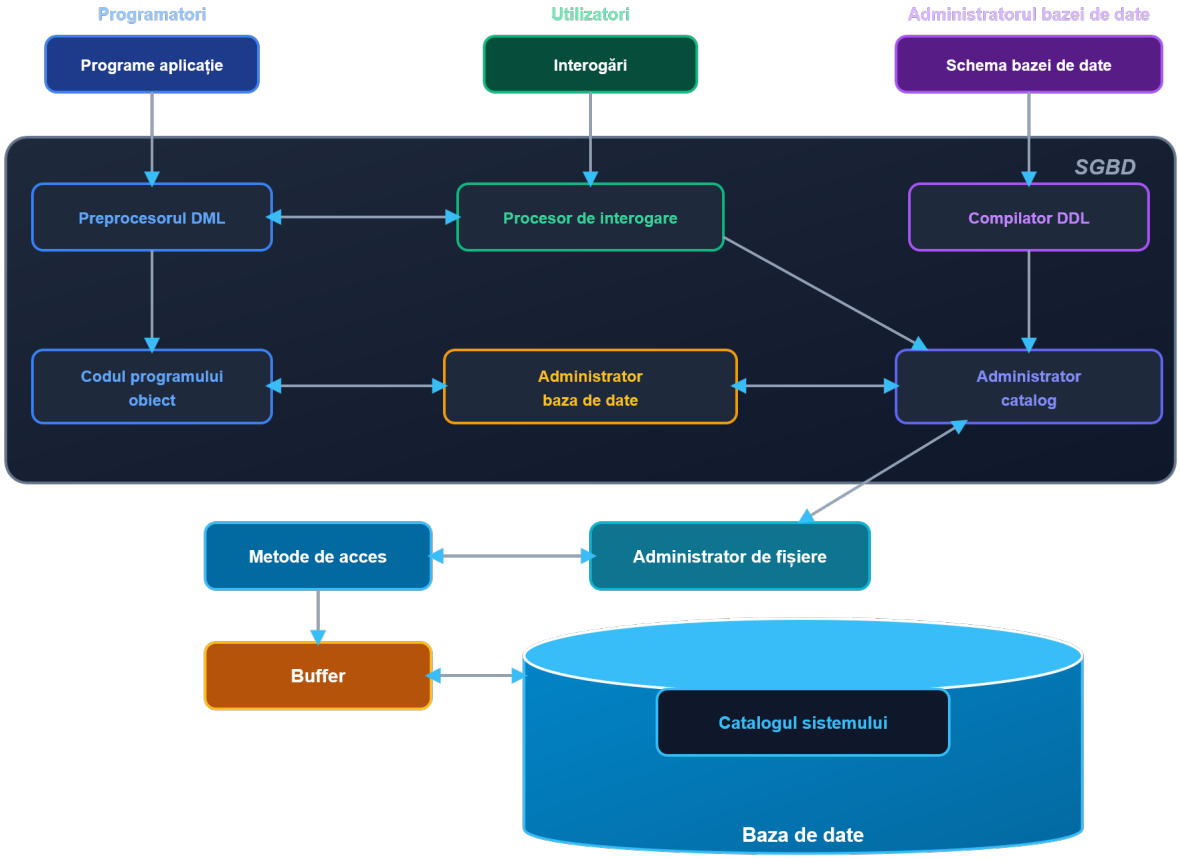


Figura 1.5. Componentele funcţionale ale unui SGBD

Principalele componente ale unui sisteme de gestiune al bazelor de date sunt (Figura 1.5):

- **Administratorul de fişiere** gestionează alocarea spaţiului pe disc, precum şi structurile de date utilizate la reprezentarea datelor pe disc;
- **Administratorul bazei de date** acceptă interogările şi examinează schemele externe şi conceptuale pentru a determina ce înregistrări sunt necesare pentru a satisface o anumită cerere, după care apelează administratorul de fişiere pentru a executa cererea;
- **Procesorul de interogare** transformă interogările într-o serie de instrucţiuni de nivel jos adresate administratorului de baze de date;
- **Preprocesorul DML** converteşte instrucţiunile limbajului de manipulare cu datele DML (*Data Manipulation Language*) dintr-un program aplicaţie în apeluri de funcţii standard ale limbajului gazdă şi interacţionează cu procesorul de interogare pentru a genera codul corespunzător;

- **Compilatorul DDL** transformă instrucțiunile limbajului de descriere a datelor DDL (*Data Definition Language*) într-un set de tabele ce conțin meta-datele. Tabelele sunt stocate în catalogul sistemului;
- **Administratorul de catalog** gestionează accesul și întreținerea catalogului de sistem. Catalogul sistemului este accesat de majoritatea componentelor sistemului de gestiune al bazei de date.

1.6. Limbaje ale bazelor de date

Pentru a elabora o bază de date un utilizator trebuie să:

- *definească schema bazei de date;*
- *aplice o colecție de operatori pentru a crea, înmagazina, extrage și modifica datele.*

Un sistem de gestiune a bazelor de date obișnuit trebuie să ofere o serie de instrumente care să ușureze activitățile specificate anterior, având următoarele componente:

- un **limbaj de definire a datelor** (*Data Definition Language* – ***DDL***), utilizat la definirea schemei bazei de date.
- un **limbaj de manipulare a datelor** (*Data Manipulation Language* – ***DML***), care permite utilizatorului manipularea obiectelor bazei de date și a relațiilor dintre acestea, în contextul schemei bazei de date.

Aceste limbaje pot varia de la un producător la altul în cadrul modelului pe care aceștia îl folosesc datorită aspectelor legate de complexitate, funcționalitate și ușurința în exploatare (interfața utilizator). De multe ori aceste limbaje sunt denumite **sublimbaje de date**, deoarece nu includ construcții pentru toate necesitățile de calcul, cum sunt cele asigurate de limbajele de nivel înalt. Majoritatea sistemelor de gestiune a bazelor de date au încorporat sublimbajul într-un limbaj de programare de nivel înalt (ex. C++). În acest caz limbajul de nivel înalt este numit **limbaj gazdă**.

Limbajul de definire a datelor permite administratorului bazei de date sau utilizatorului să descrie și să denumească entitățile din baza de date și relațiile ce pot exista între diferitele entități. Limbajul de definire a datelor reprezintă **o colecție de instrucțiuni** utilizate pentru descrierea tipurilor de date și definirea structurii bazei de date. Acesta este utilizat pentru a defini **o schemă a bazei de date** sau pentru **a modifica** una existentă.

Rezultatul compilării instrucțiunilor din limbajul de definire a datelor reprezintă **un set de tabele stocate** în fișiere speciale denumite **cataloage de sistem**. Catalogul de sistem conține **metadatele** (datele care descriu obiectele din baza de date). **Metadatele** conțin definiții ale înregistrărilor datelor și altor obiecte cerute de sistemul de gestiune a bazei de date. Sistemul de gestiune a bazei de date consultă mai întâi catalogul de sistem pentru a accesa corect datele.

Limbajul de definire a datelor conține comenzi necesare următoarelor operații: *definirea schemelor de relație; eliminarea relațiilor; crearea indecșilor; modificarea schemelor*.

Limbajul de definire a datelor permite specificarea următoarelor informații necesare în orice bază de date: *schema fiecărei relații din baza de date; domeniul de valori asociat fiecărui atribut; constrângerile de integritate; setul de indecși care se creează pentru fiecare relație în parte; informații referitoare la securitatea sistemului și la modul de acces la acesta; structura de înmagazinare fizică pe disc a datelor.*

Limbajul de manipulare a datelor asigură un set de procedee ce permit operații de bază pentru **manipularea datelor** din baza de date. Limbajul de manipulare a datelor asigură **o colecție de operatori** ce pot fi aplicați pentru a valida instanțele tipurilor de date în cadrul schemei și de a crea, modifica, șterge sau extrage astfel de instanțe.

Există **două tipuri** de limbaje de manipulare a datelor:

- **Limbaje procedurale de manipulare a datelor** (de exemplu, *limbajul Cobol cu setul de instrucțiuni DL/1*, specific modelului de date ierarhic) permit utilizatorului să comunice sistemului ce date sunt necesare și cum pot fi ele **exact** regăsite. Aceste limbaje prelucrează informația înregistrare cu înregistrare.

- **Limbaje neprocedurale de manipulare a datelor** (de exemplu, *limbajul SQL sau limbajul QBE,* specifice modelului relațional de date) permit utilizatorului să comunice sistemului ce date sunt necesare fără a specifica cum se regăsesc datele. Aceste limbaje prelucrează informația pe seturi de înregistrări și conferă o mai mare independență de date.

Întrebări de control:

1. Care este diferența principală dintre „date" și „informație" din punct de vedere al naturii lor (existență fizică/obiectivă versus caracter subiectiv/intangibil)?
2. Care au fost principalele neajunsuri majore ale metodei de organizare a datelor în fișiere de date folosite în anii '50-'60?
3. Cum a definit organizația CODASYL conceptul de bază de date în anul 1969 și ce rol are dicționarul de date (catalogul) în această structură?
4. Care sunt cele trei nivele de abstractizare ale schemelor unei baze de date și ce reprezintă fiecare?
5. În ce constă diferența dintre independența logică de date și independența fizică de date conform arhitecturii ANSI?
6. Care sunt funcțiile de bază ale unui Sistem de Gestiune a Bazelor de Date (SGBD)?
7. Care sunt principalele componente funcționale ale unui SGBD și ce rol îndeplinesc modulul *Compilator DDL* și *Preprocesorul DML*?
8. Care este diferența dintre un Limbaj de Definire a Datelor (DDL) și un Limbaj de Manipulare a Datelor (DML)
9. Care este diferența dintre limbajele DML procedurale și cele neprocedurale?

Capitolul 2. MODELE DE BAZE DE DATE

2.1. Modele de date

Înaintea construirii unei baze de date este necesară elaborarea **unui model de date** utilizat pentru reprezentarea datelor. Un ***model*** este o abstractizare și o structură ce simbolizează **toate caracteristicile entităților esențiale** ce prezintă interes pentru utilizator, o reprezentare și o reflectare a lumii reale.

Un ***model de date*** reprezintă **o colecție integrată de concepte** necesare descrierii datelor, relațiilor dintre ele, precum și a constrângerilor asupra datelor. Modelul de date este utilizat la descrierea schemei bazei de date, definind structura datelor, legăturile dintre acestea, semantica lor, precum și constrângerile impuse, deși nu este obligatoriu ca întotdeauna acestea să fie regăsite în orice model de date.

Modelele de date oferă înțelegerea descriptivă necesară definirii schemelor logice și externe, și sunt utile descrierii formale a schemei bazei de date. **Schema logică** a unei baze de date reprezintă o descriere abstractă a unei porțiuni din realitatea modelată împreună cu instanțele bazei de date.

Un **model de date** este alcătuit din **trei elemente de bază**: ***entități, atribute, relații.***

- ***Entitatea*** *reprezintă un obiect sau concept din lumea reală sau imaginară.*
- ***Atributul*** *reprezintă o caracteristică distinctă a unei entități.*
- ***Relația*** *reprezintă o interacțiune între două sau mai multe entități.*

Modelele de date corespund celor trei nivele de abstractizare: extern, conceptual și intern.

Modelul extern de date este utilizat pentru a reprezenta interpelările fiecărui utilizator. Acest model mai este cunoscut și sub denumirea de Univers al Discursului. Modelul extern de date este reprezentat prin ***modelele de date bazate pe înregistrări***.

Modelul conceptual de date reprezintă o vedere logică asupra tuturor obiectelor bazei de date și a relațiilor existente între ele. Acest model este independent de sistemul de gestiune a bazelor de date ales și se reprezentă prin ***modele de date bazate pe obiecte***.

Modelul intern de date permite transformarea schemei conceptuale într-o formă, ce poate fi înțeleasă de către sistemul de gestiune a bazelor de date. Modelul de date intern se bazează pe ***modele de date fizice***.

Fiecare model de date are propria reprezentare a datelor, dar întotdeauna un model este alcătuit dintr-o **intensie** și o **extensie**. ***Extensia*** unei baze de date se referă la setul curent de înregistrări pe care îl conține. Acest set de înregistrări nu rămâne același tot timpul existenței bazei de date, suferind diverse modificări, pe măsura introducerii, actualizării sau ștergerii de date. Partea cu caracter **permanent** în cadrul unei baze de date o reprezintă ***intensia*** sau ***schema bazei de date***. **Intensia** bazei de date descrie **structura tuplurilor** unei relații. Operațiile limbajului de manipulare a datelor pot fi efectuate numai în condițiile în care se cunoaște această structură.

Modele de date bazate pe înregistrări descriu datele la nivel conceptual. Spre deosebire de modelele orientate pe obiecte, acestea sunt folosite cu scopul de a specifica structura logică generală a bazei de date și de a oferi un nivel ridicat al descrierii implementării. Modelele sunt denumite în acest fel deoarece baza de date este alcătuită din înregistrări de același tip. Fiecare tip de înregistrare are un **număr fix de câmpuri**, fiecare câmp având, de obicei, **o lungime fixă**, ceea ce duce la simplificarea reprezentării. Aceste modele nu oferă un mecanism de reprezentare directă a codului din baza de date. Pentru a efectua interogări și actualizări asupra bazei de date se folosesc o serie de limbaje individuale separate asociate modelului. Din această categorie de modele fac parte: ***modelul ierarhic de date, modelul reticular de date, modelul relațional de date.***

Modelul ierarhic a fost primul model de date ce a fundamentat un sistem de gestiune a bazelor de date și a fost dezvoltat de către firma IBM pentru produsul său IMS, care utiliza limbajul DL/1. Modelul ierarhic lucrează cu grupuri repetitive prin utilizarea unei structuri de date ce se

bazează pe parcurgerea de sus în jos a unui arbore: datele aflate în înregistrările primare reprezintă ramurile arborelui, în timp ce datele ce formează grupurile repetitive reprezintă frunzele acestuia.

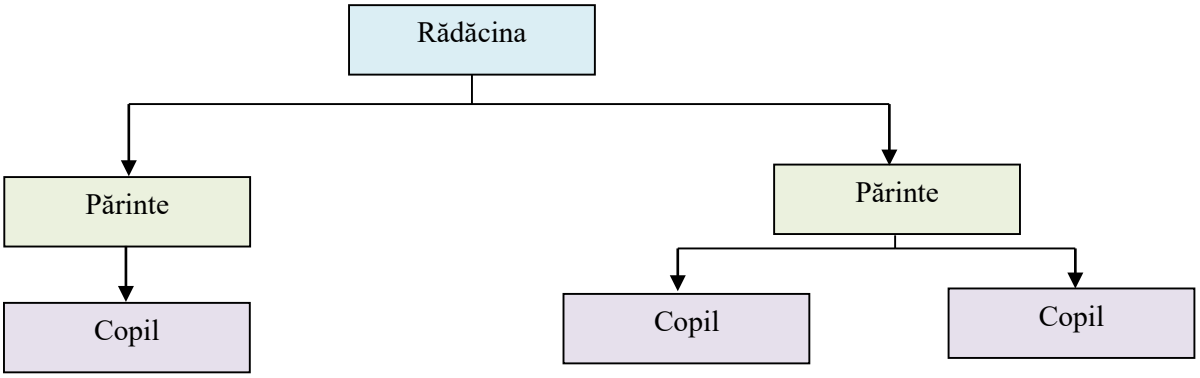


Figura 2.1. Structura unui arbore ierarhic de date

Intensia modelului ierarhic de date este reprezentată cu ajutorul unui ***arbore de definiție,*** ce reprezintă o diagramă a structurii de date în care sensul legăturilor funcționale este întotdeauna **de la nodul părinte către nodul copil** (Figura 2.1). O astfel de diagramă este un graf orientat, alcătuit cu scopul reprezentării tipurilor de entități și a relațiilor dintre acestea. Nodurile grafului corespund tipurilor de entități, iar arcele grafului reprezintă legăturile funcționale dintre tipurile de entități.

Extensia modelului ierarhic de date reprezintă **un tabel** în care fiecare linie este o înregistrare, ce corespunde unei instanțieri a tipului de entitate. În tabele **sunt permise duplicatele** și, prin urmare, pot exista două instanțieri identice ale aceluiași tip de entitate. Un singur tabel din baza de date are rolul de rădăcină a arborelui, în timp ce restul tabelelor formează mulțimea părinților și copiilor arborelui.

O **relație** într-o bază de date ierarhică este reprezentată prin intermediul **perechii părinte/copil**. În acest tip de relație, tabelul părinte poate fi asociat cu unul sau mai multe tabele copil, dar un singur tabel copil poate fi asociat doar cu un singur tabel părinte. Aceste tabele sunt asociate în mod explicit cu ajutorul unor **pointeri** sau pe baza unui **aranjament fizic al înregistrărilor în tabele.** Utilizatorul accesează datele pornind din rădăcina arborelui și parcurge un anumit drum unic până ajunge la datele căutate. O astfel de metodă de acces cere utilizatorului o foarte bună cunoaștere a structurii bazei de date.

Avantajul modelului ierarhic este acela că metodele folosite la regăsirea înregistrărilor asociate din baza de date sunt **mai simple** decât cele folosite în **modelul reticular**. Un alt avantaj al utilizării bazelor de date ierarhice este acela că utilizatorul poate extrage datele foarte rapid datorită legăturilor explicite (pointeri) definite în structura tabelelor. Integritatea referențială se obține prin crearea structurii și nu poate fi încălcată, ceea ce face ca o înregistrare din tabelul copil să fie obligatoriu asociată unei înregistrări existente în tabelul părinte, iar o înregistrare ștearsă din tabelul părinte să impună eliminarea tuturor înregistrărilor asociate din tabelul copil. **Dezavantajul** acestui tip de bază de date este că ea nu poate suporta asocierile complexe și, de aceea, deseori sunt probleme referitoare la **redundanța datelor**, deoarece este posibil să-i fie permisă introducerea de date inconsistente.

Modelul reticular (de rețea) permite elaborarea structuri unei baze de date de tip rețea, utilizând în acest scop conceptele de **noduri** și **seturi**. Un **nod** reprezintă o colecție de înregistrări, în timp ce un **set** stabilește și reprezintă relațiile din cadrul unei bazei de date de tip rețea. O astfel de construcție transparentă relaționează o pereche de noduri prin utilizarea unuia dintre ele sub denumirea de **proprietar**, iar a celuilalt sub denumirea de **membru** (Figura 2.2).

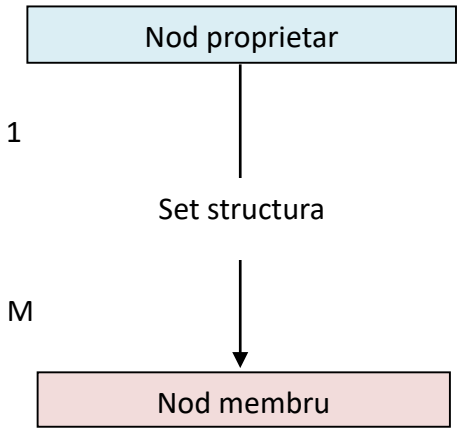


Figura 2.2. Structura de tip set

Structura de tip set este o construcție ce stabilește și reprezintă o relație din cadrul bazei de date de rețea și este o îmbunătățire remarcabilă față de relația părinte/copil. O astfel de structură suportă o **relație de unu-la-mulți**, ceea ce înseamnă faptul că o înregistrare din nodul proprietar poate fi relaționată cu una sau mai multe înregistrări ale nodului membru, dar unei singure înregistrări din nodul membru îi este asociată o singură înregistrare din nodul proprietar. Mai mult decât atât, o înregistrare a nodului membru nu poate exista fără să fie asociată unei înregistrări existente în nodul proprietar.

Între o pereche de noduri **se pot defini unul sau mai multe seturi**, iar un singur nod poate fi implicat în seturi cu alte noduri din baza de date. Utilizatorul poate accesa date din cadrul unei baze de date de tip rețea prin cea mai potrivită structură de seturi. Spre deosebire de bazele de date ierarhice, în care accesul trebuie să înceapă cu nodul rădăcină, în bazele de date de tip rețea utilizatorul poate accesa datele indiferent de nod pe baza structurilor de tip set. ***Intensia*** modelului de baze de date de tip rețea este un ***graf cu arce numerotate*** pentru a indica drumul ce trebuie parcurs, deoarece un tip de entitate copil poate fi conectat la mai multe tipuri de entități părinte sau la același tip de entitate părinte prin mai multe arce. ***Extensia*** acestui model este un ***tabel*** ce permite introducerea de înregistrări duplicat, dar înregistrările pot fi ordonate.

Bazele de date de tip rețea au performanțe excelente în cazul regăsirii seturilor de înregistrări ce aparțin unui anumit obiect, deoarece relațiile dintre înregistrări (pointeri) reprezintă o parte constitutivă a bazei de date. În același timp, se permite utilizatorilor crearea de interogări mult mai complexe decât cele ce se pot elabora prin intermediul bazelor de date ierarhice, dar viteza de consultare a bazelor de date de tip rețea scade atunci, când se dorește căutarea înregistrărilor pe baza unor criterii specificate.

Principalul dezavantaj al acestui tip de baze de date este legat de faptul că utilizatorul este obligat să **cunoască foarte bine structura bazei de date** pentru a se putea descurca cu structurile de seturi. Totodată, aplicațiile ce lucrează cu astfel de baze de date (în principal programele COBOL) trebuie să actualizeze atât valorile datelor, cât și pointerii înregistrărilor ce se adaugă, șterg sau se modifică. Necesitatea actualizării secvențiale atât a datelor, cât și a pointerilor duce la creșterea complexității proceselor în care sunt implicate tranzacții. Un alt dezavantaj este acela că nu este ușoară modificarea structurii bazei de date fără a afecta programele aplicație care lucrează cu aceasta. Deoarece, așa cum s-a arătat, o relație este definită în mod explicit sub forma unei structuri de tip set, aceasta nu poate fi modificată fără a afecta programele aplicație ce folosesc această structură la căutarea datelor.

2.2. Modelul relațional de date

Modelul relațional de date a fost dezvoltat în anul 1970 de către E. F. Codd și este un model de tip **entitate-relație,** bazat pe elaborarea unui **model conceptual**. Modelul relațional este simplu și are un **suport teoretic solid.** În cadrul modelului se utilizează un număr mic de concepte care se referă la **relații** (structuri de date bidimensionale ce au proprietăți speciale), **rânduri** (datele aflate în cadrul relațiilor), **coloane** (câmpurile datelor din rândurile corespunzătoare) și **chei** (mecanisme de identificare și asociere a rândurilor aflate în unul sau mai multe tabele). Modelul relațional se bazează pe **teoria matematică a mulțimilor**, ceea ce înseamnă faptul că toate operațiile sunt încheiate cu succes, iar rezultatele operațiilor sunt **predictibile**.

Componente de bază ale modelului relațional sunt:

- **componenta de *structură* a datelor** (relații cu proprietăți speciale);
- **componenta de *manipulare* a datelor** (operații predefinite prin care tehnologia relațională folosește un optimizator inteligent pentru a găsi calea de acces la date);
- **componenta de *integritate* a datelor** (reguli necesare protecției datelor la efectuarea unor operații incorecte).

Principalul avantaj al modelului relațional este acela că nu este necesară utilizarea atât a **pointerilor,** cât și a **datelor în cadrul tabelelor**, folosind în schimb ***relații*** pentru a accesa valori corespondente din mai multe tabele. ***O relație*** constă dintr-o **asociere între înregistrările aflate în două tabele ce au aceleași valori ale atributelor comune**. Deoarece tabelele relaționale nu conțin pointeri, datele aflate în astfel de tabele sunt independente de metodele folosite de către sistemul de gestiune a datelor în lucrul cu înregistrările tabelelor. **Baza de date relațională** constă din una sau mai multe **relații (**tabele).

Intensia modelului relațional este **o schemă relațională** cu una sau mai multe **scheme de relație**. **Fiecare schemă** de relație are **propriul nume** și **propriile atribute**. ***Extensia*** modelului relațional este **un tabel ce nu permite înregistrări duplicat.** Fiecare schemă de relație introduce un tabel în schema relațională. Modelul de date relațional folosește **tabele bidimensionale,** care reprezintă entitățile. O coloană reprezintă un **atribut** al unei entități și mai poartă denumirea de **câmp** sau **proprietate**. Un rând reprezintă **un tuplu,** care este o instanță a unui tip de entitate sau de relație, sau orice altceva din baza de date. De obicei una dintre coloanele tabelului este numită **cheie primară** și ea acceptă doar **valori unice**.

Principalele concepte ale modelului relațional sunt:

- ✓ **Obiectul** (entitatea) este ceva real existent în lumea materială (om, firmă, produs, etc.) și abstractă (noțiuni filozofice). Obiectele de același tip formează o **clasă de obiecte** (entități) și se descriu într-un tabel cu **numele clasei** (de exemplu, Student; Profesor; Firma; Factura etc.).
- ✓ **Atributul** este o caracteristică a unui obiect și se reprezintă în tabel ca o coloană cu un **nume propriu unic**. Fiecare relație conține o listă de atribute (coloane) definite pe un anumit domeniu. De ex. Furnizor (Nume, Adresă, Localitate, CodFiscal). Tabele diferite conțin atribute diferite, având scheme diferite $R_i=\{A_{1i},A_{2i},...,A_{ki}\}$. În modelul relațional **fiecare tabel** reprezintă o **relație**. Atributele unei relații au un **caracter global** în baza de date: dacă un nume denotă **două coloane în tabele distincte** în **aceeași bază de date**, atunci el **reprezintă același atribut.** Să notăm prin U o mulțime ne vidă de elemente $A_1,A_2,...,A_n$, numite ***nume de atribute*** sau ***atribute,*** preluate din **toate tabelele** unei baze de date. În această mulțime atributele cu același nume apar doar ca o singură apariție. Mulțimea $U = \{A_1,A_2,...,A_n\}$ se numește ***universul*** unei baze de date relaționale sau ***mulțime universală*** a bazei de date .Fiecare atribut A_i din mulțimea universală se caracterizează prin domeniul său de definiție **dom(A_i),** care reprezintă o **mulțime finită de valori de același tip** care le poate primi atributul A_i. . D**omeniu de valori admisibile** (DVA) se definește pentru fiecare atribut din punct de vedere **sintactic** și **semantic.** Dacă valorile atributului sunt **atomare**, atunci domeniul este unul **simplu**, iar atributul se numește **atribut atomic** sau **atribut simplu**. În caz contrar avem un **atribut compus**

- ✓ **Identificatorul primar** (cheie primară) este un atribut sau un set de atribute ale obiectului care **identifică în mod univoc** o entitate (obiect). Identificatorul primar evidențiază **o entitate unică** în cadrul unei clase de entități.
- ✓ **Cheia externă** reprezintă un atribut al unui obiect care are **același nume și tip de date** ca și identificatorul primar al unui alt obiect. Cheia externă este utilizată pentru corelarea a două tabele într-o relație de asociere de tip ***one-to-many*** între obiecte.
- ✓ **O linie** dintr-o relație tabelară se numește **tuplu.** Structura **tuplului** este formată din **toate atributele** unei relații tabelare cu schema R=**{A_1,A_2,...,A_k}**. În cazul când se selectează doar anumite atribute dintr-o relație tabelară cu schema R, atunci cortegiu de valori ale acestor atribute formează așa numita **X-valoare a tuplului**, unde X este o submulțime de atribute din mulțimea R.

  Fie R este **o submulțime** de atribute a universului U.
  - ❖ Definim ***relația* r** asupra R ca **o mulțime finită de tupluri** cu schema R.
  - ❖ ***Aritatea*** relației r este egală cu **cardinalitatea mulțimii R**.
  - ❖ ***Cardinalitatea*** relației r este **numărul de tupluri** în ea.
  - ❖ ***Mulțimea de atribute R*** se numește ***schema*** relației **r** (notată cu **sch(r) = R**).
  - ❖ ***Baza de date relațională*** este o mulțime finită de relații, notată ca **db = {r_1,...,r_m},** unde r_i este o relație cu schema R_i, $1 \leq i \leq m$.
  - ❖ ***Schema bazei de date*** **db = {r_1,...,r_m}** este **mulțimea schemelor relațiilor** ce formează baza de date și ea se notează **Db = {R_1,...,R_m},** unde R_i = sch(r_i).

Exemplu: Fie avem relația tabelară **r** cu schema **R={NrInreg, Brand, Model, Culoare, Tip, Anul}**, care este formată din **3 tupluri.** Aritatea relației **r** este egală cu 6 (mulțimea R are 6 atribute). Cardinalitatea relației **r** este egală cu 3 (3 tupluri se regăsesc în tabel).

NrInreg	Brand	Model	Culoare	Tip	Anul
XBX233	BMW	X5	Black	SUV	2021
DDT069	Toyota	Corolla	White	Sedan	2019
DOR333	Ford	Mondeo	Silver	Universal	2020

Putem identifica în mulțimea R o submulțime de atribute **X={Brand, Model, Tip},** valorile cărora formează X-valorile tuplurilor relației r.

Brand	Model	Tip
BMW	X5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal

Schema unei relații este o expresie **a proprietăților comune și invariante** ale tuplurilor ce compun relația. Schema unei relații mai este cunoscută sub denumirea de ***intensie*** a relației.

Relația tabelară se mai numește ***extensie.*** Extensia reprezintă **mulțimea tuplurilor** care compun la un moment dat relația, mulțime care este **variabilă în timp**.

O relație tabelară posedă următoarele proprietăți:

- ❖ *Într-o relație r_i* ***nu există*** *coloane cu nume* ***duplicate****, fiindcă atributele A_{ij}, $1 \leq j \leq k$, sunt elemente ale mulțimii R_i.*
- ❖ *Relația r_i* ***nu are tupluri identice****, fiindcă r_i este o* ***mulțime*** *de tupluri.*
- ❖ ***Ordinea tuplurilor*** *în r_i este* ***nesemnificativă****, fiindcă r_i este o* ***mulțime****.*
- ❖ ***Ordinea coloanelor*** *într-o relație tabelară este* ***nesemnificativă****.*
- ❖ ***Valorile atributelor*** *în r_i sunt* ***atomice,*** *fiindcă domeniile lor sunt* ***simple****.*
- ❖ Relațiile care se **stochează fizic** și formează baza de date se numesc ***relații de bază***.

Există, însă, și situații în care **extensia** nu se memorează în baza de date. Este cazul așa-numitelor ***relații virtuale***, cunoscute și sub numele de ***relații derivate*** sau ***viziuni (vederi, views)***. Relația virtuală **nu este definită explicit ca relația de bază**, prin mulțimea tuplurilor componente, ci **implicit** pe baza **altor relații**.

Relațiile de bază sunt **proiectate de administratorul bazei de date**, în timp ce **viziunile** sunt definite de **utilizatorii bazei de date**. Relațiile asupra unei mulțimi de atribute pot avea **un nume** sau pot să nu aibă, dacă ele sunt identificate în mod unic de schemele sale. **Numele relației**, de obicei, se scrie cu minuscule, de exemplu, relația **r**.

2.3. Constrângeri de integritate

Constrângerile de integritate, numite și restricții de integritate, definesc **cerințele** pe care trebuie să le satisfacă datele din baza de date pentru a putea fi considerate **corecte** și **coerente** în raport cu lumea reală pe care o reflectă. Constrângerile sunt **principalul mod de integrare** a semanticii datelor în cadrul modelului relațional. **Mecanismele de definire și verificare** ale acestor restricții reprezintă **instrumentele principale de control al semanticii datelor**.

În modelul relațional de date, constrângerile sunt studiate sub aspectul puterii lor de modelare și al posibilității de verificare a respectării lor. Constrângerile de integritate pot fi divizate în linii mari în două categorii:

- ***constrângeri de comportament;***
- ***constrângeri de dependență între date***.

Constrângerile de comportament specifică caracteristicile **independente** ale unui atribut (sau domeniu de valori). Ele exprimă **semantica** elementelor domeniilor. De exemplu, toate valorile atributului **Nota** trebuie să fie mai mari decât zero, dar nu pot depăși valoarea zece. Sau nici o persoană de vârsta 25 ani nu poate avea o vechime în muncă de 37 ani. Conform acestei restricții, valorile atributului trebuie să se încadreze între anumite limite.

Constrângerile de dependență specifică **legătura dintre atribute** (sau domeniile de valori). Putem identifica așa-numita **dependență de submulțime**. De exemplu, în relația tabelară ***angajați,*** definită pe mulțimea de atribute **{Angajat, Salariu, Departament, Manager},** apare o dependență de submulțime, deoarece un manager este în același timp și un angajat, dar nu orice angajat este manager. Prin urmare, avem că **dom(Manager) ⊆ dom(Angajat).**

Dacă presupunem că angajatul are un singur salariu, se subordonează unui singur manager direct, lucrează într-un singur departament, atunci Angajat-valorile determină **în mod unic** toate tuplurile în relația ***angajați***. Această constrângere este numită **dependență funcțională**. Există și alte tipuri de dependențe cum ar fi cele **multivaloare** și **de joncțiune.**

Relația reprezintă **o mulțime de tupluri**, iar **o mulțime nu poate conține elementele duplicate**, prin urmare, relația **nu poate** prezenta **tupluri identice**. Tuplurile sunt **unice** și trebuie să existe posibilitatea **identificării lor** în cadrul unei relații. **Identificarea** unui tuplu fără a consulta toate componentele tuplului a dus la apariția noțiunii de **cheie** sau **identificator**.

Definiție: *Fie U este o mulțime universală de atribute și este definită o submulțime R nevidă ($R \subseteq U$ și $R \neq \emptyset$). Atunci putem defini o* ***mulțime K de atribute,*** *unde $K \subseteq R$, ca* ***cheie*** *pentru schema R (sau pentru relația r cu schema R), dacă ea posedă următoarele proprietăți:*

1) ***pentru orice două tupluri t_1 și t_2 din relația r avem $t_1[K] \neq t_2[K]$;***
2) ***nici o submulțime K^1 proprie a lui K nu posedă proprietatea (1).***

Proprietatea (1), numită **restricția de unicitate a cheii**, permite K-valorilor să identifice în mod unic **toate tuplurile** dintr-o relație. Dacă însă K conține o cheie K^1 și $K \neq K^1$, atunci, cu toate că atributele din K sunt suficiente de a atinge scopul, unele din ele nu sunt necesare și pot fi eliminate din cheie fără a se afecta unicitatea cheii. Dacă K este o **submulțime proprie a unei chei**, atunci utilizarea a astfel de K-valori pentru căutarea datelor va descoperi tupluri ce coincid pe toate valorile atributelor din K.

Proprietatea (2) ne asigură că o cheie K constituie numai acele atribute ce **sunt necesare și suficiente** pentru a determina **univoc** pe celelalte. Cu alte cuvinte, K-valorile întotdeauna asigură un grad exact de informație pentru a găsi **un tuplu unic** într-o relație tabelară.

Mulțimea de atribute ce posedă proprietatea (1) se numește ***supercheie.*** Cheia este o **supercheie minimală**. Orice cheie e și supercheie. O mulțime vidă **nu poate servi** drept cheie a unei relații ce conține mai mult de un tuplu. **Orice relație** are **cel puțin o cheie**. La limită cheia este constituită fie dintr-un **singur atribut**, fie din **totalitatea atributelor** din schema relației respective.

Într-o relație pot exista **mai multe** chei. Se spune în acest caz că relația posedă mai multe ***chei candidate***. În această situație administratorul bazei de date va stabili **una din cheile candidate** să servească în mod efectiv la identificarea unică a tuplurilor. Ea va primi numele de ***cheie primară***. **Primare** se vor numi și **domeniile atributelor** ce formează o cheie primară.

Cheia primară a unei relații se numește ***cheie simplă***, dacă este constituită dintr-un singur atribut, iar atunci când este formată din mai multe atribute este denumită ***cheie compusă.*** Nu toate atributele unei chei compuse pot fi definite pe domenii primare.

O *cheie externă* reprezintă un atribut (grup de atribute) dintr-o schemă R_i definit (definite) pe același (aceleași) domeniu (domenii) ca și cheia primară a scheme R_j. Relația r_i se numește *relație care referă*, iar r_j poartă numele de *relație referită*.

Unele atribute pot avea așa numitele **valori nedefinite** sau **necunoscute** notate cu "**Null**". În cazul valorilor "**Null"** trebuie respectate constrângerile formulate prin următoarele reguli, numite **regulile de actualizare** (inserare, modificare și eliminare) a relațiilor:

- **Constrângerea entității:** *cheia primară a unei relații de bază nu poate conține valori "Null";*
- **Constrângerea referențială (de referire):** *dacă atributul A al unei chei compuse a relației r_i este definit pe un domeniu primar, atunci trebuie să existe o relație de bază r_j cu o cheie primară B, încât orice A-valoare din r_i să apară în calitate de B-valoare în r_j.*

Constrângerea entității impune că la înserarea unui tuplu, valoarea cheii să fie **cunoscută**, pentru a se putea verifica faptul că această valoare nu este deja încărcată (respectarea constrângerii de unicitate a cheii). Cu valori "**Null**" cheia își **pierde rolul de identificator de tuplu**.

Constrângerea referențială impune ca într-o relație r_i, care referă o relație r_j, valorile cheii compuse **să figureze** printre valorile cheii primare din relația r_j pentru atributele compatibile.

2.4. Modele logice orientate pe obiecte

Modelele logice orientate pe obiecte descriu datele la nivelele conceptual și extern, oferind o flexibilitate ridicată. Astfel de modele pot specifica în mod explicit constrângerile aplicate datelor și se bazează pe următoarele concepte: **entitate** – un obiect sau concept din lumea reală ce are identitate proprie; **atribut** – un set de proprietăți utilizate la descrierea entităților; **relație** – o asociere între două sau mai multe entități.

Din această categorie fac parte următoarele modele:

- ✓ ***modelul entitate-relație;***
- ✓ ***modelul orientat pe obiecte;***
- ✓ ***modelul obiectual-relațional;***

Modelul entitate-relație se bazează pe o percepție a lumii reale ca fiind alcătuită dintr-o colecție de obiecte de bază sau concepte (entități) împreună cu relațiile care se stabilesc între ele.

Fiecare entitate are asociat un **set de atribute** care o descriu, iar o relație reprezintă **o asociere** dintre două sau mai multe entități. Mulțimile tuturor entităților sau relațiilor de același tip sunt cunoscute sub denumirea de **tipuri de entități sau relații.** Un alt element important în cadrul diagramelor entitate-relație îl reprezintă precizarea **constrângerilor de cardinalitate** care exprimă numărul de entități asociate altui tip de entitate prin intermediul unui tip de relație.

Modelul orientat pe obiecte este utilizat doar în scopuri speciale, cele mai cunoscute produse de acest tip fiind: ObjectStore, Versant, Objecivity/DB, Gemstone, Ontos, O2, Jasmine, Cache. Modelul se bazează pe **o colecție de obiecte**, la fel ca în cazul modelului entitate-relație.

Un ***obiect*** conține valorile înmagazinate în cadrul unor variabile instanțiate în interiorul acestor obiecte. Spre deosebire de modelul entitate-relație, **valorile sunt ele însele obiecte.** Astfel de obiecte conțin alte obiecte imbricate până la un nivel oarecare. Obiectul mai conține elemente de cod ce operează asupra acestuia și care se numesc ***metode.*** Obiectele ce conțin același tip de valori și aceleași metode sunt grupate în ***clase***. O clasă poate fi interpretată ca fiind definiția de tip a obiectului respectiv. Singura modalitate prin care un obiect poate accesa datele altui obiect este prin ***invocarea*** metodelor acelui obiect, ceea ce este cunoscut sub numele de ***trimitere de mesaje către obiectul respectiv.***

Părțile interne ale obiectului respectiv, variabilele și metodele, nu sunt vizibile în exterior, obținându-se astfel ***două nivele de abstractizare a datelor.*** Spre deosebire de entitățile din modelul entitate-relație, fiecare obiect are un ***identificator unic*** indiferent de valorile pe care le conține. Două obiecte ce au aceleași valori sunt ***distincte (polimorfism).*** Distincția se menține și la nivelul fizic prin atribuirea unui identificator unic al obiectului respectiv.

Modelul orientat pe obiecte are toate caracteristicile limbajului de programare orientat pe obiecte, făcând ca modelul relațional să fie coborât la stadiul de depozit de date.

Modelul obiectual-relațional (cunoscut inițial sub numele de model de date relațional extins) a extins modelul relațional prin introducerea unor serii de elemente și caracteristici specifice modelului obiectual, cum ar fi: **clase, încapsulare, moștenire**. Cele mai cunoscute produse de pe piață sunt: PostgreSQL, Informix, DB2, Oracle.

Scopul acestei extinderi a fost acela de a permite bazelor de date relaționale să opereze cu tipuri complexe de date, cum ar fi: ***imagini audio, video, elemente de proiectare***. Modelul se află încă la stadiul de dezvoltare, chiar dacă este promovat de cei mai mari producători de pe piață de produse de baze de date.

2.5. Modele arhitecturale ale bazelor de date

Modelul mainframe (centralizat) inițial presupunea că baza de date este organizată și stocată integral pe un sistem performant (mainframe) de unde poate fi accesată de mai multe console utilizator. Modelul s-a dovedit performant și sigur atât în implementare, cât și în utilizare, dar au existat și câteva puncte sensibile. Problema delicată la mainframe-uri nu este numărul de utilizatori suportați, ci faptul că aplicațiile au o infrastructură rigida, a căror extindere determină implicații dure de organizare și administrare, pe lângă creșterile nedorite ale traficului de date prin rețea.

Modelul integrat reprezintă un mediu software independent, instalat pe un singur calculator, și include atât baza de date propriu-zisă, cât și interfața de acces la date (un prim exemplu de astfel de mediu integrat este dBASE), astfel că singurul utilizator va fi beneficiarul. Accesarea datelor se face fie printr-un limbaj de generația IV sau printr-un macrolimbaj, fie prin elemente de interfață (comenzi la prompter, dialoguri QBE, comenzi din meniu). Datele fiind organizate tabelar, exista posibilitatea de a proiecta aplicații relaționale.

Modelul File-Server implică utilizarea unui server de fișiere centralizat aflat undeva în rețea, care pune la dispoziția clientului unități de disc logic cu fișiere partajate. Serverul de fișiere nu are cunoștințe despre cererile logice care se prelucrează, dar acționează sub forma unui disc ce poate fi accesat prin intermediul unei rețele pentru a transfera date de la/spre client. Toate aplicațiile rulează la client. Exemple de astfel de produse sunt FoxPro, dBASE, MS Access. O bază de date de acest tip are o fiabilitate foarte scăzută deoarece ea se păstrează sub forma unui fișier într-un sistem de fișiere, fiind ușor de distrus dacă se produce o eroare în timpul unei prelucrări sau a unei pierderi de conexiune în timpul efectuării unei tranzacții. Soluțiile "file-server" nu sunt recomandate întreprinderilor de mari dimensiuni sau aplicațiilor extinse, preferându-se în acest caz soluția "client-server" care scade traficul din rețea și asigură o fiabilitate mai mare a sistemelor.

Modelul Client-Server reprezintă o bază de date centralizată care prelucrează cererile logice provenite de la client. Un astfel de server "înțelege" atât natura cererii, cât și structura și

localizarea datelor, majoritatea calculelor fiind efectuate de motorul bazei de date. Clientul are doar o interfață grafică cu care poate accesa baza de date de pe server, folosind aplicații pentru a transmite comenzi SQL serverului bazei de date, rezultatele fiind primite sub formă de tabele. Exemple de astfel de produse sunt: Oracle, Microsoft SQL Server, MySQL, Sybase, Informix.

Arhitectura "client-server" oferă o fiabilitate ridicată și o serie de alte caracteristici avantajoase ce nu pot fi oferite de arhitectura "file-server":

- **Copie de siguranță ce poate fi creată în timpul lucrului** prin utilizarea unui planificator automat ce face copii ale bazelor de date fără a fi necesară întreruperea conexiunii cu utilizatorii**.**
- **Tranzacții sigure** prin jurnalizarea tranzacțiilor, astfel încât actualizările efectuate prin intermediul unei tranzacții pot fi în orice moment refăcute sau anulate, indiferent dacă eroarea se produce la client sau la server.
- **Fiabilitate și protecție sporită a datelor**. O bază de date în model "file-server" poate fi reparată, în cazul apariției unei erori, dar în acest caz utilizatorul trebuie deconectat, ce se întâmplă foarte rar în cazul modelului "client-server".
- **Procesarea mai rapidă a interogărilor**. În cazul modelului "client-server", prelucrările se efectuează direct pe server. Prelucrarea pe server duce la o încărcare mult mai mare a acestuia decât în cazul modelului "file-server", dar traficul din rețea va fi substanțial mai scăzut.

Întrebări de control:

1. Care sunt elementele de bază ale unui model de date și prin ce se diferențiază intensia de extensia unei baze de date?
2. Cum este reprezentată o relație în modelul ierarhic de date, care sunt principalele sale avantaje legate de performanță și prin ce se caracterizează integritatea sa referențială?
3. Cum funcționează structura de tip set în cadrul modelului reticular (de rețea) și care sunt neajunsurile principale pe care le întâmpină un utilizator în explorarea și modificarea acesteia?
4. Care sunt componentele de bază ale modelului relațional și ce proprietăți fundamentale posedă o relație tabelară?
5. Care este diferența dintre o cheie primară, o supercheie și o cheie externă, precum și între o cheie simplă și una compusă?
6. În ce constau constrângerea entității și constrângerea referențială atunci când într-o bază de date apar valori nedefinite de tip „Null"?
7. Care sunt caracteristicile definitorii ale modelului orientat pe obiecte și prin ce s-a extins acesta în cadrul modelului obiectual-relațional?
8. Prin ce se diferențiază arhitectura de tip File-Server de cea Client-Server și care sunt principalele avantaje oferite de modelul Client-Server în privința fiabilității și traficului din rețea?

Capitolul 3. ELEMENTE DE ALGEBRĂ RELAȚIONALĂ

3.1. Operații tradiționale pe mulțimi

Operațiile algebrei relaționale pot fi divizate în două categorii:

- operațiile **tradiționale** pe mulțimi (*uniunea, intersecția, diferența, produsul cartezian*);
- operațiile **relaționale native** (*proiecția, selecția, joncțiunea, Θ-joncțiunea, semi-joncțiunea,).*

Operațiile binare asupra relațiilor (**uniunea, intersecția și diferența)**, necesită ca operanzii (relațiile tabelare) să fie definiți pe **scheme compatibile**. Compatibilitatea schemelor se definește în felul următor:

*Vom spune că două relații tabelare r(R) și s(S) sunt **compatibile** (sau au scheme compatibile), dacă între schemele R și S există o **corespondență biunivocă f**: pentru orice atribut A din R, există un atribut B în S, încât **dom(A)=dom(B)**, B=f(A) și A=f^{-1}B), unde f^{-1} este funcția inversă funcției f.*

Remarcă. *Două relații cu aceeași schemă sunt compatibile.*

- ***Uniunea*** a două relații compatibile r(R) și s(S), notată cu **r ∪ s**, este o relație definită pe schema **R** sau **S**, și constă din tuplurile ce aparțin relațiilor r sau s, adică

$$r \cup s = \{t \mid t \in r \vee t \in s\}.$$

Exemplu:

Brand	Model	Tip
BMW	X5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal

Relația r

Brand	Model	Tip
Audi	Q5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal
Fiat	500	Hatchback

Relația s

Brand	Model	Tip
Audi	Q5	SUV
BMW	X5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal
Fiat	500	Hatchback

Relația q=r ∪ s

Important! În relația q tuplurile care se repetă sunt incluse **o singură dată.**

Operația **uniunea** are două proprietăți:

- **Uniunea este comutativă:** **r ∪ s = s ∪ r.**
- **Uniunea este asociativă:** **(r ∪ s) ∪ q = r∪(s ∪ q)**

pentru relațiile mutual compatibile r, s și q.

Operația uniunea are două cazuri speciale. Pentru orice relație r(R) au loc:

1) **r ∪ ∅ = r** ;
2) **r ∪ s = s,** dacă r ⊆ s.

- ***Intersecția*** a două relații compatibile r(R) și s(S), notată cu **r ∩ s,** este o relație definită pe schema R sau S și constă din tuplurile ce aparțin **concomitent** relațiilor r și s, adică

$$r \cap s = \{t \mid t \in r \,\&\, t \in s\}.$$

Exemplu:

Brand	Model	Tip
BMW	X5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal

Relația r

Brand	Model	Tip
Audi	Q5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal
Fiat	500	Hatchback

Relația s

Brand	Model	Tip
Toyota	Corolla	Sedan
Ford	Mondeo	Universal

Relația q=r ∩ s

- ***Diferența*** a două relații compatibile r(R) și s(S), notată cu **r \ s,** este o relație definită pe mulțimea de atribute **R** sau **S** și are în calitate de tupluri **toate** tuplurile din relația r ce nu se regăsesc în relația s, adică

$$r \setminus s = \{t \mid t \in r \ \& \ t \notin s\}.$$

Diferența **nu are** proprietatea **comutativă**, adică

$$r \setminus s \neq s \setminus r,$$

nici **nu este asociativă**, adică

$$(r \setminus s) \setminus q \neq r \setminus (s \setminus q).$$

Un caz special este $\emptyset \setminus r = \emptyset$; altul e $r \setminus \emptyset = r$ pentru orice relație r(R).

Exemplu:

Brand	Model	Tip
BMW	X5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal

Relația r

Brand	Model	Tip
Audi	Q5	SUV
Toyota	Corolla	Sedan
Ford	Mondeo	Universal
Fiat	500	Hatchback

Relația s

Brand	Model	Tip
BMW	X5	SUV

Relația q1=r\s

Brand	Model	Tip
Audi	Q5	SUV
Fiat	500	Hatchback

Relația q2=s\r

- ***Produsul cartezian*** al două relații tabelare $\mathbf{r(A_1...A_n)}$ și $\mathbf{s(B_1...B_m)}$, notat cu **r × s**, este o **mulțime de tupluri** (și nu întotdeauna o relație) definite pe mulțimea de atribute $\mathbf{A_1...A_nB_1...B_m}$.

Tuplurile reprezintă **toate posibilele asociații** de tupluri din r și s: dacă $t_r \in r$ și $t_s \in s$, atunci concatenația $\mathbf{t_rt_s}$ este un tuplu în **r × s.**

Exemplu:

Brand
BMW
Toyota
Ford

Relația r

Tip
SUV
Sedan
Universal

Relația s

Brand	Tip
BMW	SUV
BMW	Sedan
BMW	Universal
Toyota	SUV
Toyota	Sedan
Toyota	Universal
Ford	SUV
Ford	Sedan
Ford	Universal

Relația q= r × s

Produsul cartezian al două relații tabelare ne vide **nu întotdeauna** produce o relație. Rezultatul e o relație, dacă **ambii operanzi** sunt relații cu scheme **ne vide** și **disjuncte**. Dacă, însă, relațiile-operanzi au scheme **vide** sau **non-disjuncte**, atunci produsul cartezian **nu este o relație**.

Produsul cartezian **nu este o operație comutativă**. În schimb se bucură de proprietatea **asociativă**.

3.2. Operațiile relaționale native

❖ ***Proiecția*** relației r(R) asupra mulțimii de atribute X, notată cu $\pi_X(r)$, e o relație cu schema **X** ce constă din **X-valorile** tuturor tuplurilor din r:

$$\pi_X(r) = \{t\,[X] \mid t \in r\}$$

Tupluri distincte din relația r pot deveni **identice**, când se proiectează pe o mulțime de atribute. **Tuplurile duplicate** în relația rezultat **se elimină**.

Există două cazuri speciale:

1) $X = R$. Atunci $\pi_X(r) = r$.
2) $r = \emptyset$. Atunci $\pi_X(r) = \emptyset$.

Exemplu: Fie avem relația **Automobile**

NrInreg	Brand	Model	Culoare	Tip	Anul
XBX233	BMW	X5	Black	SUV	2021
DDT069	Toyota	Corolla	White	Sedan	2019
DOR333	Ford	Mondeo	Silver	Universal	2020
AAD123	BMW	X5	Red	SUV	2022
CAR321	Audi	S4	Green	Sedan	2016
AUR001	Toyota	RAV4	White	SUV	2022
FRD035	Ford	Focus	Blue	Universal	2017

Atunci proiecția $\pi_{Brand, Tip}$**(Automobile)** va fi

Brand	Tip
BMW	SUV
Toyota	Sedan
Ford	Universal
Audi	Sedan
Toyota	SUV

Dacă mulțimea de atribute **X = ∅**, atunci proiecția **$\pi_{\emptyset}$(r)** este **nedefinită**, fiindcă schema unei relații **nu poate fi o mulțime vidă**. Schema unei relații, produsă de operația proiecția, are **cel puțin un atribut.**

Pentru cazul când **R⊂X**, operația proiecția iarăși **este nedefinită**. Mulțimea asupra căreia se face proiecția **nu poate fi mai largă** decât schema relației inițiale.

Pentru selectarea unor tupluri dintr-o relație e necesară **specificarea condițiilor de selectare**. În rezultat se obține o relație ce este o **submulțime de tupluri ale relației inițiale**.

Fie că condiția de selecție se notează prin **formula calculului propozițional, F, definită recursiv**:

1. **AθB** și **Aθa** sunt **formule**, unde A și B sunt **atribute compatibile** și **a∈dom(A)**, iar **θ∈{=, ≠, <, ≤, >, ≥}.** Aceste formule sunt **atomice**.
2. Dacă **G** și **H** sunt **formule**, atunci **conjuncția G&H**, **disjuncția G∨H**, negațiile **~G** și **~H** tot sunt **formule.**
3. Nimic altceva nu e formulă.

Vom spune că **formula F este *aplicabilă*** relației r(R), dacă orice constantă c din F este în dom(R), și orice atribut A din F este în R.

O relație tabelară r **satisface** formula F (sau F e validă în r), dacă F **e aplicabilă relației r** și orice tuplu t∈r satisface formula F în sensul că formula G obținută prin substituirea oricărui atribut A din F cu A-valoarea tuplului t **are valoarea adevăr**.

❖ ***Selecția*** relației r(R) conform formulei F, unde F e aplicabilă relației r(R), e o submulțime a relației r(R), notată cu $\sigma_F(r)$, ce constă din toate tuplurile t∈r ce satisfac formula F, adică

$$\sigma_F(r) = \{t \mid t \in r \ \& \ F(t)\}$$

Exemplu: Fie avem relația **Automobile**

NrInreg	Brand	Model	Culoare	Tip	Anul
XBX233	BMW	X5	Black	SUV	2021
DDT069	Toyota	Corolla	White	Sedan	2019
DOR333	Ford	Mondeo	Silver	Universal	2020
AAD123	BMW	X5	Red	SUV	2022
CAR321	Audi	S4	Green	Sedan	2016
AUR001	Toyota	RAV4	White	SUV	2018
FRD035	Ford	Focus	Blue	Universal	2017

Atunci selecția $\sigma_{((Tip = "SUV") \ \&\ (Anul > 2020))}$**(Automobile)** va returna tuplurile

NrInreg	Brand	Model	Culoare	Tip	Anul
XBX233	BMW	X5	Black	SUV	2021
AAD123	BMW	X5	Red	SUV	2022

Este evident că $\sigma_G(\sigma_F(r)) = \sigma_{G\&F}(r)$. Deoarece conjuncția este **comutativă**, adică $\sigma_{F\&G}(r)=\sigma_{G\&F}(r)$, atunci și compoziția a două selecții este **comutativă** și avem $\sigma_G(\sigma_F(r)) = \sigma_F(\sigma_G(r))$.

Operația selecția este **distributivă** în raport cu operațiile tradiționale pe mulțimi. Fie r și s două relații compatibile și $\gamma \in \{\cup, \cap, \setminus\}$, atunci

$$\sigma_F (r \ \gamma \ s) = \sigma_F(r) \ \gamma \ \sigma_F (s).$$

Fie **r(R)** și **s(S)** sunt două relații cu scheme disjuncte, adică **R∩S=∅**, A∈R, B∈S, și fie **θ** un element din mulțimea **{=, ≠, <, ≤, >, ≥}.** Presupunem că atributele A și B sunt **compatibile**, adică **dom(A)=dom(B).**

- ***θ-joncțiunea* (*theta-joncțiunea*)** relațiilor r(R) și s(S), notată cu **$r|x|_{A\theta B}s$,** este o mulțime de tupluri concatenate de forma t_rt_s , unde t_r∈r, t_s∈s și $t_r(A)$ θ $t_s(B)$, adică:

$$r|x|_{A\theta B}s = \{ t_rt_s \mid t_r \in r \ \& \ t_s \in s \ \& \ t_r(A) \ \theta \ t_s(B)\}$$

Condiția **R∩S=∅** în definiție **este necesară**. Dacă **R∩S≠∅**, adică schemele nu sunt disjuncte, atunci θ-joncțiunea **nu este o relație**. În cazul când θ este "=", θ-joncțiunea se mai numește ***echijoncțiune.***

Exemplu: Fie avem relațiile **Automobile** și **Leasing**, iar **dom(Anul)=dom(Year)**

NrInreg	Brand	Model	Culoare	Tip	Anul
XBX233	BMW	X5	Black	SUV	2021
DDT069	Toyota	Corolla	White	Sedan	2019
DOR333	Ford	Mondeo	Silver	Universal	2020
AAD123	BMW	X5	Red	SUV	2022
CAR321	Audi	S4	Green	Sedan	2016
AUR001	Toyota	RAV4	White	SUV	2022
FRD035	Ford	Focus	Blue	Universal	2017

LeasingMonth	Year
36	2021
60	2022
72	2022

Atunci **Automobile $|x|_{Anul=Year}$ Leasing** va fi:

NrInreg	Brand	Model	Culoare	Tip	Anul	LeasingMonth	Year
XBX233	BMW	X5	Black	SUV	2021	36	2021
AAD123	BMW	X5	Red	SUV	2022	60	2022
AUR001	Toyota	RAV4	White	SUV	2022	60	2022
AAD123	BMW	X5	Red	SUV	2022	72	2022
AUR001	Toyota	RAV4	White	SUV	2022	72	2022

- ***Joncțiunea (joncțiunea naturală)*** relațiilor r(R) și s(S) (notația uzuală **r|x|s)** este o relație cu schema **RS**. Tuplul t aparține relației rezultat, dacă există tuplurile t_r și t_s în r și s, respectiv, și satisfac $t[R]=t_r$ și $t[S]=t_s$, adică

$$r \ |x| \ s = \{t \mid t[R] = t_r \ \& \ t[S] = t_s \ \& \ t_r \in r \ \& \ t_s \in s\}.$$

Fiecare tuplu din relația rezultat este o **concatenare** a unui tuplu din r cu un tuplu din s ce au **(R∩ S)-valori egale**. Atributele cu **același nume** în schema relației rezultat se iau **o singură dată**.

Dacă R și S sunt **disjuncte**, R∩S=∅, atunci joncțiunea relațiilor r și s este identică cu **produsul cartezian** al lor, adică **r|x|s = r × s**.

Exemplu:

Contravenție			
Cod	Data	Articol	NrInmat
23455	12.02.2023	236	XGH123
25671	03.03.2023	236	ARC079
26578	03.03.2023	238	XGH123
33455	13.03.2023	241	ARC079
39081	14.03.2023	236	XGH123

Autovehicul		
NrInmat	Brand	Model
DDT344	Ford	Kuga
ARC079	Audi	A4
XGH123	Honda	Civic
DOR033	Geely	Coolray

Contravenție |x| Autovehicul

Cod	Data	Articol	NrInmat	Brand	Model
23455	12.02.2023	236	XGH123	Honda	Civic
25671	03.03.2023	236	ARC079	Audi	A4
26578	03.03.2023	238	XGH123	Honda	Civic
33455	13.03.2023	241	ARC079	Audi	A4
39081	14.03.2023	236	XGH123	Honda	Civic

Operatorul de joncțiune exclude tuplurile unei relații ce nu au corespondent in celălalt operand.

- *Operatorul de **joncțiune externă*** – asigura prezența in relația rezultat a tuturor tuplurilor unei relații, acestea fiind completate cu valori **Null** atunci, când nu au corespondent in cealaltă relație.

  Acest operator are trei variante de implementare:
  - **joncțiune externa la stânga** (Left Join) – extinde doar tuplurile primului operand;
  - **joncțiune externă la dreapta** (Right Join) – extinde doar tuplurile celui de-al doilea operand;
  - **joncțiune externă completă** (Full Join)– extinde toate tuplurile.

r_1

Angajat	Departament
Albu	Logistică
Negru	Producție
Galben	Producție

r_2

Departament	Manager
Producție	Popescu
Marketing	Sârbu

$r_1 |x|_{LEFT} r_2$

Angajat	Departament	Manager
Albu	Logistică	**Null**
Negru	Producție	Popescu
Galben	Producție	Popescu

$r_1 |x|_{RIGHT} r_2$

Angajat	Departament	Manager
Negru	Producție	Popescu
Galben	Producție	Popescu
Null	Marketing	Sârbu

$r_1 |x|_{FULL} r_2$

Angajat	Departament	Manager
Albu	Logistică	**Null**
Negru	Producție	Popescu
Galben	Producție	Popescu
Null	Marketing	Sârbu

❖ **Semi-joncțiunea** constă în construirea unei relații din cele două și e formată **numai din tuplurile unei singure relații** ce participă la joncțiune. ***Semi-joncțiunea*** relațiilor r(R) și s(S), notată cu **r|xs** sau **rx|s**, este o mulțime de tupluri determinată de expresia

$$\mathbf{r|xs = \pi_R(r|x|s)} \quad \text{sau} \quad \mathbf{rx|s = \pi_S(r|x|s)}$$

Exemplu:

Contravenție			
Cod	Data	Articol	NrInmat
23455	12.02.2023	236	XGH123
25671	03.03.2023	236	ARC079
26578	03.03.2023	238	XGH123
33455	13.03.2023	241	ARC079
39081	14.03.2023	236	XGH123

Autovehicul		
NrInmat	Brand	Model
DDT344	Ford	Kuga
ARC079	Audi	A4
XGH123	Honda	Civic
DOR033	Geely	Coolray

Contravenție x| Autovehicul = $\pi_{\text{Autovehicul}}$(Contravenție |x| Autovehicul)

NrInmat	Brand	Model
XGH123	Honda	Civic
ARC079	Audi	A4

Întrebări de control:

1. Ce condiție trebuie să îndeplinească două relații tabelare $r(R)$ și $s(S)$ pentru a fi considerate compatibile?
2. Care sunt proprietățile operației de uniune a două relații compatibile și cum se tratează tuplurile care se repetă în relația rezultat?
3. De ce diferența a două relații ($r \setminus s$) nu este o operație comutativă și ce reprezintă rezultatul ei?
4. În ce condiții produsul cartezian ($r \times s$) al două relații tabelare produce o relație validă?
5. Cum influențează eliminarea tuplurilor duplicate rezultatul operației de proiecție ($\pi_X(r)$) și în ce situații această operație devine nedefinită?
6. Cum este definită selecția ($\sigma_F(r)$) și ce proprietate are compoziția a două selecții succesive?
7. Care este diferența principală dintre o θ-joncțiune (sau echijoncțiune) și o joncțiune naturală ($r \bowtie s$)?
8. Prin ce se deosebesc variantele de joncțiune externă (Left, Right, Full Join) și semi-joncțiunea față de joncțiunea naturală clasică?

Capitolul 4. FUNDAMENTELE LIMBAJULUI SQL

4.1. Evoluția limbajului SQL

Limbajul SQL apare ca rezultat al lansării de către firma IBM în anul 1974 a proiectului System/R. În anul 1974 inginerii-programatori Donald Chamberlin și Ray Boyce, angajați ai corporației IBM, publică specificațiile unui limbaj structurat de interogare SEQUEL (Structured English as Query Language). Donald Chamberlin și Raymond Boyce, bazându-se pe modelul stabilit de E. F. Codd în lucrarea „*A Relational Model of Data for Large Shared Data Banks*", au dezvoltat limbajul de programare în propria lucrare „*SEQUEL: A Structured English Query Language*". Chamberlin și Boyce au preluat limbajele lui Codd cu scopul de a proiecta un limbaj relațional care să fie mai accesibil utilizatorilor fără o pregătire formală în matematică sau informatică. Această versiune SQL originală a fost folosită pentru a manipula și a prelua datele stocate în sistemele originale de baze de date relaționale ale IBM, cunoscute sub numele de System/R.

În anii următori SQL nu a fost disponibil public. Cu toate acestea, în 1979, Oracle, cunoscut pe atunci sub numele de Relational Software, a lansat propria sa versiune de SQL, numită Oracle V2, care a fost lansată comercial. Este important de menționat că SQL nu a fost primul limbaj de programare pentru gestionarea bazelor de date, dar el a devenit o realizare impecabilă datorită simplității, puterii și fiabilității sale. Cu toate acestea, succesul prelungit al SQL nu poate fi atribuit numai calităților sale. Standardizarea internațională nu numai că a ajutat SQL să se apropie de universalitate, dar a adăugat și atribute cheie la ceea ce a înflorit în specificațiile sale moderne. Totul a început când ANSI (American National Standards Institute) s-a implicat în standardizarea limbajului. În 1986 limbajul SQL a devenit oficial acceptat, iar Comitetul tehnic pentru baze de date ANSI (ANSI X3H2) al Comitetului pentru standarde acreditate ASC X3 a dezvoltat primul standard SQL, ANSI X3.135-1986. Mai târziu ASC X3 a devenit INCITS (Comitetul Internațional pentru Standarde de Tehnologie a Informației), o organizație de dezvoltare a standardelor, acreditată ANSI.

Acesta a fost începutul a ceea ce oamenii numesc în mod greșit standardul ANSI pentru SQL. De fapt, nu există standarde ANSI, ci doar standarde dezvoltate de comitetele aprobate de ANSI, multe funcționând în conformitate cu cerințele esențiale ANSI (standarde naționale americane). În câteva luni mai târziu ISO a publicat un standard identic din punct de vedere tehnic, ISO 9075-1987, aducând SQL, care era cândva limitat la bazele de date IBM, pe scena internațională. În perioada publicării inițiale a acestui standard au fost cu siguranță necesare specificații mai profunde cu privire la SQL, dar ANSI X3.135-1986 a ajutat la stabilirea bazei unor capacități importante pentru limbajul de codificare. Acest standard a oferit capacitatea de a invoca capabilități SQL din patru limbaje de programare: COBOL, FORTRAN, Pascal și PL/I. Aceste standarde au fost revizuite în comun, mai întâi în 1989 (ANSI X3.135-1989 și ISO/IEC 9075:1989) și în 1992 (ANSI X3.135-1992 și ISO/IEC 9075:1992). Ediția din 1989 a adăugat suport pentru două limbaje de programare suplimentare, Ada și C. Aceste ediții au devenit cunoscute ca SQL-86, SQL-89 și SQL-92.

Standardul a fost revizuit din nou în 1993 (SQL3), 2003, 2008, 2011 și 2016. Standardul SQL a apărut în mai multe părți de la începutul secolului, dar, începând cu ediția din 2003, a fost subdivizat în 9 părți, fiecare acoperind un aspect diferit al standardului general. Standardul internațional ISO/IEC 9075 pentru SQL este dezvoltat de Comitetul Tehnic Comun ISO/IEC (JTC) 1 pentru Tehnologia Informației. Ediția actuală a standardului este ISO/IEC 9075:2016.

Principalele avantaje ale limbajului SQL sunt:

- ***Independența de producătorul produsului informatic;***
- ***Portabilitatea pe diverse sisteme informatice;***
- ***Standardizarea internațională a limbajului;***
- ***Utilizarea modelului relațional de organizare a datelor;***

- ***Reprezintă un limbaj de nivel înalt complet de generația IV;***
- ***Asigură suport programat de acces la baze de date;***
- ***Furnizează răspunsuri la numeroase interogări ale utilizatorilor;***
- ***Permite modificarea dinamică a datelor și a structurii bazei de date;***
- ***Permite implementarea arhitecturilor client-server.***

4.2. Tipuri de date, operatori și identificatori

Vom examina în continuare versiunea Transact-SQL a limbajului SQL implementată în cadrul SGBD Microsoft SQL Server. Această versiune posedă un set de date de sistem permise pentru utilizare conformă standardului internațional. Dar pot fi definite și date de tip utilizator pe baza datelor de sistem ale limbajului Transact-SQL.

In cazul când două expresii SQL au tipuri de date diferite, atunci tipul de date, ordinea sortării datelor, precizia, formatul, lungimea câmpului și caracteristicile rezultatului final se determină în felul următor:

- ✓ Tipul de date al rezultatului se determină prin aplicarea regulilor de conversie a tipurilor de date din expresie;
- ✓ Sortarea rezultatului este determinată de regulile de prioritate pentru sortare, dacă avem tipuri de date `char, varchar, text, nchar, nvarchar` sau `ntext`.
- ✓ Precizia, formatul și lungimea rezultatului depinde de precizia, formatul și lungimea operanzilor din expresie.

Deosebim următoarele categorii de date ale limbajului Transact-SQL

- ❖ **date numerice exacte**: `bit, tinyint, int, smallint, bigint, numeric(p,s), decimal(p,s), money, smallmoney;`
- ❖ **date numerice aproximative:** `float(n), real;`
- ❖ **date binare:** `binary(n), varbinary(n), image;`
- ❖ **date tip șiruri de caractere**: `char(n), varchar(n), text;`
- ❖ **date tip șir de caractere Unicode**: `nchar(n), nvarchar(n), ntext;`
- ❖ **date de tip dată și timp:** `date, time, datetime, datetime2, datetimeoffset, smalldatetime;`
- ❖ **date speciale:** `cursor, rowversion, hierarchyid, uniqueidentifier, sql_variant, xml, special geometry types, speciale geography types, table.`

Datele numerice exacte

- `bit` – un întreg cu valoare 0 sau 1. Reprezintă valorile logice **False** sau **True**.
- `tinyint` – numere întregi între 0 și 255. Ocupă 1 octet.
- `smallint` – numere întregi între -2^{15} până la $2^{15}-1$. Ocupă 2 octeți.
- `int` – numere întregi între -2^{31} până la $2^{31}-1$. Ocupă 4 octeți.
- `bigint` - numere întregi între -2^{63} până la $2^{63}-1$. Ocupă 8 octeți.
- `decimal(p,s), numeric(p,s)` – numere zecimale exacte (P – numărul total de cifre, S – numărul de cifre în partea zecimală).
- `money` – unități monetare. Ocupă 8 octeți.
- `smallmoney` – unități monetare. Ocupă 4 octeți.

Datele numerice aproximative

- `float(N)` – numere zecimale periodice cu precizia N.
- `real` – corespunde `float(24)`.

Datele binare

- `binary(N)` – păstrarea numerelor hexazecimale cu lungime fixă N.
- `varbinary(N)` – păstrarea numerelor hexazecimale cu lungime variabilă de maximum N.
- `image` – pentru păstrarea documentelor Word, Excel, imagini (fișiere bitmap, GIF, JPEG).

Date tip şir de caractere

- `char(N)` – secvență text cu lungime fixă N caractere.
- `varchar(N)` – secvență text cu lungime variabilă maxim N caractere.
- `text` – texte de caractere ASCII.

Date tip şir de caractere UNICODE

- `nchar(N)` – secvență de caractere UNICODE cu lungime fixă N (până la 4000 caractere).
- `nvarchar(N)` – secvență de caractere UNICODE cu lungime variabilă maxim N caractere (până la 4000 caractere).
- `ntext` – secvență de caractere UNICODE de lungime variabilă, care nu depășește valoarea 2^{30} caractere.

Date de tip oră și dată calendaristică

- `datetime` – data în interval 01.01.1773 - 31.12.9999, ora în interval 00:00:00 – 23:59:59.997 cu precizia 0,003 secunde.
- `datetime2` – data în interval 01.01.0001 – 31.12.9999, ora în interval 00:00:00:0000000 – 23:59:59.9999999 cu precizia 0,0000001 secunde.
- `date` – data în interval 01.01.0001 – 31.12.9999.
- `time` – ora în interval 00:00:00:0000000 – 23:59:59.9999999.
- `datetimeoffset` – data în interval 01.01.1773 – 31.12.9999 împreună cu ora în interval 00:00:00 – 23:59:59.9999999.
- `smalldatetime` – data și ora în interval 01.01.1900 – 06.06.2079 cu precizia 1 minut.

Date speciale

- `cursor` – este un tip de date pentru variabile și parametri de ieșire ai unei proceduri stocate care conține referință la un cursor. Orice variabilă de tip CURSOR poate avea valoarea NULL.
- `rowversion` – este un tip de date care expune numere binare unice generate automat într-o bază de date. Acest tip de date este, în general, utilizat ca mecanism pentru ștampilarea versiunilor de rânduri de tabel. Dimensiunea de stocare este de 8 octeți. Tipul de date `rowversion` este doar un număr crescător și nu păstrează o dată sau o oră.
- `hierarchyid` – este un tip de date de sistem cu lungime variabilă pentru crearea tabelelor cu structură ierarhică sau pentru referirea la date cu structură ierarhică, reprezentând pozițiile într-o ierarhie.
- `sql_variant` – este un tip de date pentru stocarea valorilor diferitor tipuri de date SQL Server, cu excepția `text, ntext, image, time_stamp` și `sql_variant`. Se utilizează pentru coloane, parametri sau variabile din cadrul funcțiilor definite de utilizatori și poate conține rânduri cu date de diferite tipuri (de exemplu, `int, binary` și `char`).
- `table` – acest tip de date este utilizat pentru stocarea unui rezultat (mulțime de rânduri) returnat de o funcție de tip tabel, care urmează ulterior să fie prelucrat.
- `uniqueidentifier` – permite desemnarea unui identificator unic global din 16 biți pentru a asigura unicitatea rândurilor în tabele.
- `geometry` – tipul de date spațiale plane.. Acest tip reprezintă datele într-un sistem de coordonate euclidian (plat).
- `geography` – tipul de date spațiale geografice este implementat ca tip de date .NET Common Language Runtime (CLR) în SQL Server. Acest tip reprezintă date într-un sistem de coordonate elipsoidale. Tipul de date geografice SQL Server stochează date elipsoidale, cum ar fi coordonatele GPS de latitudine și longitudine.
- `xml` – date de tip XML păstrate în coloană sau variabilă XML.

În continuare sunt prezentate exemple de **constante** de tip diferit:

✓ **Constante de tip char, varchar**: `'Cahul', 'reducere 5% '`

- ✓ **Constante de tip nchar, nvarchar**: N'Cahul', N'reducere 25%'
- ✓ **Constante binare**: 0xAE, 0x (empty binary string)
- ✓ **Constante datetime**: 'December 5, 1985', '14:30:24'
- ✓ **Constante de tip integer**: 1894, 3, -39
- ✓ **Constante de tip decimal:** 2.0, -45.78
- ✓ **Constante de tip float sau real**: 101.5E5, -0.5E-2
- ✓ **Constante de tip money:** -$12, $54203.14
- ✓ **Constante de tip hierarchyid:** '/2/1/1/', '/2/1.5/1/', /0.3.-7/
- ✓ **Constante de tip uniqueidentifier:** '6F9619FF-8B86-D011-B42D-00C04FC964FF'
- ✓ **Constante de tip geometry:** 'POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))'
- ✓ **Constante de tip geography:** -122.360 47.656, -122.343 47.656

În cadrul limbajului Transact-SQL sunt definite următoarele **categorii de operatori**, cu ajutorul cărora se formează expresii SQL:

- **Operatori aritmetici** (+, -, *, /, %)
- **Operator de atribuire** (=)
- **Operatori relaționali** (=, <, >, <>, !, >=, <=)
- **Operatori logici** (AND, OR, NOT, LIKE, IN, EXISTS, CONTAINS, BETWEEN, ANY, ALL, SOME)
- **Operator de concatenare a șirurilor caracteriale** (+)

Pentru a specifica anumite explicații la codul SQL se pot utiliza comentarii. Comentariul este un **text definit de utilizator**. Textul plasat între /* și */ nu va fi compilat de către server.

În cazul comentariilor extinse pe mai multe rânduri:

- **Primul rând este marcat cu simbolurile /***
- **Următoarele – cu simbolurile **
- **Ultimul – cu simbolurile */**

Nu există restricții față de numărul și lungimea comentariilor. Se acceptă comentarii imbricate. Simbolurile /* dintr-un comentariu curent sunt tratate ca început de comentariu imbricat și necesită simboluri de sfârșit de comentariu imbricat */. Dacă ele lipsesc, se generează codul de eroare. O altă modalitate de a insera un comentariu al utilizatorului pe o linie de cod este utilizarea simbolurilor –, urmat de textul comentariului.

Numele variabilelor, funcțiilor și procedurilor stocate trebuie să respecte următoarele reguli pentru identificatorii Transact-SQL.

Numele obiectului bazei de date este denumit **identificatorul** său.

Există două clase de **identificatori**:

- ❖ **Identificatorii obișnuiți** nu sunt delimitați atunci când sunt utilizați în instrucțiuni SQL.
- ❖ **Identificatorii delimitați** sunt cuprinși între ghilimele duble (") sau paranteze ([ ]).

Atât identificatorii obișnuiți, cât și cei delimitați trebuie să conțină de la 1 la 128 de caractere. Pentru tabelele temporare locale, identificatorul poate avea maximum 116 caractere.

Un identificator de obiect este creat atunci când obiectul este definit. Identificatorul este apoi folosit pentru a face referire la obiect. Primul caracter al identificatorului trebuie să fie unul dintre următoarele:

- O literă așa cum este definită de standardul Unicode 3.2. Definiția Unicode a literelor include caractere latine de la a la z, de la A la Z, precum și litere din alte limbi.
- Caracterul de subliniere (_), semnul (@) sau semnul numeric (#).

Anumite simboluri de la începutul unui identificator au o semnificație specială în SQL Server. Un identificator obișnuit care începe cu semnul @ denotă întotdeauna o variabilă sau un parametru local și nu poate fi folosit ca nume pentru orice alt tip de obiect. Un identificator care începe cu un semn numeric (#) denotă un tabel sau o procedură temporară. Un identificator care începe cu semne de număr dublu (##) denotă un obiect temporar global. Deși semnul numeric (#) sau caracterele semnului numeric dublu (##) pot fi folosite pentru a începe numele altor tipuri de obiecte, nu se recomandă de realizat această practică.

Unele funcții Transact-SQL au nume care încep cu dublu la semnele (@@). Pentru a evita confuzia cu aceste funcții, nu ar trebui să utilizați nume care încep cu @@.

Caracterele ulterioare pot include următoarele:

- Litere așa cum sunt definite în Standardul Unicode 3.2.
- Numere zecimale din latină de bază sau din alte scripturi naționale.
- Semnul @, semnul dolar ($), semnul numeric (#) sau liniuța de subliniere (_).

Identificatorul nu trebuie să fie un cuvânt rezervat Transact-SQL. SQL Server rezervă atât versiunile cu majuscule, cât și versiunile minuscule ale cuvintelor rezervate. Când identificatorii sunt utilizați în instrucțiunile Transact-SQL, identificatorii care nu respectă aceste reguli trebuie să fie delimitați prin ghilimele duble sau paranteze. Spațiile încorporate sau caracterele speciale nu sunt permise. Caracterele suplimentare nu sunt permise.

4.3. Variabile SQL

Variabilele sunt destinate pentru păstrarea unor valori. Variabila se caracterizează prin tipul de date ce poate păstra. Variabilele utilizate trebuie inițial declarate, iar apoi să le fie atribuită valoarea.

În limbajul SQL **variabilele** locale sunt obiecte ce pot să conțină un singur tip de date. Variabilele sunt utilizate în pachete sau scenarii în caz dacă este necesar:

- *de controlat numărul de execuție a unui ciclu sau de verificat valorile de control;*
- *de păstrat datele pentru returnarea lor unei proceduri stocate sau pentru furnizarea rezultatului unei funcții.*

Spre deosebire de variabilele locale **variabilele de tip tabel** pot că conține date de tipuri diferite organizate sub formă de tabel. O categorie aparte de variabile formează **variabilele de tip cursor** care permit păstrarea unor înregistrări preluate dintr-un set de date.

Instrucțiunea **DECLARE** declară o variabilă în Transact-SQL stabilind:

- ✓ *numele variabilei care va începe cu simbolul -* **@**
- ✓ ***tipul de date*** *al variabilei care poate fi unul de sistem sau de tip utilizator*
- ✓ *valoarea inițială* ***NULL.***

```
DECLARE
{   { @local_variable [AS] data_type [ = value ] }
  | { @cursor_variable_name CURSOR }  } [ ,...n ]
  | { @table_variable_name [AS] <table_type_definition> }

<table_type_definition> ::=
    TABLE ( { <column_definition> | <table_constraint> | <table_index>
} } [ ,...n ] )
```

Instrucțiunea **SET** atribuie variabilei o valoare concretă conformă tipului ei. Utilizarea instrucțiunii SET este **metoda recomandată** de atribuire valorilor pentru variabile.

```
SET
{@local_variable {+= | -= | *= | /= | %= | &= | ^= | |= } expression }
```

Exemplu.

```
DECLARE @myvar CHAR(20), @NewBalance INT = 5;
SET @myvar = 'This is a test';
SET @NewBalance *= 10;
SELECT @myvar, @NewBalance;
GO
```

Dacă **variabila unei proceduri** returnează un **rezultat extern,** atunci ea se declară cu parametrul OUTPUT.

```
DECLARE @MyInt INT OUTPUT
```

Variabila poate primi valoare și în cadrul instrucțiunii de selecție SELECT. În acest caz trebuie de respectat următoarele reguli:

- *Dacă fraza SELECT returnează **o valoare**, atunci variabila trebuie să fie **una scalară**.*
- *Dacă instrucțiunea SELECT returnează **mai multe valori**, atunci variabila trebuie să fie de **tip tablou**.*

Exemplu 1.

```
USE AdventureWorks2019;
GO
DECLARE @rows INT;
DECLARE @MyTableVar TABLE (
    Customer VARCHAR(20)  );

SET @rows = (SELECT COUNT(*) FROM Sales.Customer);
SELECT Customer INTO @MyTableVar FROM Sales.Customer);
SELECT @rows AS NrOfCustomers;
SELECT Customer AS CustomerName FROM @MyTableVar;
GO
```

4.4. Funcții SQL

Funcțiile SQL sunt fie **deterministe**, fie **nedeterministe**. Funcțiile sunt **deterministe** atunci, când returnează întotdeauna același rezultat oricând sunt apelate, folosind un set specific de valori de intrare. Funcțiile sunt **nedeterministe** atunci, când ar putea returna rezultate diferite de fiecare dată când sunt apelate, chiar și cu același set specific de valori de intrare.

De exemplu, funcția **AVG** întoarce întotdeauna același rezultat, dar funcția **GETDATE**, care returnează valoarea curentă a datei calendaristice, întoarce întotdeauna un rezultat diferit.

Următoarele funcții încorporate sunt întotdeauna **deterministe**.

EXP	MAX	MIN	SQRT	SUM
DATEDIFF	DAY	MONTH	YEAR	POWER

4.4.1. Funcții matematice

Funcțiile matematice permit realizarea unor calcule. Argumentul funcției trebuie să fie de tip **numeric**. Funcțiile matematice pot fi utilizate în cadrul **expresiilor de calcul**. În expresii pot fi utilizați operatorii aritmetici și paranteze rotunde, pentru a schimba prioritatea executării calculelor. Prioritatea implicită este: 1. executarea funcției; 2. (–) minusul unar; 3. înmulțirea sau împărțirea; 4. adunarea sau scădere.

Exemple de funcții matematice:

ABS	COS	LOG	LOG10
EXP	PI	POWER	RAND
ROUND	SIN	SQRT	

Toate funcțiile matematice, cu excepția RAND, sunt funcții deterministe. Aceasta înseamnă că returnează aceleași rezultate de fiecare dată când sunt apelați cu un set specific de valori de intrare. RAND este determinist numai atunci, când este specificat un parametru de bază.

4.4.2. Funcții de agregare

O **funcție agregată** efectuează un calcul pe un set de valori și returnează o **singură valoare**. Cu excepția COUNT(*), funcțiile agregate ignoră valorile nule. Funcțiile agregate sunt utilizate în lista unei instrucțiuni SELECT sau în clauza HAVING. Toate funcțiile agregate sunt **deterministe**. Cu alte cuvinte, funcțiile agregate returnează aceeași valoare de fiecare dată când sunt apelate, atunci când sunt apelate cu un set specific de valori de intrare.

Exemple de funcții de agregare:

AVG	MAX	SUM
COUNT	MIN	STDEV

Exemple:

```
SELECT MAX(TaxRate) FROM SalesTaxRate;
SELECT MIN(Bonus) FROM SalesPerson;
SELECT COUNT(Email) FROM Customers;
SELECT SUM(Quantity) AS Total FROM  Sales;
SELECT AVG(Quantity) FROM Sales WHERE ProductID=302;
```

4.4.3. Funcții calendaristice (data&time)

Funcțiile calendaristice procesează datele de tip `date, datetime, datetimeoffset, datetime2, smalldatetime, time.` Pot fi utilizate numerele seriale ale datei calendaristice sau orei pentru intervalele de valori definite de tipul concret de date.

Cele mai utilizate funcții sunt:

DAY	MONTH	YEAR
GETDATE	SYSDATETIME	DATEDIFF

Exemple:

```
SELECT GETDATE( );
SELECT DAY('2019-04-30 01:01:01.1234567');
SELECT YEAR(Birthday) FROM SalesPerson;
```

4.4.4. Funcții șir de caractere

Funcțiile caracteriale efectuează o operație pe o valoare de intrare a șirului și returnează o serie sau o valoare numerică. Toate funcțiile, cu excepția funcției FORMAT sunt deterministe. Atunci când unei funcții caracteriale îi sunt transmise argumente care nu sunt valori de șir, tipul de intrare este implicit convertit într-un tip de date text.

Exemple de funcții șir de caractere:

ASCII	CONCAT	LEN	LTRIM	RIGHT	STR	UNICODE
CHAR	LEFT	LOWER	REPLACE	RTRIM	TRIM	UPPER

Exemple:

```
SELECT CONCAT ( 'Happy ', 'Birthday ', 11, '/', '25' ) AS Result;
SELECT TRIM( ‘   test   ') AS Result;
SELECT REPLACE('abcdefghicde','cde','xxx');
SELECT LEFT(Name, 5) FROM Product ORDER BY ProductID;
SELECT LEN(FirstName) AS Length FROM Customer;
```

4.4.5. Funcții logice

Funcția logică IIf returnează una din cele două valori posibile în funcție de valoarea unei expresii booleene.

Sintaxa: `IIF ( boolean_expression, true_value, false_value )`

Exemplu 1.

```
DECLARE @a int = 45, @b int = 40;
SELECT IIF ( @a > @b, 'TRUE', 'FALSE' ) AS Result;
```

Funcția logică CHOOSE returnează o valoare din listă în funcție de valoarea unui index.

Sintaxa: `CHOOSE ( index, val_1, val_2 [, val_n ] )`

Exemplu 2.

```
SELECT CHOOSE (3,'Manager','Director','Developer','Tester') AS Result;
```

4.5. Expresii SQL

O **expresie SQL** reprezintă o **combinație de simboluri** și **operatori** utilizați pentru calcularea **unei singure valori ale datelor**. Deosebim expresii **simple** (constanta, variabila, atributul, funcția scalară) și **expresii complexe**.

Expresiile complexe se formează prin unirea a **cel puțin două expresii simple** cu ajutorul operatorilor SQL. În cazul **expresiilor simple** formate din constante, variabile, atribute, funcții scalare **tipul de date al expresiei** corespunde tipului de date al elementului din expresie. La unirea a două expresii cu ajutorul operatorilor relaționali sau logici rezultatul evaluării expresiei va avea **tip logic** și poate lua următoarele valori: TRUE, FALSE sau UNKNOWN.

Expresiile complexe, formate din mai multe argumente și operatori, returnează **un singur rezultat**. În acest caz se va aplica **conversia tipului de date** (câte **două argumente** pentru o evaluare de tip) până când nu se obține **rezultatul final**. În acest caz se respectă **regulile de conversie a tipurilor de date.**

Două expresii pot fi unite cu un operator oarecare dacă ele sunt de tip **corespunzător operatorului dat** și respectă măcar una din condiții:

- ✓ *Expresii conțin date de **același tip**;*
- ✓ *Tipul de date cu prioritate inferioară poate fi **convertit implicit** în tipul de date cu prioritate superioară.*

Dacă expresiile **nu satisfac** nici una din condițiile de conversie, atunci funcția **CAST** sau funcția **CONVERT** convertește **explicit** datele cu prioritate inferioară fie în date cu tip de date superior, fie în unul intermediar, care apoi poate fi **convertit implicit** în unul mai superior. Dacă conversia **implicită** sau **explicită** este imposibil de realizat, atunci expresiile nu pot fi unite.

Sortarea expresiilor care returnează un rând de simboluri este determinată de **regulile de prioritate ale operației de sortare**.

Exemplu: *Expresii numerice și caracteriale în lista SELECT.*

```
SELECT CodProdus,  PretUnit+10 AS PretNou,
‘calitatea ‘+ Calitate AS  TipCalitate
FROM Produse
```

Întrebări de control:

1. Care este diferența de spațiu de stocare și de precizie între tipurile de date numerice smallint, int și bigint în Transact-SQL?
2. Prin ce se deosebesc tipurile de date pentru șiruri de caractere varchar(n) și nvarchar(n) în privința setului de caractere și a capacității maxime?
3. Care sunt regulile de denumire a identificatorilor în Transact-SQL în ceea ce privește primul caracter pentru variabile locale, tabele temporare și obiecte temporare globale?
4. Cum se declară și cum se atribuie o valoare unei variabile locale în Transact-SQL folosind instrucțiunile DECLARE și SET?
5. Ce deosebește o funcție deterministă de una nedeterministă în Transact-SQL și în ce categorie se încadrează funcția GETDATE()?
6. Care sunt sintaxele și modurile de funcționare ale funcțiilor logice IIF și CHOOSE furnizate de Transact-SQL?

Capitolul 5. DEFINIREA DATELOR ÎN LIMBAJULUI SQL

5.1. Definirea tabelelor inițiale

Instrucțiunea CREATE TABLE **creează structura** pentru un **tabel**. Crearea unui tabel constă în generarea structurii (schemei) sale.

Sintaxa:

```
CREATE TABLE
    { database_name.schema_name.table_name | schema_name.table_name |
    table_name }  ( { <column_definition> } [ ,... n ] ) [ ; ]
```

Exemplu 1 .

```
CREATE TABLE FirmaFurnezor
(Cod_Furnizor         int         PRIMARY KEY,
Nume_Furnizor         char(25)    NOT NULL,
Adresa                char(30),
Localitate char(20)   NOT NULL DEFAULT 'Cahul',
Cod_Propietar   int   NOT NULL REFERENCES Proprietar(Cod_Proprietar)
ON DELETE CASCADE);
```

- Clauza **NOT NULL** stabilește că **nu se admit valori vide** în colanele respective.
- Clauza **DEFAULT** permite specificarea **valorii implicite** înscrise în câmpul dat.

Câmpul **Cod_Furnizor** este declarat **identificator primar**, iar câmpul **Cod_Proprietar este identificator extern** cu referință la câmpul **Cod_Proprietar** din tabelul *Proprietar.* **Identificatorul extern** în cazul dat este supus unei **constrângeri** cu ajutorul opțiunii ***ON DELETE CASCADE***, care specifică că suprimarea (ștergerea) oricărui proprietar din tabelul *Proprietar* implică **suprimarea automată** a tuturor înregistrărilor corelate care se găsesc în tabelul *FirmaFurnizor.*

Exemplu 2.

```
CREATE TABLE Produse_Facturi_Emise
(NrFactura decimal(8)      NOT NULL,
CodProdus  decimal(6)      NOT NULL,
Cantitate  decimal(12,3)   NOT NULL DEFAULT   1,
PretUnit   money           NOT NULL,
PRIMARY KEY     (NrFactura, CodProdus),
FOREIGN KEY     (NrFactura) REFERENCES Facturi(NrFactura),
FOREIGN KEY     (CodProdus) REFERENCES Produse(CodProdus),
CHECK (Cantitate > 0));
```

Clauza **CHECK** introduce **restricția** (constrângerea) de valori pentru câmpul **Cantitate**, pentru care se admit **doar valori mai mari ca zero**.

5.2. Definirea constrângerilor și reacțiilor

În cadrul instrucțiunii **CREATE TABLE** poate fi aplicat mecanismul de **constrângere** pentru câmpurile definite. **Constrângerea** este un mecanism care asigură că valorile unui atribut (coloană) sau a unei mulțimi de atribute **satisfac o condiție declarată**.

Constrângerea poate fi aplicată cu ajutorul opțiunii **CONSTRAINT** sau a unor clauze prestabilite (**NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, DEFAULT**). Unei constrângeri definite cu ajutorul opțiunii **CONSTRAINT** i se poate da un **nume unic** (etichetă).

Constrângerile pot fi șterse, pot fi adăugate, pot fi activate sau dezactivate, dar **nu pot fi** modificate. Constrângerea poate fi definită la **nivel de colană** (câmp) sau la **nivel de tabel.**

Definirea unei constrângeri se realizează cu clauza **CONSTRAINT**.

```
<column_constraint> ::=
[ CONSTRAINT constraint_name ]
{
   { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        [ ( <column_name> [ ,... n ] ) ]
          | [ FOREIGN KEY ]
        REFERENCES [schema_name.]referenced_table_name [ (ref_column)]
        [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
        [ NOT FOR REPLICATION ]
  | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}
```

Exemplu 3.

```
CREATE TABLE FirmaFurnezor
(Cod_Furnizor   int CONSTRAINT PK_F PRIMARY KEY,
Nume_Furnizor   char(25),
Adresa          varchar(50),
Localitate      varchar(50),
Cod_Propietar   int  CONSTRAINT FK_P REFERENCES
                     Proprietar(CodProprietar));
```

- *Constrângerile previn ștergerea unui tabel dacă există dependențe.*
- *Constrângerile pot fi create odată cu tabelul sau după ce acesta a fost creat.*
- *Constrângerile pot fi activate sau dezactivate în funcție de necesități.*

Constrângerile declarative se divizează în:

- ***constrângeri de domeniu***
- ***constrângere de integritate a entității***
- ***constrângere de integritate referențiale.***

Constrângerile de domeniu definesc valori admisibile ale atributului:

- ***DEFAULT*** *– stabilește valoare implicită a câmpului;*
- ***CHECK –*** *stabilește și verifică domeniul de valori admisibile;*
- ***UNIQUE –*** *admite doar valori unice pentru câmpul dat;*
- ***NOT NULL*** *– nu acceptă valori nule în câmp.*

Constrângerea de integritate a entității PRIMARY KEY precizează **cheia primară** a unui tabel. Când se creează cheia primară se generează automat un **index unic**. Valorile cheii primare sunt **distincte** și **diferite** de valoarea ***Null***.

Constrângerea de integritate referențială FOREIGN KEY asigură **coerența** între cheile primare și cheile externe corespunzătoare. Când este definită o cheie externă se verifică:

- *dacă a fost definită o cheie primară pentru tabelul referit de cheia externă;*
- *dacă numărul coloanelor ce compun cheia externă corespunde numărului de coloane a cheii primare;*
- *dacă tipul și lungimea fiecărei coloane a cheii externe corespunde cu tipul și lungimea fiecărei coloane a cheii primare.*

La definirea unui tabel pot fi setate **reacțiile** la încălcarea constrângerilor la efectuarea operațiilor de actualizare sau ștergere a datelor din tabel.

În cazul operațiilor **de actualizare** este posibilă reacția în unul din modurile următoare:

- **CASCADE**: noua valoare pentru atributul din tabelul extern va fi atribuită tuturor liniilor corespunzătoare din tabelul intern;
- **SET NULL**: valoarea NULL este asignată atributului din tabelul intern în locul valorii modificate din tabelul extern;
- **SET DEFAULT**: atributului din tabelul intern îi va fi asignată valoarea implicită în locul valorii modificate în tabelul extern;
- **NO ACTION**: operația de actualizare este rejectată.

Opțiunile disponibile în cazul încălcării constrângerii de referință **prin ștergerea** de înregistrări din tabelul extern sunt:

- **CASCADE**: vor fi șterse toate liniile din tabelul intern corespunzătoare liniei șterse din tabelul extern;
- **SET NULL**: se asignează valoarea NULL atributului din tabelul intern în locul valorii șterse din tabelul extern;
- **SET DEFAULT**: atributul din tabelul intern va primi valoarea implicită în locul valorii șterse din tabelul extern;
- **NO ACTION:** operația de ștergere este rejectată.

*Exemplu 4. Definirea **reacțiilor.***

```
CREATE TABLE Angajati
(
NrInreg    bigint     PRIMARY KEY,
Nume       char(20),
Prenume    char(20),
Dept       char(15)   REFERENCES Departament(NumeDept),
Salariu    money DEFAULT 0,
Oras       char(15),
ON DELETE SET NULL
ON UPDATE CASCADE
UNIQUE (Nume, Prenume)
)
```

5.3. Indexarea tabelelor

Instrucțiunea CREATE INDEX creează **un index** nou pe un tabel existent. Un index este un obiect al unei baze de date care:

- ***crește viteza de execuție a cererilor;***
- ***garantează că o coloană conține valori unice.***

Un index unic este **creat automat** când în definirea unui tabel apar constrângerile ***PRIMARY KEY*** sau ***UNIQUE***.

Crearea unui index pe una sau mai multe coloane ale unui tabel se face prin instrucțiunea:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
```

Un index este **o structură pe disc** asociată tabelului sau viziunii ce conține **chei** construite pe **una sau mai multe coloane** ale tabelului sau viziunii. Aceste chei sunt păstrate într-o structură (B-arbore) care permite SQL Server să găsească rapid tuplurile din tabelele indexate.

În SQL putem crea următoarele **tipuri de indecşi**:

- **CLUSTERED** – organizează liniile tabelului în aceeași ordine ca și intrările în index. Este tip de index ideal care nu implică multe actualizări. Fiecare tabel poate conține **numai un index** de acest tip. Când tabelul conține un index clustered liniile tabelului sunt **sortate** într-o anumită ordine.
- **NONCLUSTERED** – are o structură separată de cea a tabelului și ordinea liniilor (fizică) nu coincide cu ordinea indecșilor (logică). Tabelul poate conține **mai mulți indecși** de acest tip.
- **UNIQUE** – se asigură că cheia indexului este **unică** și, prin urmare, fiecare rând din tabel sau viziune tot este unic.

Fiecare tabel poate conține maximul **999 indecși Nonclustered** și **un singur index de tip Clustered.**

Clauza **INCLUDE** permite crearea unui index de tip NONCLUSTERED extins cu coloane ce nu fac parte din cheie. Clauza **WHERE** permite crearea unui index NONCLUSTERED optimizat pentru interogări ce selectează date dintr-o submulțime bine definită care folosește un predicat pentru a indexa o porțiune de linii din tabel. În clauza opțională **WITH** se pot impune **reguli de validare** utilizate la crearea indexului.

Cheia indexului poate fi **coloana** unui tabel sau **concatenarea mai multor coloane**.

Exemplu 1.

```
CREATE UNIQUE INDEX ind_DenumireaFirmei ON Firma (DenFirma ASC);
```

Pentru a **interzice valori dublate** în câmpul sau câmpurile indexate din diferite înregistrări, se utilizează cuvântul rezervat **UNIQUE**.

Exemplu 2.

```
CREATE NONCLUSTERED INDEX ind_functionari_1 ON functionari (Nume ASC,
Sector) WHERE Sector IS NOT NULL;
CREATE UNIQUE CLUSTERED INDEX ind_functionari_2 ON functionari (Cod
ASC)  WITH DROP EXISTING;
```

Instrucțiunea **ALTER INDEX** permite **modificarea indecşilor.**

Sintaxa:

```
ALTER INDEX {index_name|ALL} ON <object> {REBUILD|DISABLE|PAUSE|ABORT}
```

Exemplu 3. Reconstituirea indexului.

```
ALTER INDEX ind_functionari_1  ON functionari REBUILD
```

Exemplu 4. Dezactivarea indexului. Un index Clustered dezactivat dezactivează accesul la tabelul respectiv. Un index dezactivat poate fi ulterior reconstruit.

```
ALTER INDEX ind_functionari_1  ON functionari DISABLE
```

Recomandări privind utilizarea indecşilor:

- *se indexează tabelele pentru care interogările selectează un număr redus de rânduri (sub 5%);*
- *se indexează tabelele care sunt interogate folosind clauze SQL simple;*
- *nu se indexează tabelele ce conţin puţine înregistrări (accesul secvenţial este mai simplu);*
- *nu se indexează tabelele care sunt frecvent actualizate, deoarece ştergerile, inserările şi modificările sunt îngreunate de indecşi;*
- *se indexează coloanele folosite frecvent în clauza WHERE sau în clauza ORDER BY;*
- *nu se indexează coloanele ce conţin date asemănătoare (puţine valori distincte);*
- *se indexează coloanele care sunt utilizate pentru a face legătura dintre tabele.*

5.4. Modificarea structurii tabelelor

Instrucţiunea ALTER TABLE permite modificarea **structurii tabelului existent**. Ea permite:

- *adăugarea (ADD) de coloane, chei (primare sau externe), constrângeri într-un tabel existent;*
- *modificarea (ALTER) coloanei unui tabel;*
- *suprimarea (DROP) coloanei unui tabel sau unui index.*

Sintaxa simplificată a instrucţiunii este:

```
ALTER TABLE table_name
{
      ALTER COLUMN column_name
    {   type_name | { ADD | DROP }
    | ADD
    {<column_definition>  | <computed_column_definition> |
<table_constraint>   } [ ,...n ]
    | DROP
     [ { [ CONSTRAINT ][ IF EXISTS ]
         { constraint_name } [ ,...n ]
           COLUMN [ IF EXISTS ]
          { column_name } [ ,...n ]
    | { ENABLE | DISABLE } TRIGGER  { ALL | trigger_name [ ,...n ] }
}
[ ; ]
```

Exemplu 1. Adăugarea unui câmp textual cu numele CodFiscal.

```
 ALTER TABLE FirmaFurnezor ADD CodFiscal char(10)
```

Exemplu 2. Suprimarea câmpului cu numele CodFiscal.

```
ALTER TABLE FirmaFurnezor DROP COLUMN  CodFiscal
```

Exemplu 3. Modificarea tipului de date a câmpului Adresa.

```
ALTER TABLE FirmaFurnezor ALTER COLUMN Adresa varchar(255)
```

Exemplu 4. Setarea valorii implicite pentru câmpul Localitate.

```
ALTER TABLE FirmaFurnezor ALTER COLUMN Localitate SET DEFAULT
'Chișinău'
```

Exemplu 5. Suprimarea valorii implicite pentru câmpul Localitate.

```
ALTER TABLE FirmaFurnezor ALTER COLUMN Localitate DROP DEFAULT
```

Exemplu 6. Definirea identificatorului primar prin constrângere pentru un tabel creat fără cheie primară.

```
CREATE TABLE  FacturiNoi
   (NrFactura   char(5),
 …);
 ALTER TABLE  FacturiNoi
     ADD CONSTRAINT  cheie_primara PRIMARY KEY (NrFactura);
```

Exemplu 7. Suprimarea unei constrângeri.

```
ALTER TABLE FacturiNoi DROP CONSTRAINT cheie_primara;
```

Exemplu 8. Suprimarea identificatorului primar.

```
ALTER TABLE FirmaFurnezor DROP PRIMARY KEY
```

Dacă se încearcă ștergerea cheii primare corelată cu o cheie externă dintr-un alt tabel, atunci această ștergere nu se poate realiza (tabelele sunt legate prin declarația de cheie externă). Ștergerea este totuși permisă dacă în comanda ALTER apare opțiunea CASCADE, care determină și ștergerea cheilor externe ce se referă la cheia primară.

Exemplu 9.

```
ALTER TABLE  Proprietar DROP PRIMARY KEY CASCADE;
```

Exemplu 10. Suprimarea cheii externe.

```
ALTER TABLE FirmaFurnezor
ADD CONSTRAINT  FK_P  FOREIGN KEY (Cod_Proprietar)
REFERENCES Proprietar(CodProprietar);
ALTER TABLE FirmaFurnezor
DROP CONSTRAINT  FK_P;
```

5.5. Crearea vizualizărilor

Vizualizarea (*view*) este un **tabel logic (virtua**l) relativ la date din una sau mai multe tabele sau vizualizări. Vizualizarea este definită plecând de la **o cerere a limbajului de interogare a datelor,** moștenind caracteristicile obiectelor la care se referă. Vizualizarea, fiind virtuală, **nu solicită** o alocare de memorie pentru date.

O vizualizare reflectă la orice moment **conținutul exact** al tabelelor de bază. Orice **modificare** efectuată asupra tabelelor **se repercutează instantaneu** asupra vizualizărilor. **Ștergerea** unui tabel implică **invalidarea** vizualizărilor asociate tabelului și **nu ștergerea** acestora.

Vizualizările sunt definite pentru:

- ✓ furnizarea unui nivel mai înalt de **securizare** a bazei de date;
- ✓ **simplificarea** formulării unei cereri;
- ✓ **mascarea complexității** datelor;
- ✓ afișarea datelor **într-o altă reprezentare** decât cea a tabelelor de bază;
- ✓ asigurarea **independenței** datelor;
- ✓ asigurarea **confidențialității** anumitor informații;
- ✓ definirea **constrângerilor** de integritate;
- ✓ **restricționarea** accesului la date.

Crearea unei vederi se realizează cu ajutorul instrucțiunii CREATE VIEW:

```
CREATE [ OR ALTER ] VIEW [schema_name.] view_name [(column [ ,...n ])]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ]  [ ; ]

<view_attribute> ::= {[ENCRYPTION] [SCHEMABINDING] [VIEW_METADATA]}
```

CHECK OPTION. Forțează toate instrucțiunile de modificare a datelor executate față de vizualizare să urmeze criteriile stabilite în fraza SELECT. Când un rând este modificat printr-o vizualizare, se va asigura că datele rămân vizibile prin vizualizare după ce modificarea este efectuată. Verificarea se aplică numai actualizărilor efectuate prin vizualizare. Nu are aplicabilitate la nicio actualizare efectuată direct în tabelele de bază ale unei vizualizări.

ENCRYPTION. Criptează intrările din tabelul de sistem *sys.syscomments* care conțin textul instrucțiunii CREATE VIEW. Utilizarea WITH ENCRYPTION împiedică publicarea vizualizării ca parte a replicării SQL Server.

SCHEMABINDING. Leagă vizualizarea la schema tabelului sau tabelelor de bază. Când se specifică SCHEMABINDING, tabelul sau tabelele de bază nu pot fi modificate într-un mod care ar afecta definiția vizualizării. Definiția vizualizării în sine trebuie mai întâi modificată sau abandonată pentru a elimina dependențele de pe tabelul care urmează să fie modificat. Când se utilizează SCHEMABINDING, fraza SELECT trebuie să includă numele din două părți (**schema.obiect**) ale tabelelor, vizualizărilor sau funcțiilor definite de utilizator la care se face referire. Toate obiectele la care se face referire trebuie să fie în aceeași bază de date. Vizualizările sau tabelele care participă la o vizualizare creată cu clauza SCHEMABINDING nu pot fi abandonate decât dacă acea vizualizare este abandonată sau modificată astfel, încât să nu mai aibă legătură cu schema. În caz contrar, se va genera o eroare. De asemenea, executarea instrucțiunilor ALTER TABLE pe tabele care participă la vederi care au legătură cu schema eșuează atunci, când aceste instrucțiuni afectează definiția vederii.

VIEW_METADATA. Specifică faptul că instanța SQL Server va returna la API-urile DB-Library, ODBC și OLE DB informațiile despre metadate despre vizualizare, în loc de tabelul sau tabelele de bază, atunci când metadatele în modul de navigare sunt solicitate pentru o interogare

care face referire la vedere. Metadatele în modul de navigare sunt metadate suplimentare pe care instanța SQL Server le returnează acestor API-uri la nivel de client. Aceste metadate le permit API-urilor de pe partea client să implementeze cursoare de pe partea client care pot fi actualizate. Metadatele în modul de navigare includ informații despre tabelul de bază căruia îi aparțin coloanele din setul de rezultate. Pentru vizualizările create cu VIEW_METADATA, metadatele în modul de navigare returnează numele vizualizării și nu numele tabelului de bază atunci, când descrie coloanele din vizualizare în setul de rezultate.

Exemplu 1. Să se genereze o vizualizare cu numele, prenumele și data nașterii proprietarilor.

```
CREATE VIEW PersoaneCasatorite AS
SELECT Nume, Prenume, DataNas  FROM  Proprietar;
```

Modificarea unei vederi se realizează cu ajutorul instrucțiunii **ALTER VIEW.**

Exemplu 2. Să se modifice vederea PersoaneCasatorite ca ea să afișeze numele, prenumele, data nașterii și sexul proprietarilor.

```
ALTER VIEW PersoaneCasatorite AS
SELECT Nume, Prenume, DataNas, Sex  FROM  Proprietar;
```

Suprimarea unei vederi se realizează prin comanda *DROP VIEW* care șterge definiția vederii din baza de date.

Sintaxa: `DROP VIEW [ schema_name .] view_name;`

Exemplu: `DROP VIEW PersoaneCasatorite`

Ștergerea vederii nu va afecta tabelele relativ la care a fost definită vederea. Aplicațiile și alte vederi, care se bazează pe vizualizarea suprimată, devin invalide. În cazul vederilor trebuie respectate anumite **restricții**:

- *Nu pot fi înserate, șterse sau actualizate date din vederi ce conțin: operatorul DISTINCT; clauzele GROUP BY, HAVING; funcții agregate; operatori de mulțimi.*
- *Nu pot fi inserate sau actualizate date care ar încălca constrângerile din tabelele de bază.*
- *Nu pot fi inserate sau actualizate valorile coloanelor care rezultă prin calcul.*

Alături de restricțiile prezentate anterior, aplicabile tuturor vizualizărilor, există **restricții specifice**, aplicabile vizualizărilor bazate pe mai multe tabele. **Regula fundamentală** este că orice operație *INSERT*, *UPDATE* sau *DELETE* pe o vedere bazată pe mai multe tabele poate modifica datele **doar din unul** din tabelele de bază.

5.6. Ștergerea obiectelor bazei de date

Instrucțiunea **DROP** șterge o bază de date, un tabel sau o vedere dintr-o baza de date sau șterge un index existent dintr-un tabel. Executarea instrucțiunii **DROP** este **ireversibilă**. **Nu poate fi făcut** un *rollback* pe această comandă.

Sintaxa instrucțiunii DROP pentru ștergerea obiectelor bazei de date:

```
- Syntax for SQL Server  and Azure SQL Database
DROP DATABASE [ IF EXISTS ] { database_name } [ ,...n ] [;]
DROP TABLE [ IF EXISTS ] { database_name.schema_name.table_name |
schema_name.table_name | table_name } [ ,...n ]  [ ; ]
```

```
DROP VIEW [ IF EXISTS ] [ schema_name .] view_name [ ...,n ] [ ; ]
DROP INDEX [ IF EXISTS ] index_name ON
{database_name.schema_name.table_or_view_name |
schema_name.table_or_view_name | table_or_view_name } [ ,...n ] [;]
```

Exemplu 1. Ştergerea tabelului Temp1: `DROP TABLE TEMP1`

Exemplu 2. Ştergerea unui index: `DROP INDEX DenumireaFirmei ON Firma`

Exemplu 3. Ştergerea unei vederi cu numele Temp2: `DROP VIEW TEMP2`

Exemplu 4. Ştergerea bazei de date AUTOMOBILE: `DROP DATABASE AUTOMOBILE`

Întrebări de control:

1. Cum se definește un tabel în SQL Server folosind instrucțiunea CREATE TABLE și care este rolul clauzelor NOT NULL, DEFAULT și CHECK?
2. Care sunt cele trei categorii principale de constrângeri declarative de date și ce rol îndeplinește opțiunea CONSTRAINT?
3. Ce condiții de validare trebuie îndeplinite pentru a stabili o constrângere de integritate referențială (FOREIGN KEY) între două tabele?
4. Care sunt reacțiile posibile la încălcarea constrângerilor de referință în cazul operațiilor de ștergere sau actualizare?
5. Care este diferența dintre un index CLUSTERED și unul NONCLUSTERED, și câte opțiuni din fiecare tip pot fi create pe un singur tabel?
6. Cum pot fi utilizate instrucțiunile ALTER INDEX și ALTER TABLE pentru a modifica structura, constrângerile sau starea indecșilor dintr-un tabel?
7. Ce reprezintă o vizualizare (VIEW) și ce efect au opțiunile WITH CHECK OPTION și WITH SCHEMABINDING la crearea acesteia?
8. Care sunt restricțiile aplicabile la executarea operațiilor de inserare, actualizare și ștergere (INSERT, UPDATE, DELETE) prin intermediul unei vizualizări?

Capitolul 6. INTEROGĂRI DE SELECȚIE

6.1. Instrucțiunea SELECT

Instrucțiunea SELECT extrage datele din baze de date păstrate pe SQL Server și le prezintă utilizatorului în **formă de tabel** sau **valoare calculată**.

Majoritatea implementărilor instrucțiunii SELECT descriu următoarele aspecte ale **setului de tupluri** care va fi afișat:

1. ***Atributele*** *tabelelor ce vor fi prezente în rezultatul afișat.*
2. ***Tabelele*** *din care se vor prelua atributele și legătura logică existentă dintre tabele.*
3. ***Condițiile de filtrarea*** *a tuplurilor din tabelele inițiale, respectarea cărora va determina includerea tuplului în selecția finală.*
4. ***Modalitatea*** *în care tuplurile selectate sunt* ***ordonate*** *în selecția finală.*

Sintaxa instrucțiunii SELECT este următoarea:

```
SELECT [ALL | DISTINCT] [TOP (expression) [PERCENT] [WITH TIES]]
      { *   | {table_name | view_name | table_alias}.*
            | {[{ table_name | view_name | table_alias }.]{column_name}
            | {expression} [[AS] column_alias | expression_alias]
       } [ ,...n ]
      [ INTO new_table ]
      [ FROM { <table_source> } [[AS] table_alias] [ ,...n] ]
      [ WHERE <search_condition> ]
      [ GROUP BY { column-expression
            | ROLLUP ( <group_by_expression> [ ,...n])
            | CUBE ( <group_by_expression> [ ,...n])
            | GROUPING SETS ( <grouping_set> [ ,...n])
            | () --calculates the grand total } [ ,...n]]
      [ HAVING < search_condition > ]
[{UNION [ALL]|EXCEPT|INTERSECT}
[<SELECT_specification> [...n ] ]
      [ORDER BY order_by_expression [COLLATE collation_name]
            [ ASC | DESC ] [ ,...n]
            [{OFFSET { integer_constant | offset_row_count_expression }
            { ROW | ROWS }  [  FETCH { FIRST | NEXT } {integer_constant
            | fetch_row_count_expression } { ROW | ROWS } ONLY  ] } ]
```

unde:

`SELECT` - cuvânt rezervat al instrucțiunii;
`*` - specificarea tuturor atributelor unui tabel;
`table_source` - specificarea tabelelor-sursă și joncțiunilor dintre ele;
`search_condition` - specificarea condițiilor de filtrare a tuplurilor;
`ALL` - afișarea tuplurilor cu dubluri;
`TOP` - afișarea unui anumit număr de tupluri specificat în expresie;
`PERCENT`- afișarea unui procent de tupluri specificat în expresie;
`WITH TIES` - afișarea tuplurilor duplicate pentru ultimul tuplu selectat cu clauza TOP;
`DISTINCT` - afișarea tuplurilor distincte;
`AS` - atribuirea unui alias (etichetă) pentru atribut, expresie sau tabel;

FROM - specificarea tabelului (relației);
INTO - specificarea numelui tabelului nou creat unde se vor înscrie tuplurile selectate;
WHERE - specificarea condițiilor de selecție (filtrare);
GROUP BY - specificarea atributelor de grupare a tuplurilor selectate;
HAVING - specificarea condiției de filtrare a tuplurilor grupate;
UNION - operatorul de reuniune a două mulțimi de tupluri cu scheme compatibile;
EXCEPT - operatorul de diferență a două mulțimi de tupluri cu scheme compatibile;
INTERSECT - operatorul de intersecție a două mulțimi de tupluri cu scheme compatibile;
ORDER BY - specificarea atributelor care determină sortarea tuplurilor;
ASC - sortarea tuplurilor în ordine crescătoare a valorilor atributului specificat;
DESC - sortarea tuplurilor în ordine descrescătoare a valorilor atributului specificat;
COLLATE - specificarea setului de caractere care va determina ordinea sortării;
OFFSET - specificarea numărului de rânduri de sărit înainte din mulțimea tuplurilor selectate, care nu vor fi afișate de la începutul liste;
FETCH - limitarea numărului de rânduri afișate din lista tuplurilor selectate după execuția clauzei OFFSET.

Exemplu 1. Selectarea listei complete cu date despre angajați, sortate după numele lor în ordine crescătoare.

```
SELECT * FROM Angajat ORDER BY Nume ASC;
```

Exemplu 2. Selectarea listei formate din trei atribute din tabelul Produse.

```
SELECT CodProdus, DenProdus, PretUnit FROM Produse
```

Exemplu 3. Afișarea listei cu produsele comercializate și suma tranzacțiilor.

```
SELECT CodProdus, PretUnit, Cantitate, Cantitate*PretUnit AS Suma
FROM  Produse, Vanzari
WHERE Produse.CodProdus=Vanzari.CodProdus
```

Exemplu 4. Afișarea listei cu produsele mai scumpe de $40, sortată descrescător după preț.

```
SELECT CodProdus, DenProdus, PretUnit FROM Produse
WHERE PretUnit > $40
ORDER BY PretUnit DESC
```

Exemplu 5. Selectarea listei cu posturile de muncă distincte ale angajaților, sortate în ordine alfabetică.

```
USE AdventureWorks2019;
GO
SELECT DISTINCT JobTitle FROM HumanResources.Employee
ORDER BY JobTitle;
GO
```

6.2. Sortarea tuplurilor în interogări

Clauza **ORDER BY** realizează **sortarea** datelor returnate de o interogare în SQL Server. Ea se folosește pentru a specifica o listă de atribute, care vor determina ordinea de afișarea a tuplurilor din setul de rezultate returnat. Sintaxa clauzei:

```
[ORDER BY order_by_expression [COLLATE collation_name]
          [ ASC | DESC ] [ ,...n]
          [{OFFSET { integer_constant | offset_row_count_expression }
          { ROW | ROWS }  [  FETCH { FIRST | NEXT } {integer_constant
          | fetch_row_count_expression } { ROW | ROWS } ONLY  ] } ]
```

unde:

`order_by_expression` – specifică o coloană sau expresie pe care să sorteze setul de rezultate al interogării. O coloană de sortare poate fi specificată ca nume sau alias de coloană, sau un număr întreg non-negativ reprezentând poziția coloanei din lista selectată. Mai multe coloane de sortare pot fi specificate. Numele coloanelor trebuie să fie unice. Secvența coloanelor de sortare din clauza ORDER BY definește organizarea setului de rezultate sortate. Adică setul de rezultate este sortat după prima coloană și apoi acea listă ordonată este sortată după a doua coloană și așa mai departe. Numele de coloane la care se face referire în clauza ORDER BY trebuie să corespundă fie unui alias de coloană sau coloană din lista selectată, fie unei coloane definite într-un tabel specificat în clauza FROM, fără ambiguități. Dacă clauza ORDER BY face referire la un alias de coloană din lista selectată, alias-ul de coloană trebuie să fie utilizat autonom.
`COLLATE collation_name` – precizează că operația ORDER BY ar trebui să fie efectuată în conformitate cu colația specificată în `collation_name` și nu în funcție de colocarea coloanei așa cum este definită în tabel sau vizualizare. Sintagma `collation_name` poate fi sau un nume de colare Windows, sau un nume de colare SQL. COLLATE se aplică numai coloanelor de tip `char, varchar, nchar` și `nvarchar`.
`ASC | DESC` – specifică faptul că valorile din coloana specificată trebuie sortate în ordine crescătoare sau descendentă. ASC sortează de la cea mai mică valoare la cea mai mare valoare. DESC sortează de la cea mai mare valoare la cea mai mică valoare. ASC este **ordinea de sortare implicită**. Valorile **Null** sunt tratate ca fiind **cele mai mici valori posibile.**
`OFFSET {integer_constant|offset_row_count_expression} {ROW|ROWS}` - specifică numărul de rânduri de sărit înainte de a începe să returneze rânduri din expresia de interogare. Valoarea poate fi o constantă întreagă sau o expresie mai mare sau egală cu zero.
`FETCH { FIRST|NEXT } {integer_constant|fetch_row_count_expression} {ROW|ROWS } ONLY` - specifică numărul de rânduri de returnat după prelucrarea clauzei OFFSET. Valoarea poate fi o constantă întreagă sau o expresie mai mare sau egală cu una.

Se recomandă evitarea specificării numerelor întregi în clauza ORDER BY ca reprezentări poziționale ale coloanelor din lista selectată. De exemplu, deși o instrucțiune de tipul

```
SELECT ProductID, Nume FROM Production ORDER BY 2
```

este valabilă, instrucțiunea nu este la fel de ușor de înțeles de către ceilalți în comparație cu specificarea numelui efectiv al coloanei.

În plus, modificările aduse listei selectate, cum ar fi modificarea ordinii coloanelor sau adăugarea de noi coloane, necesită **modificarea** clauzei ORDER BY pentru a evita rezultatele neașteptate.

Într-o instrucțiune SELECT TOP (N), **se va utiliza întotdeauna** o clauză ORDER BY. Acesta este **singurul mod** de a indica în mod previzibil care rânduri sunt afectate de TOP.

Nu există nicio limită la numărul de coloane din clauza ORDER BY. Coloanele de tip `ntext, text, image, geography, geometry` și `xml` nu pot fi utilizate într-o clauză ORDER BY.

Numele coloanelor și alias-urile specificate în clauza ORDER BY trebuie definite în lista de selectare, dacă instrucțiunea SELECT conține una dintre următoarele clauze sau operatori: UNION, EXCEPT, INTERSECT, SELECT DISTINCT. În plus, atunci când instrucțiunea include un operator UNION, EXCEPT sau INTERSECT, numele coloanelor sau alias-urile coloanei trebuie specificate în lista selectată a **primei interogări** (din stânga).

Într-o interogare care folosește operatorii UNION, EXCEPT sau INTERSECT, ORDER BY este permisă numai la **sfârșitul declarației**. Această restricție se aplică numai atunci, când specificați UNION, EXCEPT și INTERSECT într-o interogare de nivel superior și nu într-o subconsultare.

Clauza ORDER BY **nu este valabilă** în vizualizări, funcții inline, tabele derivate și sub-consultări, cu excepția cazului când se folosește ORDER BY pentru a determina rândurile returnate de clauza TOP sau clauzele OFFSET și FETCH.

OFFSET și FETCH nu sunt acceptate în vizualizări indexate sau într-o vedere definită prin utilizarea clauzei CHECK OPTION.

Exemplu 1. Utilizarea ordinii de sortare implicită (ascendentă).

```
SELECT LastName, FirstName FROM Person
ORDER BY LastName, FirstName
```

Exemplu 2. Selectarea tuplurilor distincte cu nume, prenume și sexul persoanelor.

```
SELECT DISTINCT LastName, FirstName, Gender  FROM Person
ORDER BY Gender DESC
```

Exemplu 3. Selectarea informației complete din tabelul Firma

```
SELECT * FROM Firma
ORDER BY Localitate ASC, DenFirma DESC
```

Exemplu 4. Utilizarea clauzelor OFFSET și FETCH.

```
SELECT DeptID, Name FROM Department
ORDER BY DeptID
OFFSET 5 ROWS              -- sare peste primele 5 rânduri
FETCH NEXT 10 ROWS ONLY;   -- returnează 10 rânduri
```

Exemplu 5. Utilizarea instrucțiunii UNION.

```
SELECT Name, Color, ListPrice FROM Product WHERE Color = 'Red‘
UNION
SELECT Name, Color, ListPrice FROM Product WHERE Color = 'Yellow'
ORDER BY ListPrice ASC;
```

Exemplu 6. Ordonarea tuplurilor în ordinea crescătoare a anului angajării în câmpul muncii.

```
SELECT BusinessEntityID, JobTitle, HireDate
FROM HumanResources.Employee
ORDER BY DATEPART(year, HireDate);
```

Exemplu 7. Utilizarea clauzelor TOP, PERCENT și WITH TIES.

```
SELECT TOP(5) PERCENT WITH TIES JobTitle, HireDate
FROM HumanResources.Employee
ORDER BY HireDate DESC;
```

6.3. Filtrarea tuplurilor selectate

Clauza **WHERE** se utilizează pentru **filtrarea tuplurilor** în cadrul instrucțiunii **SELECT**.

În clauză WHERE se folosesc operatori relaționali (**=, <, >, <>, <=, >=**) care compară componentele unui predicat. **Predicatele** sunt expresiile care returnează valorile logice TRUE, FALSE sau UNKNOWN. Pot fi formate predicate compuse cu ajutorul operatorilor logici NOT, AND, OR. În cadrul predicatelor pot fi utilizați și operatorii BETWEEN, CONTAINS, EXISTS, FREETEXT, IN, IS NULL, IS NOT NULL, LIKE, ALL, ANY, SOME. Predicatele sunt utilizate ca **criterii de filtrare** în clauzele WHERE și HAVING ale instrucțiunii SELECT.

Exemplu 1.

```
SELECT DenProdus, PretUnit FROM Produse WHERE PretUnit <= $400
```

Pentru a compara șirurile de caractere se utilizează operatorul de comparare (=), care compară datele de tip **text, ntext, char, nchar, varchar, nvarchar**. Operatorul = verifică corespunderea exactă a două șiruri de caractere.

Operatorul LIKE verifică asemănarea unui șir cu un model oarecare. Pentru formarea unui model de căutare se utilizează caracterul %, care semnifică o consecutivitate nedefinită de caractere. Predicatul FREETEXT se utilizează în clauza WHERE pentru realizarea căutării valorilor ce corespund criteriilor de filtrare după sens, dar nu și după scriere. Predicatul CONTAINS se utilizează în clauza WHERE pentru căutarea atributelor caracteriale și verificarea corespunderii exacte sau parțiale cu unele cuvinte sau fraze.

Exemplu 2.

```
SELECT FirstName, LastName, Phone FROM Person
WHERE EmailAddress LIKE ‘%@gmail.com’
```

Exemplu 3.

```
SELECT * FROM Firma WHERE DenFirma LIKE ‘C%’
```

Exemplu 4.

```
DECLARE @SearchWord varchar(30)
SET @SearchWord ='performance'
SELECT Description FROM Product
WHERE CONTAINS(Description, @SearchWord)
```

Exemplu 5.

```
SELECT Title FROM Document
WHERE FREETEXT (Document, 'vital safety components' )
```

Exemplu 6.

```
SELECT EmployeeKey, LastName FROM DimEmployee
WHERE LastName IN ('Smith', 'Godfrey', 'Johnson')
```

Exemplu 7.

```
SELECT * FROM Firma WHERE CodFirma BETWEEN 105 AND 130
```

Exemplu 8.

```
SELECT * FROM Firma WHERE Localitate =‘Cahul’ OR Localitate=‘Orhei’
```

Exemplu 9.

```
SELECT * FROM Firma
WHERE Localitate  NOT IN (‘Cahul’, ‘Orhei’, ‘Briceni’)
```

Exemplu 10.

```
SELECT * FROM Proprietar
WHERE Prenume = ‘Ion’ AND NOT Nume = ‘Popescu’
```

Exemplu 11.

```
SELECT FirstName, LastName, Phone FROM Person
WHERE EmailAddress IS NULL
```

În clauza WHERE pot fi formate condiții de filtrare în baza unui set de valori returnate de o frază SELECT. Dacă fraza SELECT returnează **o singură valoare**, atunci ea se utilizează în predicat împreună cu operatorul =. În cazul dacă SELECT returnează **mai multe valori**, atunci fraza SELECT se utilizează în interiorul operatorilor IN, ANY, ALL, EXISTS.

Exemplu 12.

```
SELECT Nume, Prenume FROM Angajati
WHERE Salariu = SELECT MAX(Salariu) FROM Angajati
```

Exemplu 13.

```
SELECT CodProdus, DenProdus, PretUnit FROM Produse
WHERE CodProdus IN (SELECT DISTINCT CodProdus FROM Vanzari)
```

6.4. Utilizarea valorii Null în interogări

Valoarea NULL specifică că valoarea atributului este **necunoscută** și diferă de valoarea zero sau valoare lipsă. Două valori NULL sunt **echivalente**. **Compararea** valorilor NULL sau NULL și cu o altă valoare **returnează o valoare NULL**.

Pentru valori NULL se aplică următoarele convenții:

- ✓ *sunt utilizate cuvintele rezervate **IS NULL** sau **IS NOT NULL** pentru verificarea valorii NULL în clauza WHERE a instrucțiunii SELECT;*

- ✓ *În tupluri valoare necunoscută a atributului se afișează ca **NULL**;*
- ✓ *Valorile NULL pot apărea explicit în tuplurile generate de instrucțiunile INSERT sau UPDATE prin neindicarea unui atribut în cadrul acestor fraze sau la adăugarea unui nou atribut într-un tabel existent cu instrucțiunea ALTER TABLE;*
- ✓ *Valorile NULL **nu se aplică** pentru **chei primare** sau **chei externe**.*

Operatorii SQL care lucrează cu valorile NULL sunt:

- **IS NULL** – verifică dacă valoarea unui câmp dintru tuplu este NULL
- **IS NOT NULL -** verifică dacă valoarea unui câmp dintru tuplu nu este NULL
- **NULLIF (expression, expression)** – returnează valoarea **NULL**, dacă două expresii sunt **egale** sau **prima expresie** în caz contrar.
- **COALESCE (expression, …) -** returnează prima valoare din lista argumentelor, care nu este egală cu NULL. Dacă toate valorile sunt NULL operatorul COALESCE returnează valoarea NULL.

Exemplu 1. Selectarea informației despre firmele fără adresă email.

```
SELECT * FROM Firma WHERE Email IS NULL
```

Exemplu 2.

```
SELECT NULLIF(4,4) AS Egale          -- rezultat: NULL
SELECT NULLIF(5,7) AS Diferite       -- rezultat: 5
```

Exemplu 3.

```
SELECT COALESCE(NULL, NULL, 'BMW', 'Audi')      -- rezultat: BMW
```

Întrebări de control:

1. Care sunt clauzele principale ale instrucțiunii SELECT în Transact-SQL și ce rol îndeplinește fiecare în generarea setului de date ?
2. Ce efect au opțiunile TOP, PERCENT și WITH TIES aplicate în fraza SELECT și de ce este obligatorie utilizarea clauzei ORDER BY împreună cu acestea?
3. Care este diferența dintre utilizarea cuvântului cheie DISTINCT și opțiunea implicită ALL în cadrul unei interogări de selecție?
4. Cum funcționează combinarea clauzelor OFFSET și FETCH în cadrul ORDER BY pentru realizarea paginării rezultatelor unei interogări?
5. Care sunt restricțiile privind utilizarea tipurilor de date speciale (cum ar fi ntext, text, image, xml, geography) în cadrul clauzei ORDER BY?
6. Prin ce se deosebesc operatorii și predicatele de filtrare din clauza WHERE, precum LIKE, CONTAINS, FREETEXT, BETWEEN și IN?
7. Cum este tratată valoarea NULL în cadrul operațiilor de comparare?
8. Care sunt funcțiile/operatorii T-SQL dedicați manipulării valorilor NULL și cum diferă comportamentul acestora?

Capitolul 7. INTEROGĂRI DE GRUPARE

7.1. Utilizarea funcțiilor de agregare în interogări

Funcțiile de agregare efectuează calcule asupra unei mulțimi de date și returnează o singură valoare numerică. Toate funcțiile de agregare, cu excepția COUNT(*), nu iau în considerație valorile NULL.

Funcțiile de agregare se utilizează doar:

- În lista instrucțiunii SELECT.
- În instrucțiunile COMPUTE sau COMPUTE BY.
- În clauza HAVING.

Limbajul SQL oferă următoarele **funcții de agregare**:

- **AVG** - calculează valoarea medie;
- **MIN** - determină valoarea minimală;
- **MAX** - determină valoarea maximală;
- **SUM** - calculează suma valorilor numerice;
- **COUNT** - determină numărul valorilor numerice în format `int`;
- **STDEV** - calculează abaterea medie standard.

Funcția COUNT returnează numărul de valori într-un grup. Funcția COUNT returnează totdeauna o valoare de tip int. Funcția COUNT(*) returnează numărul de valori într-un grup, **inclusiv NULL** și valori ce se **repetă**. Funcția COUNT (ALL expression) evaluează expression pentru fiecare tuplu din grup și returnează numărul valorilor ce nu sunt egale cu NULL. Funcția COUNT(DISTINCT expression)) evaluează expression pentru fiecare tuplu din grup și returnează numărul valorilor **unice** ce nu sunt egale cu NULL.

Să examinăm câteva exemple de utilizare a funcțiilor de agregare în cadrul interogărilor de selecție de totalizate. Fie avem două relații: *Angajați*(AngId, Nume, Prenume, Functie, Salariu, IdDept) și *Departament*(IdDept, NumeDept, Oras). Asupra lor pot fi aplicate următoarele interogări de selecție:

Exemple:

```
SELECT COUNT(*)AS NrContabili FROM Angajati
WHERE Functie='contabil';
SELECT COUNT(DISTINCT Salariu) AS NrSalariiDistincte FROM Angajati;
SELECT COUNT(ALL Salariu) AS NrSalarizati FROM Angajati;
SELECT AVG(Salariu) AS Salariu Mediu FROM Angajati;
GO
```

7.2. Interogări de grupare

Gruparea tuplurilor returnate de instrucțiunea SELECT se realizează cu ajutorul clauzei **GROUP BY**. Această clauză realizează gruparea tuplurilor după valorile înscrise în **una sau mai multe coloane** ale tabelului sau după valorile returnate de o expresie SQL. Ca rezultat al grupării:

- Este returnat doar **un singur tuplu** (rând) pentru **fiecare grup** creat.
- Funcțiile de agregare din lista frazei SELECT returnează **o singură valoare** calculată pentru **fiecare grup** de tupluri.

```
GROUP BY { column-expression
        | ROLLUP ( <group_by_expression> [ ,...n])
        | CUBE ( <group_by_expression> [ ,...n])
        | GROUPING SETS ( <grouping_set> [ ,...n])
        | () --calculates the grand total } [ ,...n]]
   [ HAVING < search_condition > ]
```

În sintaxa clauzei GROUP BY *column-expression* specifică o coloană sau un calcul neagregat pe o coloană. Această coloană poate aparține unui tabel, tabel derivat sau vizualizare. Coloana trebuie să apară în clauza FROM a instrucțiunii SELECT, dar nu este necesar să apară în lista SELECT. Cu toate acestea, fiecare coloană de tabel sau de vizualizare din orice expresie **neagregată** din lista instrucțiunii SELECT **trebuie să fie inclusă** în lista GROUP BY.

Expresia coloană *column-expression* **nu poate** conține:

- Un alias de coloană care este definit în lista SELECT. Poate folosi un alias de coloană pentru un tabel derivat care este definit în clauza FROM.
- O coloană de tip **text**, **ntext** sau **imagine**. Cu toate acestea, se poate utiliza o coloană de text, ntext sau imagine ca argument pentru **o funcție** care returnează o valoare a unui tip de date valid. De exemplu, expresia poate folosi SUBSTRING() și CAST(). Acest lucru se aplică și expresiilor din clauza HAVING.
- Metode de tip de date **xml**, dar poate include o funcție definită de utilizator care utilizează metode de tip de date **xml**. Poate include o coloană calculată care utilizează metode de tip de date **xml**.
- O subconsultare SELECT (subquery).
- O coloană dintr-o vizualizare indexată.

Clauza **HAVING** se folosește pentru filtrarea tuplurilor grupate după o condiție specificată. Clauza HAVING poate fi utilizată doar cu instrucțiunea SELECT în cadrul frazei GROUP BY.

În cazul dacă GROUP BY este lipsă, atunci clauza HAVING lucrează similar clauzei WHERE.

Fie este dat tabelul inițial **Sales**, asupra cărora vor fi aplicate un șir de interogări.

Country	Region	Sales
Canada	Alberta	100
Canada	British Columbia	200
Canada	British Columbia	300
United States	Montana	100

Exemplu 1. Utilizarea clauzei GROUP BY.

```
SELECT Country, Region, SUM(Sales) AS TotalSales FROM Sales
GROUP BY Country, Region;
```

Rezultat:

Country	Region	TotalSales
Canada	Alberta	100
Canada	British Columbia	500
United States	Montana	100

Exemplu 2. Utilizarea clauzei GROUP BY împreună cu clauza HAVING.

```
SELECT Country, Region, SUM(Sales) AS TotalSales FROM Sales
GROUP BY Country, Region HAVING Country='Canada';
```

Rezultat:

Country	Region	TotalSales
Canada	Alberta	100
Canada	British Columbia	500

Clauza **ROLLUP** permite să fie generate tuplurile unei fraze simple GROUP BY și să fie calculate **subtotaluri** pentru **valorile numerice** din **tuplurile grupate,** precum și **totalul general**. Pentru fiecare **valoare unică** din grup se **generează un subtotal.** Valorile grupate din coloane se adună **de la dreapta spre stânga.** La final se calculează **totalul general**.

Ordinea rezultatului final și numărul de tupluri generate este determinată de ordinea atributelor din lista ROLLUP. De exemplu, GROUP BY ROLLUP (col1, col2, col3, col4) creează grupuri pentru **fiecare combinație** de expresii de coloane din listele următoare:

- col1, col2, col3, col4
- col1, col2, col3, NULL
- col1, col2, NULL, NULL
- col1, NULL, NULL, NULL
- NULL, NULL, NULL, NULL -- Este totalul general

Exemplu 3. Utilizarea clauzei GROUP BY împreună cu clauza ROLLUP.

```
SELECT Country, Region, SUM(Sales) AS TotalSales FROM Sales
GROUP BY ROLLUP(Country, Region);
```

Rezultat:

Country	Region	TotalSales
Canada	Alberta	100
Canada	British Columbia	500
Canada	NULL	600
United States	Montana	100
United States	NULL	100
NULL	NULL	700

Clauza **CUBE** permite să fie formate tuplurile unei fraze simple GROUP BY, tuplurile cu calcule agregate ale instrucțiunii ROLLUP și tuplurile cu rezultatele calculelor încrucișate. Tuplurile generate de CUBE sunt tupluri grupate pentru toate permutările ale atributelor din lista instrucțiunii CUBE. Pentru GROUP BY CUBE (a, b) rezultatele au grupuri pentru valori unice de (a, b), (NULL, b), (a, NULL) și (NULL, NULL). Numărul de grupe generate este egal cu 2^n, unde n – numărul de atribute din lista instrucțiunii CUBE. Se formează un tuplu pentru fiecare combinație unică a valorilor atributelor din listă cu un subtotal pentru fiecare rând și un total general. Tuplurile generate de instrucțiunea CUBE nu depind de ordinea atributelor din listă.

Exemplu 4. Utilizarea clauzei GROUP BY împreună cu clauza CUBE.

```
SELECT Country, Region, SUM(Sales) AS TotalSales FROM Sales
GROUP BY CUBE(Country, Region);
```

Rezultat:

Country	Region	TotalSales
Canada	Alberta	100
NULL	Alberta	100
Canada	British Columbia	500
NULL	British Columbia	500
United States	Montana	100
NULL	Montana	100
NULL	NULL	700
Canada	NULL	600
United States	NULL	100

Clauza **GROUPING SETS** oferă posibilitatea de a combina mai multe clauze GROUP BY. Fraza GROUP BY utilizată cu clauza GROUPING SETS generează **aceeași mulțime de tupluri** ca și câteva fraze simple GROUP BY, unite cu ajutorul instrucțiunii UNION ALL.

GROUP BY ( ) specifică crearea unui grup gol, care generează totalul general. Acesta este util ca unul dintre elemente în cadrul clauzei GROUPING SETS. De exemplu, declarația de mai jos oferă rezultatul pentru vânzările totale pentru **fiecare țară** și apoi oferă **totalul general** pentru toate țările.

Exemplu 6:
```
SELECT Country, SUM(Sales) AS TotalSales FROM Sales
GROUP BY GROUPING SETS ( Country, () );
```

Rezultat:

Country	TotalSales
Canada	600
United States	100
NULL	700

Întrebări de control:

1. Cum funcționează funcțiile de agregare în privința valorilor NULL?
2. Unde pot fi utilizate funcțiile de agregare în cadrul unei interogări SQL Server?
3. Care este regula principală privind includerea coloanelor neagregate din lista SELECT în clauza GROUP BY?
4. Ce tipuri de date, expresii sau structuri sunt interzise de a fi folosite direct într-o expresie de coloană din cadrul clauzei GROUP BY?
5. Care este diferența dintre filtrarea datelor realizată prin clauza WHERE și cea efectuată prin clauza HAVING?
6. Cum generează opțiunea ROLLUP subtotalurile și totalul general, și cum influențează ordinea coloanelor din listă numărul de grupuri create?
7. Cum se determină numărul de combinații de grupuri generate de opțiunea CUBE în funcție de numărul n de atribute specifice?
8. Care este rolul clauzei GROUPING SETS și ce efect are specificarea unui grup gol () în cadrul acesteia?

Capitolul 8. JONCȚIUNI ȘI SUBCONSULTĂRI

8.1. Joncțiuni interne și externe

Joncțiunea tabelelor poate fi realizată cu clauzele **FROM** sau **WHERE**. **Se recomandă** realizarea joncțiunilor prin clauza **FROM**. Clauzele **WHERE** și **HAVING** pot conține criterii de selecție pentru realizarea **filtrării tuplurilor,** formate ca urmare a joncțiunii prin utilizarea clauzei **FROM.**

Există următoarele tipuri de joncțiuni:

- **Joncțiuni interne** (sunt operații tipice de joncțiune care utilizează operatorii relaționali = sau < > pentru stabilirea unei **corelări între tuplurile a două relații în baza valorilor comune din unul sau mai multe atribute (coloane)** din fiecare tabel).
- **Joncțiunile externe,** care pot fi de **stânga** (*left join*), de **dreapta** (*right join*) sau **totale** (*full join*).

Joncțiunea internă este o joncțiune în care se verifică **valorile comune** ale atributelor din tabele diferite prin utilizarea **operatorilor de comparare**. În standardul SQL joncțiunea internă se realizează cu clauzele **FROM** sau **WHERE**. Sunt **unicele clauze** utilizate pentru realizarea joncțiunilor în standardul SQL.

Joncțiunile interne realizate cu clauza **WHERE** se mai numesc **joncțiuni interne de tip vechi**. Dacă se utilizează operatorul = în cadrul clauzei **WHERE**, atunci acest tip de joncțiune se mai numește **echijoncțiune**. În cazul echijoncțiunei se vor selecta doar acele tupluri din tabele joncționabile, care au **valori egale** ale atributelor prin care se realizează joncțiunea tabelelor.

*Exemplu de **joncțiune cu clauza WHERE** : să se afișeze codul și denumirea produselor din relația Produse pentru care **nu există** prețuri actualizate în relația PreturiNoi.*

```
SELECT CodProdus, DenProdus
FROM Produse, PreturiNoi
WHERE Produse.CodProdus<>PreturiNoi.CodProdus
```

*Exemplu de **echijoncțiune cu clauza WHERE**: să se afișeze denumirea produselor din relația Produse, care au fost măcar o dată comercializate.*

```
SELECT DISTINCT DenProdus
FROM Produse, Vanzari
WHERE Produse.CodProdus=Vanzari.CodProdus
```

Auto-joncțiunea unei relații este un procedeu de joncțiune a unei relații **cu ea însuși**, pentru a identifica **anumite valori** ale atributelor, comparându-le cu alte valori ale acestor atribute. În cazul auto-joncțiunii sunt utilizate **aleas-uri** pentru **relații** și pentru **identificarea atributelor** în clauza **WHERE**.

Exemplu 1. Să se afișeze lista clienților din relația Customers, care locuiesc în același oraș și au același nume.

```
SELECT A.CustomerName AS Name1, B.CustomerName AS Name2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.CustomerName = B.CustomerName AND A.City = B.City
ORDER BY A.City;
```

Cuvintele rezervate **INNER JOIN** sunt utilizate pentru realizarea **joncțiunii interne** în cadrul clauzei **FROM** (metodă recomandată de joncțiune a tabelelor).

Sintaxa:

```
FROM first_table < join_type > second_table [ ON ( join_condition ) ]
```

Exemplu 2.

```
SELECT DISTINCT  Produse.DenProdus, Vanzari.Cantitate
FROM Produse INNER JOIN Vanzari
ON Produse.CodProdus = Vanzari.CodProdus
WHERE Vanzari.Cantitate >100
```

Joncțiunea a două tabele poate fi realizată după **mai multe atribute comune**. În acest caz se utilizează mai multe cuvinte rezervate **ON** combinate cu operatorii logici **OR** sau **AND**.

Sintaxa:

```
SELECT list_columns
FROM table1  INNER JOIN table2
ON table1.column1=table2.column1 [AND
(ON table1.column2=table2.column2) OR
(ON table1.column3=tabel2.column3)];
```

Exemplu 3.

```
SELECT Produse.Producator, Produse.DenProdus, ProduseNoi.PretNou
FROM Produse INNER JOIN ProduseNoi
ON Produse.Producator = ProduseNoi.Producator AND
ON Produse.DenProdus=ProduseNoi.DenProdus
```

În cadrul instrucțiunii INNER JOIN pot fi utilizate alte instrucțiuni INNER JOIN pentru a putea realiza joncțiunea dintre **trei sau mai multe relații**. Se utilizează următoarea sintaxă:

```
SELECT list_columns
FROM table1  INNER JOIN
(table2 INNER JOIN [( ] table3
[INNER JOIN [( ] tableX [INNER JOIN ...)]
ON table3.column3= tableX.columnX)]
ON table2.column2= table3.column3)
ON table1.column1= table2.column2;
```

Exemplu 4. De afișat codurile și denumirile produselor comercializate în primele trei luni ale anului 2026.

```
SELECT DISTINCT Vanzari.CodProdus, Produse.DenProdus
FROM Produse INNER JOIN (Facturi INNER JOIN Vanzari
ON Facturi.NrFactura=Vanzari.NrFactura)
ON Produse.CodProdus=Vanzari.CodProdus
WHERE YEAR(Facturi.DataFactura) = 2026
AND MONTH(Facturi.DataFactura) IN(1,2,3)
```

Joncțiunile externe returnează **toate tuplurile** ale măcar unui tabel sau viziuni specificate în clauza **FROM**, în cazul când valorile din atribute, ce determină joncțiunea, corespund **criteriilor de filtrare** din clauzele **WHERE** sau **HAVING**.

Toate tuplurile din **tabelul de stânga** din fraza **JOIN** formează **joncțiunea externă de stânga,** cele obținute din tabelul din dreapta formează **joncțiunea externă de dreapta**. Toate tuplurile din **ambele tabele** se returnează în **joncțiunea externă totală (FULL OUTER JOIN)**. Câmpurile din tuplurile joncționate, care **nu conțin date** în tabelele inițiale, se completează cu valoarea **NULL**.

Pentru joncțiuni externe cu clauza **FROM** aplicația SQL Server utilizează cuvinte rezervate ISO:

- **LEFT JOIN;**
- **RIGHT JOIN;**
- **FULL OUTER JOIN.**

Sintaxa:

```
FROM table1 [ LEFT | RIGHT| FULL OUTER ] JOIN table2
ON table1.join_column=table2.join_column
```

Exemplu 5. Să se afișeze codurile și denumirile produselor care au participat la tranzacții comerciale.

```
SELECT DISTINCT Produse.CodProdus, Produse.DenProdus
FROM Produse RIGHT JOIN Vanzari
ON Produse.CodProdus = Vanzari.CodProdus
```

Joncțiunea realizată **fără clauza WHERE**, realizează un **produs cartezian** al tuplurilor tabelelor specificate în clauza **FROM**. Același produs cartezian poate fi obținut în mod intenționat prin utilizarea cuvintelor rezervate **CROSS JOIN**. Dimensiunea mulțimii tuplurilor rezultante este calculată prin înmulțirea numărului de tupluri din primul tabel cu numărul tuplurilor existente în tabelul al doilea. Produsul cartezian sau joncțiunea încrucișată returnează **toate combinațiile posibile** ale tuplurilor din tabelele joncționate.

Exemplu 6.

```
--  Generarea unui produs cartezian
SELECT Facturi.NrFactura, Firma.DenFirma
FROM Facturi, Firma
-- Varianta echivalentă a codului
SELECT Facturi.NrFactura, Firma.DenFirma
FROM Facturi CROSS JOIN Firma
```

8.2. Interogări cu operatori pe mulțimi

Regulile de bază pentru utilizarea operatorilor pe mulțimi UNION, INTERSECT și EXCEPT sunt următoarele:

- **Numărul și ordinea atributelor** trebuie să fie **identice în toate frazele SELECT**
- **Tipurile de date** ale atributelor trebuie să fie **compatibile**.

Operatorul **UNION ALL** permite unirea a două sau mai multe mulțimi de tupluri returnate de frazele SELECT într-o **singură mulțime**. În această mulțime vor fi prezente **toate tuplurile** (cu dubluri) din toate interogările SELECT din fraza complexă. Dacă se utilizează doar **UNION,** atunci **tuplurile duplicate** se afișează ca **unul singur**.

Exemplu 1.

```
SELECT * FROM Facturi
UNION ALL
SELECT * FROM FacturiSuplimentare
```

Operatorii **EXCEPT** и **INTERSECT** returnează tupluri diferite, comparând rezultatele a două interogări **SELECT**. Operatorul **EXCEPT** returnează **toate tuplurile distincte** din interogarea **SELECT** din **stânga,** care **nu sunt prezente** în tuplurile returnate de interogarea **SELECT** din **dreapta**. Operatorul **INTERSECT** returnează **toate tuplurile distincte,** care sunt prezente **atât** în interogarea **SELECT** din **stânga**, **cât** și în interogarea **SELECT** de **dreapta**.

Dacă **tipul de date ale atributelor** din frazele **SELECT** din stânga sau dreapta operatorilor **EXCEPT** sau **INTERSECT** sunt cele **caracteriale**, atunci compararea lor se realizează după **regulile de prioritate a sortării**. Două valori **NULL** se consideră **egale**.

Numele atributelor returnate de operatorul **EXCEPT** sau **INTERSECT** corespunde cu **numele atributelor** din interogarea SELECT **de stânga**.

Numele atributelor sau etichetelor din fraza **ORDER BY** trebuie să corespundă **numelor atributelor** interogării SELECT **de stânga**.

Posibilitatea păstrării valorii **NULL** într-o coloană din tuplurile returnate de **EXCEPT** sau **INTERSECT** depinde de **constrângerile atributului** respectiv din interogarea **SELECT de stânga**.

Dacă operatorii **EXCEPT** sau **INTERSECT** sunt utilizate într-o expresie cu alți operatori, atunci ea este evaluată respectând următoarea **prioritate:** 1) **Expresiile din paranteze**; 2) Operatorul **INTERSECT**; 3) Operatorii **EXCEPT** și **UNION** executați consecutiv **de la stânga la dreapta.**

Exemplu 2.

```
SELECT * FROM ProduseNoi
EXCEPT
SELECT * FROM Produse
```

Exemplu 3.

```
SELECT CodProdus, DenProdus FROM Produse
     INTERSECT
SELECT CodProdus, DenProdus FROM ProduseNoi
ORDER BY Produse.CodProdus ASC
```

8.3. Subconsultări (subqueries)

Subconsultarea (**subquery**) reprezintă o **interogare imbricată** într-o instrucțiune **SELECT**, **INSERT, UPDATE** sau **DELETE** sau într-o **altă subconsultare**. Subconsultarea poate fi utilizată oriunde unde sunt admise expresiile SQL. Subconsultarea mai este numită **interogare internă**, iar instrucțiunea ce conține o subconsultare se numește **interogare externă**. O subconsultare **SELECT totdeauna este încadrată în paranteze rotunde**.

Subconsultarea **nu poate conține** instrucțiunile COMPUTE sau FOR BROWSE și poate conține fraza **ORDER BY numai împreună** cu clauza **TOP**.

Subconsultarea poate fi **imbricată** în fraza **WHERE** sau **HAVING** a interogării externe **SELECT, INSERT, UPDATE** sau **DELETE**, sau într-o altă subconsultare. Pot fi create

subconsultări imbricate până la **nivelul 32.** Subconsultarea poate fi utilizată oriunde se permite o expresie SQL cu condiția că ea **returnează o singură valoare**.

Dacă **tabelul** este referit într-o subconsultare **și nu este prezent** în **interogarea externă**, atunci atributele acestui tabel **nu pot fi incluse** în **lista atributelor returnate de interogarea externă.**

O subconsultare **imbricată** în instrucțiunea exterioară SELECT are următoarele componente:

- interogare SELECT obișnuită, care include componentele listei de selectare obișnuite.
- clauză FROM obișnuită, care include unul sau mai multe nume de tabel sau de vizualizare.
- clauză WHERE opțională.
- clauză GROUP BY opțională.
- clauză HAVING opțională.

Instrucțiunile care utilizează o subconsultare (subquery) posedă uzual următorul format:

```
WHERE expression [NOT] IN (subquery)
WHERE expression   comparison_operator  [ANY | ALL] (subquery)
WHERE [NOT] EXISTS (subquery)
```

Există trei tipuri de subconsultări, care:

- *se utilizează cu operatorii **IN**, **ANY** sau **ALL** pentru realizarea unei **liste de comparație;***
- *se folosesc cu operatorul = și returnează **o singură valoare**;*
- *sunt teste pentru **verificare existenței unor tupluri** și se folosesc cu operatorul **EXISTS**.*

Exemplu 1. De afișat codul și denumirea produselor care niciodată nu au fost vândute.

```
SELECT     CodProdus, DenProdus
FROM  Produse
WHERE CodProdus NOT IN (SELECT DISTINCT CodProdus FROM Vanzari);
```

Exemplu 2. De afișat informația despre facturile emise pe aceeași dată când a fost emisă factura cu numărul 1123.

```
SELECT * FROM Facturi
WHERE DataFactura IN (SELECT DataFactura
FROM Facturi WHERE NrFactura=1123)
```

Exemplu 3. De afișat informația completă despre cel mai ieftin produs la depozit.

```
SELECT *
FROM Produse
WHERE PretUnit=(SELECT MIN(PretUnit) FROM Produse)
```

O subconsultare poate fi inclusă ca argument al operatorilor **ANY** sau **ALL**. Acești operatori relaționali selectează o valoarea oarecare din lista de valori , returnată de subconsultare. Operatorii ANY și ALL pot fi combinați cu operatorii de comparație <, >, =, !=. Semnificația acestor combinări este următoarea:

< ANY ⇔ mai mic ca maximum;
> ANY ⇔ mai mare ca minimum;

= ANY ⇔ IN.

< ALL ⇔ mai mic ca minimum;
> ALL ⇔ mai mare ca maximum;
! = ALL ⇔ NOT IN.

Exemplu 4. De afișat numărul facturilor emise firmelor, care au codurile mai mari decât cel mai mare cod al firme, căreia i s-a eliberat facturi pe data 17.06.2022.

```
SELECT NrFactura
FROM Facturi
WHERE CodFirma>ALL
          (SELECT CodFirma
          FROM Facturi
          WHERE DataFactura=#17/06/2022#)
```

În cazul utilizării subconsultărilor este necesar de respectat un șir de reguli:

1. **Lista atributelor** unei subconsultări, utilizate cu un operator relațional (de comparație), poate să conțină **doar o singură expresie sau atribut** (cu excepția operatorilor **EXISTS** și **IN,** care sunt utilizați cu instrucțiunea **SELECT *** sau **o listă de atribute selectate**).
2. Dacă fraza **WHERE** a interogării externe conține un atribut, el trebuie **să fie joncționabil** cu **un atribut** din **lista de atribute** din subconsultare.
3. Atributele definite cu tipurile de date ntext, text și image **nu pot fi** utilizate în lista atributelor din subconsultare.
4. Subconsultările cu un operator relațional (după care **nu urmează ANY** sau **ALL), nu pot conține** frazele **GROUP BY** și **HAVING**, deoarece ele trebuie să returneze **doar o singură valoare.**
5. Clauza **DISTINCT nu poate fi utilizată** în subconsultare împreună cu fraza **GROUP BY**.
6. **Nu pot fi utilizate** în subconsultări instrucțiunile **COMPUTE** și **INTO**.
7. Fraza **ORDER BY** poate fi utilizată **numai împreună** cu clauza **TOP**.
8. Viziunea, creată cu o subconsultare, **nu poate fi actualizată.**
9. **Lista atributelor** unei subconsultări, care începe cu cuvântul cheie **EXISTS**, **implicit** conține semnul (*) în locul numelui atributelor.

Exemplu 5. De afișat numele furnizorilor, care au livrat produse cu prețul 220 lei.

```
SELECT NumeFurnozor FROM Furnizori
WHERE EXISTS
     (SELECT * FROM Produse
     WHERE FurnizorId = Furnizori.FurnizorId AND PretUnit = 220);
```

8.4. Interogări corelate

Majoritatea interogărilor sunt obținute prin executarea unei subconsultări și utilizarea rezultatelor ei în fraza **WHERE** din **interogarea externă**.

În **interogările corelate** subconsultarea **depinde de valorile** obținute de **interogarea externă**. Subconsultarea se execută **câte o dată** pentru **fiecare valoare** selectată de **interogarea externă**. Logica executării unei interogări corelate este prezentată în Figura 8.1.

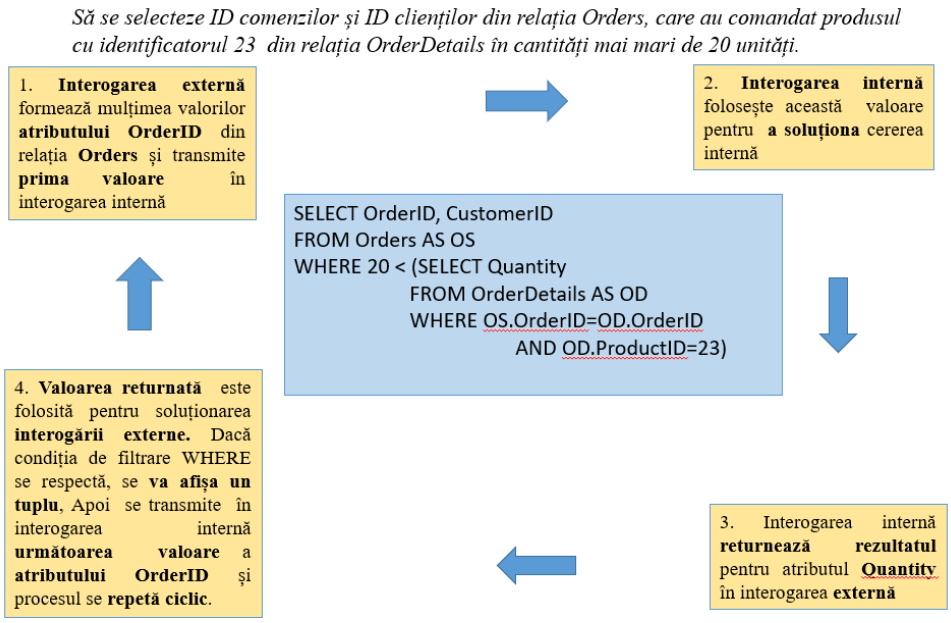


Figura 8.1. Logica executării unei interogări corelate

Ca și în cazul subconsultărilor, există un șir de reguli de respectat la elaborarea interogărilor corelate:

1. Dacă subconsultarea este utilizată cu operatorul **EXISTS**, atunci ea este folosită pentru verificarea **existenței tuplurilor respective** în tabelele sau viziuni.
2. În fraza **WHERE** a **interogării externe** se verifică existența tuplurilor generate de subconsultare.
3. Subconsultarea **nu returnează tupluri, ci doar valorile logice** TRUE sau FALSE.
4. În fața operatorului **EXISTS** nu se scrie numele atributului, constantei sau expresiei.
5. Lista atributelor unei subconsultări create cu EXISTS conține semnul asterix (*).

Exemplu 6. Determinarea producătorilor de laptop-uri care produc şi imprimate

```
SELECT DISTINCT Producator FROM Produse AS LP
WHERE TipProdus='Laptop' AND
EXISTS
(SELECT * FROM Produse
WHERE TipProdus='Printer' AND Producator=LP.Producator);
```

8.5. Expresii tabelare comune (CTE)

Expresiile tabelare comune (**CTE - Common Table Expression**) pot fi concepute ca **tupluri rezultante temporale** asupra cărora pot fi aplicate instrucțiunile **SELECT**, **INSERT, UPDATE, DELETE** sau **CREATE VIEW**.

CTE **nu se păstrează** în baza de date ca obiecte, **timpul lor de viață este limitat** pe **durata** realizării interogării. Dar, spre deosebire de interogări, CTE pot să facă referință la **ele însuși și recursiv**.

CTE sunt utilizate:

- *pentru realizarea **interogărilor recursive**.*
- *pentru **înlocuirea vederilor** în cazurile când nu este necesar de a le salva în baza de date.*
- *pentru **referințe multiple** la o mulțime de tupluri din una și aceeași instrucțiune.*

Utilizarea CTE sporește **gradul de înțelegere** și **simplifică** lucrul cu **interogările complexe**, divizându-le în blocuri constructive **logice**. Din aceste blocuri pot fi constituite CTE intermediare

mai complexe pentru realizarea unei interogări cu grad de complexitate sporit. CTE pot fi definite în subprograme ale utilizatorului (**funcții, proceduri, triggeri, viziuni**).

CTE constă din **nume**, o **listă opțională de atribute** și **o interogare SELECT** ce definește CTE. După definire **numele CTE** poate fi folosit ca **referință** în instrucțiunile **SELECT, INSERT, UPDATE** și **DELETE** la fel ca numele unui tabel sau vedere. CTE poate fi folosită în instrucțiunea **CREATE VIEW**.

O expresie tabelară comună se creează cu ajutorul operatorului **WITH**.

Sintaxa:

```
WITH query_name [(column_name1, ...)] AS (SELECT ...)
```

Pentru a defini CTE: a) se alege un nume și o listă de atribute; b) se creează o interogare CTE; c) Se utilizează numele CTE într-o interogare SQL.

Exemplu 1.

```
-- Se creează CTE
WITH TopSales_CTE (SalesPersonID, NumSales)
AS
     ( SELECT SalesPersonID, Count(*)
     FROM Sales.Orders
     WHERE SalesPersonID IS NOT NULL
     GROUP BY SalesPersonId )
-- Se utilizează CTE în interogări
SELECT *
FROM TopSales_CTE
ORDER BY NumSales DESC;
SELECT AVG(NumSales) AS "Average Sales Per Person"
FROM TopSales_CTE;
GO
```

Expresiile tabelare comune pot fi utilizate pentru crearea interogărilor recursive. **CTE recursive** sunt CTE ce se repetă de mai multe ori, returnând submulțimi de tupluri până la completarea rezultatului.

Pentru a crea o CTE recursivă:

- ***Se creează o interogare –ancoră a unui arbore recursiv;***
- ***Se adaugă operatorul UNION ALL;***
- ***Se creează o interogare recursivă cu referință la CTE.***

Sintaxa:

```
WITH <query_name> [(<column_name> [, …]) AS
(
     SELECT …        -- ancora
     UNION ALL
     SELECT …        -- recursia
)
SELECT * FROM  <query_name> [OPTION (MAXRECURSION  <loop_number>)];
```

Exemplu 2. Calcularea factorialului 5!

```
WITH f_CTE (N, Fact) AS
(SELECT  5, 1
UNION ALL
SELECT N-1,  N*Fact FROM f_CTE  WHERE N>1)
SELECT * FROM f_CTE
```

Întrebări de control:

1. Care este diferența dintre joncțiunile interne și joncțiunile externe în SQL?
2. Ce este o echijoncțiune și cum se deosebește o joncțiune internă realizată prin clauza WHERE față de una realizată prin INNER JOIN în clauza FROM?
3. În ce condiții se obține un produs cartezian?
4. Care sunt regulile de bază care trebuie respectate pentru a putea utiliza operatorii pe mulțimi UNION, INTERSECT și EXCEPT?
5. Care este diferența de funcționare dintre operatorii pe mulțimi UNION și UNION ALL?
6. Ce reprezintă o subconsultare (subquery)?
7. Cum funcționează o interogare corelată și prin ce se deosebește modul ei de execuție față de o subconsultare obișnuită (necorelată)?
8. Ce este o expresie tabelară comună (CTE), cu ce operator se creează și care sunt pașii necesari pentru definirea unei CTE recursive?

Capitolul 9. INSTRUCȚIUNI DE ACTUALIZARE A DATELOR

9.1. Instrucțiunea INSERT

Instrucțiunea **INSERT** adaugă o înregistrare sau înregistrări multiple la un tabel. Sintaxa completă a instrucțiunii:

```
-- Syntax for SQL Server and Azure SQL Database

[ WITH <common_table_expression> [ ,...n ] ]
INSERT
{       [ TOP ( expression ) [ PERCENT ] ]
        [ INTO ]
        { <object> | rowset_function_limited
          [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]  }
    {
        [ ( column_list ) ]  [ <OUTPUT Clause> ]
        { VALUES ({ DEFAULT|NULL|expression} [ ,...n ] ) [ ,...n]
        | derived_table
        | execute_statement
        | <dml_table_source>
        | DEFAULT VALUES
        }
    }
}
[;]

<object> ::=
{     [ server_name . database_name . schema_name .
      | database_name .[ schema_name ] .
      | schema_name . ]  table_or_view_name  }

<dml_table_source> ::=
    SELECT <select_list>
    FROM ( <dml_statement_with_output_clause> )
      [AS] table_alias [ ( column_alias [ ,...n ] ) ]
    [ WHERE <search_condition> ]
        [ OPTION ( <query_hint> [ ,...n ] ) ]
```

Când instrucțiunea INSERT INTO se utilizează pentru a adăuga o singura înregistrare la un tabel, codul specifică numele și valoarea fiecărui câmp din înregistrare. Trebuie specificat fiecare câmp al înregistrării, căruia i se asociază o valoare, și valoarea acelui câmp. Daca nu se specifica fiecare câmp, atunci **valoarea implicită**, specificată cu clauzele DEFAULT sau NULL, se inserează în coloanele care lipsesc. Corespondența între atributele tabelului și valorile ce urmează a fi inserate este dictată de ordinea în care termenii apar în definiția tabelului. Înregistrările se adaugă la sfârșitul tabelului.

Exemplu 1.

```
INSERT INTO Firma (CodFirma, DenFirma, FormaOrg, Localitatea, Telefon,
Email, CodProprietar)
VALUES (177, 'Rodeo','SRL','Orhei', NULL, NULL,202)
```

Daca se omite lista câmpurilor, atunci clauza VALUES trebuie sa includă o valoare pentru fiecare câmp din tabel. Pot fi adăugate concomitent mai multe tupluri într-o singură instrucțiune.

Exemplu 2.

```
INSERT INTO Firma
VALUES (177, 'Rodeo', 'SRL', 'Orhei', NULL, NULL, 202),
(178, 'Alfa', 'SA', 'Chisinau', NULL, 'email@alfa.md, 214),
(179, 'Sunshine', 'SRL', 'Cahul', '060300300', NULL, 202);
```

Instrucțiunea INSERT INTO se poate utiliza pentru a adăuga un set de înregistrări din alt tabel sau interogare. În acest caz, clauza SELECT se utilizează pentru specificarea câmpurilor care se adaugă la tabelul ținta specificat. Tabelele care conțin înregistrările care se adaugă nu vor fi afectate de interogarea de adăugare. Dacă tabelul destinație conține o cheie primară, atunci este necesar ca în acest câmp să fie adăugate valori unice și non-Null; în caz contrar SQL Server nu va adăuga înregistrările.

Exemplu 3.

```
INSERT INTO DepartamenteCahul
SELECT Dept, Oras FROM Departament WHERE Oras = 'Cahul';
```

Pentru a adăuga **un anumit număr** de tupluri dintr-un tabel în altul instrucțiunea INSERT se folosește cu clauzele **TOP** și **PERCENT**.

Exemplu 4.

```
INSERT TOP (10) INTO Tabel3
SELECT * FROM Proprietar  ORDER BY Nume
```

Exemplu 5.

```
INSERT  TOP(5) PERCENT INTO Firme_fara_Email
SELECT DenFirma, FormaOrg FROM Firma WHERE Email IS  NULL
```

În instrucțiunea INSERT **nu se includ** câmpurile care sunt formate automat de SQL Server:

- Definite cu proprietatea de incrementare automată a valorilor IDENTITY;
- Câmpurile calculate.

Pentru a returna valorile inserate ca parte a unei operații de inserare poate fi utilizată clauza OUTPUT în instrucțiunea INSERT. Sintaxa:

```
INSERT SomeTable
OUTPUT dml_select_list
INTO  (@table_variable | output_table) (column_list)
```

Clauza OUTPUT returnează informații sau expresii bazate pe fiecare rând afectat de o instrucțiune INSERT, UPDATE, DELETE sau MERGE. Aceste rezultate pot fi returnate aplicației de procesare pentru a fi utilizate în mesaje de confirmare, arhivare și alte astfel de cerințe ale aplicației. Rezultatele pot fi, de asemenea, introduse într-o variabilă de tip tabel sau un alt tabel. În plus, se poate captura rezultatele unei clauze OUTPUT într-o instrucțiune INSERT, UPDATE, DELETE sau MERGE imbricată și de inserat rezultatele într-un tabel sau vizualizare țintă.

Microsoft SQL Server creează și gestionează automat două tabele speciale: **deleted** și **inserted**. Aceste tabele temporare rezidente în memorie sunt utilizate pentru a testa efectele anumitor modificări ale datelor și pentru a stabili condițiile pentru acțiunile în cadrule declanșatoarelor DML (DML triggers). Nu este posibilă modificarea directă a datelor din aceste tabele sau efectuarea unor operațiuni de limbaj de definire a datelor (DDL) pe tabele, cum ar fi CREATE INDEX.

Tabelul **deleted** stochează copii ale rândurilor afectate în timpul declarațiilor DELETE și UPDATE. În timpul executării unei instrucțiuni DELETE sau UPDATE, rândurile sunt șterse din tabelul de declanșare și transferate în tabelul **deleted**. Tabelul **deleted** și tabelul de declanșare **nu au rânduri comune.**

Tabelul **inserted** stochează copii ale rândurilor afectate în timpul instrucțiunilor INSERT și UPDATE. În timpul unei tranzacții de inserare sau actualizare, se adaugă noi rânduri atât la tabelul **inserted**, cât și la tabelul de declanșare. Rândurile din tabelul **inserted** sunt copii ale noilor rânduri din tabelul declanșator.

O tranzacție de actualizare UPDATE este similară cu o operațiune de ștergere urmată de o operație de inserare; rândurile vechi sunt copiate mai întâi în tabelul **deleted**, iar apoi rândurile noi sunt copiate în tabelul declanșator și în tabelul **inserted**.

Exemplu 10.

```
-- Declararea variabilei de tip tabel
DECLARE @MyTableVar TABLE
(
CasareID        smallint,
Cauza           nvarchar(50),
DataCasare      datetime
);
-- Inserarea tuplului nou în Tabel_Casare
-- Atributul CasareID  din tabelul Tabel_Casare este definit cu
-- proprietatea IDENTITY și nu se specifică în lista VALUES
INSERT Tabel_Casare
     OUTPUT inserted.CasareID, inserted.Cauza, inserted.DataCasare
     INTO @MyTableVar
VALUES (N‘Eroare operator', GETDATE());
-- Afișarea tuplului inserat din variabila tip tabel
SELECT * FROM @MyTableVar;
```

9.2. Instrucțiunea DELETE

Instrucțiunea DELETE creează o interogare de ștergere, care elimină înregistrările din unul sau mai multe tabele listate în clauza FROM, care satisfac condițiilor de filtrare din clauza WHERE.

Sintaxa:

```
-- Syntax for SQL Server and Azure SQL Database
  [ WITH <common_table_expression> [ ,...n ] ]
DELETE
    [ TOP ( expression ) [ PERCENT ] ]
    [ FROM ]
    { { table_alias
      | <object>
      | rowset_function_limited
      [ WITH ( table_hint_limited [ ...n ] ) ] }
      | @table_variable
    }
    [ <OUTPUT Clause> ]
    [ FROM table_source [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                    { { [ GLOBAL ] cursor_name }
                        | cursor_variable_name
                    }
                ]
              }
            }
    ]
    [ OPTION ( <Query Hint> [ ,...n ] ) ]
[; ]

<object> ::=
{     [ server_name.database_name.schema_name.
      | database_name. [ schema_name ] .
      | schema_name. ] table_or_view_name
}
```

Sintaxa simplificata:

```
DELETE
FROM tabel
[WHERE criteriu];
```

Dacă nu se specifică clauza WHERE se vor șterge toate înregistrările din tabel. Când se utilizează DELETE, se șterg doar datele; structura tabelului și toate proprietățile tabelului, cum ar fi atributele de câmp și indexurile, rămân intacte. Pot fi utilizate subconsultări în clauza WHERE.

Instrucțiunea DELETE se poate utiliza pentru a elimina înregistrări din tabele care se afla într-o relație de tipul "unu la mai mulți" cu alte tabele. Operațiunile de tip ștergere în cascadă duc la ștergerea înregistrărilor și din tabele subordonate.

După eliminarea înregistrărilor nu se mai poate anula operațiunea. Pentru a afla ce înregistrări vor fi șterse, se examinează preventiv rezultatele unei interogări de selectare care, utilizează aceleași criterii de filtrare, apoi se execută interogarea de ștergere. Se recomandă menținerea unor copii de rezervă ale datelor înainte de a realiza operații de ștergere a datelor.

Exemplu 1. Ştergerea din tabelul Firma a înregistrării despre firma cu codul 106.

```
DELETE
FROM Firma
WHERE CodFirma=106
```

Exemplu 2 . Comanda care șterge departamentele fără angajați din tabelul Departament.

```
DELETE  FROM  Departament
WHERE Dept NOT IN (SELECT Dept FROM Angajati)
```

Exemplu 3. Ştergerea dintr-o variabilă de tip tabel.

```
DELETE FROM @ProductTotals
WHERE ProductID = 60
```

Instrucțiunea DELETE poate fi utilizată cu clauza TOP pentru a șterge un anumit număr de tupluri din tabel. Mai jos este dat un exemplu de ștergere a 2,5 procente de rânduri din tabelul Firma.

Exemplu 4. Ştergerea a 2,5 procente de rânduri din tabelul Firma.

```
DELETE TOP (2.5) PERCENT  FROM Firma
```

Instrucțiunea **TRUNCATE TABLE** șterge toate tuplurile din tabel fără să înscrie acest lucru în fișierul de log-uri. TRUNCATE TABLE este similară instrucțiunii DELETE fără clauza WHERE, dar TRUNCATE TABLE se execută mai rapid și necesită mai puține resurse de sistem și jurnale de tranzacții.

Exemplu 5.

```
TRUNCATE TABLE Departamente
```

Instrucțiunea DELETE poate fi utilizată cu opțiunea OUTPUT.

Exemplu 6.

```
CREATE TABLE dbo.table1 ( id INT, employee VARCHAR(32) );
 GO
INSERT INTO dbo.table1
VALUES (1, 'Fred') ,(2, 'Tom') ,(3, 'Sally') ,(4, 'Alice');
GO
DECLARE @MyTableVar TABLE ( id INT, employee VARCHAR(32) );
PRINT 'table1, before delete'
```

```
SELECT * FROM dbo.table1;

-- Tabelul inițial

table1, before delete
id   employee
-------------------
1    Fred
2    Tom
3    Sally
4    Alice

DELETE
FROM dbo.table1 OUTPUT deleted.* INTO @MyTableVar
WHERE id = 4 OR id = 2;
PRINT 'table1, after delete‘
SELECT * FROM dbo.table1;

-- Tabelul după ștergere

table1, after delete
id   employee
-------------------
1    Fred
3    Sally

PRINT '@MyTableVar, after delete'
SELECT * FROM @MyTableVar;
DROP TABLE dbo.table1;

-- Tuplurile din variabila tip tabel

@MyTableVar, after delete
id   employee
--------------------------------
2    Tom
4    Alice
```

9.3. Instrucțiunea UPDATE

Instrucțiunea **UPDATE** creează o interogare de actualizare care modifică valorile din câmpurile unui tabel specificat, în funcție de criterii specificate.

Sintaxa:

```
-- Syntax for SQL Server and Azure SQL Database

[ WITH <common_table_expression> [...n] ]
UPDATE
    [ TOP ( expression ) [ PERCENT ] ]
    { { table_alias | <object> | rowset_function_limited
```

```
         [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
      }
      | @table_variable  }
    SET
        { column_name = { expression | DEFAULT | NULL }
          | { udt_column_name.{ { property_name = expression
                                | field_name = expression }
                                | method_name ( argument [ ,...n ] )
                              }
          }
          | column_name { .WRITE ( expression , @Offset , @Length ) }
          | @variable = expression
          | @variable = column = expression
          | column_name { += | -= | *= | /= | %= | &= | ^= | |= }
expression
          | @variable { += | -= | *= | /= | %= | &= | ^= | |= }
expression
          | @variable = column { += | -= | *= | /= | %= | &= | ^= | |=
} expression
        } [ ,...n ]

    [ <OUTPUT Clause> ]
    [ FROM{ <table_source> } [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                      | cursor_variable_name
                  }
                ]
              }
            }
    ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
[ ; ]

<object> ::=  { [ server_name . database_name . schema_name .
    | database_name .[ schema_name ] .
    | schema_name . ]  table_or_view_name}
```

Exemplu 1.

```
UPDATE Produse SET CodProdus = 350 WHERE CodProdus=302
```

Se pot modifica **mai multe câmpuri** în același timp. Când se dorește actualizarea datelor din mai multe câmpuri, se folosește virgula ca separator între câmpuri si valori.

Exemplu 2.

```
UPDATE Firma SET Localitatea='Briceni', Email='crocus@yahoo.com'
WHERE CodFirma=109
```

Exemplu 3.

```
UPDATE Produse SET PretUnit=PretUnit + 300
```

```
WHERE CodProdus IN (302, 304, 306)
```

Exemplu 4.

```
UPDATE @ProductTotals SET Revenue = Revenue * 1.15
WHERE ProductID = 62 OR ProductID = 66
```

Dacă **nu este specificată** clauza ***WHERE*** se vor **modifica toate tuplurile**. După actualizarea înregistrărilor **nu se poate anula** acțiunea realizată.

Limbajul SQL este orientat pe operațiile cu mulțimi și acest lucru trebuie de luat în considerare când se scriu comenzi de actualizare. Să presupunem că se dorește modificarea salariilor angajaților astfel: creșterea salariilor sub 3 mii lei cu 10% și a salariilor peste 3 mii lei cu 15%. O cale de a face actualizarea este execuția următoarelor comenzi:

```
UPDATE  Angajati SET Salariu = Salariu * 1.1 WHERE Salariu <= 3000
UPDATE  Angajati SET Salariu = Salariu * 1.15 WHERE Salariu > 3000
```

Să presupunem că avem un angajat care câștigă **2800**, deci satisface condiția din prima comanda de actualizare și atributul Salariu va fi setat la **3080**. În acest moment, linia satisface de asemenea și condiția celei de-a doua actualizări, deci salariul va fi modificat din nou. Creșterea pentru salariatul respectiv va fi de **26.5%**.

Această problemă particulară poate fi rezolvată prin **inversarea** celor două operații de actualizare. În situațiile mai complexe soluția ar putea să necesite introducerea unor actualizări intermediare sau utilizarea unor **cursori**.

Instrucțiunea **UPDATE** utilizată cu opțiunea **OUTPUT** returnează datele din tuplurile modificate cu instrucțiunile **UPDATE.**

Exemplu 5.

```
DECLARE @NewTableVar TABLE (Dollars money);
UPDATE Sales.SalesPerson SET Bonus = 10000
OUTPUT  inserted.Bonus  INTO @NewTableVar;
SELECT Dollars FROM @NewTableVar;
```

Întrebări de control:

1. Ce efect are omiterea listei de câmpuri în cadrul unei instrucțiuni INSERT INTO și ce cerință impune clauza VALUES în acest caz?
2. Care sunt tipurile de câmpuri generate automat de SQL Server ce trebuie excluse din instrucțiunea INSERT?
3. Care este rolul clauzei OUTPUT și prin ce se diferențiază tabelele temporare speciale inserted și deleted gestionate de SQL Server?
4. Ce se întâmplă atunci când se execută o instrucțiune DELETE fără clauza WHERE?
5. Care sunt avantajele utilizării instrucțiunii TRUNCATE TABLE comparativ cu DELETE?
6. Cum pot fi utilizate clauzele TOP și PERCENT în cadrul instrucțiunilor INSERT și DELETE?
7. Ce consecință are omiterea clauzei WHERE într-o instrucțiune UPDATE?
8. De ce ordinea executării instrucțiunilor UPDATE consecutive este critică într-un limbaj orientat pe mulțimi?

Capitolul 10. STRUCTURI DE CONTROL ȘI PACHETE SQL

10.1. Structuri de control SQL

Limbajul SQL conține **cuvinte rezervate speciale**, cunoscute ca **limbaj de control** al execuției instrucțiunilor, care dirijează cu **ordinea executării** instrucțiunilor SQL, a unor blocuri de instrucțiuni, a funcțiilor definite de utilizator sau a procedurilor stocate. Fără acest limbaj de execuție toate instrucțiile ale SQL ar fi executate **consecutiv** în ordinea în care ele sunt scrise.

Limbajul de control al execuției permite legarea instrucțiunilor într-un **fir de execuție**, precum și elaborarea unor structuri programate similare celor din limbaje de programare de nivel înalt. Cuvintele rezervate de control se utilizează în cazul când este necesar de realizat anumite acțiuni cu ajutorul limbajului SQL.

- **Blocul** este o **instrucțiune compusă** din mai multe instrucțiuni SQL și permite tratarea loc ca **o singură instrucțiune**. **Blocul** este compus din 3 secțiuni: **declarativă (opțională), executabilă, de tratare a excepțiilor (opțională).** Există **trei tipuri** de blocuri:
  - **Blocuri anonime** – blocuri SQL fără nume în cadrul unei aplicații;
  - **Proceduri** – bloc SQL cu **nume distinct**, cu parametri de intrare, dar **fără parametri de ieșire expliciți**;
  - **Funcții** - bloc SQL cu **nume distinct** și cu parametri de intrare care returnează **un singur rezultat de ieșire**.
- **Pachetele de programe** grupează proceduri și funcții.
- **Declanșatoare pe baza de date** sunt blocuri SQL asociate unor tabele și lansate automat când se execută o **comandă de manipulare cu date**.
- **Declanșatoare de aplicație** sunt blocuri SQL asociate unui eveniment în cadrul aplicației și se lansează automat în execuție.

10.1.1. Instrucțiunea BEGIN…END

Structura de control **BEGIN...END** este destinată pentru unirea mai multor instrucțiuni ale limbajului SQL într-un bloc logic executabil.

Sintaxa instrucțiunii:

```
BEGIN
    { sql_statement | statement_block }
END
```

Exemplu:

```
BEGIN
DECLARE @Max money;
SET @Max=(SELECT MAX(PretUnit) FROM Produse);
PRINT @Max;
END
```

10.1.2. Instrucțiunea IF…ELSE

Structura de control **IF...ELSE** este utilizată pentru realizarea **unei ramificări** în algoritmul de execuție a instrucțiunilor în funcție de realizarea unor **condiții**.

Sintaxa instrucțiunii:

```
IF Boolean_expression
     { sql_statement | statement_block }
```

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

Exemplu 1.

```
IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')
       SELECT 'Weekend';
ELSE
       SELECT 'Weekday';
```

10.1.3. Instrucțiunea CASE

Instrucțiunea **CASE** realizează o **selecție condiționată**. Poate fi utilizată în instrucțiunile SELECT, UPDATE, DELETE, SET și clauzele WHERE, ORDER BY, HAVING și în lista după SELECT.

Sintaxa:

```
-- Simple CASE expression:
CASE input_expression
     WHEN when_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END

-- Searched CASE expression:
CASE
     WHEN Boolean_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

SQL Server permite doar 10 niveluri de imbricare în expresiile CASE. Expresia CASE nu poate fi utilizată pentru a controla fluxul de execuție al instrucțiunilor Transact-SQL, blocurilor de instrucțiuni, funcțiilor definite de utilizator și procedurilor stocate.

Instrucțiunea CASE își evaluează secvențial condițiile și se oprește cu prima condiție a cărei condiție este îndeplinită. În unele situații, o expresie este evaluată înainte ca o expresie CASE să primească rezultatele expresiei ca intrare. Sunt posibile erori în evaluarea acestor expresii. Expresiile agregate care apar în argumentele WHEN pentru o expresie CASE sunt evaluate mai întâi, apoi furnizate expresiei CASE.

Exemplu 1.

```
SELECT ‘imprimantă’ AS TipProdus, Model,
     CASE Color
          WHEN ‘da’ THEN ‘Color’
          WHEN ‘nu’ THEN ‘Simplă’
          ELSE ‘necunoscută’
     END  AS Color
FROM [calculatoare].[dbo].[imprimante] ORDER BY Model
```

10.1.4. Instrucțiunea WHILE

Instrucțiunea **WHILE** asigură controlul executării **repetitive finite** a unei porțiuni de cod.
Sintaxa:

```
-- Syntax for SQL Server and Azure SQL Database
  WHILE Boolean_expression
     { sql_statement | statement_block | BREAK | CONTINUE }
```

unde:
BREAK - determină o ieșire din cea mai interioară buclă WHILE. Orice instrucțiuni care apar după cuvântul cheie END, care marchează sfârșitul buclei, nu sunt executate.
CONTINUE - determină repornirea buclei WHILE, ignorând orice instrucțiuni după cuvântul cheie CONTINUE.

Exemplu 1.

```
DECLARE @Numar int
SET @Numar = 1
WHILE @Numar < 4
           BEGIN
                      SELECT @Numar AS Numarul
                      SET @Numar = @Numar + 1
           END
```

Exemplu 2.

```
DECLARE @contor  int
SET @contor = 0
-- Start buclă infinită
WHILE  (1 = 1)
     BEGIN
          SET @contor = @contor + 1
          IF (@contor  % 9 = 0)    -- condiția de ieșire din buclă
               BREAK
     PRINT 'Iterația buclei ' + CAST(@contor AS  CHAR)
     END
```

Exemplu 3.

```
DECLARE @contor int
SET @contor = 1
WHILE @contor <= 5
     BEGIN
          PRINT @contor
          SET @contor = @contor + 1
          CONTINUE;
          IF @contor  =  4                -- nu va fi executată
               BREAK;
     END
```

10.1.5. Instrucțiunea RETURN

Instrucțiunea **RETURN asigură abandonarea** procesului de execuție a instrucțiunilor și returnarea unor valori numerice întregi. **Comenzile după RETURN nu vor fi executate**.

Sintaxa: `RETURN [ integer_expression ]`

Exemplu 1.

```
DECLARE @costmax float
SET @costmax=400
IF (SELECT  max(PretUnit) FROM Produse) <@costmax
     RETURN
ELSE
     BEGIN
     PRINT 'Sunt produse al caror pret nu depaseste' +
     CAST(@costmax AS CHAR(10));
     SELECT CodProdus, DenProdus, PretUnit
     FROM Produse
     WHERE PretUnit < @costmax
     END
```

10.1.6. Instrucțiunea GOTO

Instrucțiunea **GOTO** este instrucțiunea de **salt necondiționat** la instrucțiunea **marcată cu o etichetă**. Eticheta reprezintă un text scris la începutul unui rând urmat de semnul ":". Instrucțiunea GOTO se folosește, în special, pentru ieșirea din instrucțiunile **puternic imbricate**, unde instrucțiunea **RETURN nu poate fi utilizată**.

Sintaxa: `GOTO  label`

Exemplu 1.

```
DECLARE @a  float
DECLARE @b  float
SET @a = (SELECT AVG(Pret) FROM imprimante)
SET @b = (SELECT AVG(Pret) FROM laptop_uri)
BEGIN
     IF @a >= @b GOTO Pret_mai_mare
     ELSE IF @a < @b GOTO Pret_mai_mic
END
Pret_mai_mare:  SELECT @a AS imprimente_pret_mediu_mai_mare RETURN
Pret_mai_mic:   SELECT @b AS laptop_uri_pret_mediu_mai_mare  RETURN
```

10.1.7. Instrucțiunea WAITFOR

Instrucțiunea **WAITFOR întrerupe executarea** pachetului sau procedurii și realizează trecerea în regim de așteptare până când:

- ***Nu se scurge un interval de timp;***
- ***Nu se realizează o oră concretă.***

Sintaxa:

```
WAITFOR
```

```
{
    DELAY 'time_to_pass'
  | TIME 'time_to_execute'
}
```

Exemplu 1.

```
WAITFOR TIME '22:20‘
     SELECT * FROM Produse;
WAITFOR DELAY '02:00‘
      SELECT * FROM Produse;
```

10.2. Pachete de instrucțiuni SQL

Pachetul este un grup (lot) format din una sau câteva instrucțiuni SQL, care **urmează să fie executate**. Fiecare instrucțiune din pachet se finalizează cu **punct și virgulă**. Microsoft SQL Server **compilează** instrucțiunile din pachet într-un **modul executabil** numit **plan de execuție.** Instrucțiunile din plan sunt executate **consecutiv.**

Eroarea de compilare oprește compilarea planului de execuție și instrucțiunile din pachet **nu se execută**.

Eroarea de execuție (de exemplu, depășirea registrului) duce la următoarele rezultate:

- ***Majoritatea erorilor duc la oprirea executării instrucțiunii curente și următoarelor din pachet,***
- ***Unele erori opresc executarea doar a instrucțiunii curente, de exemplu, legată de constrângere, următoarele fiind executate.***

GO - este **directiva utilitară** care informează aplicațiile SQL Server **despre sfârșitul pachetului** de instrucțiuni SQL.

Sintaxa: `GO [număr_executări]`

GO nu este o instrucțiune Transact-SQL; este o comandă recunoscută de utilitățile sqlcmd și osql, și de editorul de cod SQL Server Management Studio. Utilitarele SQL Server interpretează GO ca un semnal că ar trebui să trimită lotul curent de instrucțiuni Transact-SQL la o instanță a SQL Server. Lotul curent de instrucțiuni este compus din toate instrucțiunile introduse de la ultima GO sau de la începutul sesiunii ad-hoc sau scriptului, dacă acesta este primul GO.

O instrucțiune Transact-SQL **nu poate ocupa aceeași linie** ca o comandă GO. Cu toate acestea, linia poate conține comentarii. După directiva GO **nu se utilizează** semnul ”**;**”.

Utilizatorii trebuie să respecte regulile pentru loturi. De exemplu, orice execuție a unei proceduri stocate după prima instrucțiune dintr-un lot trebuie să includă cuvântul cheie EXECUTE. Domeniul de aplicare al variabilelor locale (definite de utilizator) este limitat la un lot și nu poate fi referit după o comandă GO.

Exemplu 1.

```
SELECT ‘imprimante’;
SELECT ‘laptor-uri’;
GO 2                  -- se execută de două ori lotul de instrucțiuni
```

Întrebări de control:

1. Care sunt cele trei secțiuni care pot alcătui un bloc de instrucțiuni SQL?
2. Prin ce se caracterizează un bloc anonim, o procedură și o funcție?

3. Cu ce scop se utilizează structura de control BEGIN…END și ce rol are într-un cod SQL?
4. Care este diferența dintre sintaxa *Simple CASE* și *Searched CASE* și ce restricții există privind nivelul de imbricare a instrucțiunii CASE în SQL Server?
5. Cum funcționează instrucțiunile BREAK și CONTINUE în cadrul unei bucle WHILE?
6. Ce efect are instrucțiunea RETURN asupra executării codului SQL și ce tip de valori poate returna?
7. În ce situații specifice este recomandată utilizarea instrucțiunii de salt necondiționat GOTO și cum se definește o etichetă?
8. Care este diferența dintre opțiunile DELAY și TIME utilizate în cadrul instrucțiunii WAITFOR?
9. Ce reprezintă directiva GO și cum afectează aceasta domeniul de aplicare (*scope-ul*) variabilelor locale?

Capitolul 11. SCHEME, PROCEDURI ȘI FUNCȚII UTILIZATOR

11.1. Administrarea schemelor

O **schemă** a bazei de date permite simplificarea procesului de gestiune a datelor și crearea **submulțimilor de obiecte** care pot fi administrate ca un tot întreg. Schemele pot fi utilizate pentru **limitarea obiectelor vizibile** și **accesului** la ele pentru **un utilizator** sau un **grup de utilizatori**. Schemele sunt **containere cu obiecte** utilizate pentru definirea spațiilor de nume (**namespaces**) ale obiectelor BD. La crearea unei baze de date SQL Server creează în mod automat o schemă **implicită** pentru plasarea în ea a obiectelor bazei de date, cu numele **dbo (** de la database objects**)**.

Scheme fac parte din corpul unei baze de date. O bază de date poate avea **mai multe scheme**. O schemă poate conține **un număr de obiecte** limitat de numărul maximal de obiecte dintr-o bază de date gestionată de SQL Server. Ca urmare a utilizării schemelor se formează o ierarhie logică de adresare la un obiect al schemei **BD.Schema.Obiectul**. De exemplu, *calculatoare.dbo.imprimante*. Acest tip de adresare este unul **obligatoriu** în cazul dacă baza de date conține mai multe scheme. În cazul unei singure scheme implicite (dbo) adresarea la un obiect al bazei de date curente poate fi realizată doar prin specificarea numelui acestui obiect.

O schemă **nu poate conține** în corpul său **o altă schemă**. O schemă **nu poate conține** obiecte cu **nume duplicat**.

O schemă nouă a bazei de date (alta decât cea implicită) poate fi creată cu ajutorul instrucțiunii **CREATE SCHEMA**. **Sintaxa** instrucțiunii este:

```
- Syntax for SQL Server and Azure SQL Database

CREATE SCHEMA schema_name_clause [ <schema_element> [ …n ] ]

<schema_name_clause> ::=
    { schema_name
    | AUTHORIZATION owner_name
    | schema_name AUTHORIZATION owner_name }

<schema_element> ::=
    {   table_definition | view_definition | grant_statement |
        revoke_statement | deny_statement     }
```

unde:
`schema_name` - este numele prin care schema este identificată în baza de date;
`AUTHORIZATION owner_name` – specifică numele utilizatorului, proprietar al schemei la nivel de bază de date, care va deține schema. Acest utilizator poate deține alte scheme și nu poate folosi schema curentă ca schemă implicită;
`table_definition` - specifică o instrucțiune CREATE TABLE, care creează un tabel în cadrul schemei. Utilizatorul care execută această instrucțiune trebuie să aibă permisiunea CREATE TABLE pe baza de date curentă;
`view_definition` – specifică o instrucțiune CREATE VIEW care creează o vedere în cadrul schemei. Utilizatorul care execută această instrucțiune trebuie să aibă permisiunea CREATE VIEW pe baza de date curentă;
`grant_statement` – specifică o instrucțiune GRANT care acordă permisiuni pentru orice element securizat, cu excepția noii scheme;
`revoke_statement` – specifică o instrucțiune REVOKE care revocă permisiunile pentru orice element securizat, cu excepția noii scheme;

deny_statement – specifică o instrucțiune DENY care interzice permisiunile pentru orice element securizat, cu excepția noii scheme.

Exemplu 1.

```
USE Comert;
GO
CREATE SCHEMA  Tranzactii AUTHORIZATION Anna
CREATE TABLE FacturiNoi
(
NrFactura int,
DataFactura date,
CodFirma int,
PRIMERY KEY(NrFactura)
)
GRANT SELECT ON SCHEMA:: Tranzactii TO Ion
DENY  SELECT ON SCHEMA:: Tranzactii TO Petru;
GO
```

Exemplu 2.

```
USE Comert;
GO
SELECT Tranzactii.FacturiNoi.NrFactura, dbo.Firma.DenFirma
FROM Tranzactii.FacturiNoi, dbo.Firma
WHERE Tranzactii.FacturiNoi.CodFirma=dbo.Firma.CodFirma
```

Pentru **modificarea** schemei prin **deplasarea** unui obiect într-o altă schemă se utilizează instrucțiunea **ALTER SCHEMA**.

Sintaxa:

```
ALTER SCHEMA  destination_schema_name
           TRANSFER schema_name.object_name
```

Exemplu 3.

```
USE Comert;
GO
ALTER SCHEMA  Facturare
           TRANSFER Tranzactii.FacturiNoi
```

Afișarea schemelor prezente într-o bază de date este posibilă cu fraza SELECT:

```
SELECT * FROM sys.schemas;
```

Ștergerea unei scheme din baza de date se realizează cu instrucțiunea **DROP SCHEMA** în sintaxa :

```
DROP SCHEMA schema_name
```

Important! Inițial este necesar de **șters toate obiectele** din schema care urmează să fie **ștearsă**.

Exemplu 4.

```
USE Comert;
GO
DROP TABLE FacturiNoi;
DROP SCHEMA Tranzactii;
GO
```

11.2. Administrarea sinonimilor

Sinonimul este un obiect al bazei de date care:

- Oferă un **nume alternativ** unui obiect din baza de date ce **există** pe un **server local** sau **îndepărtat**;
- Oferă un **nivel de abstractizare** care **protejează** o aplicație-client de **modificările** ce pot fi făcute **numelor obiectelor** din baza de date sau **locațiilor** serverelor.

Sinonimele pot fi utilizate în **interogări** sau **proceduri stocate** în locul **numelor originale** ale obiectelor compuse din 2, 3 sau 4 părți.

Numele complex ***Nume_server.Nume_BD.Nume_schema.Nume_obiect*** poate fi înlocuit cu un **sinonim**. Un sinonim aparține unei **scheme**. Numele sinonimului trebuie să fie **unic**. Sinonimul poate fi creat pentru: ***tabele, viziuni, proceduri stocate, funcții de sistem, funcții CLR, proceduri stocate CLR***. Un sinonim nu poate fi obiect de bază pentru un alt sinonim.

Doar **numele** sinonimului execută legătura cu un obiect al bazei de date. Existența obiectului, tipul și verificarea dreptului de acces către obiect se face **în timpul rulării** instrucțiunii SQL, astfel obiectul de bază poate fi modificat, suprimat sau înlocuit cu alt obiect, care va avea același nume ca și obiectul de bază original.

Un sinonim este creat cu ajutorul instrucțiunii **CREATE SYNONYM**. **Sintaxa** instrucțiunii este:

```
-- SQL Server Syntax

CREATE SYNONYM [ schema_name_1. ] synonym_name FOR <object>

<object> :: =
{[  server_name.[database_name].[schema_name_2].
  | database_name.[schema_name_2].
  | schema_name_2.
  ] object_name  }
```

Exemplu 1.

```
USE calculatoare;
GO
CREATE SYNONYM notebook FOR calculatoare.dbo.laptop_uri
GO
SELECT * FROM notebook;
GO
```

Pentru a **șterge** un sinonim se utilizează instrucțiunea **DROP SYNONYM** în sintaxa:

```
DROP SYNONYM  [schema_name.] synonym_name
```

Numele schemei este **opțional** dacă se suprimă sinonimul din **schema implicită** asociată utilizatorului curent.

Exemplu 2.

```
USE calculatoare;
GO
DROP SYNONYM notebook
```

Pentru a modifica un sinonim **existent**, se **șterge** cel vechi și se creează unul **nou**.

11.3. Proceduri stocate SQL

O procedură stocată în Microsoft SQL Server este similară procedurilor din limbajele de programare. Procedurile stocate:

- ❖ Acceptă parametri de intrare și returnează multiple valori în formă de parametri de ieșire;
- ❖ Conțin instrucțiuni de programare care execută operații asupra bazei de date, inclusiv apelul altor proceduri;
- ❖ Returnează valoare de stare a procedurii sau grupului de proceduri pentru a indica succesul sau eșecul (și cauzele lui) procedurii.

Deosebim:

- ✓ **Proceduri stocate definite de utilizator** (elaborate în limbajul SQL sau ca o trimitere la **o metodă** a limbajului **Common Language Runtime** (CLR) al platformei **.NET Framework,** care se păstrează într-o bază de date, cu excepția ***Resource database***);
- ✓ **Proceduri stocate temporare** (**locale** sau **globale** stocate în **tempdb. Procedurile temporale locale** au un prefix comun unic (#) și sunt vizibile numai pentru conexiunea curentă a utilizatorului și sunt șterse la închiderea sesiunii. **Procedurile temporare globale** au prefix comun (##) și sunt vizibile **tuturor utilizatorilor** după crearea lor și sunt șterse la sfârșitul **ultimii sesiuni** ce a utilizat procedura)
- ✓ **Proceduri stocate ale sistemului** sunt create în prealabil și execută **sarcini administrative**. Ele pot fi executate în orice bază de date și se păstrează fizic în ***Resource database*** și au prefixul ***sp_*** **.** Procedurile stocate ale sistemului apar logic în ***sys schema*** fiecărui sistem de bază de dată definită de utilizator. Permisele GRANT, DENY, REVOKE se aplică procedurilor stocate ale sistemului. În majoritatea cazurilor procedurile stocate ale sistemului returnează valoarea **zero**, fapt ce indică că procedura este executată cu **succes**, sau altă valoare în caz de **eșec**.

Există un șir de reguli de respectat în cazul utilizării procedurilor stocate:

- ***Nu pot fi utilizate*** *în cadrul unei proceduri stocate instrucțiunile: USE, CREATE SCHEMA, CREATE RULE, CREATE/ALTER TRIGGER, CREATE/ALTER VIEW, CREATE/ALTER PROCEDURE, CREATE/ALTER FUNCTION, CREATE DEFAULT, CREATE AGGREGATE*
- *În cadrul procedurii pot fi referite* ***tabele*** *și* ***tabele temporare*** *create înainte de referirea la ele în procedură.*
- ***Tabelul temporar local*** *creat în procedură poate fi referit numai în cadrul ei și dispare la părăsirea procedurii.*
- *Dacă o procedură* ***apelează*** *o altă procedură stocată, atunci procedura apelată* ***poate accesa toate obiectele create de prima procedură****, inclusiv tabelele temporare.*
- *Dacă se* ***apelează*** *o procedură stocată îndepărtată care modifică o instanță SQL server îndepărtată, aceste modificări* ***nu pot fi desfășurate înapoi*** *(rolled back).*
- ***Numărul maxim*** *de parametri într-o procedură stocată poate fi* ***2100****.*
- ***Numărul*** *variabilelor locale este limitat de* ***volumul memoriei***
- ***Dimensiunea*** *maximă a proceduri stocate este de* ***250 MB****.*

O procedură stocată se creează cu ajutorul instrucțiunii **CREATE PROCEDURE** în sintaxa:

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }
    [schema_name.] procedure_name [ ; number ]
    [ { @parameter_name [ type_schema_name. ] data_type }
        [ VARYING ] [ NULL ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ] [ ,…n ]
[ WITH <procedure_option> [ ,…n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ …n ] [ END ] }
[;]

<procedure_option> ::= [ENCRYPTION] [RECOMPILE] [EXECUTE AS Clause]
```

unde:

ENCRYPTION – indică faptul că Database Engine va converti textul original al instrucțiunii CREATE PROCEDURE într-un format criptat, care nu este direct vizibilă în nicio vizualizare a catalogului cu metadate. Utilizatorii care nu au acces la tabelele de sistem sau la fișierele bazei de date nu pot prelua textul criptat. Cu toate acestea, textul va fi disponibil pentru utilizatorii privilegiați care pot accesa tabelele de sistem sau pot accesa direct fișierele bazei de date. Utilizarea acestei opțiuni împiedică publicarea funcției ca parte a replicării SQL Server. Această opțiune nu poate fi specificată pentru funcțiile CLR;
RECOMPILE – indică faptul că Database Engine nu memorează în cache un plan de interogare pentru această procedură, forțând să fie compilat de fiecare dată când este executat. Această opțiune nu poate fi utilizată când este specificat FOR REPLICATION sau pentru procedurile CLR;
EXECUTE AS – specifică contextul de securitate în care se execută procedura;
FOR REPLICATION – specifică faptul că procedura este creată pentru replicare. O procedură creată cu opțiunea FOR REPLICATION este folosită ca filtru de procedură și este executată numai în timpul replicării. Parametrii nu pot fi declarați dacă este specificat FOR REPLICATION. FOR REPLICATION nu poate fi specificat pentru procedurile CLR. Opțiunea RECOMPILE este ignorată pentru procedurile create cu FOR REPLICATION.

*Exemplu 1. Procedură **fără parametri***

```
USE calculatoare;
GO
IF OBJECT_ID (‘test’, ‘P’) IS NOT NULL DROP PROCEDURE test;
GO
CREATE PROCEDURE test
AS SELECT Model, Pret FROM pc_uri WHERE Pret!=600;
```

*Exemplu 2. Procedură **cu parametri**.*

```
USE calculatoare;
GO
CREATE PROCEDURE test1
          @Pret  float=500,
          @Model  varchar(4)=‘12%’
```

```
AS
     SELECT Model, Pret FROM dbo.pc_uri
     WHERE Pret>@Pret  AND Model LIKE @Model;
```

Procedura se lansează în execuție cu instrucțiunea EXECUTE în sintaxa:

```
EXECUTE <procedure_name> <parameter_value> [, …]
```

Exemplu 3:

```
EXECUTE test1;
EXECUTE test1 @Pret=600;
EXECUTE test1 600;
EXECUTE test1 @Pret=600, @Model='1%';
EXECUTE test1 @Model='1%', @Pret=600;
EXECUTE test1 600, '1%';
```

Executarea unei proceduri stocate cu **variabile de ieșire** implică **declararea** acestor variabile **în procedură** cu cuvântul rezervat **OUTPUT** și **apelarea** procedurii cu specificarea **OUTPUT** pentru variabila locală de ieșire. **Variabila de ieșire** a procedurii trebuie să fie **declarată** preventiv în aplicația-client.

Exemplu 3. Procedură cu parametri de ieșire.

```
USE calculatoare;
GO
DECLARE @Pret_mediu float;
EXECUTE test2 'Laptop_uri', @Pret_mediu OUTPUT;
IF @Pret_mediu>250
BEGIN
          PRINT 'Pretul mediu pe tabel este egal cu '
               +RTRIM(CAST(@Pret_mediu AS varchar(10)))+'$ .'
END;
GO
```

După ce procedura este creată, **textul procedurii** este plasat în tabelul de sistem ***SysComments***, de unde el poate fi extras cu ajutorul procedurii de sistem ***sp_helptext***, care extrage codul inițial al procedurii.

Exemplu 9. Afișarea textului procedurii test1.

```
USE calculatoare;
GO
sp_helptext test1;
```

De asemenea, pot fi utilizate următoarele vederi de sistem pentru obținerea informațiilor referitoare la proceduri:

- **sys.sql_modules** - returnează definiția unei proceduri Transact-SQL. Textul unei proceduri create cu opțiunea ENCRYPTION nu poate fi vizualizat utilizând vizualizarea catalog **sys.sql_modules**.
- **sys.assembly_modules** - returnează informații despre o procedură CLR.

- **sys.parameters -** returnează informații despre parametrii definiți într-o procedură.
- **sys.sql_expression_dependencies, sys.dm_sql_referenced_entities, sys.dm_sql_referencing_entities -** returnează obiectele la care se face referire printr-o procedură.

Modificarea unei proceduri se realizează cu instrucțiunea **ALTER PROCEDURE**.

Exemplu 4. Modificarea procedurii test1.

```
USE calculatoare;
GO
ALTER PROCEDURE test1
@Pret float,                -- au fost anulate valorile implicite
@Model varchar(4)           -- pentru parametrii procedurii
AS
     SELECT Model, Pret FROM dbo.pc_uri
          WHERE Pret>@Pret  AND Model LIKE @Model;
```

11.4. Funcții definite de utilizator

Utilizatorul poate defini **propriile funcții** într-o bază de date SQL Server. O funcție definită grupează în cadrul unui obiect SQL independent un **set de instrucțiuni SQL** și returnează **o singură valoare** asociată cu numele funcției. Deosebim:

- **Funcții scalare;**
- **Funcții de tip tabel simplu;**
- **Funcții de tip tabel multi-instrucțiune**

Funcțiile scalare returnează **un singur rezultat scalar**. El poate fi de orice tip cu excepția **text, ntext, image** și **timestamp**. Se utilizează următoarea **sintaxă** pentru crearea funcției scalare:

```
-- Transact-SQL Scalar Function Syntax
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
[ NULL ] [ = default ] [ READONLY ] }  [ ,...n ]  ] )
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[ ; ]

-- Transact-SQL Function Clauses
<function_option>::=
{   [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
  | [ INLINE = { ON | OFF }]  }
```

unde:

ENCRYPTION - indică faptul că Database Engine va converti textul original al instrucțiunii CREATE FUNCTION într-un format criptat, care nu este direct vizibilă în nicio vizualizare a catalogului cu metadate. Utilizatorii care nu au acces la tabelele de sistem sau la fișierele bazei de date nu pot prelua textul criptat. Cu toate acestea, textul va fi disponibil pentru utilizatorii privilegiați care pot accesa tabelele de sistem sau pot accesa direct fișierele bazei de date. Utilizarea acestei opțiuni împiedică publicarea funcției ca parte a replicării SQL Server. Această opțiune nu poate fi specificată pentru funcțiile CLR;
SCHEMA BINDING - specifică faptul că funcția este legată de obiectele bazei de date la care face referire. Când se specifică SCHEMABINDING, obiectele de bază nu pot fi modificate într-un mod care ar afecta definiția funcției. Definiția funcției în sine trebuie mai întâi modificată sau abandonată pentru a elimina dependențele de obiectul care urmează să fie modificat. Legarea funcției de obiectele la care face referire este eliminată numai atunci când are loc una dintre următoarele acțiuni: funcția este abandonată; funcția este modificată prin utilizarea instrucțiunii ALTER cu opțiunea SCHEMABINDING nespecificată. O funcție poate fi legată de schemă numai dacă sunt adevărate următoarele condiții: funcția este o funcție Transact-SQL; funcțiile și vizualizările definite de utilizator la care face referire funcție sunt, de asemenea, legate de schemă; obiectele la care face referire funcția sunt referite folosind un nume din două părți; funcția și obiectele la care face referire aparțin aceleiași baze de date; utilizatorul care a executat instrucțiunea CREATE FUNCTION are permisiunea REFERENCES pe obiectele bazei de date la care face referire funcția;
RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT - specifică atributul **OnNULLCall** al unei funcții scalare. Dacă nu este specificat, CALLED ON NULL INPUT este utilizat implicit. Aceasta înseamnă că corpul funcției se execută chiar dacă NULL este trecut ca argument. Dacă RETURNS NULL ON NULL INPUT este specificat într-o funcție CLR, aceasta indică faptul că SQL Server poate returna NULL atunci, când oricare dintre argumentele pe care le primește este NULL, fără a invoca efectiv corpul funcției;
EXECUTE_AS - specifică contextul de securitate în care este executată funcția definită de utilizator. Se poate de controlat ce cont de utilizator folosește SQL Server pentru a valida permisiunile pentru orice obiect de bază de date la care se face referire de către funcție. EXECUTE AS nu poate fi specificat pentru funcțiile inline cu valori de tabel;
INLINE = { ON | OFF } - specifică dacă scalarul returnat de funcția scalară trebui să fie aliniat sau nu. Această clauză se aplică numai funcțiilor scalare definite de utilizator. Clauza INLINE nu este obligatorie.

Exemplu 1.

```
USE Comert;
GO
IF OBJECT_ID('GetTotal','FN') IS NOT NULL DROP FUNCTION GetTotal;
GO
CREATE FUNCTION GetTotal(@FJO char(4))
RETURNS int
BEGIN
     DECLARE @Total int
     SET @Total= (SELECT COUNT(*) FROM Firma WHERE FormaOrg=@FJO)
     RETURN @Total
END
SELECT  dbo.GetTotal('SRL') AS Numar_SRL
```

Funcția de tip tabel simplu returnează ca rezultat un **set de date**. Ea poate fi utilizată pentru returnarea unor date în funcție de **anumiți parametri** utilizați ca variabile ale funcției. Funcția de

tip tabel simplu poate conține **doar o singură interogare** în corpul său. Clauza **ORDER BY** poate fi utilizată numai cu **TOP**, **iar fiecare câmp calculat selectat** trebuie să aibă o **etichetă**. **Sintaxa:**

```
-- Transact-SQL Inline Table-Valued Function Syntax
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
[ NULL ]  [ = default ] [ READONLY ] }  [ ,...n ]  ]  )
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] select_stmt [ ) ]  [ ; ]

-- Transact-SQL Function Clauses
<function_option>::=
{   [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
  | [ INLINE = { ON | OFF }]  }
```

Exemplu 2.

```
USE Comert;
GO
CREATE FUNCTION GetProd(@Pret money)
RETURNS TABLE
WITH  ENCRYPTION
AS
RETURN
(SELECT CodProdus, DenProdus, PretUnit FROM Produse
WHERE PretUnit=@Pret);
SELECT * FROM GetProd(150);
```

Funcțiile de tip tabel multi-instrucțiune pot să conțină **mai multe instrucțiuni SQL** în calitate de corp și returnează un tabel ca rezultat. **Sintaxa**:

```
-- Transact-SQL Multi-Statement Table-Valued Function Syntax
CREATE [ OR ALTER ] FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
[ NULL ]  [ = default ] [READONLY] }  [ ,...n ]  ] )
RETURNS @return_variable TABLE <table_type_definition>
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN
    END
[ ; ]
```

```
-- Transact-SQL Function Clauses
<function_option>::=
{   [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
  | [ INLINE = { ON | OFF }]  }
```

Exemplu 3.

```
USE calculatoare;
GO
CREATE FUNCTION GetModTip(@Prod char(1))
           RETURNS @ret TABLE
                (Producator char(1), ModTip varchar(15))
           AS
           BEGIN
                INSERT @ret
                     SELECT Producator, Model+'   '+Tip
                     FROM produse
                     WHERE Producator=@Prod;
                RETURN
           END;
GO
SELECT * FROM GetModTip('E%');
```

Modificarea unei funcții se realizează cu ajutorul instrucțiunii **ALTER FUNCTION**.

Exemplu 4.

```
ALTER FUNCTION GetProd(@Pret money, @Cod  int)
RETURNS TABLE
WITH  ENCRYPTION
AS
RETURN
(SELECT CodProdus, DenProdus, PretUnit FROM Produse
WHERE PretUnit=@Pret AND CodProdus!=@Cod);
```

Suprimarea unei funcții se realizează cu instrucțiunea DROP FUNCTION în sintaxa:

```
DROP FUNCTION [<schema-name>.]<function_name> [, …]
```

Exemplu 6.

```
DROP FUNCTION GetProd
```

Întrebări de control:

1. Ce este o schemă în SQL Server, care este numele schemei implicite create automat la generarea unei baze de date și cum se definește structura ierarhică de adresare a unui obiect?
2. Cum se pot transfera obiectele existente dintr-o schemă în alta?
3. Ce condiție obligatorie trebuie îndeplinită înainte de a șterge o schemă?
4. Ce rol îndeplinește un sinonim în SQL Server?

5. Cum se realizează modificarea unui sinonim existent?
6. Care sunt diferențele esențiale dintre procedurile stocate temporare locale (cu prefixul #) și cele globale (cu prefixul ##) ?
7. Ce restricții de utilizare există în cadrul procedurilor stocate (menționați cel puțin 3 instrucțiuni interzise)?
8. Ce efecte au opțiunile ENCRYPTION și RECOMPILE atunci când sunt specificate la crearea unei proceduri stocate?
9. Cum se declară și cum se apelează parametri de ieșire în cadrul procedurilor stocate?
10. Care sunt cele trei tipuri principale de funcții definite de utilizator în SQL Server?
11. În ce condiții poate fi aplicată opțiunea SCHEMABINDING unei funcții și ce restricții impune aceasta asupra obiectelor de bază la care face referire?
12. Care este diferența de structură și sintaxă dintre o funcție de tip tabel simplu (*inline*) și o funcție de tip tabel multi-instrucțiune (*multi-statement*)?

Capitolul 12. TRANZACȚII ȘI DECLANȘATOARE SQL

12.1. Tranzacții SQL

O **tranzacție SQL** reprezintă un șir de operații executate cu o entitate logică de execuție.

Tranzacția este caracterizată prin așa proprietăți ca**: atomicitate, integritate, izolare** și **durată.**

Atomicitatea tranzacției determină executarea tuturor operațiilor din ea sau nici una din ele. **Integritatea** tranzacției determină faptul că după executarea ei datele din baza de date rămân integre. **Izolarea** tranzacției determină faptul că modificările executate de o tranzacție nu depind de modificările executate în paralel de o altă tranzacție. **Durata** tranzacție determină faptul că modificările realizate de o tranzacție sunt permanente în cadrul sistemului.

La executarea unor tranzacții SQL Server actualizează un **registru** cu înscrierea **stării inițiale** ale datelor ce urmează să fie modificate în cadrul tranzacției. Toate modificările efectuate asupra datelor se **memorizează** în memoria tampon **cash**, iar înscrierea lor pe disc se realizează numai când este activat **un punct de control** în baza de date și numai după se sunt făcute modificările respective în **registrul tranzacțiilor efectuate**.

Instrucțiunea **BEGIN TRANSACTION** stabilește începutul unei **tranzacții locale** și incrementează valoarea variabilei de sistem **@@TRANSCOUNT** cu valoarea 1. Cu toate că **BEGIN TRANSACTION** lansează o **tranzacție locală**, ea nu va fi inclusă în registrul tranzacțiilor până când nu este executată o instrucțiune de modificare a datelor, de exemplu, **INSERT**, **DELETE** sau **UPDATE**. Instrucțiunea **BEGIN TRANSACTION** creează în sistem **un punct de control** (restore point), care garantează integritatea datelor din baza de date. În caz de eroare se **restabilește** starea bazei de date corespunzătoare acestui punct de control. Fiecare tranzacție se execută până nu se finalizează fără erori și se execută instrucțiunea **COMMIT TRANSACTION**, care înscrie modificările realizate în baza de date. În caz de eroare toate modificările sunt suprimate cu instrucțiunea **ROLLBACK TRANSACTION**.

Tranzacția locală inițiată cu **BEGIN TRANSACTION** se transformă în una **distribuită** dacă instrucțiunile **INSERT**, **UPDATE** sau **DELETE** se referă la un **tabel îndepărtat** de pe alt server sau se execută o **procedură stocată pe un alt server.** O tranzacție distribuită poate fi creată explicit cu instrucțiunea **BEGIN DISTRIBUTED TRANSACTION.**

Instrucțiunea **COMMIT TRANSACTION** finalizează o tranzacție de succes prin **înscrierea** tuturor modificărilor pe disc, **eliberarea** resurselor sistemului utilizate în tranzacție și **micșorează** valoarea înscrisă în variabila **@@TRANSCOUNT** cu 1 și termină tranzacția activă. În cazul **tranzacțiilor imbricate** resursele sistemului sunt eliberate doar atunci, când este finalizată tranzacția **externă.** Executarea instrucțiunii **COMMIT TRANSACTION** când variabila **@@TRANSCOUNT** are valoare 0 va genera o eroare, deoarece **nu există o tranzacție inițiată**. După executarea instrucțiunii **COMMIT TRANSACTION nu poate** fi utilizată instrucțiunea **ROLLBACK TRANSACTION**, deoarece datele modificate deja sunt parte componentă a bazei de date.

Sintaxa instrucțiunii **BEGIN TRANSACTION** este următoarea:

```
--Applies to SQL Server and Azure SQL Database

BEGIN { TRAN | TRANSACTION }
    [ { transaction_name | @tran_name_variable }
      [ WITH MARK [ 'description' ] ]  ]  [ ; ]
```

unde:

`transaction_name` - este numele atribuit tranzacției. Acest nume trebuie să respecte regulile pentru identificatori, dar identificatorii mai mari de 32 de caractere nu sunt permise. Se recomandă

de utilizat numele tranzacțiilor numai pe perechea cea mai exterioară de instrucțiuni imbricate BEGIN...COMMIT sau BEGIN...ROLLBACK. Numele tranzacției este întotdeauna diferențiat cu majuscule și minuscule, chiar și atunci, când instanța SQL Server nu face distincție între majuscule și minuscule;
`@tran_name_variable` - este numele unei variabile definite de utilizator care conține un nume de tranzacție valid. Variabila trebuie să fie declarată cu un tip de date char, varchar, nchar sau nvarchar. Dacă variabilei îi sunt transmise mai mult de 32 de caractere, vor fi utilizate numai primele 32 de caractere; caracterele rămase vor fi trunchiate;
`WITH MARK [ 'description' ]` -specifică faptul că tranzacția este marcată în jurnal cu descrierea dată . O descriere mai lungă de 128 de caractere este trunchiată la 128 de caractere înainte de a fi stocată în tabelul **msdb.dbo.logmarkhistory**. Dacă se folosește WITH MARK, trebuie specificat un nume de tranzacție. WITH MARK permite restaurarea unui jurnal de tranzacții la o marca numită.

Sintaxa instrucțiunii **COMMIT TRANSACTION** este următoarea:

```
COMMIT [ { TRAN | TRANSACTION }
[ transaction_name | @tran_name_variable ] ] [ ; ]
```

Sintaxa instrucțiunii **ROLLBACK** este următoarea:

```
ROLLBACK { TRAN | TRANSACTION }
     [ transaction_name | @tran_name_variable
     | savepoint_name | @savepoint_variable ]   [ ; ]
```

Exemplu 1. Începutul și finalizarea unei tranzacții fără nume.

```
BEGIN TRANSACTION;
DELETE FROM HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT;
```

Exemplu 2. Începutul și anularea unei tranzacții fără nume.

```
CREATE TABLE ValueTable (id INT);
BEGIN TRANSACTION;
       INSERT INTO ValueTable VALUES(1);
       INSERT INTO ValueTable VALUES(2);
ROLLBACK;
```

12.2. Declanșatoare SQL

Declanșatorul (*trigger*) este un tip aparte de procedură stocată care este executată **automat** la realizarea unui eveniment pe SQL Server. Deosebim **declanșatoare DML, declanșatoare DDL și declanșatoare de logare.**

Declanșatoarele DML sunt executate în cazul când utilizatorul încearcă să execute modificări în date utilizând limbajul DML. Astfel de evenimente sunt instrucțiunile **INSERT**, **UPDATE** sau **DELETE,** aplicate unui tabel sau viziuni. Declanșatoarele DML sunt utilizate pentru **verificarea constrângerilor** și **integrității datelor.**

Declanșatoarele DDL sunt executate la lansarea în execuție a instrucțiunilor **CREATE, ALTER, DROP, GRANT, REVOKE, DENY** și a unui șir de proceduri stocate de sistem, care realizează operații similare.

Declanșatoarele de logare reacționează la evenimentul **LOGON** la realizarea unei sesiuni de autorizare a utilizatorilor.

Există **trei tipuri** de **declanșatoare DML**: **AFTER**, **INSTEAD OF** și **CLR**

- **AFTER** indică faptul că declanșatorul DML se va activa **după ce toate operațiile din fraza SELECT** vor fi executate cu succes, inclusiv verificarea constrângerilor.
- **INSTEAD OF** indică faptul că declanșatorul DML este utilizat **în locul** instrucțiunii SQL. **INSTEAD OF** nu poate fi indicat pentru declanșatoare DDL sau declanșatoare **LOGON**.
- Un declanșator **CLR** va executa una sau mai multe metode .**NET Framework.**

Declanșatoarele DML folosesc **două tabele temporale,** rezidente în memorie, care sunt create automat de SQL Server la realizarea operațiilor **DELETE, UPDATE** și **INSERT**, ce conțin starea până la executarea operațiilor de ștergere (***deleted)*** sau starea după inserare sau modificare (***inserted).***

Sintaxa unui declanșator DML este următoarea:

```
-- SQL Server Syntax
-- Trigger on an INSERT, UPDATE, or DELETE statement to a table
-- or view (DML Trigger)
CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]  [ NOT FOR REPLICATION ]
AS { sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier
[ ; ] > }

<dml_trigger_option> ::= [ ENCRYPTION ] [ EXECUTE AS Clause ]
<method_specifier>   ::=  assembly_name.class_name.method_name
```

Exemplu 1. Declanșator DML de afișare a unui mesaj la operațiile de actualizare asupra unui tabel.

```
CREATE TRIGGER reminder1
ON Sales.Customer
AFTER INSERT, UPDATE, DELETE
AS RAISERROR ('Notify Customer Relations', 16, 10);
GO
```

Sintaxa unui declanșator DDL este următoarea:

```
-- Trigger on a CREATE, ALTER, DROP, GRANT, DENY,
-- REVOKE or UPDATE statement (DDL Trigger)
CREATE [ OR ALTER ] TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH <ddl_trigger_option> [ ,...n ] ]
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS { sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier
>  [ ; ] }

<ddl_trigger_option> ::=  [ ENCRYPTION ]  [ EXECUTE AS Clause ]
```

Exemplu 2. Declanşator DDL care anulează operaţiile de modificare sau de ştergere a tabelelor bazei de date Comerţ cu afişarea informaţiei despre acţiunea anulată .

```
USE Comert;
GO
CREATE TRIGGER protectie
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
PRINT 'Modificarea schemei bazei de date este interzisa'
SELECT EVENTDATA().value
     ('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]','nvarchar(max)')
ROLLBACK;
GO
```

Sintaxa unui declanşator **LOGON** este următoarea:

```
-- Trigger on a LOGON event (Logon Trigger)
CREATE [ OR ALTER ] TRIGGER trigger_name
ON ALL SERVER
[ WITH <logon_trigger_option> [ ,...n ] ]
{ FOR| AFTER } LOGON
AS { sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier
>  [ ; ] }

<logon_trigger_option> ::=   [ ENCRYPTION ]  [ EXECUTE AS Clause ]
```

Exemplu 3. Declanşator LOGON care anulează o nouă logare pe server a utilizatorului user3 dacă acesta deja este logat activ la 3 sesiuni pe servere diferite.

```
USE  Comert;
GO
CREATE  LOGIN user3 WITH  PASSWORD='parola3';
GO
CREATE TRIGGER sesiune_user3
ON ALL SERVER WITH EXECUTE AS 'user3'
FOR LOGON
AS
BEGIN
     IF ORIGINAL_LOGIN()='user3' AND
          (SELECT COUNT(*) FROM sys.dm_exec_sessions
          WHERE is_user_process=1 AND original_login_name='user3')>3
ROLLBACK;
END;
```

Activarea şi dezactivarea declanşatoarelor poate fi realizată cu ajutorul instrucţiunilor ENABLE TRIGGER şi DISABLE TRIGGER.

- **La nivel de tabel**

```
USE Comert;
GO
```

```
DISABLE TRIGGER  tuplu_nou_creat ON Firma;
GO
ENABLE TRIGGER  tuplu_nou_creat ON Firma;
GO
```

- **La nivel de bază de date**

```
USE Comert;
GO
DISABLE TRIGGER  protectie ON DATABASE;
```

- **În cadrul instrucțiunii ALTER**

```
USE Comert;
GO
ALTER TABLE  Firma DISABLE TRIGGER  tuplu_nou_creat;
```

- **La nivel de server**

```
USE Comert;
GO
DISABLE TRIGGER sesiune_user3 ON ALL SERVER;

USE Comert;
GO
ENABLE TRIGGER  ALL ON ALL SERVER;
```

Lista declanșatoarelor din baza curentă se păstrează în viziunile ***sys.triggers*** și ***sys.trigger_events.*** Lista declanșatoarelor la nivel de server se păstrează în viziunile ***sys.server_triggers*** și ***sys.server_trigger_events.***

Procedura stocată ***sp_helptrigger*** afișează **lista declanșatoarele** aplicate unui **tabel.**

```
SP_HELPTRIGGER  Firma
```

Conținutul unui **trigger** poate fi aflat în felul următor:

```
SP_HELPTRIGGER tuplu_nou_creat
```

12.3. Cursoare SQL

Operațiile dintr-o bază de date relațională acționează pe un set complet de rânduri. De exemplu, setul de rânduri returnat de o instrucțiune SELECT este format din toate rândurile care îndeplinesc condițiile din clauza WHERE a instrucțiunii. Acest set complet de rânduri returnat de instrucțiune este cunoscut ca **setul de rezultate**. Aplicațiile, în special aplicațiile online interactive, nu pot funcționa întotdeauna eficient cu întregul set de rezultate ca unitate. Aceste aplicații au nevoie de un mecanism care să funcționeze cu un rând sau un bloc mic de rânduri la un moment dat. **Cursoarele** sunt o extensie la seturile de rezultate care oferă acel mecanism.

Cursoarele extind procesarea rezultatelor prin:

- Permite poziționarea pe anumite rânduri ale setului de rezultate.
- Preluarea unui rând sau a unui bloc de rânduri din poziția curentă în setul de rezultate.
- Sprijinirea modificărilor datelor la rândurile de la poziția curentă în setul de rezultate.
- Sprijinirea diferitelor niveluri de vizibilitate la modificările făcute de alți utilizatori în datele bazei de date care sunt prezentate în setul de rezultate.

- Furnizarea de instrucțiuni Transact-SQL în scripturi, proceduri stocate și declanșează accesul la datele dintr-un set de rezultate.

Cursoarele Transact-SQL se bazează pe sintaxa DECLARE CURSOR și sunt utilizate în principal în scripturile Transact-SQL, procedurile stocate și declanșatoarele. Cursoarele Transact-SQL sunt implementate pe server și sunt gestionate de instrucțiuni Transact-SQL trimise de la client la server. Ele pot fi, de asemenea, conținute în loturi, proceduri stocate sau declanșatoare.

Sintaxa extinsă Transact-SQL:

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

SQL Server acceptă patru tipuri de cursor.

Cursor FORWARD-ONLY (Numai înainte). Un cursor numai înainte este specificat ca FORWARD_ONLY și READ_ONLY și nu acceptă derularea. Acestea se mai numesc și cursore firehose și acceptă doar preluarea în serie a rândurilor de la începutul până la sfârșitul cursorului. Rândurile nu sunt preluate din baza de date până când nu sunt parcurse în întregime. Efectele tuturor instrucțiunilor INSERT, UPDATE și DELETE făcute de utilizatorul curent sau comise de alți utilizatori, care afectează rândurile din setul de rezultate sunt vizibile pe măsură ce rândurile sunt preluate de pe cursor.

Deoarece cursorul nu poate fi derulat înapoi, majoritatea modificărilor aduse rândurilor din baza de date după ce a fost preluat rândul nu sunt vizibile prin cursor. În cazurile în care o valoare utilizată pentru a determina locația rândului în setul de rezultate este modificată, cum ar fi actualizarea unei coloane acoperite de un index grupat, valoarea modificată este vizibilă prin cursor.

Cursor STATIC. Setul complet de rezultate ale unui cursor static este construit în tempdb atunci când cursorul este deschis. Un cursor static afișează întotdeauna setul de rezultate așa cum era atunci, când cursorul a fost deschis. Cursoarele statice detectează puține modificări sau deloc, dar consumă relativ puține resurse în timpul derulării.

Cursorul nu reflectă nicio modificare făcută în baza de date care afectează fie apartenența setului de rezultate, fie modificări ale valorilor din coloanele rândurilor care alcătuiesc setul de rezultate. Un cursor static nu afișează rânduri noi inserate în baza de date după deschiderea cursorului, chiar dacă acestea corespund condițiilor de căutare ale instrucțiunii SELECT a cursorului. Dacă rândurile care compun setul de rezultate sunt actualizate de către alți utilizatori, noile valori ale datelor nu sunt afișate în cursorul static. Cursorul static afișează rândurile șterse din baza de date după deschiderea cursorului. Nicio operație UPDATE, INSERT sau DELETE nu este reflectată într-un cursor static (cu excepția cazului în care cursorul este închis și redeschis), nici măcar modificările efectuate folosind aceeași conexiune care a deschis cursorul.

Cursor KEYSET (set de chei). Apartenența și ordinea rândurilor dintr-un cursor bazat pe set de chei sunt fixe atunci când cursorul este deschis. Cursoarele conduse de seturi de chei sunt controlate de un set de identificatori unici, chei, cunoscute sub numele de set de chei. Cheile sunt construite dintr-un set de coloane care identifică în mod unic rândurile din setul de rezultate. Setul de chei este setul de valori ale cheilor din toate rândurile care s-au calificat pentru instrucțiunea

SELECT la momentul deschiderii cursorului. Setul de taste pentru un cursor bazat pe setul de taste este construit în tempdb atunci când cursorul este deschis.

Cursor DINAMIC. Cursoarele dinamice sunt opusul cursoarelor statice. Cursoarele dinamice reflectă toate modificările aduse rândurilor din setul lor de rezultate atunci când derulați prin cursor. Valorile datelor, ordinea și apartenența rândurilor din setul de rezultate se pot modifica la fiecare preluare. Toate instrucțiunile UPDATE, INSERT și DELETE făcute de toți utilizatorii sunt vizibile prin cursor. Actualizările sunt vizibile imediat dacă sunt făcute prin intermediul cursorului folosind fie o funcție API, cum ar fi SQLSetPos, fie clauza Transact-SQL WHERE CURRENT OF. Actualizările efectuate în afara cursorului nu sunt vizibile până când nu sunt confirmate, cu excepția cazului în care nivelul de izolare a tranzacției cursorului este setat la citire neconfirmată.

În cadrul sintaxei instrucțiunii DECLARE CURSOR se utilizează:

- `select_statement` - este o instrucțiune SELECT standard care definește setul de rezultate al cursorului. Cuvintele cheie FOR BROWSE și INTO nu sunt permise în select_statement a unei declarații de cursor;
- `SCROLL` - specifică faptul că sunt disponibile toate opțiunile de preluare (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). SCROLL nu poate fi specificat dacă este specificat și FAST_FORWARD. Dacă SCROLL nu este specificat, atunci este disponibilă doar opțiunea de preluare NEXT și cursorul devine FORWARD_ONLY;
- `READ_ONLY` - împiedică actualizările efectuate prin intermediul acestui cursor. Cursorul nu poate fi referit într-o clauză WHERE CURRENT OF dintr-o instrucțiune UPDATE sau DELETE. Această opțiune înlocuiește capacitatea implicită a unui cursor de a fi actualizat;
- `UPDATE [OF column_name [,...n]]-` definește coloane actualizabile în cadrul cursorului. Dacă este specificat `OF <nume_coloană> [, <... n>]`, numai coloanele listate permit modificări. Dacă UPDATE este specificat fără o listă de coloane, toate coloanele pot fi actualizate;
- `LOCAL` - specifică faptul că domeniul de aplicare al cursorului este local pentru lotul de instrucțiuni, procedura stocată sau declanșatorul în care a fost creat cursorul. Numele cursorului este valabil numai în acest domeniu. Cursorul poate fi referit prin variabilele de cursor locale din lot, procedură stocată sau declanșator sau un parametru OUTPUT al procedurii stocate. Un parametru OUTPUT este utilizat pentru a trece cursorul local înapoi lotului de apelare, procedurii stocate sau declanșatorului, care poate atribui parametrul unei variabile de cursor pentru a face referire la cursor după terminarea procedurii stocate. Cursorul este implicit dealocat atunci, când lotul de instrucțiuni, procedura stocată sau declanșarea se termină, cu excepția cazului în care cursorul a fost trecut înapoi într-un parametru OUTPUT. Dacă este transmis înapoi într-un parametru OUTPUT, cursorul este dealocat atunci, când ultima variabilă care face referire la el este dealocată sau iese din domeniul de aplicare;
- `GLOBAL` - specifică faptul că domeniul de aplicare al cursorului este global pentru conexiune. Numele cursorului poate fi referit în orice procedură stocată sau lot de instrucțiuni executat de conexiune. Cursorul este doar implicit dealocat la deconectare;
- `FORWARD_ONLY` - specifică faptul că cursorul se poate deplasa numai înainte și poate fi derulat de la primul la ultimul rând. FETCH NEXT este singura opțiune de preluare acceptată. Toate declarațiile de inserare, actualizare și ștergere făcute de utilizatorul curent (sau comise de alți utilizatori) care afectează rândurile din setul de rezultate sunt vizibile pe măsură ce rândurile sunt preluate. Deoarece cursorul nu poate fi derulat înapoi, totuși, modificările aduse rândurilor din baza de date după ce a fost preluat rândul nu sunt vizibile prin cursor. Cursoarele numai înainte sunt dinamice în mod implicit, ceea ce înseamnă că toate modificările sunt detectate pe măsură ce rândul curent este procesat. Acest lucru asigură deschiderea mai rapidă a cursorului și permite setului de rezultate să afișeze actualizările făcute pentru tabelele de bază. În timp ce cursoarele numai înainte nu acceptă derularea înapoi, aplicațiile pot reveni la începutul setului de rezultate prin închiderea și redeschiderea cursorului. Dacă FORWARD_ONLY este

specificat fără cuvintele cheie STATIC, KEYSET sau DYNAMIC, cursorul funcționează ca un cursor dinamic. Când nu sunt specificate nici FORWARD_ONLY, nici SCROLL, FORWARD_ONLY este implicit, cu excepția cazului în care sunt specificate cuvintele cheie STATIC, KEYSET sau DYNAMIC. Cursoarele STATIC, KEYSET și DYNAMIC sunt implicit SCROLL. Spre deosebire de API-urile bazei de date precum ODBC și ADO, FORWARD_ONLY este acceptat cu cursoare STATIC, KEYSET și DYNAMIC Transact-SQL;

- `FAST_FORWARD` - specifică un cursor FORWARD_ONLY, READ_ONLY cu optimizări de performanță activate. FAST_FORWARD nu poate fi specificat dacă este specificat și SCROLL sau FOR_UPDATE. Acest tip de cursor nu permite modificarea datelor din interiorul cursorului;
- `READ_ONLY` - împiedică actualizările efectuate prin intermediul acestui cursor. Cursorul nu poate fi referit într-o clauză WHERE CURRENT OF dintr-o instrucțiune UPDATE sau DELETE. Această opțiune înlocuiește capacitatea implicită a unui cursor de a fi actualizat;
- `SCROLL_LOCKS` - specifică faptul că actualizările poziționate sau ștergerile efectuate prin intermediul cursorului sunt garantate cu succes. SQL Server blochează rândurile pe măsură ce sunt citite în cursor pentru a asigura disponibilitatea lor pentru modificări ulterioare. SCROLL_LOCKS nu poate fi specificat dacă este specificat și FAST_FORWARD sau STATIC;
- `OPTIMISTIC` - specifică că actualizările poziționate sau ștergerile efectuate prin cursor nu reușesc dacă rândul a fost actualizat de când a fost citit în cursor. SQL Server nu blochează rândurile pe măsură ce sunt citite în cursor. În schimb, utilizează comparații ale valorilor coloanei de marcaj temporal sau o valoare a sumei de control dacă tabelul nu are o coloană marcaj temporal, pentru a determina dacă rândul a fost modificat după ce a fost citit în cursor. Dacă rândul a fost modificat, încercarea de actualizare sau ștergere poziționată eșuează. OPTIMISTIC nu poate fi specificat dacă este specificat și FAST_FORWARD;
- `TYPE_WARNING` - specifică faptul că un mesaj de avertizare este trimis către client atunci, când cursorul este convertit implicit de la tipul solicitat la altul.

Un cursor creat se deschide cu instrucțiunea OPEN:

```
OPEN { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Returnarea unui rând specificat din cursor se realizează cu instrucțiunea FETCH:

```
FETCH
          [ [ NEXT | PRIOR | FIRST | LAST
                    | ABSOLUTE { n | @nvar }
                    | RELATIVE { n | @nvar }
               ]
               FROM
          ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

Instrucțiunea CLOSE închide un cursor deschis, eliberând setul de rezultate curent și eliberând orice blocare a cursorului ținute pe rândurile pe care este poziționat cursorul. Instrucțiunea CLOSE lasă structurile de date disponibile pentru redeschidere, dar preluarea și actualizările poziționate nu sunt permise până când cursorul nu este redeschis. Instrucțiunea CLOSE trebuie să fie aplicată pe un cursor deschis. CLOSE nu este permisă pe cursoarele care au fost doar declarate sau sunt deja închise.

```
CLOSE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Instrucțiunea DEALLOCATE elimină o referință de cursor. Când ultima referință de cursor este dealocată, structurile de date care cuprind cursorul sunt eliberate de Microsoft SQL Server.

```
DEALLOCATE { { [ GLOBAL ] cursor_name } | @cursor_variable_name }
```

Exemplu 1.

```
USE AdventureWorks2012;
GO
DECLARE contact_cursor CURSOR FOR
SELECT LastName FROM Person.Person
WHERE LastName LIKE 'B%'
ORDER BY LastName;
OPEN contact_cursor;
-- Perform the first fetch.
FETCH NEXT FROM contact_cursor;

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
      BEGIN
   -- This is executed as long as the previous fetch succeeds.
            FETCH NEXT FROM contact_cursor;
      END
CLOSE contact_cursor;
DEALLOCATE contact_cursor;
GO
```

Întrebări de control:

1. Care sunt cele patru proprietăți fundamentale care caracterizează o tranzacție SQL și ce reprezintă fiecare dintre ele?
2. Care este diferența dintre efectele comenzilor COMMIT TRANSACTION și ROLLBACK TRANSACTION asupra datelor și a resurselor sistemului?
3. Care sunt cele trei categorii principale de declanșatoare (triggers) în SQL Server în funcție de tipul de eveniment la care reacționează?
4. Prin ce se deosebesc declanșatoarele DML de tip AFTER față de cele de tip INSTEAD OF?
5. Ce rol au tabelele temporare inserted și deleted în cadrul declanșatoarelor DML?
6. Care sunt caracteristicile principale care diferențiază un cursor STATIC de un cursor DYNAMIC?
7. Care este ordinea corectă a etapelor din ciclul de viață al unui cursor T-SQL (de la definire până la eliberarea resurselor din memorie) și ce instrucțiuni se folosesc la fiecare pas?

Capitolul 13. CICLUL DE VIAȚĂ AL UNEI BAZE DE DATE

13.1. Etapele de proiectare și implementare a bazei de date

Ciclul de viață al unei baze de date reprezintă setul de pași, tehnici, metode și instrumente utilizate pentru transpunerea modelului de date într-un model fizic. Ciclul de viață conține mai multe etape interconectate, reprezentate în Figura 13.1.

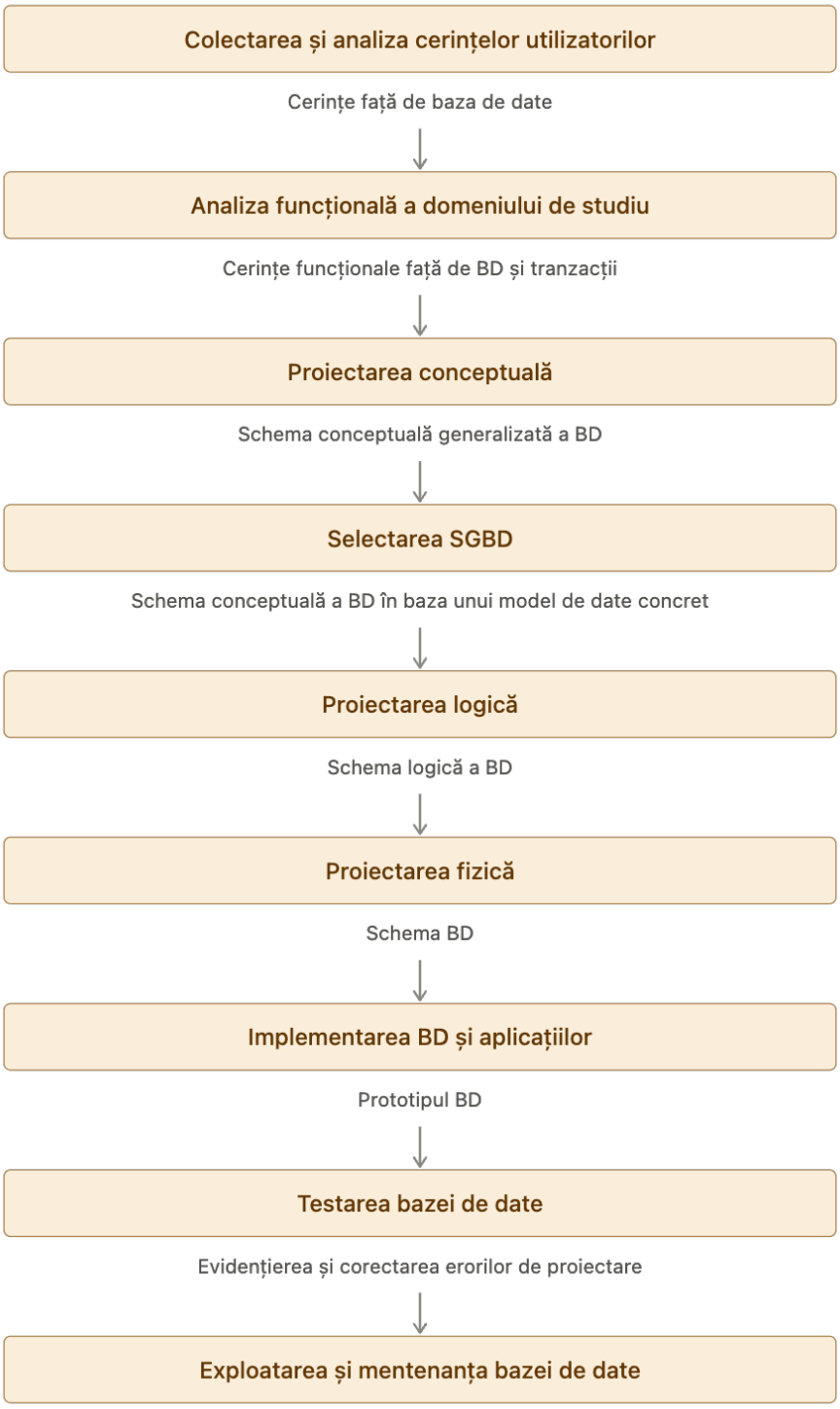


Figura 13.1. Etapele ciclului de viață a bazei de date

Fiecare etapă se caracterizează prin specificul său, activitățile caracteristice și rezultatul final al acestor activități. La prima etapă de colectare și analiză a cerințelor utilizatorilor sunt analizate documentele din domeniul de obiecte analizat și se identifică viitoarele interpelări, care urmează să fie adresate bazei de date. Aceste cerințe față de baza de date devin obiectul unei analize minuțioase cu scopul identificării tuturor regulilor de afaceri, specifice domeniului de obiecte, și formulării cerințelor funcționale, care trebuie să satisfacă viitoarea baza de date.

Activitățile de proiectare a bazei de date încep cu etapa de proiectare conceptuală, în cadrul căreia se identifică toate entitățile domeniului de obiecte proiectat și toate relațiile funcționale existente între aceste entități. Etapa se finalizează cu elaborarea schemei conceptuale a bazei de date și documentației de proiect conexe acestei scheme. În continuare urmează luarea deciziei de selectare a modelului de organizarea a datelor în baze de date și, ca urmare a acestei deciziei, selectarea unui SGBD concret capabil să implementeze acest model.

La etapa de proiectare logică a bazei de date schema conceptuală este rafinată în continuare și modificată conform exigenților modelului de organizarea a bazelor de date selectat, obținându-se schema logică a bazei de date. În această schemă sunt identificate toate atributele entităților și relațiilor; sunt stabiliți identificatorii interni și externi; sunt elaborate constrângerile de integritate și referențiale, și identificate domeniile de valori ale atributelor; sunt elaborate regulile de operare ale bazei de date conform regulilor de afaceri ale domeniului de obiecte și algoritmii specifici de calcul a valorilor derivate și totalizatoare. În cazul bazelor de date relaționale și obiect-relaționale etapa de proiectare logică se finalizează cu normalizarea relațiilor și obținerea schemei logice finale a bazei de date.

La etapa de proiectare fizică se realizează activitățile de proiectare a structurii interne a fișierelor bazei de date conform schemei logice a bazei de date. Sunt create toate tabelele bazei de date și implementate toate constrângerile de integritate și referențiale, obținându-se schema bazei de date. Urmează activitățile de elaborarea a interpelărilor utilizatorilor și aplicaților cu baze de date. Se selectează un subset de date reale din domeniul de obiecte și se introduc în baza de date, obținându-se un prototip al bazei de date, asupra căruia se aplică un sistem complex de activități de testare și evaluare a conținutului BD cu scopul determinării gradului de corespundere prototipului cerințelor funcționale ale proiectului și depistării neconformităților și erorilor de proiectare. După analiza și corectarea erorilor depistate se completează baza de date cu toate datele din domeniul de obiecte proiectat și se transmite pentru exploatare operațională, asigurându-se mentenanța necesară pe parcursul utilizării ei.

13.2. Etapa de analiză și specificare a cerințelor

Etapa de analiză a cerințelor implică examinarea domeniului de interes ce urmează a fi modelat, intervievarea beneficiarului bazei de date, obținerea de informații despre sisteme asemănătoare, analiza cerințelor de dezvoltare viitoare și obținerea de informații despre întregul domeniu luat în ansamblu. Colectarea și analiza reprezintă o etapă extrem de importantă în proiectarea unei baze de date efectuată pentru a:

- *dezvolta și înțelege sistemul ce urmează a fi creat;*
- *dezvolta și înțelege domeniul de interes ce urmează a fi modelat;*
- *construi un sistem nou;*
- *reevalua un proces;*
- *instrui utilizatorii și managerii;*
- *comunica modul de funcționare a sistemului;*
- *evalua controlul.*

La etapa de analiză managerii, proiectanții și consultanții folosesc o serie de tehnici de creare a diagramelor și graficelor informative. Unele dintre cele mai folosite sunt:

- ***Diagrame de conținut*** – prezintă în mod simplificat felul în care un proces intră în legătură cu entitățile aflate în afara procesului.

- ***Diagrame logice de flux de date*** – prezintă reprezentarea logică a procesului. Sunt descrieri abstracte care nu specifică nici cine efectuează o anumită activitate și nici locul în care se află anumite elemente. Dacă procesul nu este elementar, acesta poate fi descompus sau partiționat. Astfel de diagrame se modifică foarte rar.
- ***Diagrame fizice de flux de date*** – prezintă atributele fizice ale procesului. Nu prezintă atributele logice ale procesului. Ele se pot modifica relativ des și nu pot fi descompuse.
- ***Schema logică a sistemului*** – prezintă prelucrarea informațiilor (intrări, ieșiri, înmagazinarea datelor, activități, fluxuri logice) și procesele operaționale (entități, fluxuri fizice, operații). Aceasta este cea mai veche tehnică.
- ***Diagrame entitate-relație*** – în care se specifică toate entitățile unui domeniu de activitate și relațiile în care sunt antrenate aceste entități

La etapă de analiză și specificare a cerințelor se încearcă cât mai clar și mai detaliat să fie definit viitorul sistem proiectat. Definirea viitorului sistem reprezintă identificarea scopului și a limitelor bazei de date. Aici sunt prezentate cele mai multe dintre interpelările (viziunile) utilizatorilor asupra bazei de date conform activităților informaționale derulate în cadrul sistemului informațional al unității economice. Definirea viitorului sistem necesită:

- Identificarea utilizatorilor finali care pun întrebări și efectuează tranzacții în baza de date;
- Identificarea tipului de interfață a utilizatorului ce urmează a fi elaborată (*grafică; bazată pe formulare; de tip SQL; de tip generator de rapoarte; bazată pe meniuri*)
- Identificarea și analiza activităților informaționale cu implicarea bazei de date;
- Identificarea tuturor interpelărilor ce urmează să fie aplicate bazei de date;
- Identificarea resurselor necesare pentru dezvoltarea proiectului (*resurse umane; resurse software, resurse hardware, alte baze de date existente în sistemul informatic al unității economice, resurse organizaționale (documentația de reglementare a activităților informaționale în cadrul întreprinderii), procesele tehnologice existente de prelucrare a fluxurilor informaționale în cadrul întreprinderii);*
- Identificarea metodelor și tehnicilor de refacere și arhivare care vor fi aplicare bazei de date **(***refacerea se efectuează în timpul funcționării sau inactivității sistemului informatic; dimensiunea copiei de siguranță a bazei de date; categoria de date care urmează să fie periodic arhivate; strategia de arhivare care va fi utilizată, etc.);*
- Identificarea cerințelor pentru sistemul de securitate a bazei de date **(** *elaborarea sistemului de roluri și permisiuni la nivel de bază de date și server; divizarea utilizatorilor în categorii și grupe; elaborarea sistemului de identificare prin parole a utilizatorilor, etc.).*

13.3. Procesul de proiectare a bazei de date

Proiectarea unei baze de date reprezintă un proces ce implică dezvoltarea și rafinarea structurii unei baze de date pe baza cerințelor formulate de către beneficiarul bazei de date și a cerințelor deduse pe baza efectuării analizei domeniului ce urmează a fi modelat. Principalul obiectiv urmărit la proiectarea unei baze de date este asigurarea consistenței, integrității și preciziei datelor. Dacă proiectul unei baze de date este incorect, atunci este foarte greu de extras informațiile necesare, fiind posibilă obținerea de informații false.

În scopul realizării unui proiect corect al unei baze de date trebuie urmărite o serie de obiective, cum ar fi:

- baza de date trebuie să înmagazineze **datele necesare** obținerii de informații descrise în faza de analiză a cerințelor precum și **orice interogare** ce poate fi pusă la un moment dat de către utilizator;
- **tabelele** trebuie să fie construite **corect și eficient**; fiecare tabel din baza de date reprezentând o singură entitate și fiind compus din **câmpuri distincte**, păstrând o **redundanță minimă a datelor**, fiecare **înregistrare** dintr-un tabel putând fi identificată cu ajutorul unei **valori unice**;

- **integritatea datelor** se impune la nivel de **tabel, câmp, relație**; aceste nivele de integritate ajută la garantarea faptului că structurile de date împreună cu valorile acestora sunt valide și precise în orice moment;
- **baza de date** trebuie să suporte **regulile specifice domeniului pe care îl modelează**; datele trebuie să ofere **informație validă și precisă**, având tot timpul înțeles;
- la proiectarea bazelor de date trebuie să se prevadă **dezvoltările viitoare**; structura bazei de date trebuie să fie ușor de modificat sau extins odată cu modificarea cerințelor impuse.

Este greu de realizat toate aceste obiective permanent, dar un proiect corect al unei baze de date trebuie să aibă în vedere obținerea cât mai rapidă a rezultatelor.

Planificarea bazei de date reprezintă controlul activităților ce permit realizarea efectivă și eficientă a etapelor de proiectare a unei baze de date. În acest scop se realizează următoarea documentație:

- **identificarea scopului bazei de date**;
- **identificarea obiectivelor bazei de date** (identificarea fiecărei activități individuale ce trebuie suportată de baza de date).

Proiectarea bazei de date cuprinde următoarele trei etape (Figura 13.2):

- ***proiectarea conceptuală a bazei de date;***
- ***proiectarea logică a bazei de date;***
- ***proiectarea fizică a bazei de date.***

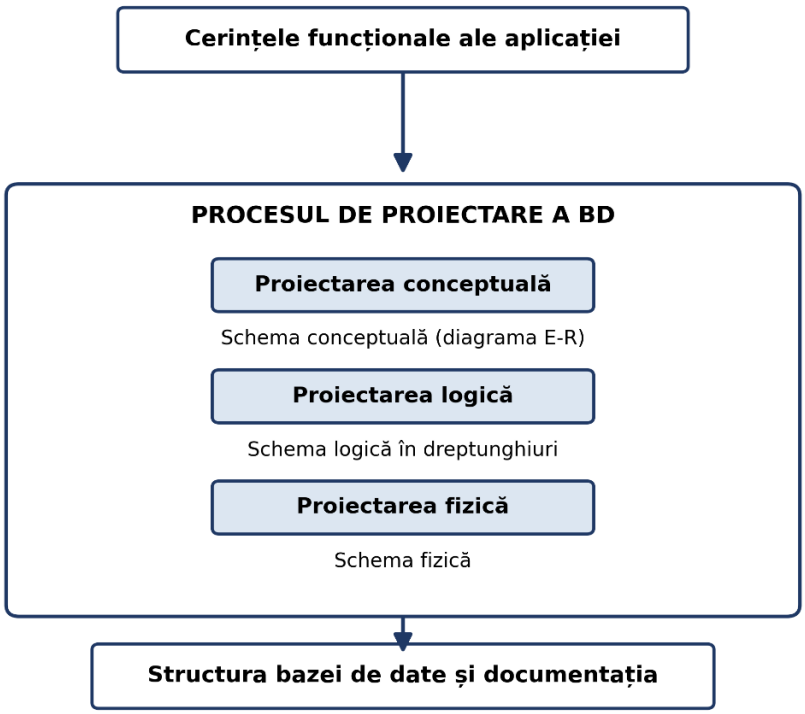


Figura 13.2. Etapele procesului de proiectare a bazei de date

Proiectarea conceptuală reprezintă procesul de construire a unui model al informațiilor utilizat în cadrul unui domeniu de interes, independent de toate considerațiile fizice. Primul pas în proiectarea bazei de date este cel de definire a cerințelor datelor specifice acelui domeniu. În acest scop este importantă definirea:

- ***tipurilor de informații care vor fi utilizate în baza de date;***
- ***categoriilor de informații necesare sistemului și aplicațiilor;***
- ***regulilor specifice domeniului (tipuri de relații dintre entități);***
- ***constrângerilor aplicate (de integritate și referențiale);***

- ***tipurilor de rapoarte generate;***
- ***scopului principal al tuturor informațiilor;***
- ***nivelului necesar de securitate a informațiilor;***
- ***informațiilor ce urmează a fi extinse în viitor;***
- ***construirea diagramelor entitate-relație.***

Proiectarea logică reprezintă procesul de constituire a unui **model al informațiilor** utilizat la modelarea unui domeniu de interes bazat pe un anumit model specific de date (de exemplu, modelul relațional), dar independent de orice alte considerații fizice. Proiectarea logică a bazei de date ajută în continuare la **definirea și achiziționarea cerințelor**. Proiectarea logică a bazei de date presupune descrierea fiecărui element de informație și a relațiilor dintre aceste elemente de informație. Un model logic poate determina dacă proiectul conține toate informațiile ce trebuie să fie extrase și reflectă relațiile necesare care se pun în concordanță cu regulile domeniului de interes.

Modelul logic al bazei de date descrie dimensiunea, forma sistemelor necesare bazei de date și arată necesitățile informaționale și operaționale ale domeniului de interes.

Pentru crearea modelului logic al bazei de date este necesară:

- **definirea tabelelor** în funcție de cerințele specifice domeniului pe care îl modelează (așa cum s-a stabilit în modelul conceptual);
- **determinarea conținutului fiecărui tabel**;
- **determinarea valorilor specifice fiecărei coloane și constrângerilor de integritate;**
- **determinarea cheilor primare și cheilor externe**;
- **determinarea relațiilor dintre tabele și constrângerilor referențiale**;
- **normalizarea tabelelor până la forma normală trei (cerința minimă).**

După încheierea definițiilor tabelelor este obligatorie reluarea discuțiilor cu beneficiarul bazei de date pentru a revedea întreaga activitate desfășurată și asigura că sunt reprezentate cele mai importante entități, denumirile tabelelor și atributelor sunt potrivite și inteligibile, iar atributele din tabele sunt corecte.

Proiectarea fizică reprezintă procesul de descriere a implementării bazei de date **pe mediile de stocare**. Sunt descrise **structurile de stocare și metodele de acces** utilizate pentru a obține un acces eficient la date. Proiectul fizic al bazei de date efectuează și o rafinare a modelului logic; acesta **transpune modelul logic** într-un **sistem relațional de gestiune a bazelor de date**. În această fază este obligatorie examinarea modului în care utilizatorul accesează baza de date.

Se recomandă separarea modelului logic de modelul fizic astfel încât modificările să poată fi documentate în conformitate cu modelul fizic pentru a corespunde constrângerilor. În acest fel, modelul logic rămâne **independent de tehnologie**, ținând cont doar de cerințele specifice domeniului pe care îl modelează.

La etapa de proiectare fizică se determină:

- ***datele folosite în mod curent***;
- ***coloanele*** ce urmează a fi ***indexate*** pentru a obține un acces mai rapid la date;
- ***spațiul memoriei necesar*** precum și cel prevăzut pentru creșterea dimensiunilor bazei de date;
- dacă ***denormalizarea bazei*** de date va duce la creșterea performanțelor acesteia;
- **cererile de funcționalitate** (*procesarea secvențială a tuplurilor; tuplurile ce îndeplinesc o anumită condiție impusă prin intermediul unei valori; tuplurile inserate sau eliminate);*
- **obiective de performanță** (*evitarea pierderii inutile de spațiu; primirea într-un timp cât mai scurt a răspunsului).*

În procesul de proiectare a bazelor de date proiectanții sistemului folosesc anumite **strategii de proiectare a aplicațiilor** cum ar fi:

- **Proiectarea de sus în jos**: se pornește cu cel mai înalt nivel de abstractizare după care urmează rafinarea (de exemplu, se stabilesc entitățile, după care se formează subclasele, iar apoi se adaugă atributele).

- **Proiectarea de jos în sus**: se pornește cu abstractizările de bază, după care urmează combinarea acestora (de exemplu, atributele se grupează formând entități). În acest fel se pot forma concepte elementare, colecții de concepte elementare, scheme.
- **Proiectarea din interior spre exterior**: este o formă specială de proiectare de jos în sus care se axează pe un set central de concepte (se selectează cele mai importante concepte, schema inițială, se creează schemele intermediare din care se deduce apoi schema finală).
- **Proiectarea mixtă**: se pornește mai întâi cu proiectarea de sus în jos, urmată de proiectarea dinspre interior spre exterior sau de jos în sus.

13.4. Implementarea și întreținerea bazei de date

Implementarea reprezintă realizarea fizică a bazei de date și a aplicațiilor care folosesc baza de date. Implementarea bazei de date se poate face folosind ***limbajul de definire a datelor existent în cadrul sistemului de gestiune a bazelor de date.*** Instrucțiunile limbajului de definire a datelor creează structura bazei de date și fișierele bazei de date. Programele aplicație sunt implementate cu ajutorul **limbajului de manipulare a datelor** și, posibil, cu ajutorul unui limbaj gazdă de programare cu ajutorul căruia se creează ecrane, meniuri, formulare și rapoarte pe de o parte, iar pe de altă parte se introduc elementele de control a securității și integrității datelor.

Pentru a crea o aplicație cu baze de date trebuie ales cel mai potrivit tip de sistem de gestiune a bazei de date. Acesta trebuie ales în așa fel încât să corespundă atât necesităților actuale, cât și celor viitoare. În acest scop trebuie făcută o apreciere asupra caracteristicilor sistemului raportate la cerințele bazei de date și a aplicației (aplicațiilor) create pe baza acesteia. Se recomandă ca SGBD să fie selectat după faza de modelare conceptuală și înainte de faza proiectării fizice pentru a avea determinările necesare.

La selectarea unui SGBD este necesar de avut în vedere:

- **Costul SGBD**, care se bazează pe:
  - *costul de achiziție a programelor;*
  - *costul mentenanței;*
  - *costul de achiziție a componentelor hardware;*
  - *costul de creare a bazei de date sau a conversiei acesteia;*
  - *costul legat de personal;*
  - *costul de pregătire a personalului;*
  - *costul de operare.*
- **Modelul de date implementat**, care depinde de:
  - *structura și utilizarea datelor;*
  - *familiarizarea programatorilor și utilizatorilor finali cu sistemul;*
  - *disponibilitatea serviciilor oferite de către producător, a programelor de comunicare, a programelor de introducere a datelor, a instrumentelor de proiectare și monitorizare etc*.

După selectarea SGBD concret și elaborarea structurii fizice a bazei de date se trece la **proiectarea programelor de aplicații** cu baza de date. În această fază trebuie creată interfața cu utilizatorul precum și programele aplicație care folosesc datele din baza de date. În acest scop este necesară:

- **proiectarea tranzacțiilor**:
  - *tranzacții folosite la extragerea datelor;*
  - *tranzacții folosite la actualizarea datelor;*
  - *tranzacții mixte (pentru extragere și pentru actualizarea datelor).*
- **proiectarea formularelor și rapoartelor**. În acest scop este necesar de cunoscut:
  - *interogările folosite, tranzacțiile și aplicațiile ce trebuie create;*
  - *frecvența apariției interogărilor și tranzacțiilor, constrângerile impuse interogărilor și tranzacțiilor, și performanța;*
  - *frecvența operațiilor de actualizare și profilului de actualizare (dinamic sau static, aplicarea unor strategii de prevedere a actualizărilor posibile).*
- **crearea interfeței cu utilizatorul și a aplicațiilor care lucrează cu datele din baza de date.**

După finalizarea elaborării aplicațiilor se elaborează prototipul bazei de date. **Prototipul** reprezintă realizarea unui model de lucru al bazei de date, el fiind folosit de către utilizatori la identificarea părților tari și a slăbiciunilor sistemului, ajutând la elaborarea de amendamente la proiectul teoretic. Pe baza unui astfel de model se clarifică mai bine atât cerințele domeniului care se modelează, cât și fezabilitatea proiectului. Prototipurile sunt relativ ieftine, putând fi folosite următoarele strategii:

- *crearea unui prototip al cerințelor (prototip creat după faza de analiză a cerințelor);*
- *crearea unui prototip evolutiv (utilizat la fiecare etapă de proiectare).*

În această fază este obligatorie dezvoltarea unui model de lucru al bazei de date pentru:

- ***a verifica caracteristicile de bună funcționare;***
- ***a identifica caracteristicile improprii;***
- ***a identifica îmbunătățirile posibile;***
- ***a identifica noile obiecte ale bazei de date (omise din greșeală la etapele precedente).***

Prototipul bazei de date, care trebuie să conțină date reale din domeniul de obiecte proiectat, este utilizat pentru testarea sistemului elaborat. **Testarea** reprezintă procesul de rulare a unor programe de aplicație cu scopul de a descoperi eventualele erori ale bazei de date. Baza de date trebuie testată cu atenție, stabilindu-se diverse strategii de testare elaborate metodic, cu atenție și rigoare. Pentru testare trebuie folosite seturi reale de date și dacă este posibil, verificările să se facă pe alte sisteme decât cel folosit la proiectare. Testarea se face atât pentru descoperirea eventualelor erori, cât și pentru descoperirea de noi modalități de îmbunătățire a performanțelor

Principalele măsurători efectuate în procesul de testare se bazează pe:

- ***timpul de răspuns la o interogare sau o tranzacție;***
- ***evaluarea tranzacțiilor (câte tranzacții se pot procesa pe secundă/milisecundă);***
- ***timpul de generare a unui raport al întregii baze de date;***
- ***timpul de achiziționare a datelor noi;***
- ***elemente de analiză și experimentale (*** *gradul de utilizare al microprocesorului și a dispozitivelor de intrare/ieșire, media interogărilor introduse pe minut și determinarea costului; numărul de tranzacții pe secundă*).

După realizarea testării și lichidarea tuturor erorilor de proiectare, depistate în procesul de testare, baza de date se completează cu toate datele necesare pentru gestiunea unității economice și se predă pentru exploatare. Preventiv se realizează instruirea utilizatorilor desemnați din cadrul întreprinderii, care vor folosi baza de date și aplicațiile elaborate.

Întreținerea operațională a bazei de date reprezintă procesul de monitorizare și întreținere a sistemului ce urmează după instalarea acestuia. Această etapă implică:

- *monitorizarea performanțelor bazei de date; dacă acestea devin inacceptabile, baza de date trebuie reorganizată;*
- *întreținerea și actualizarea bazei de date atunci, când este necesar (pot să apară cereri noi).*

Structura bazei de date nu este greu de modificat dacă proiectarea a fost făcută corespunzător, altfel vor trebui depuse eforturi suplimentare. Dacă baza de date a fost corect proiectată, atunci:

- **este ușor de modificat și de întreținut structura acesteia -** modificările efectuate asupra unui câmp sau tabel nu vor afecta alte câmpuri sau tabele ale bazei de date;
- **datele sunt ușor de modificat -** modificările efectuate asupra unei valori dintr-un anumit tabel al bazei de date nu vor afecta valorile altor câmpuri din alte tabele. Într-o structură corectă, modificările trebuie făcute într-un singur loc;
- **informația se extrage cu ușurință din baza de date -** tabelele sunt corect construite, orice relație stabilită între ele fiind valabilă;
- **aplicațiile sunt ușor de proiectat și elaborat –** aplicațiile elaborate vor conține instrucțiunile de manipulare cu datele și nu sunt afectate de necesitatea reproiectării structurii bazei de date datorită unei proiectări incorecte inițiale a bazei de date.

Întrebări de control:

1. Care sunt etapele principale ale ciclului de viață al unei baze de date?
2. Ce rol are etapa de colectare și analiză a cerințelor utilizatorilor?
3. Ce activități specifice implică definirea viitorului sistem la etapa de analiză și specificare a cerințelor?
4. Care sunt principalele obiective care trebuie urmărite în faza de proiectare pentru a asigura consistența, integritatea și precizia datelor dintr-o bază de date?
5. În ce constă proiectarea conceptuală a unei baze de date și ce elemente fundamentale trebuie definite în cadrul acesteia?
6. Care sunt pașii obligatorii pentru realizarea modelului logic al bazei de date și la ce nivel de normalizare trebuie aduse tabelele?
7. Ce decizii și optimizări specifice de performanță se iau în cadrul etapei de proiectare fizică a bazei de date?
8. Care sunt cele patru strategii de proiectare a aplicațiilor folosite de proiectanții de sisteme?
9. Ce factori economici și tehnici trebuie luați în considerare atunci când se selectează un Sistem de Gestiune a Bazelor de Date (SGBD)?
10. Ce scop servește dezvoltarea unui prototip al bazei de date?

Capitolul 14. PROIECTAREA CONCEPTUALĂ A BAZEI DE DATE

14.1. Modelul Entitate-Relație

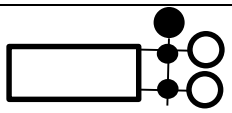
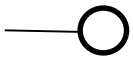
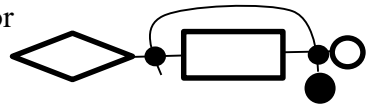
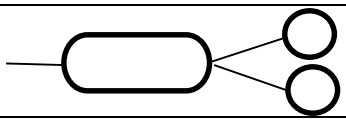
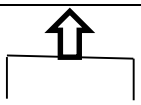
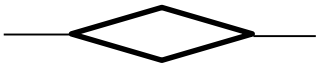
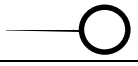
Modelul Entitate-Relație (Entity-Relationship Model) a fost elaborat în anul 1976 de P.S. Chen și reprezintă un **model conceptual de nivel înalt** al unei baze de date, care definește **mulțimile de entități** și **asocierile** dintre ele, dar **nu impune** nici un mod specific de structurare și prelucrare (gestiune) a datelor. Elementele esențiale ale modelului E-R, folosit în proiectarea bazelor de date, sunt **entitățile (entities**) și **asocierile** dintre acestea **(relationships).**

O entitate este **„orice ce poate fi identificat în mod distinctiv"**. O entitate se refera la un aspect al realității obiective care poate fi deosebit de restul universului și poate reprezenta un obiect fizic, o activitate, un concept, etc. Orice entitate este descrisă prin atributele sale. **Un atribut** este o **proprietate** care descrie un anumit aspect al unei entități.

O relație (asociere) este o **corespondență funcțională** între entități din două sau mai multe mulțimi de entități. **Gradul** unei relații este dat **de numărul de mulțimi** de entități asociate. Asocierile pot fi **binare** (între 2 mulțimi de entități) sau **multiple**.

Modelul Entitate – Relație este un model de date conceptual ce pune la dispoziție o serie de *construcții* ce descriu datele necesare unei aplicații într-o manieră ușor de înțeles și independentă de criteriile de gestiune și organizare a datelor din sistem. Construcțiile modelului E-R definesc schemele ce descriu modul de organizare a datelor și dictează care *apariții de date* (valori pe care baza de date le poate stoca la diferite momente de timp) sunt legale și corecte. Aceste scheme se mai numesc **diagrame Entitate-Relație.**

Construcțiile de bază ale modelului Entitate-Relație sunt următoarele:

Entitate 	Identificator intern simplu 
Relație 	Identificator intern compus 
Atribut simplu (atomar) 	Identificator extern 
Atribut compus 	Generalizare 
Cardinalitatea unei relații n_1,N_1 n_2,N_2 	Submulțime 
Cardinalitatea unui atribut n, N 	

Construcțiile modelului Entitate-Relație sunt folosite pentru elaborarea ***diagramelor Entitate-Relație*** (în continuare, diagrame E-R). Un exemplu de diagramă E-R este prezentat în Figura 14.1.

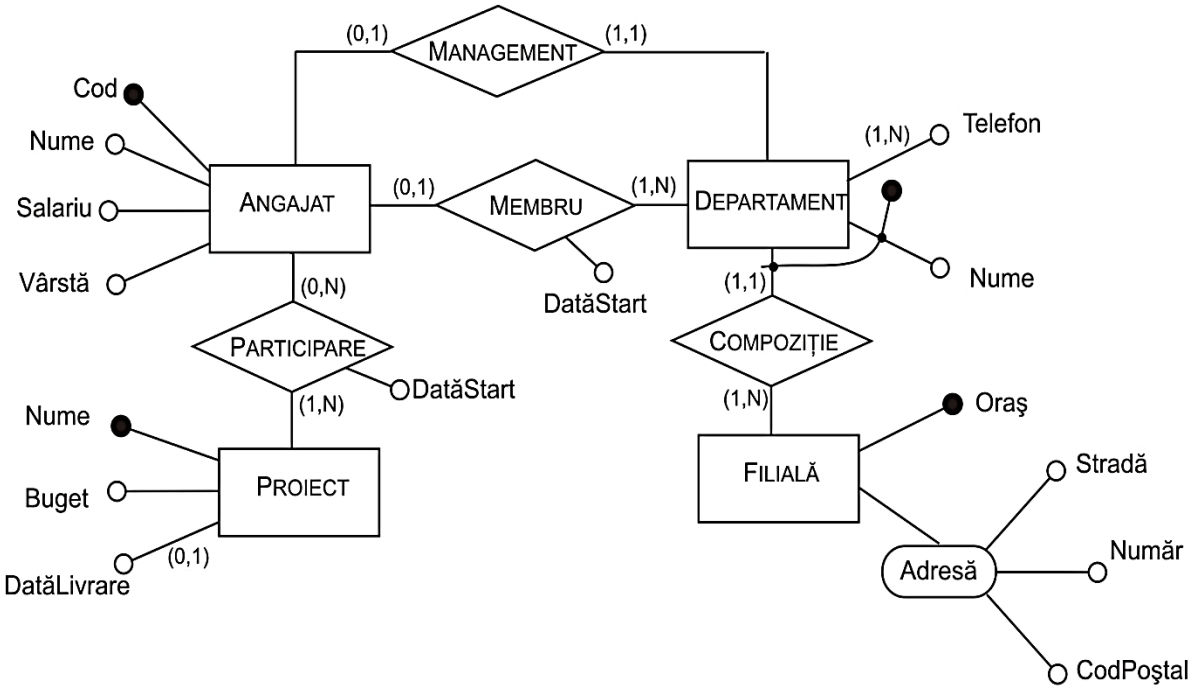


Figura 14.1. Exemplu de diagramă Entitate-Relație

14.2. Entități, atribute, relații în diagrame E-R

Entitățile formează **clase de obiecte** ce au proprietăți **comune** și existentă **autonomă.** Fiecare entitate ce caracterizează printr-un nume sub formă de substantiv la singular, care se scrie cu litere majuscule în interiorul unui dreptunghi. La nivelul întregii diagrame E-R numele entității trebuie să fie unic. De exemplu, ORAȘ, DEPARTAMENT, ANGAJAT, PRODUS, PROIECT sunt exemple de entități într-o aplicație cu bază de date. **Apariția** unei entități este un **obiect concret** din clasa de obiecte reprezentate de entitatea respectivă. Astfel, orașele Chișinău și Cahul sunt două **apariții** ale entității ORAȘ, iar lucrătorii Popescu, Ionescu și Moraru sunt trei apariții ale entității ANGAJAT.

În proiectarea bazelor de date se consideră doua categorii de entități: **entități normale** (**puternice**, **tari**, **obișnuite** – *regular entities)* și ***entități slabe*** *(dependente - weak entities).* **Entitățile normale (tari)** au o **existentă proprie** în cadrul modelului, în timp ce **entitățile slabe** nu pot exista decât dacă există o entitate normală (puternică) cu care ele sunt asociate într-**o relație slabă**. De exemplu, o entitate SUBALTERN poate să reprezinte o persoană care depinde de un angajat-supervizor al unei instituții (adică se află în subordinea acestuia). O entitate SUPERVIZOR este o entitate puternică, deoarece ea există în mod normal în modelul activității instituției, în timp ce o entitate SUBALTERN este o entitate slabă, care va apărea în modelul Entitate-Relație doar după ce se va înregistra entitatea SUPERVIZOR. Entitățile slabe se reprezintă în diagramele E-R în dreptunghiuri cu contur dublu (Figura 14.2).

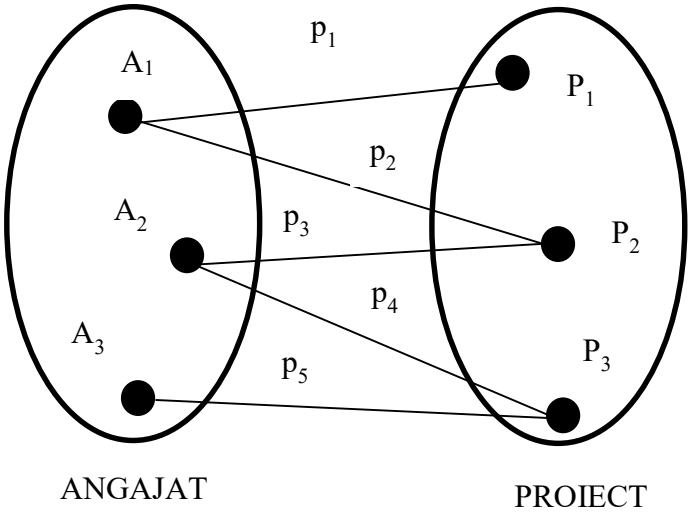


Figura 14.2. Simboluri grafice pentru entități în diagrame Entitate-Relație

Relațiile reprezintă **legături logice între două sau mai multe entități.** Relațiile se prezintă în diagrame E-R sub formă de romburi în care sunt scrise denumirile relațiilor. Denumirea unei relații reprezintă un substantiv la singular scris cu litere majuscule, care reflectă esența legăturii existente între entitățile implicate în această relație. Numele relației este **unic** pentru diagrama E-R.

De exemplu, PARTICIPARE este un exemplu de relație care există între entitățile ANGAJAT și PROIECT.

Ca și în cazul entităților, relațiile se caracterizează prin aparițiile sale, însă în acest caz **o apariție a unei relații** reprezintă **o mulțime de tupluri** formate din aparițiile entităților implicate în relație. Exemplu de apariții pentru relația PARTICIPARE, existentă între entitățile ANGAJAT și PROIECT, este prezentată în Figura 14.3. Unele și aceleași entități pot participa în relații diferite (Figura 14.4).



PARTICIPARE	
ANGAJAT	PROIECT
A_1	P_1
A_1	P_2
A_2	P_2
A_2	P_3
A_3	P_3

Figura 14.3. Aparițiile relației PARTICIPARE sub formă de mulțime de tupluri

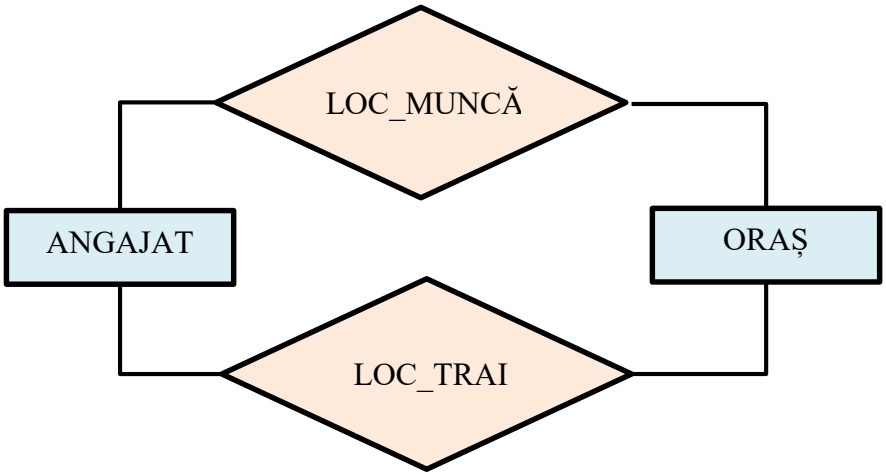
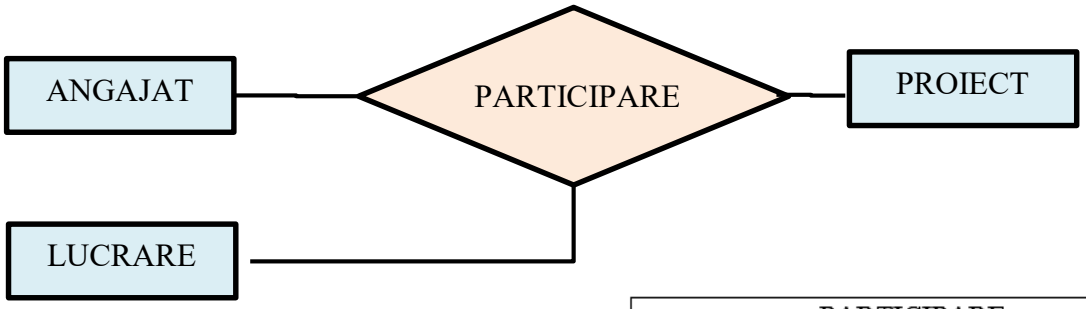


Figura 14.4. Relații diferite dintre aceleași entități

Mulțimea aparițiilor unei relații este o **relație matematica** între **mulțimile de apariții ale entităților implicate** (este o submulțime a **produsului cartezian** al celor două mulțimi de apariții ale entităților implicate) (Figura 14.3). Astfel este asigurat faptul ca nici un tuplu nu va apărea de două ori printre aparițiile unei relații. Acest aspect are consecințe importante: relația PARTICIPARE (Figura 14.3) **nu are capacitatea** de a raporta faptul ca un angajat oarecare **participă de mai multe ori** în același proiect cu **lucrări diferite** (deoarece aceasta ar trebui să producă perechi identice în aparițiile relației PARTICIPARE). În acest caz, pentru a putea specifica lucrările efectuate în cadrul proiectului de către angajat, ar trebui ca în relația PARTICIPARE să fie inclusă o **entitate nouă** LUCRARE, legată de entitățile ANGAJAT și PROIECT prin intermediul a două relații binare (Figura 14.5).



PARTICIPARE		
ANGAJAT	PROIECT	LUCRARE
A1	P1	L1
A1	P1	L2
A1	P2	L2
A1	P3	L3
A2	P1	L1
A3	P1	L3
A3	P3	L3

Figura 14.5. Relații multiple dintre entități

Între entitățile unui domeniu de obiecte pot exista ***relații recursive,*** care reprezintă relații între o entitate și ea însăși. Ca exemplu poate servi relația recursivă COLEG a entității ANGAJAT, care conectează perechi de oameni care lucrează împreună. Această relație este una **simetrică** (Figura 14.6). Pe de altă parte, relația recursivă MOȘTENIRE a entității REGE este una **asimetrică**, deoarece este necesar de asociat identificatori pentru liniile de moștenire ale relației recursive.

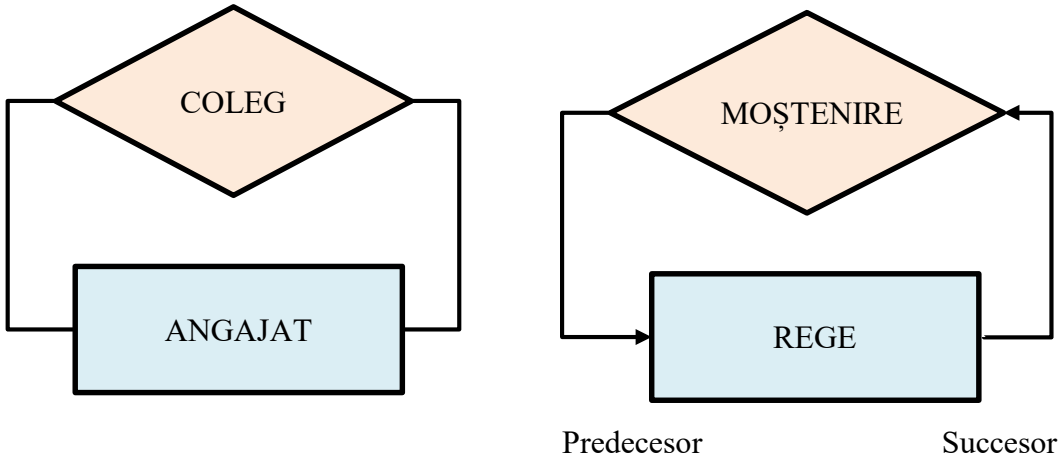


Figura 14.6. Relații recursive simetrice (stânga) și asimetrice (dreapta)

Atributele descriu proprietățile elementare ale **entităților** sau **relațiilor.** De exemplu, **Nume, Prenume** sunt atribute ale entității **STUDENT**, iar **Data, Nota** sunt atribute pentru **relația EXAMEN** (Figura 14.7).

Un atribut asociază fiecărei apariții a unei entități (sau relație) **o valoare,** aparținând unei mulțimi de valori, denumite ***domeniul atributului.*** Domeniul conține **valori admisibile** pentru atribut. De exemplu, domeniul pentru atributul **Nume** poate fi orice șir de caractere de lungime 20*,* iar domeniul pentru atributul **Nota** poate fi orice număr întreg între 1 și 10 inclusiv. Domeniile **nu sunt reprezentate grafic**, ele fiind de obicei descrise în **documentația** asociată diagramelor E-R.

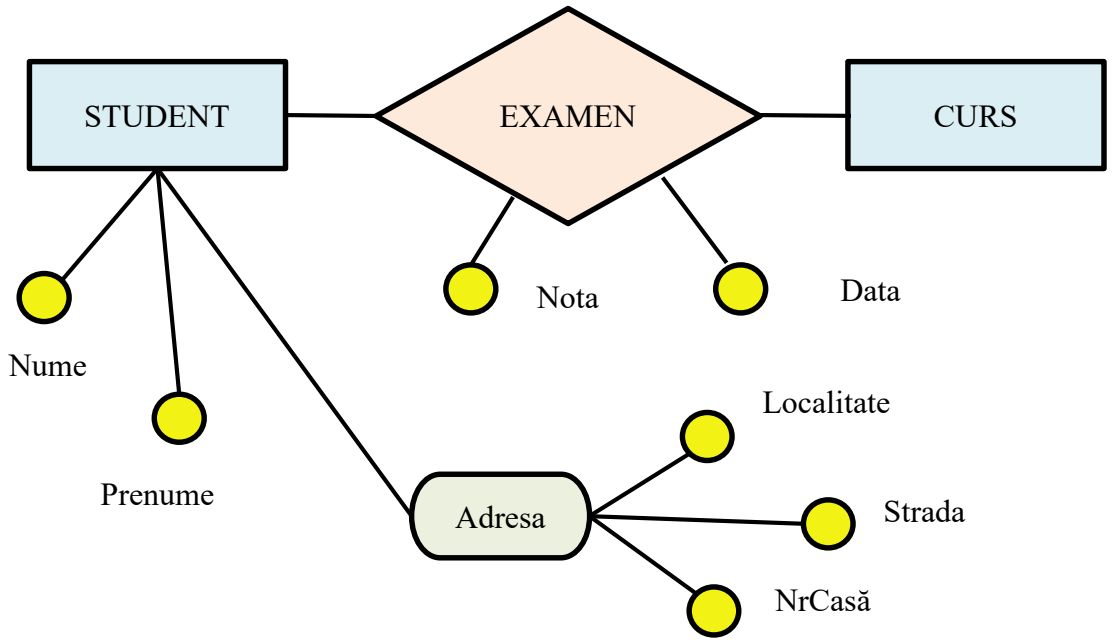


Figura 14.7. Atribute simple și compuse

Atributele pot fi ***simple,*** fiind reprezentate printr-o valoare atomară a apariției atributului, sau ***compuse,*** fiind reprezentate printr-o mulțime de atribute simple ale aceleași entități sau relații, ale căror înțelesuri sau utilizări sunt strâns conectate, ca, de exemplu, atributul **Adresa** (Figura 14.7).

14.3. Cardinalitatea relațiilor și atributelor

Cardinalitatea de participare a entităților într-o relație oarecare este specificată pentru fiecare entitate participantă la relație. **Cardinalitatea asocierii** exprimă numărul **minim** și **maxim** de apariții ale unei entități cu cealaltă entitate asociată. Într-o diagramă E-R cardinalitățile minime și maxime ale participărilor entităților în relații sunt specificate în paranteze rotunde în ambele părți ale simbolului grafic al relației. Este posibil să fie atribuită cardinalității unei relații orice valoare întreagă egală sau mai mare ca 0, singura cerință fiind ca valoarea minimă a cardinalității să fie mai mică decât cea maximă (Figura 14.8). În majoritatea cazurilor este suficientă utilizarea a trei valori: **0, 1 sau N** (N este o valoare întreagă mai mare ca 1 cu semnificația funcțională:

- *cardinalitatea minima 0 – participarea la relație a entității este **opțională**;*
- *cardinalitatea minima 1 – participarea la relație a entității este **obligatorie**;*
- *cardinalitatea maxima 1 – entitatea participă cu o **singură apariție distinctă** în relația de asociere cu o altă entitate;*
- *cardinalitatea maxima N – entitatea participă cu **mai multe apariții distincte** în relația de asociere cu o altă entitate.*

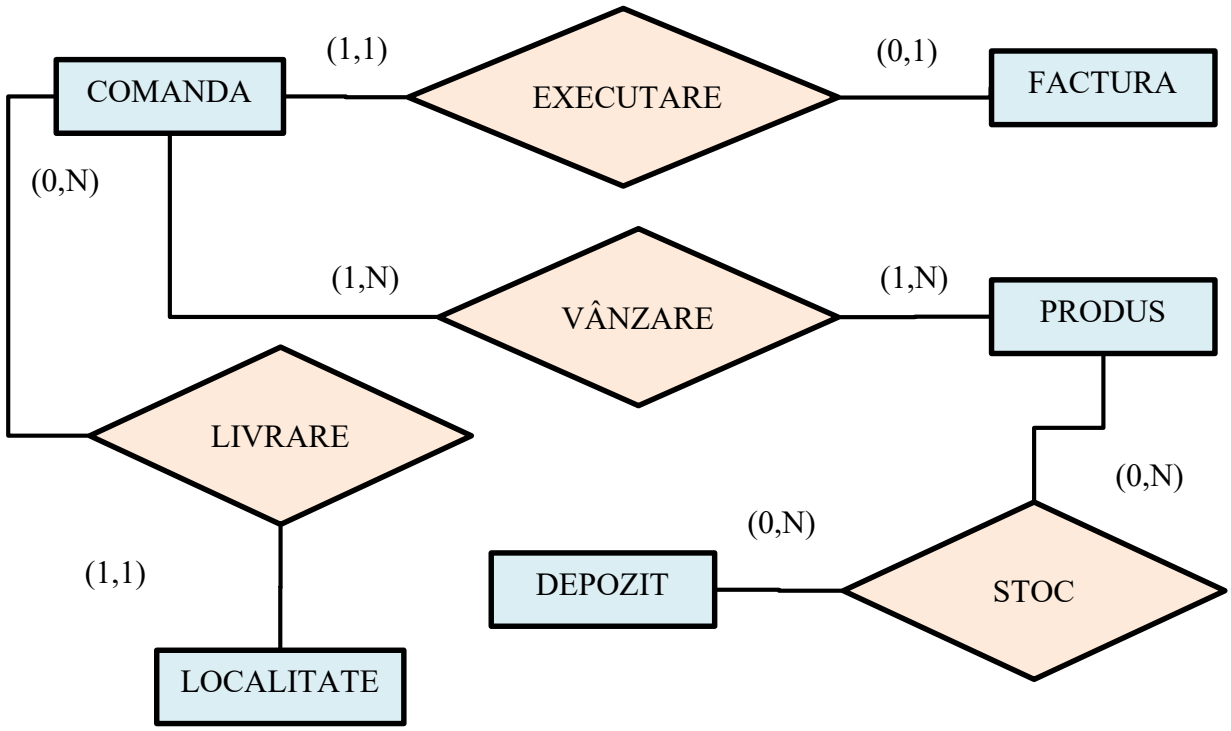


Figura 14.8. Reprezentarea cardinalității relațiilor dintre entități

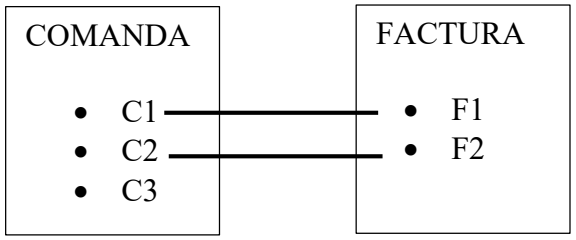
În exemplu din Figura 14.8 cardinalitatea (1,1) atribuită entității COMANDA înseamnă că o factură poate fi rezultatul executării a minim unei comenzi (1,...) și a unui număr maxim de tot o comandă (...,1). Cardinalitatea (0,1) atribuită entității FACTURA înseamnă că o comandă poate fi executată prin maxim o factură (...,1) sau prin nici una (0,...) (este cazul când comanda este pregătită pentru executare, dar factura încă nu s-a eliberat). Similar, entitatea COMANDA participă cu cardinalitatea (1,N) în relația VÂNZARE, desemnând faptul că tranzacțiile de vânzare a produselor se realizează în baza minim unei comenzi, dar pot să existe mai multe comenzi pentru același produs. Cardinalitatea (1,N) a entității PRODUS desemnează faptul că orice comandă în mod obligatoriu trebuie să refere cel puțin un produs (1,...), dar poate să includă mai multe produse diferite (...,N).

Dacă în relație ambele entități participă cu cardinalitatea minimă 1, atunci aceste asocieri poartă denumirea de **asocieri totale**. Asocierile totale nu acceptă valoarea **Null** în mulțimea tuplurilor relației. Dacă una sau ambele entități participă în relație cu cardinalitatea minimă 0, atunci aceste asocieri se numesc **asocieri parțiale**. Aparițiile relațiilor cu asocieri parțiale dintre entități pot să conțină în mulțimea tuplurilor returnate tupluri ce conțin valori **Null** (Figura 5.9).

Cardinalitatea de participare a entităților într-o relație se determină din analiza domeniul de obiecte modelat și regulile de afaceri (*business rules*) specifice acestui domeniu, care stabilesc anumite constrângeri de participare a entităților în relații, de exemplu, că o comandă poate solicita vânzarea mai multor produse, dar un produs poate figura în comenzi diferite (Figura 14.9). Determinarea cardinalității opționale se bazează pe faptul dacă relația analizată acceptă valorile lipse (valorile Null) în asocierile dintre aparițiile entităților participante în relație.

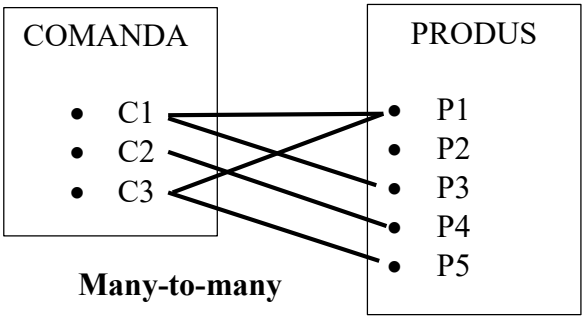
Relațiile dintre entități pot fi caracterizate prin **numărul de entități distincte** care participă la asociere (**relații binare, relații recursive, relații multiple**) și **cardinalitatea relației (asocierii)** (Figura 5.9), care în funcție de cardinalitățile maxime ale entităților implicate în relații, poate fi:

- ***asocierea unu-la-unu (one-to-one)***
- ***asocierea unu-la-mai-multe (one-to-many)***
- ***asocierea mai multe-la-mai-multe (many-to-many)***



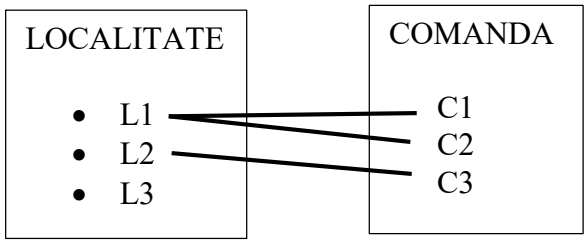
EXECUTARE	
COMANDA	FACTURA
C1	F1
C2	F2
C3	Null

One-to-one



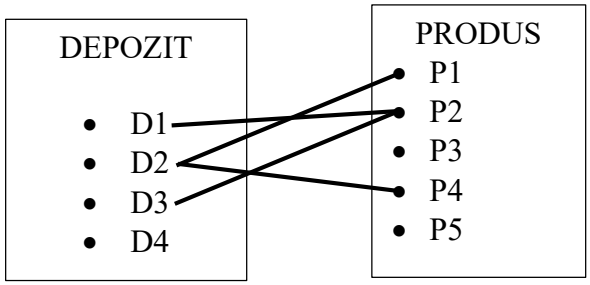
VÂNZARE	
COMANDA	PRODUS
C1	P1
C1	P3
C2	P4
C3	P1
C3	P5

Many-to-many



LIVRARE	
LOCALITATE	COMANDA
L1	C1
L1	C2
L2	C3
L3	Null

One-to-many



STOC	
DEPOZIT	PRODUS
D1	P2
D2	P1
D2	P4
D3	P2
D4	Null
Null	P3
Null	P5

Many-to-many

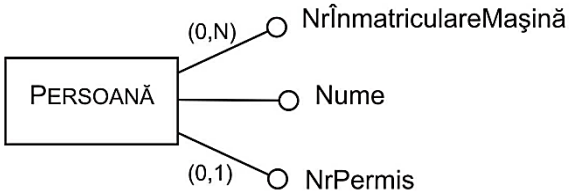
Figura 14.9. Tipuri de asocieri dintre entități

Relațiile **one-to-one** definesc o asociere unu la unu între entități. Relațiile **one-to-many** apar atunci, când cardinalitatea maximă a unei entități este 1, iar cardinalitatea maximă a celeilalte entități este N. Relațiile **many-to-many** posedă cardinalități maxime egale cu N pentru ambele entități (Figura 14.8 și Figura 14.9). Sunt rare cazurile în care participarea este obligatorie pentru toate entitățile implicate în relație din motivul că când se adăugă o nouă apariție a unei entități, în general, aparițiile corespunzătoare ale altor entități, legate de aceasta **nu sunt cunoscute** încă sau **nu există.** De exemplu, când se primește o nouă comandă, nu există încă o factură eliberată de contabilitate și, prin urmare, nu este posibilă construirea unei apariții pentru relația EXECUTARE care conține noua comandă (Figura 14.9). În relațiile N-are, entitățile implicate au aproape întotdeauna cardinalitatea maximă egală cu N.

Cardinalitatea atributelor descrie **numărul minim, respectiv maxim de valori ale atributelor asociate** fiecărei apariții a unei entități sau relații. În general, cardinalitatea unui atribut

este (1,1) și este omisă pe schema E-R. În acest caz atributul este o funcție ce asociază o singură valoare fiecărei apariții a entității sau relației. Totuși, valoarea unor atribute poate fi necunoscută (**Null)** sau pot exista **valori diferite** ale atributului asociate unei apariții a entității. Aceste situații pot fi reprezentate prin alocarea cardinalității minime egală cu 0 în primul caz și, respectiv, cu alocarea cardinalității maxime egală cu N în al doilea caz.

În Figura 14.10 sunt prezentate cardinalitățile alocate entității PERSOANĂ. Atributul **Nume** este un atribut obligatoriu cu o singură apariție pentru o persoană concretă, care poate să posede sau nu permis de conducere a automobilului. În caz de posedare a unui permis el este document unic (nu poți deține mai multe permise naționale conform legislației în vigoare). Permisul de conducere se identifică prin atributul **NrPermis** cu cardinalitatea (0,1), iar o apariție unică a atributului **Nume** poate fi asociată cu o apariție unică a atributului **NrPermis** sau cu o valoare **Null**. O persoană oarecare poate să nu posede în proprietate privată nici un automobil (chiar dacă posedă permis de conducere) sau poate să fie proprietar de mai multe automobile (chiar dacă nu posedă permis de conducere). Din aceste considerente o singură apariție a atributului **Nume** poate fi asociată cu N apariții ale atributului **NrÎnmatriculareMașină**.



PERSOANĂ		
Nume	NrPermis	NrÎnmatriculareMaşină
Albu	988222333	DDT322
Negru	Null	NEG001
Negru	Null	NEG002
Corbu	777555111	Null
Graur	Null	Null

Figura 14.10. Exemple de cardinalitate ale atributelor entității PERSOANĂ

Într-un mod similar participării unei apariții a unei entități într-o relație, putem spune ca:

- un atribut cu cardinalitatea minimă egală cu zero este opțional pentru entitatea asociată (sau relația asociata);
- un atribut este obligatoriu în cazul în care cardinalitatea minimă este unu;
- un atribut posedă mai multe valori dacă el are cardinalitatea maximă N.

Atributele cu cardinalitatea maximă N trebuie folosite cu precauție deoarece ele reprezintă situații care pot fi modelate, câteodată, prin entități adiționale, legate prin relații one-to-one sau many-to-many cu entitățile la care se refera. De exemplu, presupunem că entitatea PERSOANĂ posedă un atribut Calificare cu cardinalitatea (0,N) (o persoana poate să nu aibă nicio calificare sau poate avea mai multe calificări). Calificarea este un concept ce poate fi atribuit mai multor persoane și este naturală modelarea acestui concept cu ajutorul unei entități CALIFICARE, care se asociază cu entitatea PERSOANĂ printr-o relație many-to-many. Așa cum va fi explicat în capitolele următoare, relațiile many-to-many sunt surse de anomalii ale datelor într-o bază de date relațională și se descompun în două relații de tip one-to-many.

14.4. Identificatori interni și externi

Un identificator permite identificarea unică a aparițiilor unei entități în cadrul unei relații. În cadrul unei diagrame E-R sau scheme logice fiecare entitate trebuie în mod obligatoriu să posede un identificator. Identificatorii entităților se clasifică în ***identificatori interni*** și ***identificatori externi***.

Identificatorul intern este format din unul sau mai multe **atribute proprii** ale entității și mai este cunoscut sub numele de **cheie primară**. Identificatorul intern poate fi unul **simplu** (format dintr-un singur atribut) sau unul **compus** (format din mai multe atribute proprii ale entității cu condiția că fiecare dintre ele să aibă cardinalitatea (1,1)) (Figura 14.11).

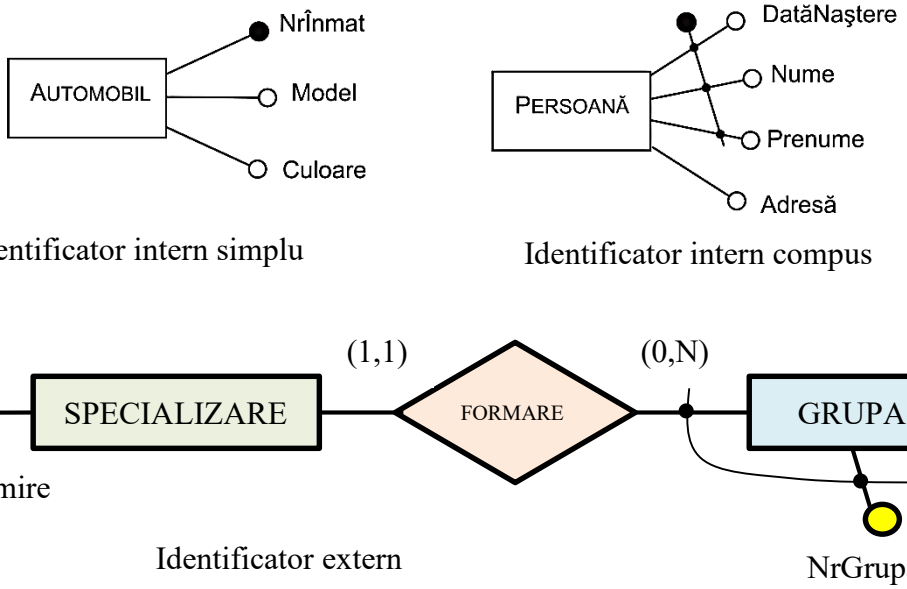


Figura 14.11. Exemple de identificatori ai entităților

Identificatorul extern reprezintă unul sau mai multe atribute ale entități opuse din cadrul relației cu care entitatea dată formează o asociere de tip one-to-many și care impreună cu atributele proprii ale entității formează **cheia primară compusă**, având și rolul suplimentar de **cheie externă**. În Figura 14.11 entitatea GRUPA utilizează în calitate de identificator intern atribut NrGrupa (numărul grupei), care nu posedă capacitatea deplină de identificare într-un singur mod posibil a aparițiilor distincte ale entității GRUPA, deoarece în cadrul unei universități pot fi formate mai multe grupe cu același număr de grupă (de exemplu, 221), dar care se referă la specializări diferite. În acest caz, pentru a putea identifica corect o grupă academică la nivel de universitate este necesară utilizarea unui **identificator extern** (#Denumire) în componența **cheii primare compuse** (#Denumire,#NrGrupa) a entității GRUPA, care face referire la entitatea SPECIALIZARE (de exemplu, TI221, CIB221, BA221, FB221 etc). Utilizarea unui identificator extern în componența cheii primare compuse a entității se indică în diagrama E-R printr-un arc, care unește toate atributele proprii ale entității din componența cheii cu linia de participare la relație (Figura 14.11). Un identificator extern poate referi o altă entitate, care la rândul sau este identificată extern, atâta timp cât **nu se generează cicluri.**

O entitate E poate fi identificată în mod unic de o altă entitate W doar dacă ambele entități sunt implicate într-o relație de asociere în care E participă cu cardinalitatea (1,1).

14.5. Generalizări și ierarhii

O generalizare reprezintă legăturile logice existente între o entitate părinte (EP) și una sau mai multe entități copii (EC_1, EC_2, ... , EC_n). **Entitatea EP** este mai generală, în sensul că EC_1, ..., EC_n sunt **cazuri particulare** ale entității EP. De exemplu, entitatea PERSOANĂ este o generalizare pentru entitățile BĂRBAT și FEMEIE (Figura 14.12).

Generalizările se caracterizează prin următoarele proprietăți:

- fiecare apariție a unei entități copil este o apariție a entității părinte;
- fiecare proprietate a entității părinte (atribut, identificator, relație, generalizare) este, de asemenea, o proprietate a entității copil.

De exemplu: daca entitatea PERSOANĂ are atributele Nume și Vârsta, atunci entitățile BARBAT și FEMEIE au, de asemenea, aceste atribute. Mai mult decât atâta, identificatorul CNP pentru entitatea PERSOANĂ este și un identificator valid pentru entitățile BARBAT și FEMEIE. Această proprietate se numește ***moștenire a proprietăților*** (Figura 14.12).

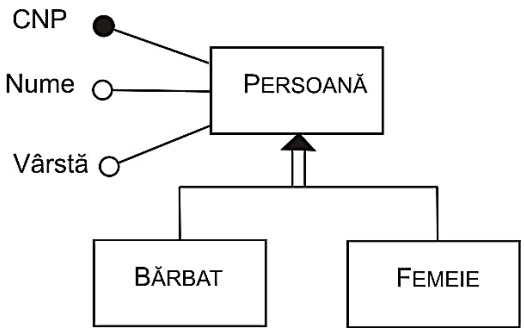


Figura 14.12. Moștenirea proprietăților prin generalizare

Generalizările pot fi **totale** sau **parțiale**, **exclusive** sau **suprapuse**. Generalizarea **totală** stabilește că fiecare apariție a entității părinte este, de asemenea, o apariție a unei entități copil. În caz contrar generalizarea este **parțială**. Generalizarea **exclusivă** stabilește că fiecare apariție a entității părinte este **cel mult o apariție** a unei entități copil distincte. În caz contrar generalizarea este suprapusă.

De exemplu, generalizarea PERSOANĂ pentru entitățile BARBAT și FEMEIE este **totală,** deoarece persoanele pot fi numai bărbați sau numai femei, și concomitent **exclusivă,** deoarece o persoană este fie bărbat, fie femeie. În același timp, generalizarea VEHICUL pentru entitățile AUTOMOBIL și BICICLETĂ este o generalizare **parțială** (exista și alte tipuri de vehicule) și **exclusiva** (ori e automobil, ori e bicicletă). Generalizarea PERSOANĂ pentru entitățile STUDENT și ANGAJAT este **parțială** și **suprapusă** (există studenți care sunt și angajați). Generalizarea **suprapusă** poate fi transformată ușor într-o generalizare **exclusivă** prin adăugarea unei sau mai multor **entități copil** pentru a reprezenta entitățile ce se suprapun. În ultimul exemplu pentru generalizarea PERSOANĂ se poate adaugă entitatea nouă STUDENTANGAJAT pentru a obține o generalizare **exclusiva**.

În general, o entitate poate fi implicată în **mai multe generalizări** diferite, care formează structuri ierarhice cu mai multe nivele. Generalizările cu mai multe nivele se numesc ***ierarhii*** (Figura 14.13). În diagrame E-R generalizările exclusive se indică cu simbolul generalizării în care săgeata este total hașurată, iar generalizările suprapuse se reprezintă cu simbolul generalizări în care se reprezintă doar conturul săgeții (Figura 14.13). Dacă o generalizare are doar o singură entitate copil, atunci entitatea copil poartă numele de ***submulțime*** a entității părinte. În Figura 14.13 entitatea MANAGERPROIECT este o submulțime a entității ANALIST.

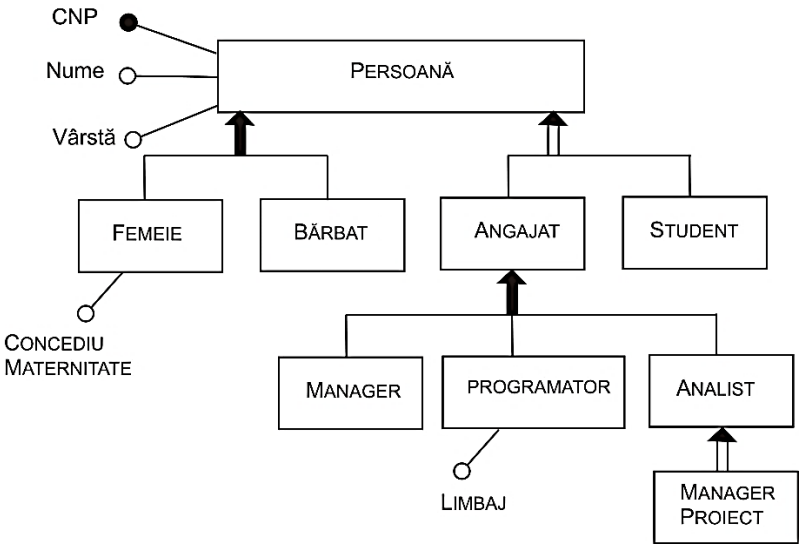


Figura 14.13. Generalizări și ierarhii

14.6. Documentația pentru diagramele E-R

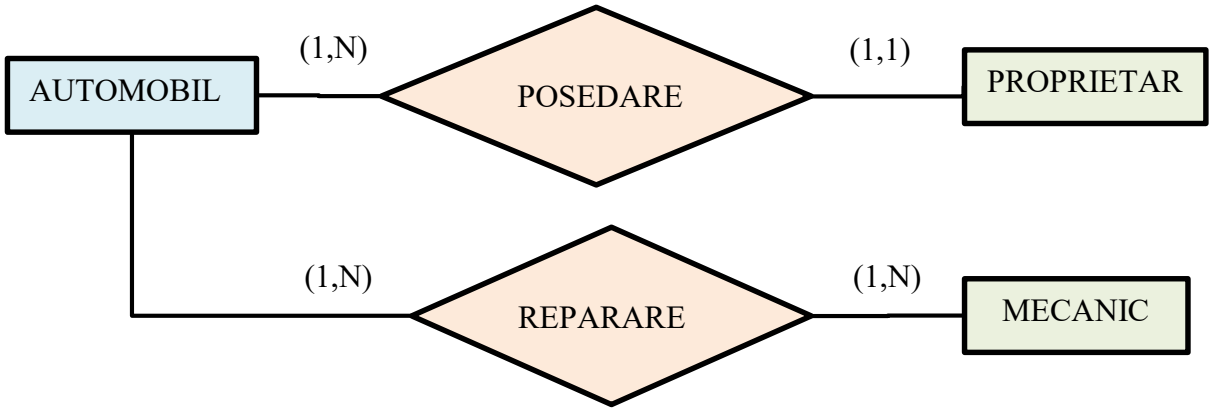
O diagramă E-R în majoritatea cazurilor este insuficientă pentru a descrie toate aspectele unei baze de date în detaliu. Într-o diagramă E-R sunt precizate doar numele conceptelor, iar acest fapt este **insuficient** pentru a explica semnificația acestor concepte. Spre exemplu, dintr-o diagramă E-R care conține entitatea PROIECT nu este clar la ce tip de proiect se referă entitatea: la un proiect **intern** al companiei sau la un proiect **extern** la care participă compania. Mai mult decât atâta, dacă diagrama E-R este una **complexă**, se poate întâmpla să nu fie reprezentate toate proprietățile conceptelor într-un mod inteligibil. Adăugarea în diagrama E-R a tuturor atributelor entităților reprezentate ar reduce semnificativ accesibilitatea și lizibilitatea diagramei.

Pe de altă parte, este imposibilă reprezentarea unor proprietăți ale datelor prin intermediul diagramelor E-R. Nu pot fi reprezentate, de exemplu, următoarele constrângeri de integritate ale entităților: 1) un angajat poate fi manager doar în departamentul de care aparține; 2) un angajat nu poate avea un salariu mai mare decât managerul departamentului de care aparține. În calitate de concluzie menționăm că modelul E-R **nu furnizează mijloace potrivite** pentru reprezentarea **constrângerilor complexe** impuse datelor. Din motivele precizate mai sus, o diagramă E-R trebuie completată cu o **documentație** care facilitează **interpretarea** diagramei și permite **descrierea proprietăților** care nu pot fi exprimate direct prin construcțiile puse la dispoziție de modelul E-R.

O diagrama E-R în prima sa formă, neprelucrată, conține o serie de **relații**, pe care proiectantul le-a identificat pe baza **întrebărilor posibile ale utilizatorilor.** Este util să stabilim **toate interogările posibile** ale bazei de date înainte de a trece la optimizarea diagramei entitate-relație. Este posibil să **nu fie identificate toate entitățile** sau **atributele** necesare pentru realizarea unor interogări. Care sunt informațiile pe care vrem sa le furnizeze baza de date? Aceasta este întrebarea de la care se pleacă în cazul analizei relațiilor identificate. Ce elemente lipsesc din modelul de date construit? Este a doua întrebare la care trebuie să răspundem, analizând entitățile și proprietățile lor. Acestea pot să treacă ușor neobservate dacă nu există o comunicare permanentă între proiectanți și reprezentanții beneficiarilor bazei de date.

Atunci când se realizează analiza relațiilor dintre entitățile domeniul de obiecte modelat este util de reprezentat diagrama E-R **fără atributele** entităților, mai ales dacă numărul de entități și atribute cu care lucrăm este foarte mare. Analiza relațiilor dintre entități se realizează prin urmărirea pas-cu-pas a **interogărilor utilizatorilor** la care vrem să răspundă baza de date sau, în mod sistematic, folosind **matricea relațiilor** (Figura 14.14)**. Matricea relațiilor** permite **identificarea** în mod sistematic a tuturor relațiilor **dintre entități**, dacă o citim pe linii și pe coloane, inclusiv și în cazul relațiilor „în buclă”. Matricea relațiilor se elaborează în strictă

corespundere cu relațiile identificate în diagrama E-R. Matricea relațiilor **nu pune în evidență opționalitatea și cardinalitatea** relațiilor. Toate celulele trebuie analizate, linie cu linie, încercând să identificăm dacă sunt prezente toate relațiile necesare pentru elaborarea interogărilor. Nici o celulă nu trebuie să rămână necompletată, dacă între cele două entități a fost identifică o relație. **Nu este obligatoriu** să existe relații între oricare două entități, dar lipsa unei astfel de relații trebuie analizată și stabilită pe **baza regulilor de afaceri** care descriu aplicația (Figura 14.14).



	PROPRIETAR	AUTOMOBIL	MECANIC
PROPRIETAR	=====	posedă	=====
AUTOMOBIL	aparține	=====	este reparat de
MECANIC	=====	repară	=====

Figura 14.14. Diagrama E-R și matricea relațiilor

Regulile de operare sunt uneltele utilizate de analiști de sistem pentru a descrie **proprietățile** unei aplicații, care **nu pot fi exprimate direct** cu ajutorul modelului conceptual. Aceasta abordare permite specificarea regulilor unei aplicații. Exemplu de regulă de operare: ***un angajat nu poate avea un salariu mai mare decât managerul său.***

O regula de operare poate fi o:

- **descriere a unui concept** relevant al aplicației (o **definire precisă** a entităților, atributelor sau relațiilor unui model E-R);
- **constrângere de integritate** aplicată datelor din baza de date;
- **derivare**, ce reprezintă un concept care poate fi obținut pe baza unei deducții sau a unui calcul matematic din alte concepte ale diagramei E-R. De exemplu, atributul **SumaTranzacției** poate fi obținut ca **produs** al atributelor **Preț** și **Cantitate.**

Regulile pentru **descrierea conceptelor** se exprimă, în general, într-un **limbaj natural.** Regulile ce descriu **constrângerile de integritate și derivările** folosesc **definiții formale.** Însă notația de tipul **if** <*condiție*> ***then*** <*acțiune*> nu este potrivită pentru a exprima o regulă. O structură mai potrivită pentru a exprima o regulă de operare sub forma unei aserțiuni poate fi:

<concept> trebuie/nu trebuie <expresie a conceptelor>

unde conceptele corespund sau unui concept al diagramei E-R la care se referă, sau unui concept derivat.

Mai jos sunt prezentate câteva exemple ale regulilor de operare pentru exprimarea **constrângerilor de integritate**:

RO1 *Managerul departamentului trebuie sa aparțină departamentului.*

RO2 *Un angajat nu trebuie sa aibă salariul mai mare decât managerul departamentului căruia îi aparține.*

RO3 *Un departament din filiala Cahul trebuie condus de un angajat cu vechimea muncii în companie mai mare sau egală cu 5 ani.*

Regulile de operare care descriu **derivări** pot fi exprimate prin specificarea operațiilor (matematice sau de alt ordin), care permit obținerea conceptelor derivate. O structură posibilă este următoarea: **<concept> este obținut prin <operații asupra conceptelor>**

De exemplu, dacă entitatea DEPARTAMENT are un atribut derivat **NumărDeAngajați**, atunci se poate introduce o regulă de forma:

RO4 *Numărul angajaților dintr-un departament este obținut prin numărarea angajaților care aparțin departamentului respectiv.*

Documentația pentru conceptele variate reprezentate într-o diagramă E-R poate fi organizată sub forma unui **dicționar de date** (Figura 14.15). Acesta este format din **doua tabele**:

- ✓ **primul tabel** descrie **entitățile din diagrama E-R** (denumirile entităților, definiții informale în limbaj natural, lista tuturor atributelor cu o descriere a acestora și identificatorii posibili);
- ✓ **al doilea tabel** descrie **relațiile din diagrama E-R** (denumirile relațiilor, descrierea lor informală, lista atributelor cu descrieri posibile și lista entităților implicate în relație împreuna cu cardinalitățile de participare la relație).

Entitate	Descriere	Atribute	Identificator
ANGAJAT	Angajaţi care lucrează în companie	CNP, Nume, Prenume, Vârstă	CNP
PROIECT	Proiectele companiei la care lucrează angajaţii	Nume, Buget, DatăLivrare	Nume
DEPARTAMENT	Departamentele filialei companiei	Telefon, Nume	Nume, FILIALĂ
FILIALĂ	Filiala companiei într-un oraş	Oraş, Adresă (Număr, Stradă, CodPoştal)	Oraş

Relaţie	Descriere	Entităţi implicate	Atribute
MANAGEMENT	Asociază un manager cu un departament	Angajat (0,1) Departament (1,1)	
MEMBRU	Asociază un angajat cu un departament	Angajat (0,1) Departament (1,N)	DatăStart
PARTICIPARE	Asociază angajaţii cu proiectele	Angajat (0,N) Proiect (1,N)	DatăStart
COMPOZIŢIE	Asociază un departament cu o filială	Departament (1,1) Filială (1,N)	

Figura 14.15. Dicționarul de date

Dicționarul de date poate fi utilizat și pentru documentarea unor **constrângeri** impuse datelor și **regulilor de operare**. Poate fi elaborat **un tabel suplimentar** în care se înscriu **regulile de operare** organizate după tipul lor (Figura 14.16)**.** Regulile de operare se exprimă în formele precizate mai sus. Este important să fie reprezentate **toate regulile de operare** care descriu **constrângerile** ce **nu sunt exprimate în diagrama E-R**. Poate fi folositoare și reprezentarea regulilor deja prezentate în schemă.

Constrângeri	
RO1	*Managerul departamentului trebuie sa aparţină departamentului.*
RO2	*Un angajat nu trebuie sa aibă salariul mai mare decât managerul departamentului căruia îi aparţine.*
RO3	*Un departament din filiala Cahul trebuie condus de un angajat cu vechimea muncii în companie mai mare sau egală cu 5 ani.*
Derivări	
RO4	*Numărul angajaților dintr-un departament este obținut prin numărarea angajaților care aparțin departamentului respectiv.*

***Fi*gura 14.16. Tabelul cu regulile de operare**

Întrebări de control:

1. Ce este modelul Entitate-Relație (E-R) și care sunt elementele sale esențiale?
2. Care este diferența dintre o entitate puternică (normală) și o entitate slabă (dependentă) și cum sunt reprezentate acestea grafic într-o diagramă E-R?
3. Cum se definesc aparițiile unei relații?
4. Ce sunt relațiile recursive?
5. Ce reprezintă cardinalitatea unei relații, ce semnificație au valorile 0, 1 și N pentru limitele minime/maxime și prin ce se deosebesc asocierile totale de cele parțiale?
6. Care este diferența dintre atributele simple și cele compuse?
7. Cum se reprezintă în diagrame E-R caracterul obligatoriu, opțional sau multivaloric al atributului?
8. Prin ce se deosebește un identificator intern (simplu sau compus) de un identificator extern și în ce condiții o entitate poate fi identificată extern?
9. Ce reprezintă o generalizare într-o diagramă E-R, ce este moștenirea proprietăților și cum se clasifică generalizările?
10. Din ce cauze o diagramă E-R este insuficientă pentru descrierea completă a unei baze de date și de ce este necesară o documentație explicativă?
11. Ce rol are matricea relațiilor în analiza și optimizarea unei diagrame E-R și ce informații furnizează aceasta?
12. Ce sunt regulile de operare, ce structuri sintactice se folosesc pentru exprimarea lor?
13. Din ce tabele este compus un dicționar de date?

Capitolul 15. PROIECTAREA LOGICĂ A BAZEI DE DATE

15.1. Reprezentarea entităților în dreptunghi

Există câteva modalități de reprezentare a unei diagrame Entitate-Relație. Prima modalitate a fost propusă de către Chen și utilizează construcțiile reprezentate în Figura 15.1 și Figura 15.2.

În reprezentarea lui Chen entitățile din diagramele E-R se reprezintă în dreptunghiuri în care se specifică numele lor la singular, iar asocierile existente între aceste entități se reprezintă prin romburi (Figura 15.1). Se identifică grafic entitățile puternice și entitățile slabe, relațiile la care ele participă și toate atributele care la caracterizează. Atributul se reprezintă prin simbolul unui oval în care se scrie numele său la singular. Numele atributului care participă la formarea identificatorului entității puternice se subliniază cu linie continuă, iar a entității slabe – cu linie întreruptă. Atributele multivaloare (compuse) se specifică în ovale cu contur dublu, iar atributele derivate (obținute prin calcule) se specifică în ovale cu contur întrerupt.

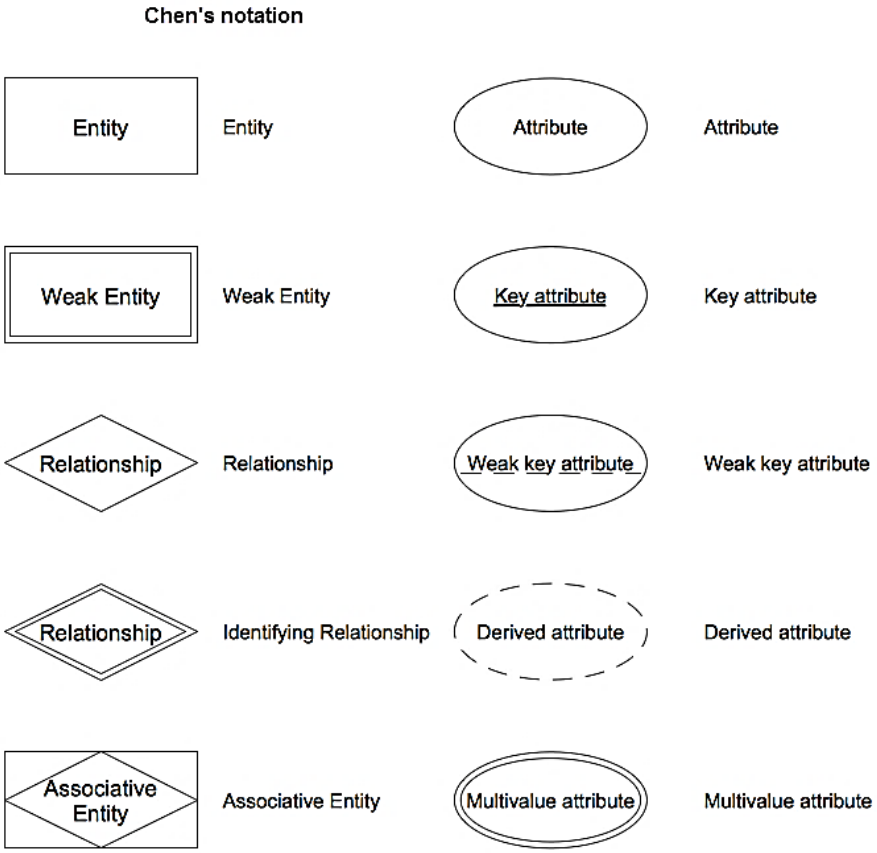


Figura 15.1. Reprezentarea entităților și atributelor în notarea lui Chen [1]

Cardinalitatea relațiilor în diagramele Chen se specifică cu ajutorul cardinalităților minimale și maximale înscrise în paranteze rotunde **din ambele părți** ale simbolului relației. Dacă ele nu se specificau în paranteze, atunci cardinalitatea se reprezenta printr-o linie continuă (pentru participare obligatorie) sau cu ajutorul liniei întrerupte (pentru participare opțională). În partea stângă a liniei se specifica cardinalitatea minimă, iar în partea dreaptă se indica cardinalitatea maximă. (Figura 15.2). Dacă cardinalitatea minimă nu e specificată, atunci participarea entității în relație este opțională. Relațiile recursive se specifică cu linie dublă (Figura 15.2).

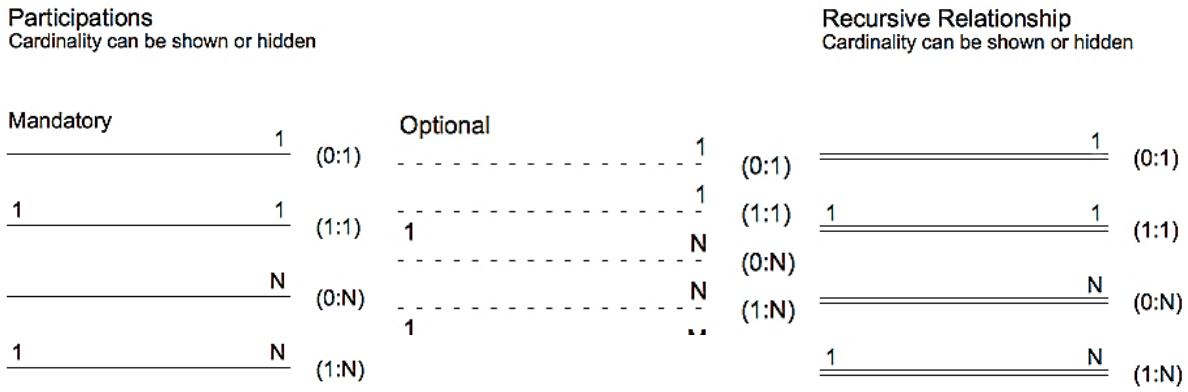


Figura 15.2. Reprezentarea cardinalității relațiilor în notarea lui Chen [1]

Dacă în cadrul diagramelor Chen se reprezentau un număr mare de entități, fiecare având un număr mare de atribute, atunci reprezentarea grafică a diagramelor E-R devenea una greoaie și dificil de urmărit și analizat. În acest caz se preferă convenția de reprezentare Crow's Foot a entităților, în care atributele unei entități sunt înscrise toate în dreptunghiul asociat acesteia și se aplică următoarele reguli:

- O entitate se reprezintă printr-un dreptunghi (*softbox*), în care se scrie numele entității. Numele entității este întotdeauna un substantiv la singular și se scrie în partea de sus a dreptunghiului cu majuscule (Figura 15.3).
- O entitate reprezintă o clasă de obiecte și pentru orice entitate există mai multe ***instanțe ale sale. O instanță*** a unei entități este un obiect, persoană, eveniment particular din clasa de obiecte care formează entitatea.
- Atributele entității se scriu unul sub altul în același dreptunghi, sub numele entității, cu litere minuscule (Figura 15.4).
- Cheia primară simplă sau compusă a entității se scrie prima în lista de atribute și se marchează cu semnul numărului (#) sau cu PK (Primary Key) în dreptul fiecărui atribut al cheii.
- Un atribut obligatoriu al entității **se marchează cu un „asterisc" (*).**
- Un atribut cu valoare opțională **se marchează cu un „ cerculeț" *(o).***

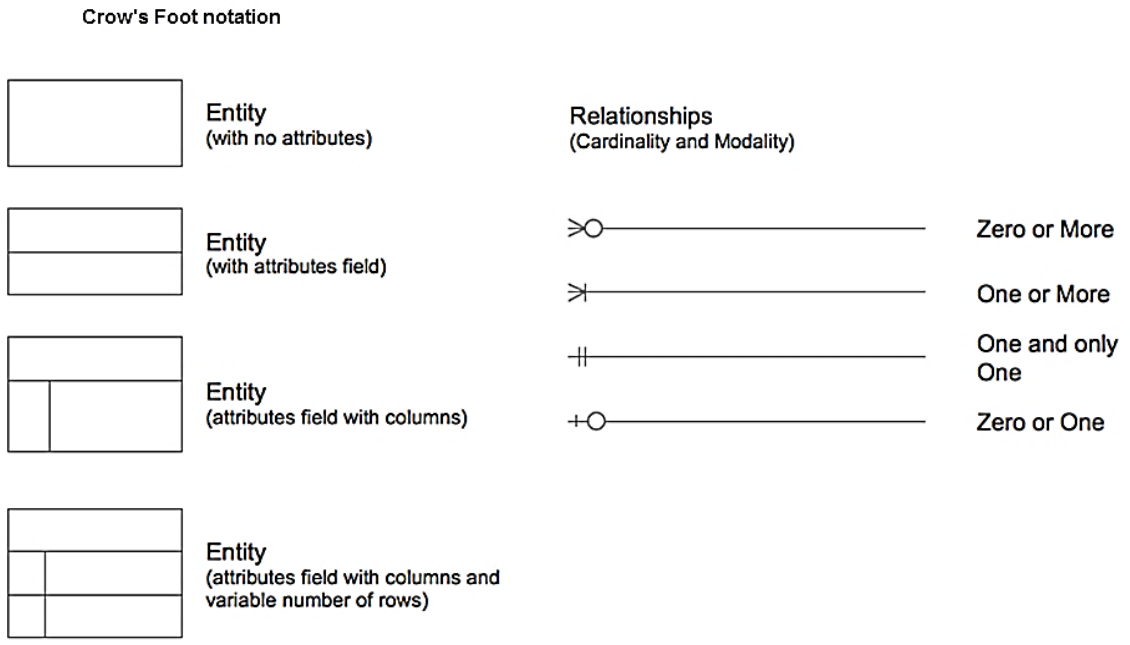


Figura 15.3. Reprezentarea entităților și cardinalității relațiilor în notarea Crow's Foot [1]

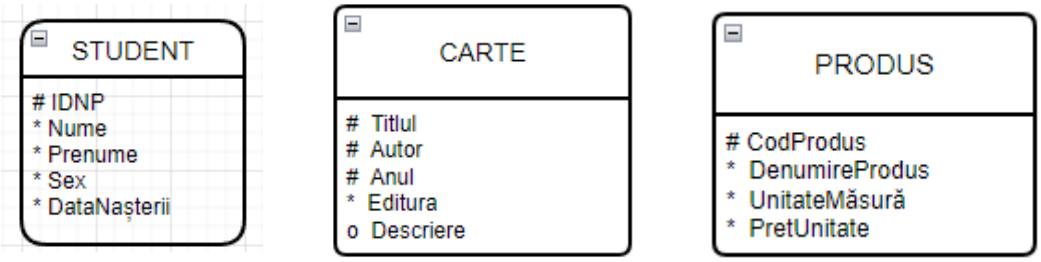


Figura 15.4. Reprezentarea entităților în dreptunghi

Un **subtip de entitate** este o clasificare a unei entități care are **caracteristici comune cu entitatea generală,** precum atribute și relații. Subtipurile se reprezintă în cadrul hărții relațiilor ca entități în interiorul altei entități. Atributele și relațiile comune tuturor subtipurilor se vor reprezenta la nivelul **supertipului (entității generale). Atributele și relațiile** supertipului vor fi moștenite de către subtipuri. Un subtip poate avea la rândul sau alte subtipuri incluse (Figura 15.5).

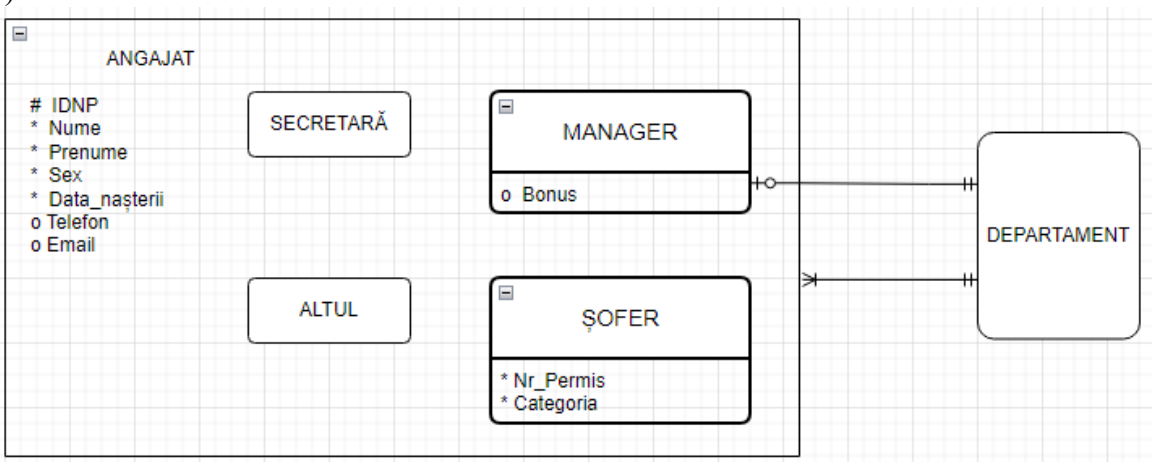


Figura 15.5. Reprezentarea subtipurilor de entități

Subtipurile trebuie să respecte două reguli importante:

- Subtipurile trebuie să reflecte **toate cazurile posibile** de instanțe ale supertipului. Orice instanță a supertipului trebuie să aparțină unui subtip. Pentru a asigura acest lucru diagramele E-R în dreptunghi conțin un subtip ALTUL (Figura 15.5) pentru a acoperi toate situațiile și pentru a permite viitoarele dezvoltări ale modelului.
- Subtipurile trebuie să se **excludă** reciproc. Această regulă se traduce pe exemplul din Figura 15.5 în faptul că un angajat nu poate fi, de exemplu, și manager, și secretară în același timp.

15.2. Reprezentarea relațiilor în notarea Crow's Foot

În notarea Crow's Foot se utilizează trei simboluri pentru a reprezenta cardinalitatea relațiilor la care participă entitatea:

Un inel reprezintă cardinalitatea "zero"	○
O linie verticală reprezintă cardinalitatea "unu"	|
Crow's foot („talpa corbului") reprezintă cardinalitatea "mulți" sau "infinit"	⋞

Aceste simboluri sunt utilizate în perechi pentru a reprezenta cele patru tipuri de cardinalitate pe care le poate avea o entitate într-o relație. Elementul **interior** al notației reprezintă valoarea **minimă**, iar elementul **exterior** (cel mai apropiat de entitate) reprezintă valoarea **maximă** (Figura 15.3). Folosind aceste simboluri pot fi indicate toate tipurile de asocieri posibile dintre entitățile diagramei E-R (Figura 15.6).

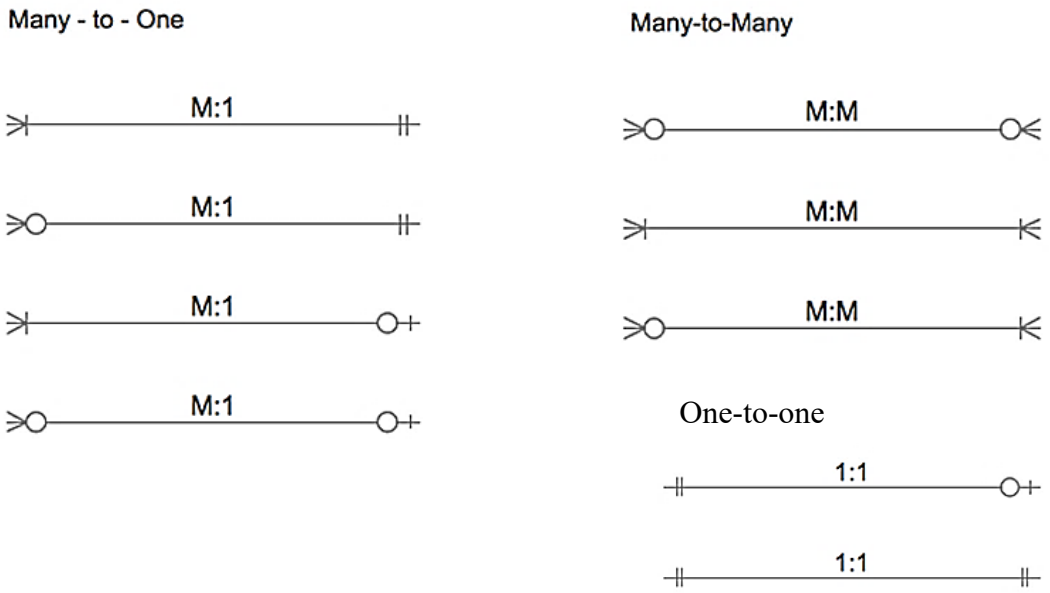


Figura 15.6. Reprezentarea cardinalității relațiilor în notarea Crow's Foot [1]

Relația dintre două entități se reprezintă printr-o linie, etichetată cu numele relației (Figura 15.7). În notarea Crow's Foot linia de legătură dintre entități este **continuă** (Figura 15.6)**.** Participarea **totală (obligatorie)** a entității în relație se indică prin bararea relației cu bara internă (care specifică cardinalitatea minimă 1), iar participarea **parțială** (**opțională**) a entității în relație se indică cu simbolul cercului (care specifică cardinalitatea minimă 0). Simbolurile externe (cele care sunt mai aproape de entitate) specifică cardinalitatea maximă a participării entităților. Dacă se utilizează o bară, atunci cardinalitatea maximală este 1. Dacă se utilizează un capăt ramificat în trei („talpă de corb"), atunci entitatea are o **participare multiplă** la această relație (cardinalitatea maximală este N).

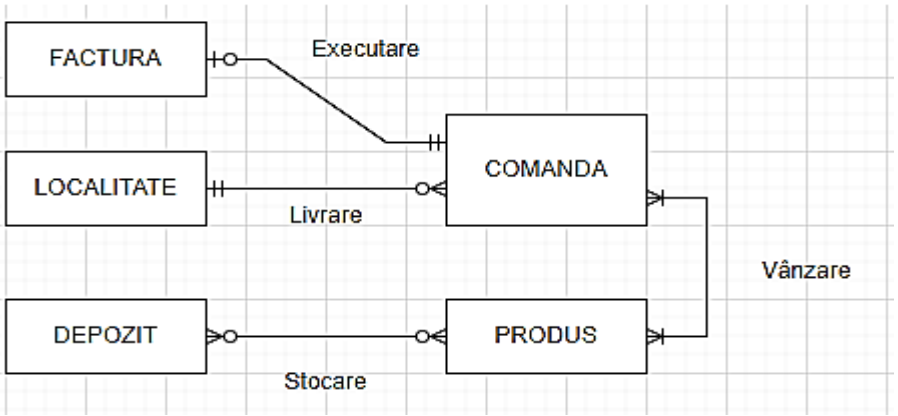


Figura 15.7. Diagrama E-R din Figura 14.8 în notarea Crow's Foot

Să examinăm în continuare diferite tipuri de asocieri ale entităților și reprezentarea lor în diagrame E-R în dreptunghi. Relațiile **one-to-one** sunt relații ideale pentru modelarea unei baze de date relaționale, dar acest tip de relație este destul de rar întâlnit în domeniul de obiecte modelat (Figura 15.8). Uneori astfel de relații pot fi modelate transformând una dintre entități într-un atribut al celeilalte entități.

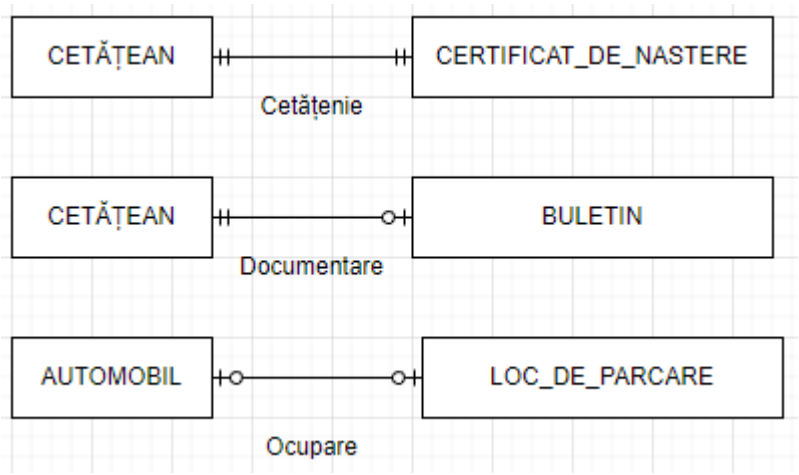


Figura 15.8. Relațiile one-to-one în notarea Crow's Foot

Relațiile **one-to-many** sunt cel mai des întâlnite tipuri de relații care apar în rezultatul modelării bazei de date (Figura 15.9). Relațiile **many-to-many** apar în prima fază a proiectării bazei de date (Figura 15.10), însă ele trebuie sa fie ulterior eliminate din modelul logic al bazei de date.

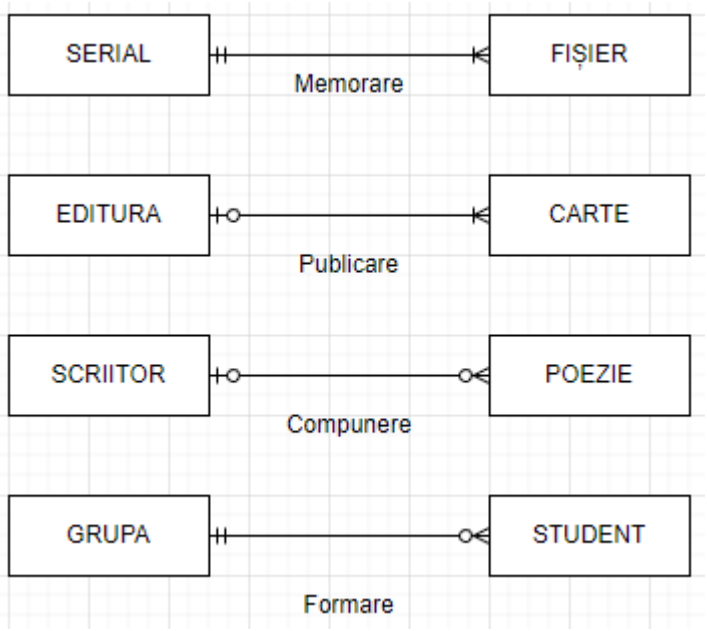


Figura 15.9. Relațiile one-to-many în notarea Crow's Foot

Spunem ca o relație este una **non-transferabilă** dacă o asociere între două instanțe ale celor două entități, odată stabilită, nu mai poate fi modificată. Non-transferabilitatea unei relații se reduce la faptul ca valorile cheii străine corespunzătoare relației respective ***nu pot fi modificate***.

Condiția de non-transferabilitate a unei relații este asigurată prin codul de program. De aceea trebuie să documentăm această restricție în documentația de proiect. În diagrame E-R o **relație non-transferabilă** se notează cu un romb pe linia corespunzătoare relației (Figura 15.11), înspre entitatea a cărei cheie străină nu este permis să o modificăm (adică în partea cu **many** a unei relații one-to-many). În cazul nostru este normal ca o notă dată unui student să nu poată fi apoi transferată unui alt student (Figura 15.11).

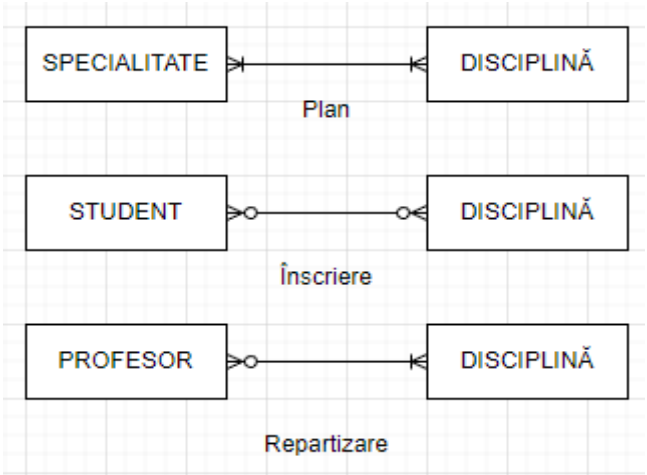


Figura 15.10. Relațiile many-to-many în notarea Crow's Foot

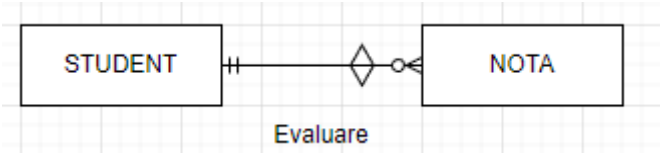


Figura 15.11. Reprezentarea unei relații non-transferabile

În unele situații, relațiile se pot exclude reciproc, adică dintr-un grup de relații, la un moment dat doar una dintre ele poate avea loc. De exemplu, un cont anume la o bancă este deținut fie de o persoana fizică, fie de o firmă, dar nu de ambele tipuri de clienți simultan (Figura 15.12).

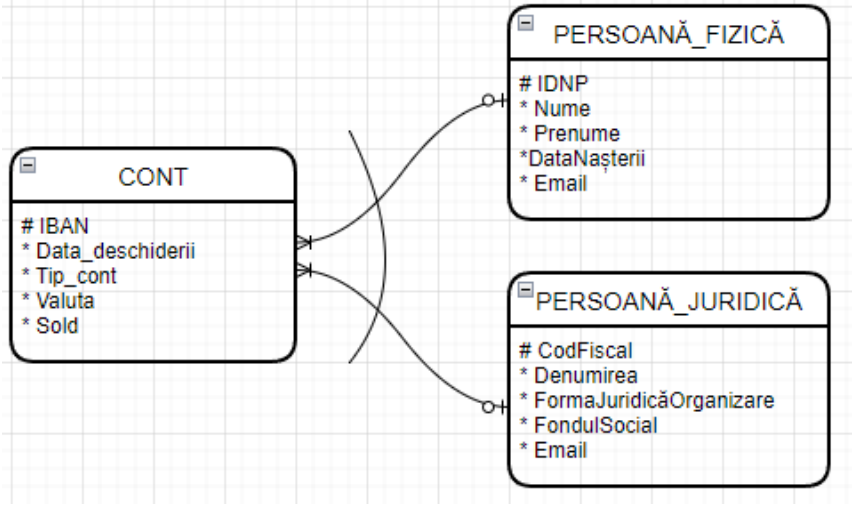


Figura 15.12. Reprezentarea unei relații exclusive obligatorie

Un grup de **relații exclusive** este reprezentat în harta relațiilor printr-un **arc peste relațiile** care fac parte din respectivul grup. Toate relațiile ce fac parte din grupul de relații exclusive trebuie sa aibă **aceeași opționalitate**. Un arc aparține unei **singure entități**, adică va include doar relații care pleacă de la aceeași entitate. O entitate poate avea mai multe arce, dar o anumită relație nu poate face parte decât dintr-un singur arc. Exista doua tipuri de relații exclusive:

- relații exclusive ***obligatorii*** în care toate relațiile ce fac parte din arcul respectiv sunt ***obligatorii,*** ceea ce înseamnă că de fiecare dată, una dintre relații are obligatoriu loc (Figura 15.12);
- relații exclusive **opționale** în care toate relațiile ce fac parte din arc sunt ***opționale.*** În acest caz de fiecare dată are loc cel mult una dintre relații, existând varianta că pentru o instanță a entității căreia aparține arcul sa nu aibă loc nici una din relațiile din grupul respectiv (Figura 15.13).

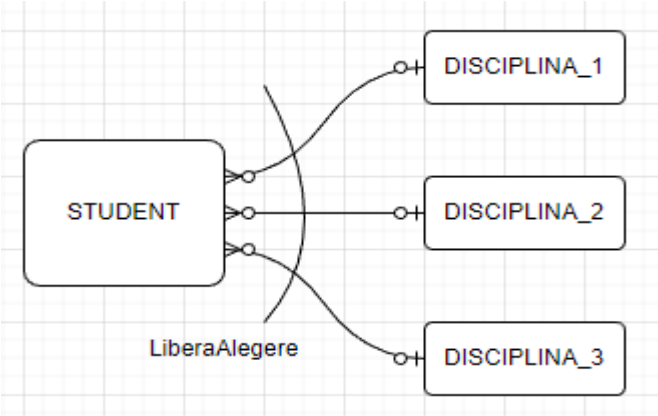


Figura 15.13. Reprezentarea unei relații exclusive opționale

O altă categorie de relații o reprezintă **relațiile ierarhice**. O relație ierarhică se formează în cadrul relațiilor **one-to-many** imbricate în cazurile când pentru identificarea unei entități este necesară utilizarea unui identificator extern în cadrul cheii compuse a entității. Un bun exemplu este acela al localizării camerelor din căminele dintr-un campus studențesc, prin numărul de cămin, prin numărul etajului și numărul camerei. Această identificare reprezintă o ierarhie pe **trei nivele** (cămin, etaj, cameră) (Figura 15.14).

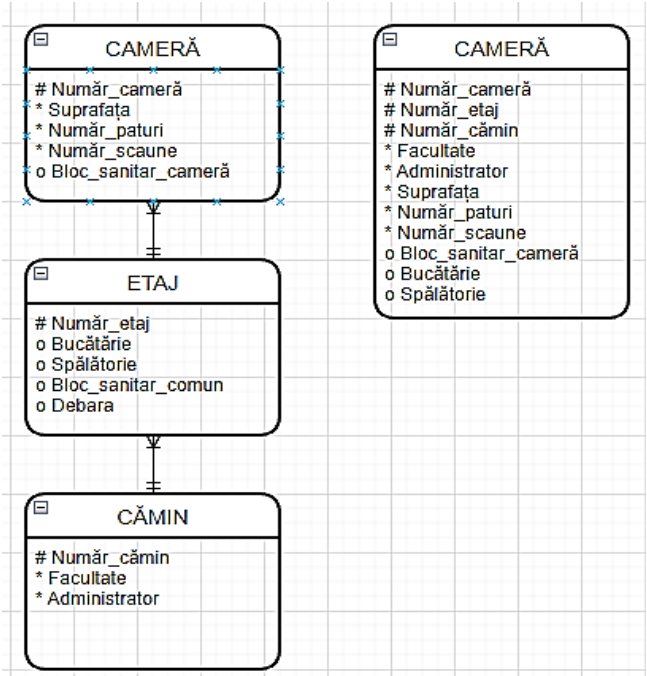


Figura 15.14. Relații ierarhice între entități

Se observa că identificarea unei camere din campus nu se face doar prin numărul camerei, ci prin toate cele trei chei primare ale entităților implicate în ierarhie: *Număr_cămin, Număr_etaj, Număr_cameră.* Acest lucru necesită participarea **obligatorie** a entităților în relațiile din ierarhie, fapt reprezentat prin **relații barate (**cu cardinalitatea minimă 1**)** între entitățile din ierarhie. Relația barată în cadrul relațiilor ierarhice specifică faptul că identificatorul intern al entității ierarhic superioare este parte componentă a cheii primare compuse a entității subordonate, având și rolul de identificator extern (Foreign Key).

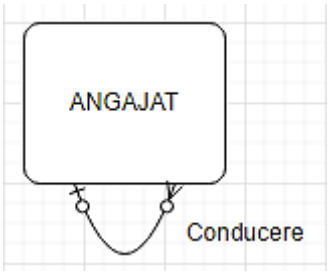


Figura 15.15. Reprezentarea unei relații recursive

În Figura 15.5 a fost prezentat un supertip de entitate ANGAJAT cu atributele comune celor patru subtipuri, care modelează situația în cadrul unui organizații comerciale. Însă fiecare angajat din cadrul unei organizații poate fi condus de un manager, care este o instanță a entității ANGAJAT. Astfel, apare o relație de la entitatea ANGAJAT la ea însăți, adică o **relație recursivă** (Figura 15.15).

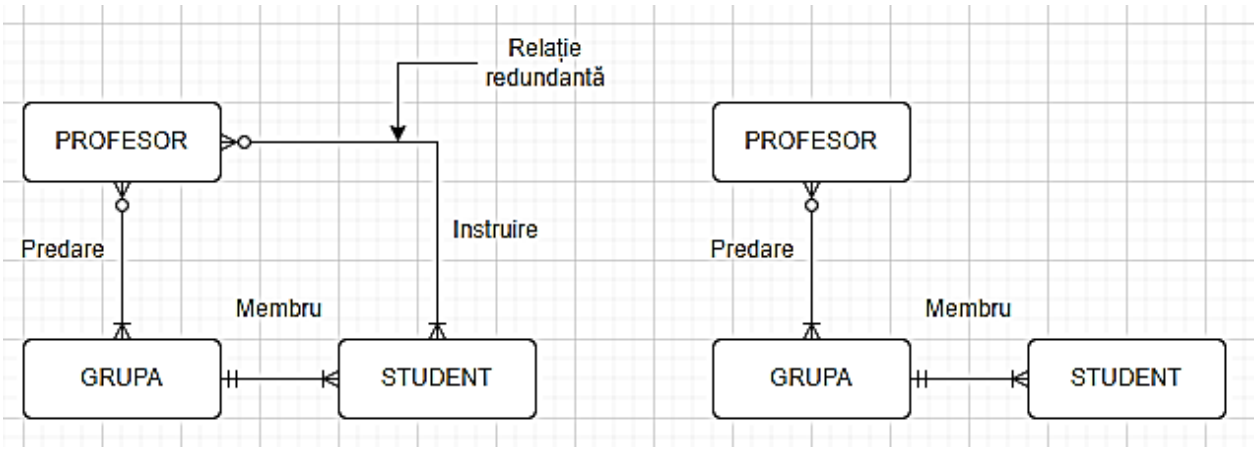


Figura 15.16. Eliminarea unei relații redundante

Atunci când o relație poate fi dedusă din alte relații se spune că această relație este **redundantă**. În cazul relațiilor din Figura 15.16 se observă că un student face parte dintr-o grupă academică, iar la această grupă predau mai mulți profesori. Prin urmare, relația *Instruire* dintre entitățile PROFESOR și STUDENT nu mai este necesară, deoarece putem deduce profesorii care îi predau unui student, aflând profesorii care predau în grupa din care face parte studentul. Relația *Instruire* este redundantă și poate fi eliminată din diagrama E-R (Figura 15.16).

15.3. Capcane de conectare și rezolvarea relațiilor many-to-many

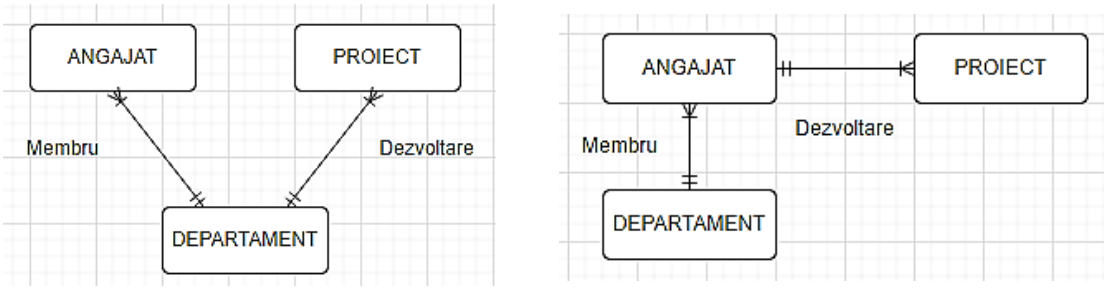
În cazul în care se analizează relațiile dintre entități din diagrama E-R trebuie să fie luate în considerație următoarele **reguli:**

- **Regula 1.** ***În diagrama finală E-R se păstrează cu prioritate relațiile de tip one-to-one.***
- **Regula 2.** ***În diagrama finală E-R nu se admit relații many-to-many, deoarece crește riscul erorilor de actualizare a datelor din baza de date.***
- **Regula 3.** ***În diagrama finală E-R nu se admit relații one-to-many care creează capcane de conectare de tip evantai.***
- **Regula 4**. ***În diagrama finală E-R se pot păstra unele relații redundante, dacă prin aceasta se elimină riscul apariției unei capcane de conectare de tip întrerupere.***

În proiectarea modelului conceptual de date, pot să apară ambiguități de interpretare a relațiilor în cadrul diagramelor E-R, care pot să conducă la imposibilitatea soluționării unor

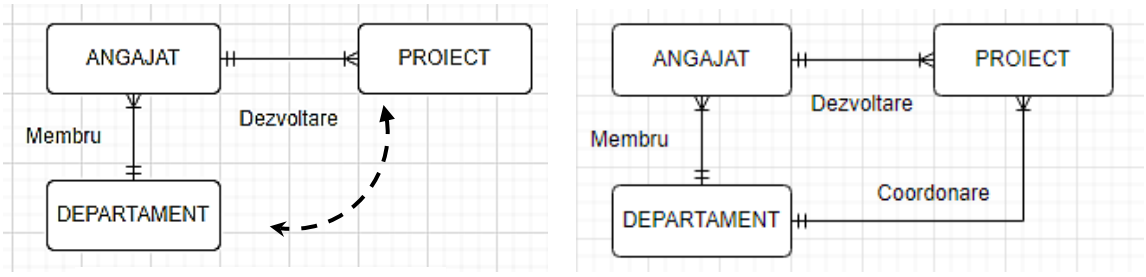
interogări ale utilizatorilor asupra bazei de date. Aceste probleme ale modelului E-R se numesc **capcane de conectare.** Capcanele de conectare sunt de două tipuri:

- **Capcanele de tip evantai** apar atunci, când aceeași entitate este implicată în mai multe relații de tip **one-to-many** și relațiile dintre entități devin ambigue. De exemplu, din diagrama E-R reprezentată în Figura 15.17 nu putem deduce exact în care proiecte este implicat un anumit angajat. Prin modificarea ordinii de reprezentare a entităților implicate în aceste relații se obține o structura de tip „arbore" prin care se elimină ambiguitatea diagramei E-R.
- **Capcanele de tip întrerupere** sunt cauzate de **absența** reprezentării unor relații în modelul E-R, care sunt necesare pentru elaborarea interogărilor utilizatorilor. În diagrama E-R din Figura 15.17, în care a fost eliminată capcana de tip evantai, lipsește relația directă dintre entitățile DEPARTAMENT și PROIECT, necesară pentru identificarea departamentului care se ocupă de dezvoltarea și coordonarea unui proiect concret. Introducând relația nouă *Coordonare*, se creează o anumită **redundantă** (Figura 15.18), care este mai preferabilă în unele situații, decât lipsa unor relații directe dintre entități și, ca urmare, necesitatea de a scrie interogări complexe asupra bazei de date (se identifică mai întâi angajatul responsabil de proiect, iar apoi se identifică departamentul în care lucrează acest angajat).



Capcană de tip evantai — Eliminarea capcanei

Figura 15.17. Eliminarea capcanei de tip evantai



Capcană de tip întrerupere — Eliminarea capcanei

Figura 15.18. Eliminarea capcanei de tip întrerupere

Relațiile **many-to-many** permanent apar în prima variantă a schemei conceptuale a bazei de date, însă ele trebuie să fie eliminate din schema conceptuală finală. Să examinăm relația dintre entitățile COMANDA și PRODUS (Figura 15.19). Un cumpărător poate comanda mai multe produse într-o singură comandă, iar un produs poate fi solicitat în mai multe comenzi în cantități diferite. Mai mult decât atâta, pot fi comandate cantități diferite pentru produse diferite în cadrul aceleași comenzi. În acest caz, în care listă de atribute trebui plasat atributul Cantitate? Dacă înscriem atributul Cantitate în lista atributelor entității COMANDA**,** atunci nu vom putea specifica cantități diferite pentru produse diferite, deoarece unei instanțe a entității COMANDA îi corespund mai multe instanțe ale entității PRODUS. Invers, dacă înscriem atributul Cantitate în lista atributelor entității PRODUS, atunci toate comenzile asociate ar trebui să refere aceeași cantitate comandată și nu putem specifica cantități diferite în comenzi diferite.

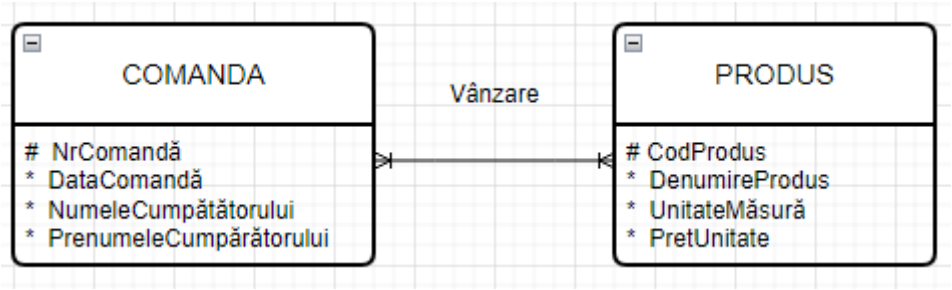


Figura 15.19. Relația many-to-many

Rezolvarea unei relații **many-to-many** constă în introducerea unei noi entități, numită ***entitate de intersecție,*** care se asociază cu entitățile originale cu câte o relație **one-to-many** (Figura 15.20).

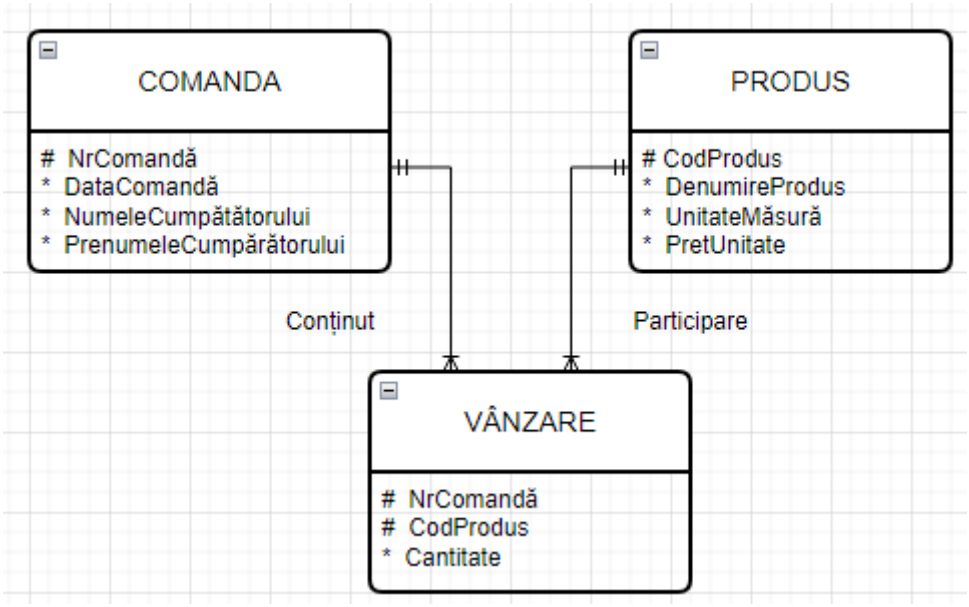


Figura 15.20. Rezolvarea relațiilor many-to-many

La rezolvarea relațiilor many-to-many se realizează următoarele acțiuni:

1) Se identifică **entitatea de intersecție,** numele căreia poate fi unul nou, sau format din numele relației originale (în cazul nostru vom crea entitatea de intersecție VÂNZARE), sau format prin combinarea denumirilor entităților originale (de exemplu, COMANDA_PRODUS)**.**
2) Se elaborează relațiile noi **one-to-many** respectând următoarele cerințe:
   a. **opționalitatea** relațiilor care pleacă din **entitatea de intersecție** sunt întotdeauna **obligatorii** în această parte, iar în partea spre entitățile originale **se păstrează** opționalitatea relațiilor **inițiale**;
   b. **cardinalitatea** ambelor relații este de tip **one-to-many**, iar partea cu **many** va fi **întotdeauna** spre **entitatea de intersecție**.
   c. se definesc **nume** noi pentru relațiile elaborate.
3) Se adaugă **atribute noi** în cadrul entității de intersecție, dacă ele sunt necesare. Pentru exemplul nostru vom adăuga atributul *Cantitate*.
4) Se stabilește **identificatorului** pentru entitatea de intersecție și cheile externe spre entitățile originale sau se formează un **identificator compus** din identificatorii entităților inițiale, la care putem adaugă atribute ale entității de intersecție. În exemplul nostru, identificatorul entității de intersecție este format din *#NrComandă* și *#CodProdus*.

15.4. Modelarea datelor istorice

Pe parcursul exploatării bazei de date pot să intervină anumite modificări în domeniul de obiecte modelat. Orice entitate se poate schimba de-a lungul timpului, precum și relațiile dintre entități se pot schimba. D exemplu, prețul produsului comercializat se poate modifica destul de des ca urmare a inflației, anotimpul, companiilor promoționale etc. Valoarea atributului *PreţUnitate* al entității PRODUS se poate modifica de-a lungul timpului. Iar descrierea entității PRODUS din Figura 15.19 nu poate reflecta această modificare.

Dacă este important să ducem cont de istoricul prețurilor unui produs, atunci atributul *PreţUnitate* trebuie transformat într-o nouă entitate (Figura 5.21). Atributul *DataStop* este unul opțional, deoarece nu totdeauna se cunoaște data când se va modifica prețul produsului.

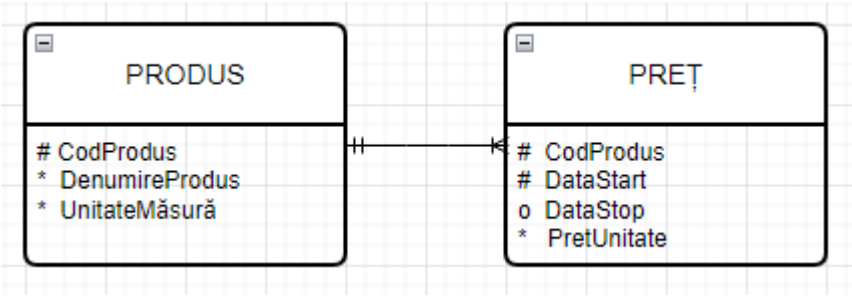


Figura 15.21. Modelarea istoricului prețului unui produs

Un alt exemplu este modelarea procesului de împrumut de carte în cadrul unei biblioteci. În acest caz este important de reținut istoricul tuturor împrumuturilor, deoarece pe baza acestora, se pot afla domeniile de interes ale cititorilor și planifica achizițiile viitoare de carte, precum și determinarea uzurii cărților. Analizând această activitate putem obține o relație **many-to-many** între entitățile **CARTE** și **CITITOR** (Figura 15.22)**.** Fiecare carte poate fi împrumutată de mai mulți cititori (evident nu în același timp) și fiecare cititor poate împrumuta mai multe cărți. Însă în bibliotecă pot fi cărți care niciodată nu au fost împrumutate și înregistrați cititori care nu au făcut nici un împrumut de carte (de exemplu, au citit doar ziare sau au folosit doar Internetul în sălile de lectură).

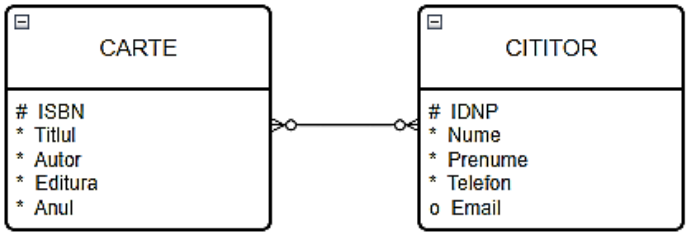


Figura 15.22. Reprezentarea împrumutului de cărți în bibliotecă

Fiind o relație **many-to-many** relația *Împrumut* trebuie descompusă în două relații **one-to-many** (Figura 15.23). Însă este oare această soluție corectă? Poate oare un cititor să împrumute 2 exemplare de carte? Având același ISBN al doilea exemplar ar genera o valoare identică pentru cheia tuplului, fapt care este interzis într-o bază de date relațională. Soluțiile posibile pentru această situație sunt prezentate în Figura 15.24.

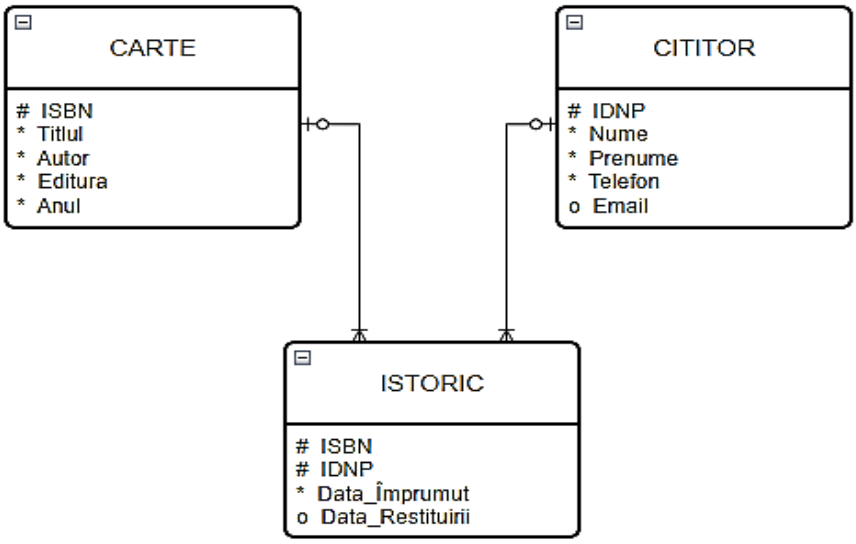


Figura 15.23. Rezolvarea relației Împrumut

În prima variantă se presupune că pentru fiecare împrumut în parte se completează câte o fișă separată care are un număr unic. Aceasta va determina utilizarea unei chei primare compuse din atributele *#ID, #ISBN, #IDNP*. Obținem modelul din partea stângă a Figurii 15.24. Sau se adaugă în entitatea de intersecție un nou atribut *Număr_exemplare* și se obține modelul din partea dreaptă a Figurii 15.24.

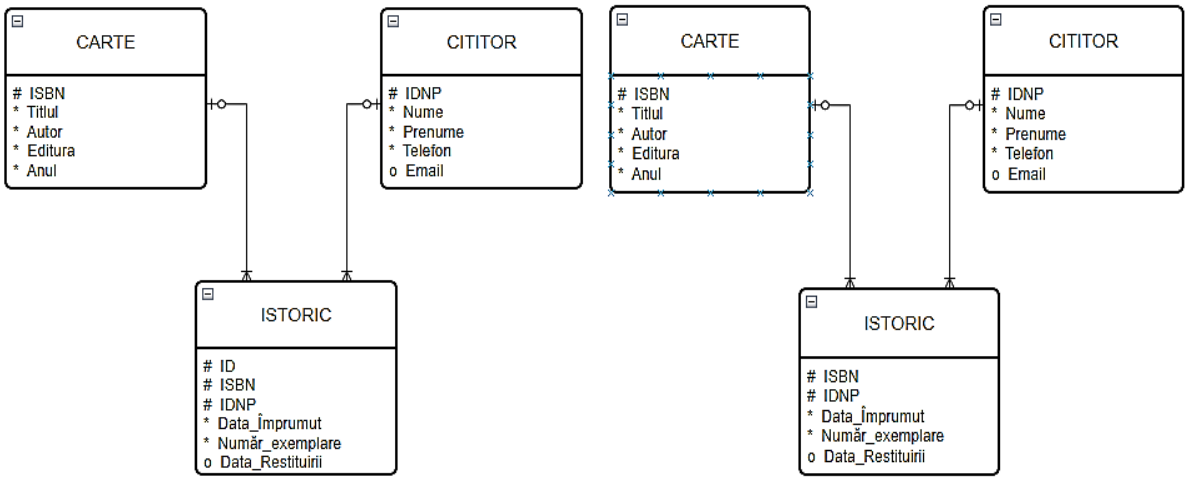


Figura 15.24. Modelarea istoricului împrumutului de cărți în bibliotecă (2 variante)

15.5. Modelarea generică a bazelor de date

În cazul în care regulile afacerii se schimbă foarte des, este relativ dificil de anticipat și de modelat entitățile bazei de date. În astfel de situații, cu evoluții nepredictibile ale afacerii, se poate folosi cu succes un **model generic al bazei de date.** Modelarea generică a bazei de date este avantajoasă din mai multe motive:

- Modelarea generică este flexibilă și permite definirea de noi entități și atribute pe măsură ce specificul activității descrise se schimbă în timp.
- Modelarea generică folosește un număr considerabil redus de entități, numite **entități generice,** iar atributele acestor entități pot fi definite pe parcurs.
- Modelul generic al bazei de date se poate adapta ușor și rapid la noile condiții de lucru.

Modelarea generică definește **o entitate generică** care posedă **atribute generice. Atributele generice** urmează sa ia valori specifice pe măsură ce activitatea descrisă în baza de

date se diversifică. În calitate de exemplu, vom considera entitatea generică PRODUS, cu un set extins de atribute, în care să includem toate atributele posibile de la toate tipurile de articole de îmbrăcăminte (*id, denumire, mărime, lungime, culoare, material, gen, circumferința_taliei, bust, lungime_ mânecă, circumferința_gât, stil, altele).* Toate aceste 13 atribute au caracter opțional și pentru descrierea unui anumit produs se completează doar câmpurile specifice, iar pentru cele care nu îl caracterizează se admite lipsa valorii (se înscrie valoarea Null).

Se pot defini și subtipuri ale entității generice dacă putem anticipa care sunt tipurile de entități care le definim pentru activitatea unui magazin de haine pe durata exploatării bazei de date. Ca neajuns al acestui mod de abordare putem menționa faptul că are loc o creștere nedorită a numărului de atribute și a dimensiunii tabelului asociat entității, care va conține 13 coloane.

Modelarea generică permite o generalizare mai aprofundată a noțiunilor. De exemplu, putem crea entitatea generică TIP_PRODUS cu șase atribute generice *atribut_1, atribut_2, atribut_3, atribut_4, atribut_5, atribut_6* care au menirea de a descrie atributele specifice pentru diferite tipuri de haine. În tabelul asociat entității TIP_PRODUS se trec denumirile atributelor generice, iar valorile acestor atribute se specifică într-un alt tabel, corespunzător unei entități *VALOARE_ATRIBUT* (Figura 15.25).

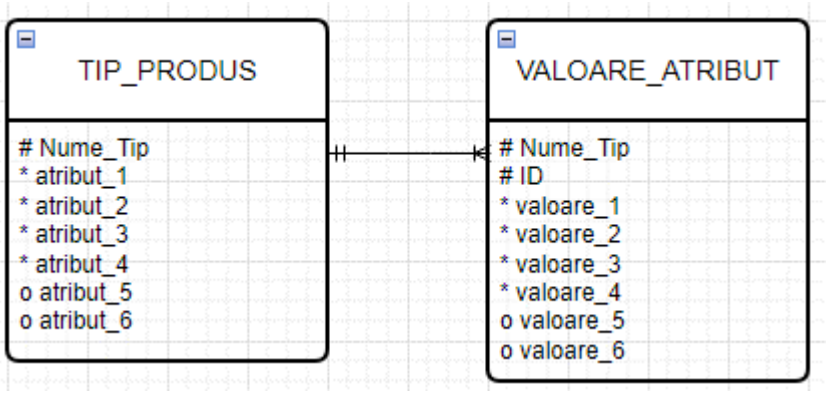


Figura 15.25. Modelarea generică a entităților bazei de date

Acest mod de descriere reduce numărul atributelor și menține dimensiuni rezonabile ale tabelelor din baza de date. Conform diagramei din Figura 15.25, un articol va avea minimum patru atribute specificate și maximum șase, care se aleg dintr-un set extins de atribute. Tabelul cu tipurile de produse va avea șapte coloane în loc de treisprezece. Valorile atributelor pentru diferite tipuri de produse se specifică în alt tabel cu denumiri generice ale coloanelor (Figura 15.26).

TIP_PRODUS						
Nume_Tip	atribut_1	atribut_2	atribut_3	atribut_4	atribut_5	atribut_6
Mantou	ID	Mărime	Material	Culoare	Gen	
Cămașă	ID	Mărime	Material	Culoare	Circumferința_gât	Gen
Maiou	ID	Mărime	Material	Culoare	Gen	Stil
Pantaloni	ID	Mărime	Material	Culoare	Gen	Lungime

VALOARE_ATRIBUT						
Nume_Tip	atribut_1	atribut_2	atribut_3	atribut_4	atribut_5	atribut_6
Mantou	01	M	Lână	Roșu	Femeie	
Cămașă	02	XXL	Bumbac	Albă	44	Bărbat
Maiou	03	L	Bumbac	Verde	Bărbat	T-shirt
Pantaloni	04	M	Denim	Sură	Bărbat	Scurți

Figura 15.26. Definirea atributelor generice pentru tipuri de produse

Modelul generic prezentat la începutul paragrafului pentru entitatea PRODUS stochează informațiile despre produse într-un singur tabel, cu foarte multe înregistrări, greu de urmărit, și este necesară sortarea pe mai multe nivele a datelor pentru separarea informațiilor despre un anumit tip de produs. De aceea, în unele cazuri se recurge la folosirea unor relații recursive care să permită stocarea informațiilor despre diferite categorii de obiecte, în tabele diferite (Figura 15.27). Pentru exemplul anterior, în care se modelează baza de date a unui magazin, în loc de două entități generice (TIP_PRODUS, VALOARE_ATRIBUT*),* se pot considera mai multe entități, cu atribute și valori specifice:

- *TIP_PRODUS (# id_tip_produs, * nume_tip)*
- *PRODUS (# id_produs, * id_tip_produs, * nume_produs)*
- *ATRIBUT (# id_atribut, # id_tip_produs, * nume_atribut)*
- *VALOARE(# id_produs, # id_atribut, # id_tip_produs, * valoare_atribut)*

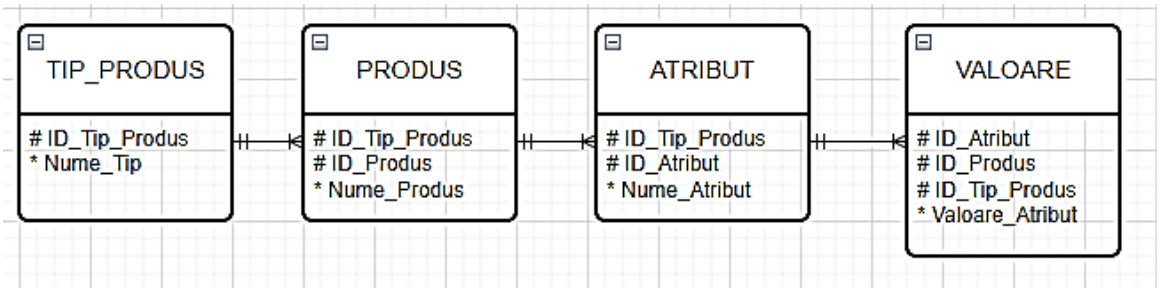


Figura 15.27. Modelarea generică recursivă a entităților bazei de date

Acest model generic „generalizat" este deosebit de flexibil, în sensul că permite adaptarea numărului de atribute la diferitele tipuri de produse sau de entități, în general. Spre deosebire de primul model generic, care consideră un număr maxim prestabilit de atribute, invariabil de la un tip la altul, modelul generalizat poate include un număr variabil de atribute pentru diversele tipuri considerate. Modelul poate fi ușor modificat în timp, pe măsură ce se extinde activitatea descrisă în baza de date. Numărul de coloane din tabelele asociate este fix, în timp ce numărul de înregistrări din tabele va creste.

Prin realizarea unor modele de date generice, se prelungește durata de viață a bazei de date și se simplifică munca administratorilor de date. Totuși, complexitatea și costurile de implementare a unei baze de date pe baza unui model generic sunt considerabil crescute comparativ cu abordarea tradițională.

15.6. Modelul logic al bazei de date relaționale

După elaborarea diagramei E-R în dreptunghi (Figura 15.28) se realizează transformarea ei într-un model logic al unei baze de date relaționale conform exigenților modelului relațional de date. Această transformare se realizează în baza unor reguli de transformare, numite și reguli de mapare.

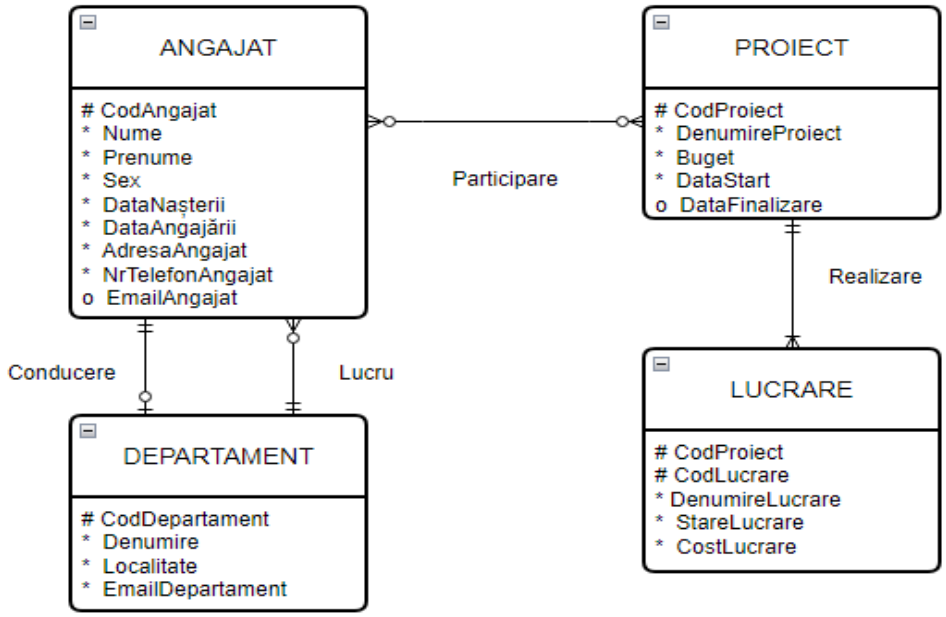


Figura 15.28. Diagrama E-R în dreptunghi în notarea Crow's Foot

- ***Regula 1.*** **Entitățile independente devin tabele independente**. Numele tabelului este numele entității scris la plural cu litere minuscule. Cheia primară a tabelului **nu conține** chei externe. De exemplu, entitatea independentă ANGAJAT generează un tabel independent **Angajați** pentru care atributul *#CodAngajat* este cheia primară. Pentru a specifica atributele care formează cheia primară se folosește notarea PK (Primary Key) și se subliniază numele atributului (Figura 15.29).
- ***Regula 2.*** **Entitățile dependente devin tabele dependente**. Cheia primară a entităților dependente conține cheia primară a entității de care depinde plus unul sau mai multe atribute adiționale. De exemplu, cheia primară a entității LUCRARE este formată din *#CodProiect* (cheia primară a entității PROIECT) și *#CodLucrare*. În același timp atributul *#CodProiect* are și rolul de cheie externă în tabelul **Lucrări** (Figura 15.29).
- ***Regula 3.*** **Un supertip cu mai multe subtipuri este transformat într-un singur tabel, dacă are mai multe atribute comune decât cele specifice fiecărui subtip.**
- ***Regula 4.*** **Subtipurile entităților devin tabele subordonate (dependente) dacă subtipurile au mai multe atribute distincte, decât comune cu supertipul.** Cheia primară a supertipului devine cheie primară în fiecare tabel de subtip. Orice relație a supertipului se marchează printr-o cheie externă în toate tabelele subtipurilor.
- ***Regula 5.*** **Relațiile one-to-one și one-to-many devin chei externe**. Astfel, relația *Conducere* devine o coloană nouă cu denumirea *CodManager* in tabelul entității DEPARTAMENT și relația *Lucru* devine o coloană nouă cu denumirea *CodDepartament* in tabelul entității ANGAJAT, ambele exercitând funcția de cheie externă (Figura 15.29). Pentru notarea atributului care are rol de cheie externă se utilizează notarea FK (Foreign Key). Dacă atributul exercită și rolul de atribut primar (participă la formarea cheii primare), atunci se utilizează notarea dublă PK, FK. În unele surse bibliografice pot fi întâlnite diagrame E-R în notarea lui Chen în care se folosește simbolul „X" la capătul liniei relației pentru a marca plasamentul cheii externe și simbolul „X" pentru specificarea faptului că cheia externă este conținută de cheia primară.
- ***Regula 6.*** **Relația many-to-many devine entitate de intersecție.** Entitatea de intersecție se reprezintă sub forma unui tabel special care are 2 chei externe (câte una pentru fiecare tabel asociat). **Cheia primară** a entității de intersecție este formată prin **compunerea celor 2 chei externe** (opțional pot fi și atribute adiționale). Entitatea de intersecție se reprezintă în diagrama E-R sub forma unui dreptunghi cu contur întrerupt sau punctat (Figura 15.29).

- ***Regula 7.*** **Un atribut simplu (cu valori atomare) devine o coloana în tabelul entității**.
- ***Regula 8.*** **Un atribut atomar multivaloare devine tabel dependent ce are cheia primară compusă, formată din cheia primară a entității și atributul multivaloare.** De exemplu, un ANGAJAT poate avea mai multe numere de telefon (personale și de serviciu), prin urmare atributul *NrTelefonAngajat* este atribut multivaloare care va genera un tabel dependent **Telefoane** (Figura 15.29).
- ***Regula 9.*** **Un atribut compus devine tabel dependent format din atributele atomare componente, care are cheia primară a entității de origine.** În cazul nostru un angajat are o singură adresă de reședință, iar atributul compus *AdresaAngajat* devine un tabel dependent **Adrese,** format din coloanele *#CodAngajat, Localitate, Strada, NrBloc, NrApartament* (Figura 15.29).

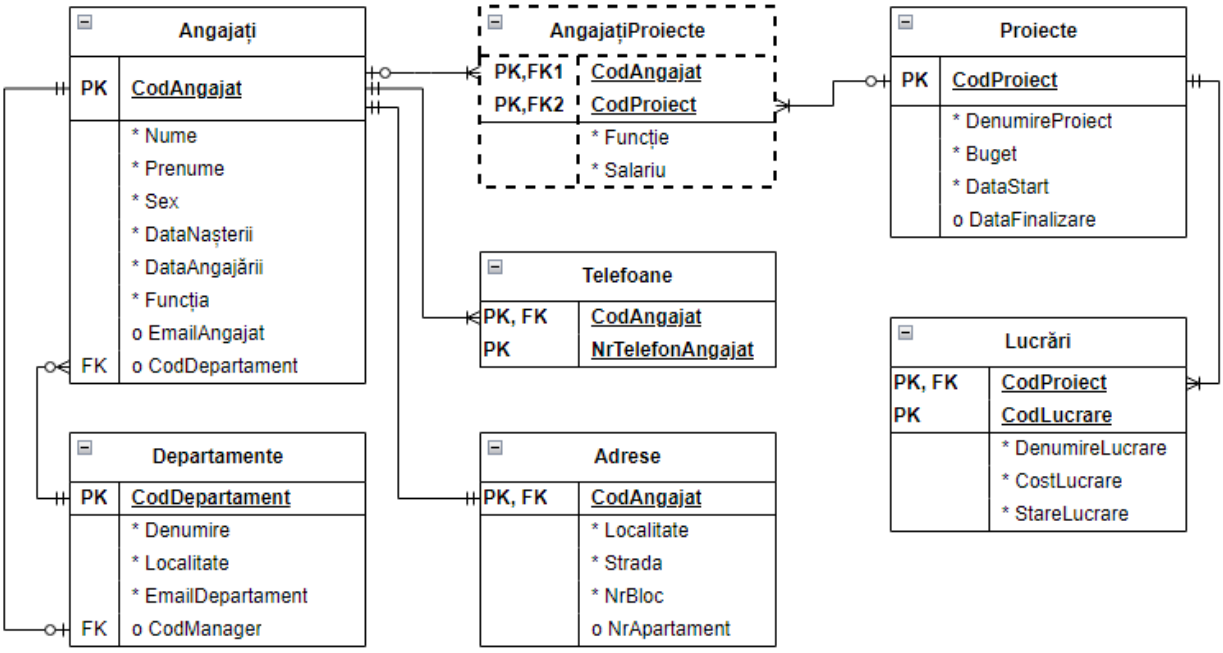


Figura 15.29. Modelul logic al bazei de date relaționale

Întrebări de control:

1. Care sunt regulile de denumire și formare a entităților și atributelor în convenția de reprezentare Crow's Foot (softbox)?
2. Ce simboluri grafice sunt utilizate în notarea Crow's Foot pentru a indica cardinalitățile (zero, unu, mulți) și cum sunt ele combinate pentru a arăta valorile minime și maxime ale unei relații?
3. Care sunt cele două reguli fundamentale pe care trebuie să le respecte subtipurile de entități într-o diagramă E-R și cum se asigură acoperirea tuturor cazurilor de instanțe?
4. Ce reprezintă o relație non-transferabilă, cum se marchează grafic pe o diagramă E-R și prin ce mijloace este asigurată această restricție la nivel de bază de date?
5. Care este diferența dintre relațiile exclusive obligatorii și cele exclusive opționale în notarea Crow's Foot și cum se marchează grafic un grup de relații exclusive?
6. Ce sunt capcanele de conectare (de tip evantai și de tip întrerupere) și cum se pot elimina acestea din modelul E-R?
7. Care sunt pașii necesari pentru rezolvarea unei relații *many-to-many* ($M:N$) și ce rol are entitatea de intersecție în această etapă?
8. Ce avantaje și dezavantaje prezintă modelarea generică a bazelor de date și în ce situații de afaceri este recomandată utilizarea acesteia?
9. Conform regulilor de mapare (transformare) a diagramei E-R în modelul logic relațional, cum sunt transformate atributele multivaloare și cele compuse?

Capitolul 16. DEPENDENȚE FUNCȚIOANALE

16.1. Descompunerea relațiilor cu pierderi de informații

Fie **U** este o **schemă de relație**. Un set de scheme de relații $\{R_1, R_2, ..., R_n\}$ reprezintă o **descompunere** a lui *U* dacă și numai dacă se respectă:

$$U = R_1 \cup R_2 \cup \ldots \cup R_n$$

și dacă la reunire **nu se pierde informația**.

O descompunere {*R, T*} a lui *U* se face **fără pierdere de informație** (referitor la setul de constrângeri) dacă **U = R|x|T,** adică U este rezultatul joncțiunii relației R cu relația T pentru **orice instanțe** ale lui *R, T* și *U*. În caz contrar, descompunerea se realizează **cu pierdere de informații** și pentru relația U este veridică expresia:

$$U \subseteq R|x|T$$

Descompunerea unui tabel în tabele mai mici se realizează în scopul reducerii redundanțelor și evitării anomaliilor. O astfel de descompunere implică anumite riscuri care, în principal, se manifestă sub două aspecte:

- ***posibila pierdere de informații;***
- ***posibila pierdere a dependențelor dintre date.***

r = {A, B}

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$ $r_1 = \{A\}$

B
1
2

$\Pi_B(r)$ $r_2 = \{B\}$

A	B
α	1
α	2
β	1
β	2

$r_1|x|r_2$

Figura 16.1. Descompunerea cu pierderi de informații

Să examinăm un exemplu de descompunere cu **pierdere de informații.** Fie avem o relație r cu schema R={A,B} și aplicăm asupra ei două operații de proiecție $\Pi_A(r)$ și $\Pi_B(r)$. În rezultatul acestor operații se vor obține două relații r_1 și r_2 cu schemele $R_1 = \{A\}$ și $R_2 = \{B\}$ (Figura 16.1). Joncțiunea acestor relații $\mathbf{r_1|x|r_2}$ nu restabilește relația inițială r (referitoare la mulțimea tuplurilor), prin urmare a avut loc o descompunere cu pierdere de informații. Acest lucru se datorează faptului, că schemele de relații R_1 și R_2 sunt **disjuncte**, adică $R_1 \cap R_2 = \emptyset$. În acest caz joncțiunea relațiilor r_1 și r_2 este identică **cu produsul cartezian** al acestor relații, adică $\mathbf{r_1 |x| r_2 = r_1 \times r_2}$.

Să examinăm un exemplu de descompunere cu **pierdere de dependențe** dintre date. Fie avem o relație r cu schema R={**A**, B, C} și cheia **A**, în care există următoarele dependențe dintre atributele schemei: A→B, B→C, A→C. În dependența B→C atributul B nu este cheie pentru schema relației, ceea ce înseamnă că poate exista o redundanță considerabilă în relația r. Descompunem schema în două proiecții cu schemele R1(**A**,B) și R2(**A**,C). Descompunerea este fără pierderi, deoarece atributul comun **A** este o cheie pentru R1 și R2. Însă descompunerea nu păstrează dependențele deoarece au fost păstrate dependențele A→B și A→C , dar s-a pierdut dependența B→C.

16.2. Dependențe funcționale

Fie $R(A_1, A_2, ...A_n)$ este o **schemă relațională** și fie *X, Y* sunt **submulțimi de atribute** ale lui R. Vom spune că ***X determină funcțional Y*** (sau Y depinde funcțional de X), dacă pentru **orice** relație **r** (instanță a lui R) **nu există două tupluri,** care să aibă **aceleași valori** pentru **atributele lui X** și să aibă **valori diferite** pentru **cel puțin un atribut din Y**.

Cu alte cuvinte, fiecare apariție (mulțime a tuplurilor) a lui X poate fi asociată **unei singure** apariții a lui Y. **Două apariții identice** ale lui X **nu pot fi asociate** decât **aceleiași apariții** a lui Y. **O valo*are a lui X determină unic o valoare a lui Y.*** Notația utilizată este ***X→Y.*** X este numit ***determinant,*** iar Y este numit ***determinat (dependent).*** Dependența funcțională X→Y este ***trivială,*** dacă Y⊆X. Implicit, **în orice relație** R există o dependență funcțională de forma ***cheia lui R → celelalte atribute ale lui R.***

O **relație universala** este **o relație** ce grupează **toate atributele,** care **modelează sistemul real** cercetat. Notăm prin **F *mulțimea dependențelor*** considerate de proiectantul bazei pentru o schemă relațională sau pentru o relație universală. De exemplu, să examinăm o relație universală **Produse(IdF, NumeF, AdresaF, IdP, NumeP, Qty).** Atunci putem scrie următoarele dependențe funcționale, aplicând anumite **constrângeri din viața reală (**produsul e livrat de un singur furnizor**)**:

IdP → NumeP, Qty, IdF, NumeF, AdresaF
IdF → NumeF, AdresaF

Aceste dependențe arată că:

- dacă două produse au același cod IdP, atunci este vorba despre același produs;
- dacă două produse sunt livrate de furnizorul cu codul IdF, atunci valorile pentru numele și adresa furnizorului trebuie să fie aceleași.

Dependențele funcționale permit **exprimarea constrângerilor** ce nu pot fi exprimate prin intermediul **super cheilor** (cheia primară, cheia candidat). **Dependențele funcționale** sunt **constrângeri** aplicate **mulțimii de relații** din baza de date și exprimă **faptele din lumea reală**. O mulțime *F* de dependențe funcționale poate fi folosită în două moduri:

- ***pentru a specifica constrângerile aplicate relațiilor;***
- ***pentru a verifica dacă relațiile mai sunt valabile în cazul aplicării mulțimii de dependențe funcționale.***

Plecând de la **o mulțime de proprietăți formale** ale dependențelor, numite **reguli de deducție,** poate fi obținută **mulțimea maximală de dependențe asociate** lui *F*. Această mulțime definește **închiderea lui *F.*** Există o multitudine de reguli de deducție. Pentru a se putea face o prezentare formală a acestora, **trei dintre ele** au fost alese în calitate de **axiome,** iar restul se pot deduce pornind de la aceste 3 axiome. Cele **trei axiome** se mai numesc și **axiomele lui Armstrong.**

Fie X, Y, Z, W sunt mulțimi de atribute ale unei scheme relaționale R. **Axiomele lui Armstrong** sunt următoarele:

- **Axioma 1. Reflexivitate: X→X Dacă Y⊆X, atunci X→Y.**

Toate dependentele funcționale care rezultă din această axiomă sunt numite **dependențe triviale**.

- **Axioma 2. Creșterea determinantului:**
  1. **Dacă X→Y și X⊆Z, atunci Z→Y.**
  2. **Dacă X→Y și W⊆Z, atunci X∪Z→Y∪W.**
  3. **Dacă X→Y, atunci X∪Z→Y∪Z.**
  4. **Dacă X→Y, atunci X∪Z→Y .**
- **Axioma 3. Tranzitivitate: Dacă X→Y și Y→Z, atunci X→Z.**

În baza axiomelor lui Armstrong se pot demonstra o serie de **reguli de inferență** pentru dependente funcționale:

- **Regula 1. Descompunere: Dacă X→Y și Z⊆Y, atunci X→Z .**

Regula descompunerii ne permite să **rescriem** un set de dependențe funcționale astfel, încât să obținem doar dependențe care au în partea dreapta **doar un singur atribut,** adică să obținem dependente funcționale în formă canonică.

- **Regula 2. Reuniune: Dacă X→Y și X→Z, atunci X→Y∪Z .**

Regula reuniunii stabilește că din N reguli obținute prin descompunere se poate obține dependența inițială, **fără pierderea** vreunei corelații existente.

❖ **Regula 3. Pseudo-tranzitivitate: Dacă X→Y şi Y∪W→Z, atunci X∪W→Z.**

Nu toate dependentele funcționale sunt folositoare pentru modelarea relațională. O dependenta funcțională $X \rightarrow Y$ se numește **dependență funcțională totală,** dacă și numai dacă **nu există** nici o **submulțime proprie** $X' \subset X$, astfel încât $X' \rightarrow Y$.

Dacă însă există o submulțime proprie $X' \subset X$, astfel încât $X' \rightarrow Y$, atunci dependența funcțională $X \rightarrow Y$ este o **dependență funcțională parțială.** În Axioma 2 dependența $Z \rightarrow Y$ este o dependenta funcțională parțială.

In cazul **dependențelor funcționale totale**, axiomele lui Armstrong se reduc la **o axiomă unică:**

❖ **Axioma de pseudo-tranzitivitate**: **Dacă X→Y şi Y∪W→Z, atunci X∪W→Z.**

Axioma de pseudo-tranzitivitate este **o regulă de deducție completă** pentru dependențele funcționale totale cu următoarele proprietăți:

- ✓ *pseudo-tranzitivitatea implică tranzitivitatea (în cazul când W=∅);*
- ✓ *reflexivitatea nu poate fi utilizată pentru a obține dependențe funcționale totale;*
- ✓ *reflexivitatea și pseudo-tranzitivitatea implică creșterea determinantului.*

Mulțimea dependențelor funcționale totale din F formează **închiderea pseudo-tranzitiva F+** a acestei mulțimi și este formată din **reuniunea** mulțimilor dependentelor funcționale totale care pot fi obținute din F, folosind **axioma de pseudo-tranzitivitate**. Două mulțimi de dependențe funcționale totale sunt **echivalente,** dacă au închideri pseudo-tranzitive **identice**.

Pentru a modela scheme relaționale se consideră **mulțimi minimale** de dependențe funcționale totale, capabile să genereze **toate închiderile pseudo-tranzitive**. Aceste mulțimi definesc **acoperiri minimale**. **O mulțime de dependente funcționale totale F*,** asociată unei mulțimi de atribute *A,* definește ***o acoperire minimală,*** dacă satisface următoarele proprietăți:

- ***nici o dependență funcțională din F* nu este redundantă;***
- ***toate dependențele funcționale totale între submulțimi ale lui A sunt în închiderea pseudo-tranzitiva a lui F*.***

Orice mulțime de dependențe totale are **cel puțin o acoperire minimală.** Alegerea acoperirii minimale este punctul de start în modelarea schemelor relaționale.

16.3. Forma canonică a dependențelor funcționale

Vom spune că **o mulțime de dependențe funcționale** este în **formă canonică** dacă:

- **Orice dependență** are în partea dreapta **un singur atribut**.
- Mulțimea de dependențe **este minimală**, adică nici una dintre dependențe nu poate fi dedusă din celelalte.

Să analizăm un exemplu. Fie R = {A,B,C,D,E} este o schemă de relație și F este o mulțime de dependențe funcționale asociată:

F = { AB→CDE, C→DE }

După aplicarea **regulii de descompunere** vom obține:

F = { AB→C, AB→D, AB→E, C→D, C→E }

Mulțimea dependențelor funcționale obținute **nu este minimală,** deoarece dependențele AB→D și AB→E se pot deduce prin tranzitivitate din dependența AB→C împreună cu dependențele C→D și C→E. Rezultă că forma canonică a mulțimii F este:

F = { AB→C, C→D, C→E }

Să analizăm un alt exemplu. În cazul relației **Produse(IdP, NumeP, Qty, IdF, NumeF, AdresaF, IdF)** din paragraful precedent putem identifica următoarea mulțime de dependențe funcționale:

F = { IdP→NumeP, Qty, IdF, NumeF, AdresaF, IdF→NumeF, AdresaF }

Forma canonica a mulțimei F este:

F = { IdP→NumeP, IdP→Qty, IdP→IdF, IdF→NumeF, IdF→AdresaF }

Au fost eliminate două dependențe redundante: **IdP→NumeF** și **IdP→AdresaF**

Dependențele funcționale între atributele unei mulțimi universale pot fi reprezentate **grafic**.

Fie avem o mulțime universală de atribute $A=\{A_1, A_2,\ldots A_n\}$ pentru care există o mulțime de **dependențe funcționale canonice** $\{X_i \rightarrow A_j\}$, unde X_i *este* ***o submulțime*** a lui ***A. Graful dependenților funcționale*** este un graf direcționat astfel:

1) Fiecare **atribut** A_j se reprezintă printr-un **singur nod** cu eticheta A_j ;
2) **Dependența funcțională** de forma $A_i \rightarrow A_j$ se reprezintă printr-o **săgeată**, care leagă nodul A_i cu nodul A_j ;
3) Dependența funcțională de formă $X_i \rightarrow A_j$, unde $X_i=\{A_{i1}, A_{i2},\ldots A_{ip}\}$ cu p>1 , se reprezintă print-un **nod auxiliar dreptunghiular** cu eticheta X_i, în care intră **o mulțime de săgeți** de la nodurile $A_{i1}, A_{i2},\ldots A_{ip}$ și pleacă **o singură săgeată** spre nodul A_j .

De exemplu, fie avem o relație cu schema R(K1, K2, X, Y), în care au fost identificate următoarele dependențe funcționale canonice: K1→X și K1K2→Y. Reprezentarea grafică a acestor dependențe funcționale este arătată în Figura 16.1.

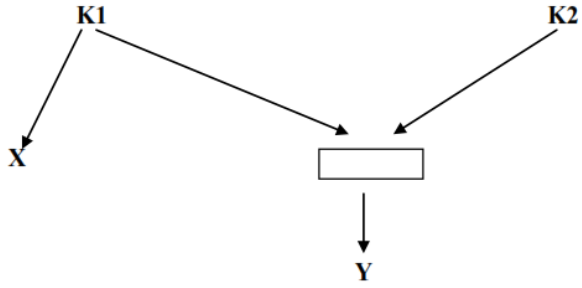


Figura 16.1. Graful dependențelor funcționale

16.4. Dependențe multivalorice

Dependența multivalorică (***Multivalued Dependency - MVD***) apare atunci, când două atribute dintr-un tabel sunt independente unele de altele, dar depind de al treilea atribut. Dependența multivalorică poate să apară în tabele cu minim trei coloane (Figura 16.2).

Automobil	Anul	Culoarea
Toyota	2015	Albă
Toyota	2015	Neagră
Audi	2016	Albă
Audi	2016	Neagră

Figura 16.2. Dependența multivalorică

Definiția formală a dependenței multivalorice este următoarea. Fie R este o schemă de relație definită pe mulțimea de atribute $A=\{A_1, A_2,\ldots A_n\}$ și avem două submulțimi de atribute $\alpha \subseteq R$ și $\beta \subseteq R$. Dependența multivalorică $\alpha \rightarrow\rightarrow \beta$ (”α multidetermină pe β”) există în schema R dacă pentru toate perechile de tupluri t_1 și t_2 din relația r(R) are loc egalitatea $t_1[\alpha]=t_2[\alpha]$ și există tupluri t_3 și t_4 încât:

Condiția 1: $t_1[\alpha]= t_2[\alpha]= t_3[\alpha]= t_4[\alpha]$
Condiția 2: $t_1[\beta]= t_3[\beta]$, $t_2[\beta]= t_4[\beta]$
Condiția 3: $t_1[R\setminus(\alpha\cup\beta)]=t_4[R\setminus(\alpha\cup\beta)]$, $t_2[R\setminus(\alpha\cup\beta)]= t_3[R\setminus(\alpha\cup\beta)]$

Să examinăm un studiu de caz (Figura 16.3). În acest exemplu schema relației este **R(Nume, Curs, Hobby)** și este formată din trei atribute. Atributele **Curs** și **Hobby** sunt multivaloric dependente de atributul **Nume**, deoarece pentru aceeași valoare ”*Albu*” au valori diferite pentru perechea **Curs** și **Hobby**. Să demonstrăm că se respectă condițiile pentru dependențele multivalorice din definiția formală de mai sus. Pentru studiul nostru de caz **α={Nume}, β={Curs}, γ=R\(α∪β)={Hobby}.**

Condiția 1 este satisfăcută: $t_1[\alpha]= t_2[\alpha]= t_3[\alpha]= t_4[\alpha]=$*'Albu'*

Condiția 2 este satisfăcută: $t_1[\beta]= t_3[\beta] =$*'SQL'*, $t_2[\beta]= t_4[\beta]=$*'Python'*
Condiția 3 este satisfăcută: $t_1[\gamma]= t_4[\gamma]=$'*Box*', $t_2[\gamma]= t_3[\gamma]=$'*Muzica*'
Toate cele trei condiții pentru existența dependenței multivalorice sunt satisfăcute și obținem că **Nume →→Curs** ($\alpha \rightarrow\rightarrow \beta$) și **Nume →→Hobby** ($\alpha \rightarrow\rightarrow \gamma$).

	Nume (α)	Curs (β)	Hobby ($\gamma= R\setminus(\alpha\cup\beta)$)
t_1	*Albu*	*SQL*	*Box*
t_2	*Albu*	*Python*	*Muzica*
t_3	*Albu*	*SQL*	*Muzica*
t_4	*Albu*	*Python*	*Box*

Figura 16.3. Exemplu de dependență multivalorică

O dependență multivalorică X →→Y se numește **multidependență elementară**, dacă ea are mulțimi minimale de atribute în partea stângă și dreaptă, adică nu există $X'\subset X$ și $Y'\subset Y$ astfel încât $X'\rightarrow\rightarrow Y'$. Dependențele multivalorice **Nume →→Curs** și **Nume →→Hobby** sunt exemple de multidependențe elementare.

Pentru dependențele multivalorice există o mulțime completă de axiome, care permit obținerea tuturor dependențelor multivalorice prin deducerea lor dintr-o mulțime dată M de multidependențe. Fie avem o schemă relațională R, definită pe mulțimea de atribute $A=\{A_1, A_2,\ldots A_n\}$, cu submulțimile de atribute V,W,X,Y,Z.

- **Axioma 1. Dacă $Y\subset X$, atunci $X\rightarrow Y$.**
- **Axioma 2. Dacă $X\rightarrow Y$, atunci $X\cup Z\rightarrow Y\cup Z$.**
- **Axioma 3. Dacă $X\rightarrow Y$ și $Y\rightarrow Z$, atunci $X\rightarrow Z$.**
- **Axioma 4. Dacă $X\rightarrow\rightarrow Y$, atunci $X\rightarrow\rightarrow R\setminus(X\cup Y)$.**
- **Axioma 5. Dacă $X\rightarrow\rightarrow Y$ și $V\subset W$, atunci $X\cup W\rightarrow\rightarrow Y\cup V$.**
- **Axioma 6. Dacă $X\rightarrow\rightarrow Y$ și $Y\rightarrow\rightarrow Z$, atunci $X\rightarrow\rightarrow(Z\setminus Y)$.**
- **Axioma 7. Dacă $X\rightarrow Y$, atunci $X\rightarrow\rightarrow Y$.**
- **Axioma 8. Dacă $X\rightarrow\rightarrow Y$, $Z\rightarrow W$, $W\subset Y$ și $Y\cap Z=\emptyset$, atunci $X\rightarrow W$.**

Există o serie de reguli care se pot deduce din axiome. Toate consideră existența unei scheme de relație R, iar X, Y, Z, W sunt submulțimi ale lui R:

- **R1. Reuniune: Dacă $X\rightarrow\rightarrow Y$ și $X\rightarrow\rightarrow Z$, atunci $X\rightarrow\rightarrow Y\cup Z$.**
- **R2. Pseudo-tranzitivitate: Dacă $X\rightarrow\rightarrow Y$ și $W\cup Y\rightarrow\rightarrow Z$, atunci $W\cup X\rightarrow\rightarrow Z\setminus(W\cup Y)$.**
- **R3. Pseudo-tranzitivitate mixtă: Dacă $X\rightarrow\rightarrow Y$ și $X\cup Y\rightarrow\rightarrow Z$, atunci $X\rightarrow\rightarrow Z\setminus Y$.**
- **R4. Diferența: Dacă $X\rightarrow\rightarrow Y$ și $X\rightarrow\rightarrow Z$, atunci $X\rightarrow\rightarrow Y\setminus(Z\cup X)\rightarrow\rightarrow Z\setminus Y$.**
- **R5. Intersecție: Dacă $X\rightarrow\rightarrow Y$ și $X\rightarrow\rightarrow Z$, atunci $X\rightarrow\rightarrow Y\cap Z$.**
- **R6. Eliminare atribute comune: Dacă $X\rightarrow\rightarrow Y$, atunci $X\rightarrow\rightarrow Y\setminus X$.**
- **R7. Toate atributele: Dacă $X\cup Y=R$, atunci $X\rightarrow\rightarrow Y$ și $Y\rightarrow\rightarrow X$.**
- **R8. Reflexivitate: Dacă $Y\subseteq X$, atunci $X\rightarrow\rightarrow Y$.**

Aceste axiome și reguli se pot folosi pentru calculul închiderii unei mulțimi de dependente funcționale și multivalorice.

16.5. Dependențe de joncțiune

În teoria bazelor de date, o dependență de joncțiune este o **constrângere** asupra setului de relații ale unei scheme a bazei de date. Un tabel T este supus unei dependențe de joncțiune dacă el poate fi întotdeauna reconstituit prin joncțiunea mai multor tabele, fiecare având un subset de atribute ale lui T. Dacă unul dintre tabelele din joncțiune are toate atributele tabelului T, dependența de joncțiune se numește **trivială**.

Dependența de joncțiune joacă un rol important în procesul de normalizare a relațiilor unei baze de date relaționale, deoarece se poate de dovedit că o schemă de relație R poate fi descompusă

prin proiecție în mai multe tabele cu schemele R_1, R_2, ..., R_n **fără pierdere de informații**, dacă există dependența de joncțiune pe schema R, numită ***(R_1, R_2, ..., R_n)**.

Un alt mod de a descrie o dependență de joncțiune este să spunem că relațiile din dependența de joncțiune sunt **independente** unele de altele. Spre deosebire de cazul dependențelor funcționale, nu există o axiomatizare solidă și completă pentru dependențele de joncțiune.

Definiția formală a dependenței de joncțiune este următoarea: O relație **r(R)** satisface dependența de joncțiune ***(R_1, R_2, ..., R_n)** dacă are loc joncțiunea la stânga (Left Join)

$$r(R) = JOIN_{i=1}^{n}(\Pi_{R_i}(r))$$

În cazul când o schemă oarecare R_i este chiar R se obține o dependență de joncțiune trivială. Pentru descompunerea în două proiecții (n=2) se obțin dependențe multivalorice. Dependența multivalorică **X→→Y** în relația cu schema R(X,Y,Z) (care determină și dependența **X→→Z** conform axiomei 4), corespunde dependenței de joncțiune ***(XUY, XUZ).** Și invers, dependența de joncțiune ***(R_1,R_2)** corespunde dependenței multivalorice **R_1∩R_2→→ R_1\(R_1∩R_2).**

Nr	Client	Pizza	Curier
1	Popa	Napolitană	Roibu
2	Albu	Mexicană	Sârbu
3	Negru	Fresca	Sârbu
4	Roșu	Furiosa	Roibu
5	Popa	Funghi	Lupu
6	Roșu	Rustică	Roibu
7	Grosu	Saleami	Lupu

Figura 16.4. Tabelul Comenzi

Să examinăm un exemplu de dependență de joncțiune (Figura 16.4). În tabelul **Comenzi** pot fi identificate următoarele dependențe funcționale: **Nr→Client, Nr→Pizza, Nr→Curier.** Descompunând prin proiecții tabelul **Comenzi** se vor obține trei **relații independente** care generează o dependență de joncțiune ***((Nr, Client), (Nr, Pizza), (Nr, Curier))** (Figura 16.5).

Nr	Client
1	Popa
2	Albu
3	Negru
4	Roșu
5	Popa
6	Roșu
7	Grosu

Nr	Pizza
1	Napolitană
2	Mexicană
3	Fresca
4	Furiosa
5	Funghi
6	Rustică
7	Saleami

Nr	Curier
1	Roibu
2	Sârbu
3	Sârbu
4	Roibu
5	Lupu
6	Roibu
7	Lupu

Figura 16.5. Proiecțiile tabelului Comenzi

Dacă în tabelul **Comenzi** se adaugă suplimentar constrângerea că **fiecare client are propriul său curier**, atunci poate să existe o dependență de joncțiune ***((Nr, Client), (Nr, Pizza), (Nr, Curier), (Client, Curier))**. În același timp, și dependența de joncțiune ***((Nr, Client, Curier), (Nr, Pizza))** ar fi valabilă. Acest lucru face evident că doar a avea o dependență de joncțiune nu este suficient pentru a normaliza o schemă de bază de date.

Întrebări de control:

1. Care sunt riscurile principale asociate cu descompunerea unui tabel în tabele mai mici și ce condiție trebuie îndeplinită pentru ca o descompunere să fie fără pierderi de informație?
2. Cum se definește formal o dependență funcțională ?
3. Care sunt cele trei axiome ale lui Armstrong folosite pentru deducerea dependențelor funcționale?

4. Care este diferența dintre o dependență funcțională totală și una parțială?
5. Ce este o acoperire minimală a unei mulțimi de dependențe funcționale și care sunt proprietățile ei?
6. Ce condiții trebuie să îndeplinească o mulțime de dependențe funcționale pentru a fi în formă canonică?
7. Ce este o dependență multivalorică (MVD) și în ce tip de tabele poate să apară?
8. Care este condiția pentru ca o dependență multivalorică să fie considerată multidependență elementară?
9. Când devine trivială o dependență de joncțiune?

Capitolul 17. NORMALIZAREA BAZELOR DE DATE

17.1. Esența normalizării bazelor de date

Normalizarea precede etapa de implementare a bazei de date deoarece aceasta implică unele modificări de structură în modelul creat, cum ar fi schimbări de entități, atribute sau relații. **Forma nenormalizată (**UNF – Unnormalized Form) a unui tabel dintr-o bază de date relațională include date repetitive. Un astfel de tabel nu este considerat „relație" în baza de date și baza de date nu poate fi considerată ca fiind una relațională.

Datorită faptului că forma nenormalizată a unui tabel conține date repetitive (redundante) în procesul gestionării acestor date pot să apară diverse erori de actualizare. **Erorile de actualizare** se clasifică în trei categorii:

- **erori de inserare** de date noi sau înregistrări noi, cauzate de incoerența unor relații din schema relațională;
- **erori de modificare** a datelor**,** care sunt cauzate de dublarea datelor în tabel;
- **erori de ștergere** a datelor**,** care au ca consecință pierderea unor date din baza de date**.**

Anomaliile și erorile de actualizare, care apar în procesul gestionării bazei de date, se produc datorită dependențelor funcționale care există între datele din cadrul relațiilor bazei de date. Să examinăm o relație Avioane, care se află în formă nenormalizată, asupra căreia este aplicată o constrângere: ***toate avioanele de același model au aceeași capacitate de locuri de pasager*** (Figura 17.1). Datorită dependenței **Model→Capacitate** relația Avioane conține un șir de anomalii: anomalii de inserare, modificare sau ștergere a datelor; redundanță în date; probleme de re-conexiune a datelor.

#A	Model	Capacitate	Aeroport
1	A320neo	174	Chișinău
2	A320neo	174	Chișinău
3	A320neo	174	Iași
4	E190	106	Chișinău
5	B737-800	189	Iași
6	B737-800	189	Iași

Figura 17.1. Relația nenormalizată Avioane

Dacă analizăm atent datele din tabelul Avioane, atunci putem identifica următoarele anomalii:

1. Relația Avioane conține **redundanță logică**. De exemplu, tuplul (A320neo, 174) apare de trei ori, iar tuplul (B737-800, 189) apare de două ori. Adăugarea noilor înregistrări referitoare la modelele de avioane deja existente în tabel va duce la creșterea acestei redundanțe.
2. Relația Avioane cu constrângerea aplicată poate să genereze **o anomalie de inserție** în cazul, când s-a procurat un nou avion Boeing B737-800 cu **166 de locuri**. Adăugarea acestei înregistrări în relația Avioane va duce la **încălcarea constrângerii** că toate avioanele de același model au aceeași capacitate de locuri de pasager.
3. Relația Avioane poate să genereze o **anomalie la ștergere.** În cazul dacă se șterge înregistrarea pentru care #A este egal cu 4, atunci se pierde informația că avionul Embraer E190 are capacitatea de 106 locuri.
4. Relația Avioane poate să genereze o **anomalie la modificare.** Dacă în urma modernizării unui avion Airbus A320neo a fost mărită capacitatea de locuri până la 210, atunci acest lucru nu poate fi reflectat în tabel, deoarece ar duce la încălcarea constrângerii aplicate că toate avioanele de același model au aceeași capacitate de locuri de pasager.
5. În cazul descompunerii prin proiecții a relației Avioane în mai multe scheme relaționale poate să apară **anomalia de re-conexiune a datelor**, care se manifestă prin pierderea de

informații. De exemplu, în cazul a două proiecții AVIOANE1=PROJECT(AVIOANE, #A, Model) și AVIOANE2=PROJECT(AVIOANE, Model, Capacitate, Aeroport) (Figura 17.2) încercarea de a restabili relația inițială Avioane prin aplicarea operației JOIN(AVIOANE1, AVIOANE2) va duce la **pierderea de informații** prin generarea unui tuplu eronat (3, A320neo, 174, Chișinău).

#A	Model
1	A320neo
2	A320neo
3	A320neo
4	E190
5	B737-800
6	B737-800

#Model	Capacitate	#Aeroport
A320neo	174	Chișinău
A320neo	174	Iași
E190	106	Chișinău
B737-800	189	Iași

Figura 17.2. Proiecțiile relației nenormalizată Avioane

În procesul modelării unei baze de date relaționale, o etapă importantă o reprezintă **normalizarea relațiilor conceptuale (Codd, 1972),** adică obținerea de relații „moleculare“ **fără a pierde nimic din informație** pentru a elimina:

- ***redundanța datelor;***
- ***anomaliile reactualizării datelor;***
- ***rezolvarea problemei re-conexiunii datelor.***

Tehnica normalizării permite obținerea **unei scheme conceptuale rafinate** printr-un proces de **ameliorare progresiva a unei scheme conceptuale inițiale** a bazei de date relaționale. După fiecare etapă de ameliorare, relațiile bazei de date ating un anumit grad de perfecțiune, deci se află într-o anumită **formă normală**. Trecerea unei relații dintr-o formă normală în alta, presupune eliminarea unor **dependențe nedorite de un anumit tip**, care sunt transformate in dependențe admisibile, adică dependențe care **nu provoacă anomalii**.

Normalizarea este un procedeu de identificare a setului optim de relații, *bazat pe* ***dependentele funcționale*** *dintre atribute, care are ca scop crearea unui model logic de date valid, neredundant, care să elimine riscurile de apariție a anomaliilor de reactualizare a datelor în baza de date.*

Procesul de ameliorare a schemei conceptuale **trebuie:**

- *să garanteze* ***conservarea datelor,*** *adică în* ***schema conceptuală finală*** *trebuie să figureze* ***toate datele din cadrul schemei inițiale****;*
- *să garanteze* ***conservarea dependențelor dintre date,*** *adică in* ***schema finală*** *fiecare dependență trebuie să aibă* ***determinantul și determinatul*** *în schema aceleiași relații;*
- *să reprezinte o* ***descompunere minimală a relațiilor inițiale,*** *adică* ***nici una din relațiile*** *care compun schema finală* ***nu trebuie*** *să fie conținută într-o* ***altă relație din această schemă****.*

17.2. Normalizarea prin descompunere a relațiilor

Există **două metode** pentru a modela bazele de date relaționale fără anomalii sau pierderi de informație.

- **Schema descompunerii** pleacă de la o **schemă relațională universală** ce conține toate atributele bazei de date. Schema **se descompune** prin proiecții succesive în **sub-relații**. Descompunerea **se oprește** când continuarea ei **ar duce la pierderi de informații**. Algoritmii de descompunere se bazează, în general, pe descrierea formală a dependenței dintre atribute.
- **Schema sintezei** pleacă de la o mulțime de **atribute independente** din **mulțimea universală a bazei de date**. Utilizând proprietăți de semantică și legături între atribute **se**

pot compune noi relații astfel, încât acestea să nu sufere de anumite anomalii. Algoritmii se bazează, in general, pe teoria grafurilor pentru a reprezenta legătura intre atribute.

Teoria matematica a normalizării prin descompunere a relațiilor a fost elaborată de E.F. Codd. **Normalizarea** este un **proces reversibil** de transformare a unei relații, în relații de structură mai simplă, numite **forme normale**. ***O relație este într-o formă normală particulară dacă ea satisface o mulțime specificată de constrângeri.*** Procesul normalizării se realizează plecând de la **o relație universală** ce conține **toate atributele** sistemului de modelat, plus o mulțime de anomalii. **Orice formă normală** se obține aplicând o **schemă de descompunere.**

Exista **doua** tipuri de descompuneri:

- **Descompunere ce conserva dependentele.** Această descompunere presupune desfacerea relației *R* in proiecțiile $R_1, R_2, ..., R_n$ astfel, încât dependentele lui *R sunt echivalente* (au închideri pseudo-tranzitive identice) cu reuniunea dependentelor din $R_1, R_2, ..., R_n$.
- **Descompunere fără pierdere de informații (*L-join).*** Această descompunere presupune desfacerea relației *R într-o mulțime de* proiecții $R_1, R_2, ..., R_n$ astfel, încât pentru orice realizare a lui *R* este adevărată relația:

$$r(R) = JOIN_{i=1}^{n}(\Pi_{R_i}(r))$$

Acest lucru înseamnă că relația inițială *R* poate fi reconstituită prin joncțiunea naturală a relațiilor obținute prin descompunere. Formele normale ale relațiilor sunt obținute prin descompuneri fără pierderi de informații.

O descompunere fără pierdere de informație se realizează prin aplicarea **regulii Casey-Delobel**. Fie avem o schemă de relație R, iar **α, β, γ** sunt submulțimi de atribute ale lui R. Dacă există a dependență funcțională **α→β**, atunci relația inițială r(R) poate fi obținută prin joncțiunea naturală a două proiecții:

$$\boldsymbol{r(R) = JOIN}(\Pi_{\boldsymbol{\alpha\cup\beta}}(\boldsymbol{r}), \Pi_{\boldsymbol{\alpha\cup\gamma}}(\boldsymbol{r}))$$

unde:

α ∪ β – este mulțimea atributelor care participă în dependența funcțională;

α ∪ γ – este reuniunea determinantului cu atributele ce nu participă în dependența funcțională.

Să aplicăm regula Casey-Delobel pentru relația Avioane pentru a scăpa de anomalia de re-conexiune a datelor. În cazul nostru constrângerea aplicată relației inițiale impune următoarele corelări: α={Model}, β={Capacitate}, γ={A#, Aeroport}. Aplicând regula Casey-Delobel se obțin schemele AVIOANE1(Model#, Capacitate) și AVIOANE2(A#, Model, Aeroport), care sunt scutite de anomalia de reconstituire a datelor inițiale.

17.3. Forme normale

Există 6 forme normale ale bazei de date relaționale (Figura 17.3). Fiecare dintre cele 6 forme normale este mai restrictivă ca predecesoarea sa.

- Prima formă normală (FN1).
- A doua formă normală (FN2).
- A treia formă normală (FN3).
- Forma normală Boyce-Codd (FNBC).
- A patra formă normală (FN4).
- A cincea formă normală (FN5).

Rețeta clasică a normalizării relațiilor unei baze de date este următoarea:

- ✓ Se inventariază **toate atributele** și se constituie **relația universală** (mulțimea tuturor atributelor bazei de date);
- ✓ Se determină **cheia primară** a relației universale și se reprezintă **toate dependențele funcționale** existente în relația universală;
- ✓ Se inventariază **toate dependențele funcționale** relevante și ireductibile dintre atributele relației universale;
- ✓ Se elimină **dependențele parțiale** prin ruperea relației universale în relații cu scheme mai simple prin aplicarea **regulii Casey-Delobel**;

- ✓ Se elimină din relațiile obținute **dependențele tranzitive** prin spargerea succesivă a relațiilor în altele cu scheme din ce în ce mai simple, astfel încât nici unul din tabelele finale să nu conțină dependențe parțiale sau tranzitive;
- ✓ Se identifică eventualele **dependențe multivalorice** și, dacă există asemenea dependențe, se operează noi descompuneri ale tabelelor;
- ✓ Se încearcă determinarea eventualelor **dependențe de joncțiune.**

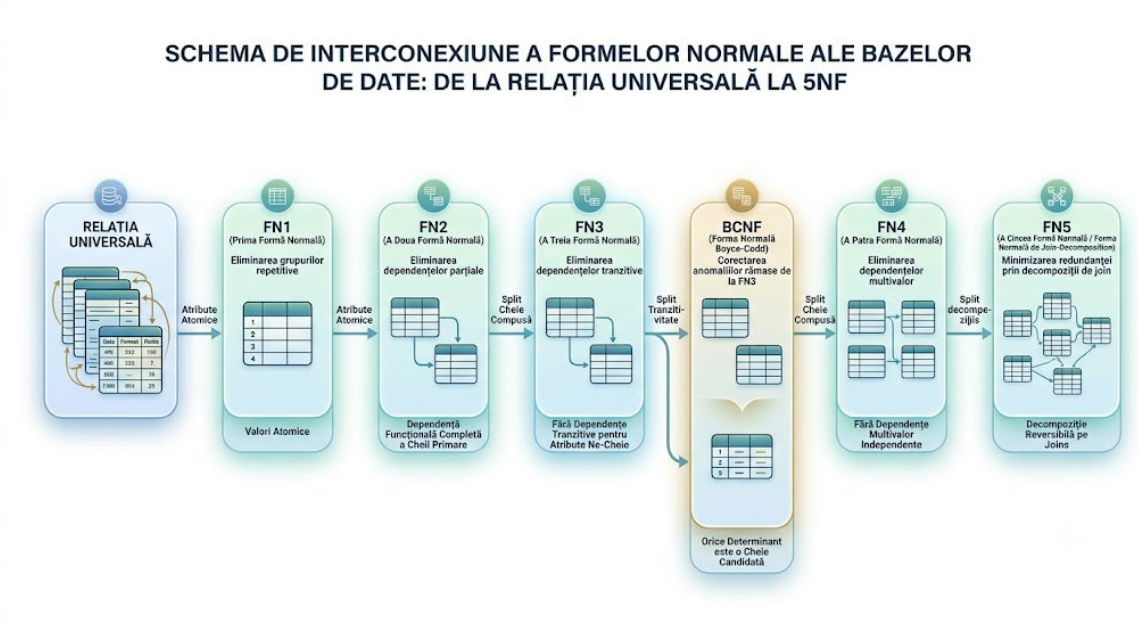


Figura 17.3. Interconexiunea formelor normale ale bazei de date relaționale

17.3.1. Forma normală 1

Se spune că o relație (tabel) este in **prima forma normală (FN1),** dacă fiecărui atribut care o compune îi corespunde o **valoare indivizibilă (atomică). O bază de date** este în forma normală unu (FN1) dacă **toate relațiile (tabelele)** sunt în forma normală unu.

Scopul formei normale unu este de a **simplifica structura unei relații** prin obținerea asigurării că ea nu conține date care mai pot fi descompuse sau date generatoare de valori repetitive, ceea ce înseamnă că **nici un atribut nu poate avea o mulțime de valori**. Prin acțiunea specifică de descompunere, atributele ce nu respectă aceste condiții sunt plasate **în relații separate**, păstrându-se atributele de legătură care au același tip de date și aceeași dimensiune.

Pentru a transforma o relație în prima formă normală trebuie să fie respectate câteva reguli simple :

- ❖ Regula 1: **Atributele** au valori **atomare** de un anumit tip de date**.**
- ❖ Regula 2: Tipul datelor pentru **domeniul de valori** ale atributului e constant în timp.
- ❖ Regula 3: Atributele au **denumiri unice.**
- ❖ Regula 4: **Ordinea** tuplurilor în relație (tabel**) nu contează.**
- ❖ Regula 5:Fiecare relație (tabel) are o **cheie primară.**

Să examină un exemplu de relație **Proprietari** (Figura 17.4). Relația Proprietari este o relație nenormalizată, deoarece atributul *Automobil* conține mai multe valori semnificative în același câmp (nu este atomar). Există mai multe variante posibile de aducere a relației **Proprietari** la prima formă normală (FN1) (Figura 17.5). În prima variantă se modifică tuplurile relației **Proprietari** prin înscrierea valorilor atomare ale atributului *Automobil* și stabilirea cheii compuse *#Proprietar, #Automobil.* În varianta a doua se modifică structura relației prin adăugarea atributelor atomare în funcție de numărul maximal de automobile în posesia unui proprietar

(atributele noi trebuie să accepte valoarea **Null**). A treia variantă presupune descompunerea relației inițiale în mai multe relații cu aceeași schemă și aceeași cheie, dar denumiri diferite

# Proprietar	Automobil
Sârbu	Audi, Ford
Albu	Renault
Negru	Mercedes, BMW, Tesla

Figura 17.4. Relația nenormalizată Proprietari

Varianta 1

# Proprietar	**# Automobil**
Sârbu	Audi
Sârbu	Ford
Albu	Renault
Negru	Mercedes
Negru	BMW
Negru	Tesla

Varianta 2

# Proprietar	Auto_1	Auto_2	Auto_3
Sârbu	Audi	Ford	
Albu	Renault		
Negru	Mercedes	BMW	Tesla

Varianta 3 (3 tabele)

Automobile_1

# Proprietar	Automobil
Sârbu	Audi
Albu	Renault
Negru	Mercedes

Automobile_2

# Proprietar	Automobil
Sârbu	Ford
Negru	BMW

Automobile_3

# Proprietar	Automobil
Negru	Tesla

Figura 17.5. Normalizarea relației Proprietari în FN1

Aducerea unei relații în forma normală unu duce la creșterea redundanței datelor, deoarece vor exista multe coloane cu aceleași date în mai multe rânduri, dar fiecare rând în ansamblu va fi unic.

17.3.2. Forma normală 2

O relație R este în **forma normală doi** (FN2) dacă și numai dacă:

- *Relația R este în FN1;*
- *Fiecare atribut care **nu este cheie** (nu participă la cheia primară) este dependent **de întreaga cheie primară.***

A doua forma normală cere ca **toate atributele** unui tabel să fie dependente funcțional de **toate atributele** cheii primare. Cu alte cuvinte, să **lipsească dependența parțială** a atributelor de cheia primară. Dacă **unul** sau **mai multe atribute** sunt dependente funcțional numai de **o parte a cheii primare**, atunci ele trebuie sa fie separate in **tabele diferite.**

Începând cu a doua formă normală, relațiile **pot fi descompuse** în relații **mai simple**, în scopul diminuării problemelor legate de stocare și actualizare. În general, atunci când într-o relație cu schema R există o dependență funcțională de tip X→Y și X **nu este cheie candidată**, este sigur că relația conține o anumită doză de redundanță.

Savantul Jan Heath a demonstrat că **orice relație** cu schema alcătuită din trei atribute, notată R(X,Y,Z), în care există dependența funcțională X→Y, poate fi descompusă în două relații cu schemele R1(X,Y) și R2(X,Z),care reprezentă proiecțiile relațiile inițiale pe atributele (X,Y) și,

respectiv, (X, Z). Esențial este faptul că descompunerea să se facă **fără pierdere de informații**, adică schema R să poată fi "recompusă" prin joncțiunea schemelor R1 și R2, obținându-se toate tuplurile relației inițiale. Astfel, se poate rezolva problema cheii primare a relației universale, în cazurile încălcării restricției de entitate (un atribut din cheia primară prezintă valori nule). Pentru a obține o relație în FN2 se poate aplica **regula Casey-Delobel** de descompunere a relațiilor (Figura 17.6).

Dacă o relație în FN1 are **o cheie primară** formată dintr-un **singur atribut**, atunci relația este **automat in FN2**, deoarece dependența funcțională a atributelor față de cheie este una totală.

Relația **ParticipareProiecte** în FN1

# **IdAngajat**	FuncțieBază	# **Proiect**	FuncțieProiect
A1	Developer	P1	Supervizor
A1	Developer	P2	Consultant
A2	Programator	P1	Tester
A2	Programator	P3	Coder
A3	Analist	P3	Supervizor
A4	Inginer	P3	Consultant

Relația **Funcții** în FN2

# **IdAngajat**	FuncțiaBază
A1	Developer
A2	Programator
A3	Analist
A4	Inginer

Relația **FuncțiiProiecte** în FN2

#IdAngajat	**# Proiect**	FuncțieProiect
A1	P1	Supervizor
A1	P2	Consultant
A2	P1	Tester
A2	P3	Coder
A3	P3	Supervizor
A4	P3	Consultant

Figura 17.6. Descompunerea în sub-relații în FN2

17.3.3. Forma normală 3

O relație este în forma normală 3 (FN3) dacă și numai dacă:

- Relația este in forma normală 2;
- Fiecare atribut, care nu participă la formarea cheii relației, nu depinde tranzitiv de un alt atribut.

Cu alte cuvinte, o relație este în FN3 dacă și numai dacă fiecare atribut non-cheie depinde doar de **întreaga cheie** și nimic altceva.

Pentru a obține relații în FN3 se aplică regula Casey-Delobel. Fie avem o relație r(R) cu schema R(K,X,Y,Z) în care atributul Y depinde tranzitiv de cheie prin atributul X, adică K→X→Y. Datorită dependenței funcționale X→Y schema relațională R nu este în FN3 și trebuie descompusă în două proiecții R1(X,Y) și R2(K,X,Z), care sunt în forma normală 3.

Să examinăm un studiu de caz (Figura 17.7). Fie avem relațiile **Studenți**, **Cursuri** și **Punctaje**, toate în forma normală 2. Relațiile **Studenți** și **Cursuri** sunt și în FN3. Cheia relației **Punctaje** este o **cheia compusă** *#StudentID, #CursID.* Atributul *Examen* depinde atât de student, cât și de curs. De exemplu, un student la arhitectură sau design vestimentar va avea la anumite cursuri examen de atelier, iar studentul de la filologie nu. Unele cursuri prevăd și examene practice, altele – doar examene teoretice.

Relația **Studenți**

# StudentID	Nume	Grupa
10	Albu	CIB201
11	Albu	TI201

Relația **Cursuri**

# CursID	Curs
1	Java
2	Python
3	C++

Relația **Punctaje**

# StudentID	# CursID	Puncte	Examen	Total
10	1	70	Proiect	200
10	2	75	Teoretic	170
11	1	80	Practic	130

Figura 17.7. Relații în FN2

Atributul *Total* este un atribut care depinde de atributul *Examen*, deoarece punctajul total se calculează în mod diferit în funcție de tipul examenului. De exemplu, practicile pot fi notate cu note mai mici în timp ce subiectele teoretice pot fi notate cu note mai mari, sau invers. Sau pot fi folosite scale diferite pentru diferite tipuri de examene, sau ponderi diferite ale punctajelor acumulate în punctajul final. Atributul *Examen* este un atribut simplu și nu face parte din cheia primară, iar atributul *Total* depinde tranzitiv de acesta. Prin urmare, există o dependență tranzitivă *#StudentID,#CursID→Examen→Total* (Figura 17.7). Pentru a obține FN3 relația **Punctaje** se descompune în două relații **Punctaje** și **Examene** (Figura 17.8).

Relația **Punctaje**

# StudentID	# CursID	Puncte	# ExamenID
10	1	70	1
10	2	75	2
11	1	80	3

Relația **Examene**

# ExamenID	Examen	Total
1	Proiect	200
2	Teoretic	170
3	Practic	130

Figura 17.8. Relații în FN3

Alt exemplu de relație care nu satisface condițiile pentru FN3 este prezentat în Figura 17.9. Deoarece fiecare rând din tabel trebuie să ne spună cine a câștigat un anumit turneu într-un anumit an, cheia compusă {*#Turneu, #An*} este un set minim de atribute garantat pentru a identifica unic un rând. Adică, {*#Turneu, #An*} este o cheie candidată pentru relația **Învingători**.

Relația inițială **Învingători (FN2)**

# Turneu	**# An**	Campion	DataNaștere
Indiana Invitational	1998	Al Fredrickson	21.07.1975
Cleveland Open	1999	Bob Albertson	28.09.1968
Des Moines Masters	1999	Al Fredrickson	21.07.1975
Indiana Invitational	1999	Chip Masterson	14.03.1977

Relația **Învingători (FN3)**

# Turneu	**# An**	Campion
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Relația **Campioni (FN3)**

# Campion	DataNaștere
Al Fredrickson	21.07.1975
Bob Albertson	28.09.1968
Chip Masterson	14.03.1977

Figura 17.9. Normalizarea până la FN3

Încălcarea FN3 are loc deoarece atributul non-primar *DataNaștere* depinde în mod tranzitiv de cheia candidat *{#Turneu, #An}* prin atributul non-primar *Campion*. Aceasta face ca tabelul inițial să fie vulnerabil la inconsecvențe logice, deoarece nimic nu împiedică aceeași persoană să fie afișată cu date de naștere diferite în înregistrări diferite. Pentru a exprima aceleași fapte fără a încălca FN3, este necesar descompus tabelul inițial **Învingători** în două relații (Figura 17.9).

17.3.4. Forma normală Boyce-Codd

O schemă de relație R se află în forma normală Boyce-Codd (FNBC) dacă, pentru **toate dependențele funcționale** ce au loc în R şi sunt de forma $X \rightarrow Y$ în care $X \subseteq R$ şi $Y \subseteq R$ sunt îndeplinite condițiile:

- ***X → Y*** este ***trivială.***
- ***X*** este o ***cheie candidat*** a schemei de relație **R,** astfel încât ***X→R.***

Cu alte cuvinte, **fiecare atribut** trebuie să depindă **de cheie, de întreaga cheie şi de nimic altceva.** FNBC este **o generalizare** a formelor normale doi şi trei. Forma normală Boyce-Codd e o **versiune puțin mai restrictivă** de forma normala 3. Pentru ca o relație să fie în forma normală Boyce-Codd, ea trebuie să satisfacă **două condiții**:

- Să fie în **FN3.**
- Pentru **orice d**ependență funcțională **A→B** identificată într-o schemă de relație**, A** trebuie să fie o **supercheie (să aibă valori unice).**

Fie avem relația **Cursuri** în FN3 (Figura 17.10). În această relație atributele *#StudentID, #Curs* formează cheia primară și ne permite găsirea tuturor tuplurilor din tabel. Important este că pentru relația **Cursuri** este aplicată constrângerea că **un profesor** predă **o singură materie**, dar o disciplină poate avea **profesori diferiți**. Prin urmare, există **o dependență între curs și profesor**, unde cursul depinde de numele profesorului.

Tabelul **Cursuri** satisface **prima formă normală**, deoarece toate valorile sunt **atomice**, numele coloanelor sunt **unice** și toate valorile stocate într-o anumită coloană sunt **din același domeniu.** Acest tabel satisface **formă normală doi**, deoarece **nu există dependență parțială**. Și **nu există o dependență tranzitorie**, prin urmare, tabelul satisface și a **3-a formă normală**. Dar relația **Cursuri** nu este în forma normală Boyce-Codd. Atributele **#***StudentID, #Curs* formează **cheia primară** a relației, ceea ce înseamnă că atributul *Curs* este un **atribut primar.** Dar mai există o dependență funcțională, și anume *Profesor→Curs.* Atributul *Profesor* este un atribut **non-primar** și nu este **cheie candidat,** fapt ce încalcă cerințele FNBC (Figura 17.10).

Relația inițială **Cursuri** în FN3

# StudentID	# Curs	Profesor
101	Java	ProfesorJava
101	C++	ProfesorC++
102	Java	ProfesorJava2
103	SQL	ProfesorSQL
104	Java	ProfesorJava

Relația **Studenți** în FNBC

# StudentID	# P_ID
101	1
101	2
102	3
103	4
104	1

Relația **Profesori** în FNBC

# P_ID	Profesor	# Curs
1	ProfesorJava	Java
2	ProfesorC++	C++
3	ProfesorJava2	Java
4	ProfesorSQL	SQL

Figura 17.10. Normalizarea până la FNBC

Pentru ca relația **Cursuri** să satisfacă FNBC ea trebuie descompusă în două proiecții **Studenți** și **Profesori** (să nu uităm de constrângerea aplicată că un profesor predă o singură disciplină). În relația **Studenți** nu există atribute non-primare și ea automat satisface cerințele pentru FNBC. În relația **Profesori** atributul non-primar *Profesor* a căpătat statutul de **cheie**

candidat, deoarece datorită constrângerii există dependența *Profesor→{P_ID, Curs},* și relația **Profesori** satisface cerințele pentru FNBC (Figura 17.10).

Relația inițială **Birouri** în FN1

# AngajatID	# NrTelefon	NrBirou
1	2-01	301
2	2-02	302
3	2-01	301
4	2-04	304
5	2-03	303
6	2-02	302

Relația **Telefoane_1** în FNBC

# NrBirou	NrTelefon
301	2-01
302	2-02
303	2-03
304	2-04

Relația **Telefoane_2** în FNBC

# AngajatID	# NrBirou
1	301
2	302
3	301
4	304
5	303
6	302

Figura 17.11. Normalizarea până la FNBC

Trebuie de remarcat faptul că nu întotdeauna este posibilă descompunerea unei relații în FNBC cu **păstrarea dependențelor funcționale**. Să examinăm relația **Birouri***(#AngajatID, #NrTelefon, NrBirou)* în FN1 cu constrângerile aplicate: a) fiecare angajat are un birou; b) fiecare birou are un telefon staționar; c) într-un birou pot activa mai mulți angajați (Figura 17.11). În relația inițială **Birouri** există dependența funcțională *{AngajatID, NrTelefon}→NrBirou.* Această dependență este una parțială, deoarece datorită constrângerii ”fiecare birou are un telefon staționar” există o dependență parțială de cheie *NrTelefon→NrBirou* și relația este în FN1. În același timp se observă existența dependenței valide *NrBirou→NrTelefon,* în care atributul *NrBirou* este o cheie candidat. Descompunerea relației inițiale în două proiecții prin aplicarea regulii Casey-Delobel, pornind de la dependența funcțională *NrBirou→NrTelefon,* va genera două relații **Telefoane**_1(*#NrBirou, NrTelefon*) și **Telefoane**_2(*#NrBirou, #AngajatID*), ambele în FNBC (Figura 17.11). Ambele relații conservă datele și permit restabilirea relației inițiale **Birouri**. Totuși, dependența funcțională *{AngajatID, NrTelefon}→NrBirou* a fost pierdută (nu se regăsește în nici una din proiecțiile obținute în rezultatul descompunerii relației inițiale). Prin urmare, relațiile în FNBC păstrează datele, dar nu păstrează întotdeauna dependențele funcționale.

Să examinăm un alt exemplu (Figura 17.12). Fie avem o relație ***Cărți****(Autor, AnulNașterii, Titlu, Categorie, Preț).* Cheia candidat pentru această relație este *{Autor, Titlu},* iar însuși relația se află în FN1, deoarece există următoarele dependențe funcționale parțiale față de cheie: *Autor→AnulNașterii și Titlu→{Categorie, Preț}.* Descompunerea relației **Cărți** în două relații **Autori**(*#Autor, AnulNașterii*) în FNBC și **Cărți2**(*#Autor, #Titlu, Categorie, Preț)* în FN1 este una incorectă, deoarece în relația **Cărți2** se păstrează dependența parțială de cheie. Pentru a obține relații în FNBC este necesar ca atributele din partea stângă a celor două dependențe funcționale netriviale (*Autor* în prima și *Titlu* în a doua) să fie super chei în relațiile obținute prin descompunere. Pentru a asigura această cerință relația **Cărți2** se descompune în alte două proiecții conform regulii Casey-Delobel, iar rezultatul final este:

- **Autori**(*#Autor, AnulNașterii*) în FNBC pentru DF: *Autor→AnulNașterii*
- **InfoCarte**(*#Titlu, Categorie, Preț*) în FNBC pentru DF: *Titlu→{Categorie, Preț}.*
- **Titluri***((#Autor, #Titlu)* în FNBC

Relația inițială **Cărți** în FN1

# Autor	AnulNașterii	# Titlu	Categorie	Preț
Popescu	1969	SQL	Manual	95
Ionescu	1975	Python	Suport	78
Onea	1986	Java	Compendiu	55
Popescu	1969	MongoDB	Suport	68
Ionescu	1975	PHP	Manual	105

Relația **Autori**

# Autor	AnulNașterii
Popescu	1969
Ionescu	1975
Onea	1986

Relația **Titluri**

# Autor	# Titlu
Popescu	SQL
Ionescu	Python
Onea	Java
Popescu	MongoDB
Ionescu	PHP

Relația **InfoCarte**

# Titlu	Categorie	Preț
SQL	Manual	95
Python	Suport	78
Java	Compendiu	55
MongoDB	Suport	68
PHP	Manual	105

Figura 17.12. Normalizarea până la FNBC

Pentru ca o relație să fie adusă în FNBC nu trebuie în mod obligatoriu ea să fie in FN3. Se pot aduce în BCNF și relații aflate în FN1 sau FN2. Acest lucru este posibil întrucât dependențele funcționale parțiale și cele tranzitive sunt tot dependențe non-cheie, adică dependențe ai căror determinanți nu sunt chei candidat.

Algoritmul de aducere a unei relații cu schema R din FN1 în FNBC este următorul:

- **Pasul 1**. Dacă relația conține **cel mult două atribute**, atunci schema R este în FNBC și algoritmul s-a terminat.
- **Pasul 2**. Dacă relația conține mai mult de două atribute, atunci se consideră toate perechile posibile (X,Y) de atribute distincte din mulțimea atributelor A.
- **Pasul 3**. Se determină $A1^+$, închiderea mulțimii A1=A-{X,Y}.
- **Pasul 4.** Dacă pentru orice pereche (X,Y) se realizează că $X \notin A1^+$, atunci relația cu schema R este în FNBC și algoritmul s-a terminat.
- **Pasul 5**. În caz contrar (pentru orice pereche (X,Y), în care X aparține $A1^+$), relația cu schema R nu este în FNBC.
- **Pasul 6**. Se reduce progresiv schema relației și se reia algoritmul explorând relațiile obținute.

17.3.5. Forma normală 4

Forma normală patru (FN4) se bazează pe conceptul de **dependență multivalorică**. O dependență multivalorică apare doar în relațiile ce **au cel puțin trei coloane**. Dacă una dintre coloane are rânduri ale căror valori corespund unei singure valori ale unui rând dintr-o altă coloană, atunci se spune că a apărut o dependență multivalorică.

O relație se află în forma normală patru (FN4) dacă şi numai dacă se află în forma normală Boyce-Codd şi dacă nu are dependențe funcționale multivalorice.

Forma normală 4 elimina redundanțele datorate relațiilor ***many-to-many***. Dacă relația conține **cel mult două** atribute și respectă cerințele FN1, atunci relația respectă automat cerințele pentru FN2, FN3 și FNBC, în ea nu pot apărea dependențe multivalorice și, prin urmare, relația este în FN4.

O relație în FNBC este în FN4 dacă orice multidependență elementară de forma $X \rightarrow\rightarrow Y$, X este o **supercheie** în schema R. Aducerea relațiilor în FN4 presupune eliminarea dependențelor multivalorice în cazul când sunt mai mult de una în cadrul unei relații.

#Pizzeria	#Blat	#Livrare
Corso	Subțire	Centru
Corso	Subțire	Botanica
Corso	Subțire	Ciocana
Corso	Gros	Centru
Corso	Gros	Botanica
Corso	Gros	Ciocana
Oliva	Gros	Botanica
Oliva	Gros	Râșcani
Oliva	Subțire	Botanica
Oliva	Subțire	Râșcani
Andy's	Clasic	Centru
Andy's	Clasic	Botanica

Figura 17.13. Relația Livrări în FNBC

Să examinăm un exemplu de relație cu dependențe multivalorice (Figura 17.13). Fiecare restaurant livrează un tip de pizza într-o arie de distribuție. Cheia primară a relației **Livrări** este formata din mulțimea atributelor {*#Pizzeria, #Blat, #Livrare*} și în relație nu există atribute non-cheie, prin urmare, relația se află în FNBC. Dar deoarece varietățile de pizza oferite de un restaurant sunt **independente** de ariile de distribuție, există **redundanțe** in tabelul **Livrări**. De exemplu, pentru restaurantul Corso se menționează de 3 ori că se oferă pizza cu blat *"Subțire"* și "*Gros*", se repetă de două ori ariile de distribuție. De asemenea, dacă dorim să adăugăm un nou tip de pizza cu blat "*Clasic*" în meniul restaurantului Corso, va trebui sa adăugăm 3 înregistrări, cate una pentru fiecare arie de distributie. Același lucru se va întâmpla dacă restaurantul Andy's își va extinde livrările în alte sectoare ale orașului Chișinău sau va extinde sortimentul de pizza.

În relația **Livrări** se observă **două** dependențe multivalorice *Pizzeria→→Blat* și *Pizzeria→→Livrare*, care trebuie să fie separate în relații separate, care vor fi în forma normală 4 (Figura 17.14).

Relația **Pizzerii**

#Pizzeria	#Blat
Corso	Subțire
Corso	Gros
Oliva	Gros
Oliva	Subțire
Andy's	Clasic

Relația **Livrări**

#Pizzeria	#Livrare
Corso	Centru
Corso	Botanica
Corso	Ciocana
Oliva	Botanica
Oliva	Râșcani
Andy's	Centru
Andy's	Botanica

Figura 17.14. Relații în FN4

17.3.6. Forma normală 5

A **cincea formă normală** se bazează pe conceptul de **dependență de joncțiune**. **Dependența de joncțiune** este o proprietate ce **garantează** că nu se generează înregistrări **false** la **reunirea** relațiilor obținute prin descompunere.

O relație se află în forma normală cinci dacă ea nu poate fi descompusă în alte relații fără a pierde informații. Cu alte cuvinte, dacă se adaugă un rând suplimentar unei relații care nu se află în forma normală cinci şi dacă această relație se descompune în alte relații, prin refacerea relației inițiale se obțin înregistrări false.

O dependență joncțiune *$(R_1, R_2, ... ,R_n)$ reprezintă o **constrângere** aplicată relației R, care arată faptul că fiecare **instanță** ***r(R)*** trebuie să aibă **pierdere** de informații prin descompunerea în

relații R_1, R_2, ... , R_n. O **dependență multivalorică** reprezintă un caz **special** al unei dependențe de cuplare în care $n = 2$. Dependența de joncțiune *(R_1, R_2,..., R_n) este o dependență de joncțiune *trivială,* dacă unele relații $R_i = R$.

Definiția formală pentru FN5 este următoarea: o schemă de relație R se află în **forma normală cinci** referitor la o mulţime F de **dependențe funcționale, multivalorice şi de joncțiune** dacă pentru fiecare dependență de joncțiune netrivială *$(R_1, R_2, ... , R_n)$ din F, **fiecare schemă de relație R_i** este o **supercheie** pentru schema R.

O relație este în FN5 dacă ea este în FN4 și **nu conține** nicio dependență de joncțiune, iar joncțiunea relației ar trebui să fie fără pierderi. FN5 este satisfăcută atunci, când toate relațiile sunt descompuse în cât mai multe relații posibile pentru a evita redundanța. Să studiem un caz concret al relației **Planificare** în FN4 (Figura 17.15). În această relație cheia este una compusă formată din toate cele trei atribute. Să presupunem că adăugăm un nou semestru ca semestrul 3, dar nu știm despre curs și cine va preda cursul respectiv, așa că înscriem în câmpurile *Profesor* și *Curs* valoarea NULL. Dar toate cele trei coloane împreună acționează ca o cheie primară, așa că nu putem lăsa alte două coloane goale, deoarece se încalcă constrângerea pentru cheie. Relația **Planificare** nu este în FN5**,** deoarece are o dependență de conexiune netrivială *{{Curs, Profesor}, {Semestru, Profesor}, {Semestru, Curs}}, cu toate acestea, subseturile de atribute {Curs, Profesor}, {Semestru, Profesor}, {Semestru, Curs} **nu sunt superchei** ale relației inițiale.

Așa că vom descompune această relație în trei relații noi: P1, P2 și P3, care se vor afla în FN5 (Figura 17.16).

# Curs	# Profesor	# Semestrul
Informatica	Albu	Semestru 1
Informatica	Ionescu	Semestru 1
Matematica	Ionescu	Semestru 1
Matematica	Popescu	Semestru 2
Chimia	Negru	Semestru 1

Figura 17.15. Relația Planificare

Relația **P1**

# Curs	# Profesor
Informatica	Albu
Informatica	Ionescu
Matematica	Ionescu
Matematica	Popescu
Chimia	Negru

Relația **P2**

# Semestrul	# Curs
Semestru 1	Informatica
Semestru 1	Matematica
Semestru 2	Matematica
Semestru 1	Chimia

Relația **P3**

# Semestrul	# Profesor
Semestru 1	Albu
Semestru 1	Ionescu
Semestru 2	Popescu
Semestru 1	Negru

Figura 17.16. Relații în FN5

Să examinăm o altă relație cu numele **Oferte** (Figura 17.17). Cheia primară a relației este una compusă formată din cele trei coloane. Tabelul este în FN4, deoarece nu există dependențe multivalorice: nicio coloană (care în sine nu este o cheie candidată sau o supercheie) nu este un determinant pentru celelalte două coloane.

# Salesman	# Brand	# Product
Schneider	Acme	Fen
Schneider	Acme	Breadbox
Jones	Robusto	Cuțite
Jones	Robusto	Fen
Jones	Robusto	Breadbox
Jones	Robusto	Umbrelă
Ferguson	Robusto	Fen

Ferguson	Robusto	Telescope
Ferguson	Acme	Fen
Ferguson	Acme	Lampă
Ferguson	Nimbus	Dulap

Figura 17.17. Relația Oferte

În absența oricăror reguli care să restricționeze combinațiile posibile valide de vânzător ambulant, marcă și tip de produs, tabelul cu trei atribute de mai sus este necesar pentru a modela corect situația. Să presupunem, totuși, că se aplică următoarea regulă: fiecare vânzător are în sortimentul său o listă limitată de firme și o listă limitată de tipuri de mărfuri, și oferă bunuri din lista de produse fabricate de firme de pe lista firmelor.

Adică vânzătorul nu are dreptul de a comercializa mărfuri de orice brand. Dacă vânzătorul V are dreptul de a comercializa mărfuri ale companiei B și dacă vânzătorul V are dreptul de a comercializa mărfuri de tip P, atunci în sortimentul vânzătorului V se includ mărfuri de tip P ale companiei B, cu condiția ca brandul B să producă bunuri de tip P. Dacă vânzătorul este reprezentant oficial al brandului B1 și brandului B2, iar produsul P se află în oferta comercială a sa, atunci, presupunând că marca B1 și marca B2 fac ambele produse de tip P, vânzătorul ambulant trebuie să ofere produse P atât de marca B1, cât și de marca B2. O astfel de restricție poate fi cauzată, de exemplu, prin faptul că lista tipurilor de mărfuri ale vânzătorului este limitată de licențele de care dispune sau de cunoștințele și calificările necesare pentru vânzarea lor, iar lista companiilor fiecărui vânzător este determinată prin acorduri de parteneriat (Figura 17.17).

Relația de mai sus nu poate exclude situațiile în care **această restricție va fi încălcată**. Nimic nu împiedică introducerea de date privind vânzarea de bunuri pe care firma nu le produce deloc, nici introducere datelor privind vânzarea de bunuri ale brandului pe care vânzătorul nu are dreptul să le comercializeze, nici introducerea datelor despre vânzarea de mărfurilor care lipsesc în oferta vânzătorului.

Ca și în cazul precedent, relația **Oferte** nu se află în FN5, deoarece are o dependență de conexiune netrivială *{{Vânzător, Brand}, {Brand, Produs}, {Vânzător, Produs}}, cu toate acestea, subseturile de atribute {Vânzător, Brand}, {Brand, Produs} și {Vânzător , Produs} **nu sunt superchei** ale relației inițiale. În acest caz, este posibil de împărțiți tabelul **Oferte** în trei relații diferite în FN5 (Figura 17.18).

Doar în situații rare un tabel în FN4 nu este conform cu FN5. De exemplu, când tabelele descompuse sunt ciclice. Acestea sunt situații în care o constrângere complexă din lumea reală, care guvernează combinațiile valide de valori ale atributelor din tabelul FN4, nu este implicită în structura acelui tabel. Dacă un astfel de tabel nu este normalizat la FN5, sarcina menținerii coerenței logice a datelor din tabel trebuie să fie suportată parțial de aplicația responsabilă pentru inserări, ștergeri și actualizări ale acestuia. În acest caz există un risc crescut ca datele din tabel să devină inconsecvente. FN5 a tabelului exclude posibilitatea unor astfel de inconsecvențe.

Relația **Vânzători**

#Vânzător	#Produs
Schneider	Fen
Schneider	Breadbox
Jones	Cuțite
Jones	Fen
Jones	Breadbox
Jones	Umbrelă
Ferguson	Telescope
Ferguson	Fen
Ferguson	Lampă
Ferguson	Dulap

Relația **Branduri**

#Vânzător	#Brand
Schneider	Acme
Jones	Robusto
Ferguson	Robusto
Ferguson	Acme
Ferguson	Nimbus

Relația **Produse**

#Brand	#Produs
Acme	Fen
Acme	Breadbox
Acme	Lampă
Robusto	Cuțite
Robusto	Fen
Robusto	Breadbox
Robusto	Umbrelă
Robusto	Telescope
Nimbus	Dulap

Figura 17.18. Relații în FN5

Un tabel T este în a cincea formă normală (FN5) sau în formă normală de unire prin proiecție (FN/UP) dacă nu poate avea o descompunere fără pierderi în orice număr de tabele mai mici. Este exclus cazul în care toate tabelele mai mici după descompunere au aceeași cheie candidată ca și tabelul T.

17.3.7. Generalizări pentru procesul de normalizare

- **FN1** elimină redundanțele datorate **valorilor multiple** (non-atomare) ale atributelor.
- **FN2** elimină redundanțele datorate **dependenței parțiale** a atributelor care nu participă la o cheie, față de cheile unei relații. Se suprimă dependențele funcționale care nu sunt totale.
- **FN3** elimina redundanțele datorate **dependenței tranzitive** a atributelor față de cheia relației. Se suprimă dependențele funcționale tranzitive.
- **FNBC** elimină redundanțele datorate **dependenței funcționale**. Se suprimă dependențele în care partea stângă nu este o supercheie.
- **FN4** elimină redundanțele datorate **dependențelor multivalorice.** Se suprimă toate dependențele multivalorice care nu sunt și dependențe funcționale.
- **FN5** elimină redundanțele datorate dependenței ciclice. Se suprimă toate **dependențele de joncțiune** care nu sunt implicate de o cheie.
- **FNBC, FN4 și FN5** respectă regula că **orice determinant este o cheie.** Dependentul cu care se definește determinantul este o dependență funcțională pentru FNBC, o dependență multivalorică pentru FN4 sau o dependență de joncțiune pentru FN5.
- **Descompunerea** unei relații **FN2** în **FN3 conservă datele și dependențele**, pe când descompunerea unei relații FN3 în FNBC și, respectiv, a unei relații FNBC în FN4 conservă **doar datele**.

17.4. Normalizarea relațiilor prin metoda de sinteză

O relație în FN3 poate fi obținută și prin aplicarea **algoritmului de sinteză** a relațiilor. Algoritmul realizează o descompunere ce conservă dependențele. Algoritmul de sinteza construiește **o acoperire minimală *F*** a **dependențelor funcționale totale**. Se execută următori pași:

- Se constituie **relația universală** (mulțimea tuturor atributelor bazei de date).
- Se determină **toate dependențele funcționale** existente în relația universală.
- Se elimină atributele **redundate** din dependențele funcționale.
- Se elimină dependențele funcționale **redundante**.
- Mulțimea F este partiționata in **grupuri** F_i , astfel încât în fiecare grup F_i să fie incluse dependențele funcționale care au același membru stâng și să nu existe două grupuri cu același membru stâng.
- Se identifică perechile de chei echivalente (X,Y) în raport cu F (dacă ele există). Două mulțimi de atribute X și Y sunt chei echivalente dacă în mulțimea de dependențe există atât dependența X→Y, cât și dependența tip Y→X.
- Pentru fiecare pereche de chei echivalente se identifică grupurile F_i (care conține în partea stângă X) și F_j (care conține în partea stângă Y) și în baza lor se formează un grup nou F_{ij}, care va conține dependențele funcționale cu membrul stâng (X,Y). Se elimină grupurile F_i și F_j , iar în locul lor va fi luat grupul F_{ij}.
- Fiecare grup F_i și F_{ij} produce o schemă FN3.

Exemplu:

- Pasul 1. Fie este identificată următoarea mulțime universală de atribute {C, F, N, P, T, U}.
- Pasul 2. Fie în această mulțime universală există următoarele dependențe funcționale:
  f1: {F}→{N}; f2: {F}→{P}; f3: {P,F,N}→{U};
  f4: {P}→{C}; f5: {P}→{T}; f6: {C}→{T}; f7: {N}→{F}

- Pasul 3. Se suprimă atributele și dependențele funcționale. Din analiza dependențelor f1, f2 și f3 prin aplicarea axiomei de pseudo-tranzitivitate putem constata că atributele P și N sunt atribute redundate, iar dependența f3 capătă forma: {F}→{U}.
- Pasul 4. Se elimină dependențele funcționale redundante. Se observă că dependența f5 poate fi obținută din dependențele f4 și f6.

  f1: {F}→{N}; f2: {F}→{P}; f3: {F}→{U};
  f4: {P}→{C}; f6: {C}→{T}; f7: {N}→{F}

- Pasul 5. Gruparea dependențelor cu același membru stâng.

  G1={f1, f2, f3}; G2={f4}; G3={f6}; G4={f7}

- Pasul 6. Identificarea perechilor de chei echivalente (dacă ele există). În cazul nostru există două dependențe f1 și f7 care conțin perechea (F,N).
- Pasul 7. Regruparea dependențelor funcționale. Din G1 și G4 se obține F1.

  F1={f1, f2, f3, f7}; F2={f4}; F3={f6}

- Pasul 8 . Definirea schemelor relațiilor în FN3.

  R1={#F, N, P, U}; R2={#P, C}; R3={#C, T}

Întrebări de control:

1. Care sunt cele trei categorii de erori de actualizare ce pot apărea în cazul gestionării datelor dintr-un tabel aflat în formă nenormalizată (UNF)?
2. Ce condiții/reguli trebuie să îndeplinească o relație pentru a fi considerată în Prima Formă Normală (FN1)?
3. Care este diferența conceptuală fundamentală dintre A Doua Formă Normală (FN2) și A Treia Formă Normală (FN3)?
4. Ce este regula Casey-Delobel și cum ajută aceasta la obținerea o descompunere fără pierderi de informații?
5. Care este condiția suplimentară pe care trebuie să o îndeplinească o relație aflată în FN3 pentru a atinge Forma Normală Boyce-Codd (FNBC)?
6. Ce sunt dependențele multivalorice și în ce formă normală sunt ele eliminate pentru a evita redundanța de tip *many-to-many*?
7. Care este conceptul de bază ce stă la baza Forme Normale Cinci (FN5) și ce risc previne aducerea unei relații în FN5?
8. Care sunt cele două metode (sau scheme) principale utilizate pentru modelarea bazelor de date relaționale fără anomalii sau pierderi de informații?
9. Ce tip de conservare garantează descompunerea unei relații din FN2 în FN3, spre deosebire de descompunerea din FN3 în FNBC?
10. Care sunt pașii principali parcurși în algoritmul de sinteză a relațiilor pentru obținerea schemelor în FN3?

Capitolul 18. PROIECTAREA FIZICĂ A BAZELOR DE DATE

18.1. Etapele proiectării fizice a bazei de date

Proiectarea logică a bazei de date constă în rafinarea modelului conceptual prin normalizarea relațiilor bazei de date și transpunerea acestuia într-un model logic de date, ce corespunde unui anumit tip de SGBD țintă (relațional, ierarhic, în rețea sau orientat-obiect). Ca rezultat se obține un model complet al bazei de date, care permite definirea tuturor interpelărilor (vederilor) utilizatorilor și menținerea integrității bazei de date. Prin integrarea în modelul logic a schemelor externe pentru vederile utilizatorilor și a modelelor de date logice locale se obține **modelul de date logic global**. În această etapă se elimină problemele generate de utilizarea unor termeni identici pentru obiecte diferite sau termeni diferiți pentru aceleasi obiecte.

Proiectarea fizică a bazei de date reprezintă procesul de descriere a implementării bazei de date **pe mediile de stocare**. Sunt descrise **structurile de stocare şi metodele de acces** la date. **Proiectul fizic** al bazei de date efectuează o rafinare suplimentară a **modelului logic** prin transpunerea modelul logic într-un sistem de gestiune a bazelor de date relaționale. În această fază este obligatorie examinarea detaliată a modului în care utilizatorul accesează baza de date.

Crearea **modelului fizic** presupune următorii pași de luare a deciziilor:

- **Adaptarea structurii de stocare în mediul SGBD**
  - ✓ *transformarea tabelelor în fișiere;*
  - ✓ *determinarea ordinii coloanelor în tabele;*
  - ✓ *definirea cheilor și indecșilor pentru tabele;*
  - ✓ *determinarea blocărilor și constrângerilor pentru date;*
  - ✓ *determinarea spațiul liber necesar pentru stocarea datelor;*
  - ✓ *determinarea schemelor pentru transferul tabelelor în baza de date.*
- **Analiza evenimentelor din baza de date**
  - ✓ *revizuirea evenimentelor înlănțuite prin modelul logic;*
  - ✓ *alcătuirea unei liste de priorități ale evenimentelor pentru a înțelege importanța acestora;*
  - ✓ *identificarea regulilor care se aplică celor mai importante evenimente;*
  - ✓ *identificarea criteriilor de căutare pe baza cărora se accesează tabelul, în cazul fiecărui eveniment sau tabel accesat;*
  - ✓ *identificarea cerințelor de sortare pe baza cărora se accesează tabelul, în cazul fiecărui eveniment sau tabel accesat;*
  - ✓ *identificarea coloanelor destinație pe baza cărora se accesează tabelul, în cazul fiecărui eveniment sau tabel accesat;*
  - ✓ *estimarea numărului de rânduri căutate raportate la numărul total de rânduri din tabel pe baza cărora se accesează tabelul, în cazul fiecărui eveniment sau tabel accesat.*
- **Analiza căilor de acces ale evenimentelor**
  - ✓ *definirea procedurilor logice pe baza cărora un sistem de gestiune a bazelor de date poate selecta anumite rânduri, poate obține coloanele cerute, prelucrând acele coloane şi rânduri în maniera corespunzătoare.*
- **Reglarea opțiunilor invizibile**
  - ✓ *În cele mai multe sisteme de gestiune a bazelor de date opțiunile invizibile pe care le poate regla proiectantul se referă la scanări, ordinea rândurilor în tabele, indexarea coloanelor.*
- **Reglarea structurilor vizibile de date**
  - ✓ *Reglarea unei structuri vizibile de date constă în modificarea structurii datelor astfel, încât acestea s-ar putea să nu mai corespundă modelului logic. Cele mai obișnuite modalități de reglare a structurilor vizibile de date sunt: stocarea coloanelor ce reprezintă copii exacte ale altor coloane; stocarea coloanelor cu ajutorul cărora se efectuează calcule pe baza unei formule; stocarea coloanelor ce se obțin prin deducție (cu ajutorul unei reguli).*

18.2. Modelul fizic al bazei de date

Transpunerea modelului conceptual într-un model care să descrie structurile bazei de date reprezintă faza de **modelare fizica a datelor.** Acest proces mai este denumit și „mapare" (*mapping).* În cazul bazelor de date relaționale, din **modelul de date logic global** se obtin **modelele tabelelor relaționale**, se deduc **constrângerile impuse datelor** și **necesitățile de securitate** ale sistemului.

Fiecarei entități i se asociază un **tabel,** numit și relatie**,** pe care îl descriem prin **capul (titlul) de tabel**. Denumirea tabelului este dată, în general, de **forma de plural** a numelui entității. De exemplu, entitatea *ANGAJAT* se asociază cu tabelul *Angajați* (Figura 18.1). **Fiecare tabel** va avea tot atâtea coloane câte **atribute** are entitatea respectivă. În modelul tabelului se înscriu, pe lângă denumirile atributelor, **constrângerile de opționalitate** și **caracterul de cheie** al fiecărui atribut.

Tabelul **Angajați**						
Tip cheie	Opți-onalitate	Denumirea coloanei	Denumirea completă	Tip de date	Constrângeri	Alte condiționări
PK	*	CA	Codul angajatului	Char(5)	Secvență de 5 cifre între 10000 și 99999	CHECK
	*	Nume	Nume	Varchar(20)		INDEX
	*	Prenume	Prenume	Varchar(20)		
	*	Sex	Sexul	Char(1)	'F' - feminin, 'M'-masculin	CHECK
	*	DataN	Data nașterii	Datetime	zz-ll-aaaa	
	*	DataA	Data angajări	Datetime	zz-ll-aaaa	
	*	FuncAng	Funcția angajatului	Varchar(20)		
	o	EmailAng	Email angajatului	Varchar(50)	Corporativ N.P@ase.md	CHECK
FK	o	CD	Codul departamentului	Char(3)	Referă tabelul Departamente	Valori corelate sau Null

Figura 18.1. Proiectarea structuri tabelului Angajați

Este util ca în modelul fizic să se folosească **denumiri prescurtate** ale atributelor, astfel încât transcrierea modelului fizic în limbaj de programare, să se facă mai simplu. În acest caz în modelul fizic este necesar să se includă și o descriere a fiecărei denumiri prescurtate. De asemenea, în modelul fizic se include **tipul datelor** pentru **fiecare atribut** și **dimensiunea dorită**, opțiunile de tranzactionare, valorile implicite (DEFAULT) și diferite condiționări (Figura 18.1).

Pentru anumite tabele des utilizate pentru crearea viziunilor utilizatorilor, se recomandă specificarea **indecșilor**. **Indexul** este un element specific al bazei de date prin care se accelerează procesul de căutare și de extragere a informațiilor din tabelele bazei de date. Crearea unui index implică ocuparea unui **spațiu de memorie suplimentar** și din această cauză este indicat să se creeze un index doar într-un caz bine justificat.

Modelul fizic al bazei de date se elaborează în baza diagramei logice în dreptunghi ajustată la exigențele modelului relațional de date (vezi Figura 15.29). Această diagramă se obține prin aplicarea regulilor de mapare aplicate schemei conceptuale a bazei de date și au fost examinate detaliat în paragraful 15.6. Suplimentar la aceste reguli de mapare în documentația tehnică a proiectului fizic se specifică anumite condiționări care nu pot fi indicate în diagrama E-R în dreptunghi.

De exemplu, dacă un supertip de entitate are mai multe atribute comune decât specifice fiecărui subtip, atunci atributele subtipurilor devin opționale, urmând ca la completarea tabelului, să se înscrie date doar în coloanele corespunzătoare atributelor unui singur subtip. Apare aici o constrângere de tip XOR, care pentru cazul a doar 2 suptipuri cu câte un atribut se exprima printr-o condiționare de forma: CHECK (atribut1 IS NOT NULL AND atribut2 IS NULL) OR atribut1 IS NULL AND atribut2 IS NOT NULL).

Un alt exemplu este cazul relațiilor cu caracter **netransferabil (**marcate în diagrama E-R cu un romb). În cazul lor este necesar de indicat în modelul fizic o constrângere referitoare la nemodificarea valorilor înscrise în tabel **pe coloana cheii externe**. Aceasta se traduce prin **interzicerea actualizării** sau **ștergerii** datelor de pe acea coloană prin definirea reacțiilor la aceste acțiuni (ON UPDATE NO ACTION, ON DELETE NO ACTION) sau prin scrierea unor declanșatoare DML care vor anula aceste acțiuni.

Aceste două exemple demonstrează cât de important este menționarea în modelul fizic a **tuturor detaliilor impuse de regulile de afaceri** pentru ca programatorii sa poată înscrie toate constrângerile impuse de ele.

După proiectarea structurii tuturor tabelelor normalizate ale bazei de date se obține modelul fizic al bazei de date (Figura 18.2), care ulterior se implementează sub formă de schemă a bazei de date cu ajutorul instrucțiunilor limbajului DDL implementat în cadrul SGBD selectat pentru proiectare. Modelul fizic se completează cu regulile de operare definite în dicționarul de date, care descriu diferite aspecte ale prelucrării datelor în domeniul modelat și care vor servi pentru proiectarea corectă a tranzacțiilor în cadrul bazei de date. Cu ajutorul instrucțiunilor limbajului DML al SGBD selectat se elaborează prototipul bazei de date cu schema proiectată, care este utilizat pentru testare și identificarea erorilor de proiectare, precum și pentru elaborarea și testarea tuturor interpelărilor utilizatorilor conform caietului de sarcini elaborat la etapa de analiză a cerințelor funcționale.

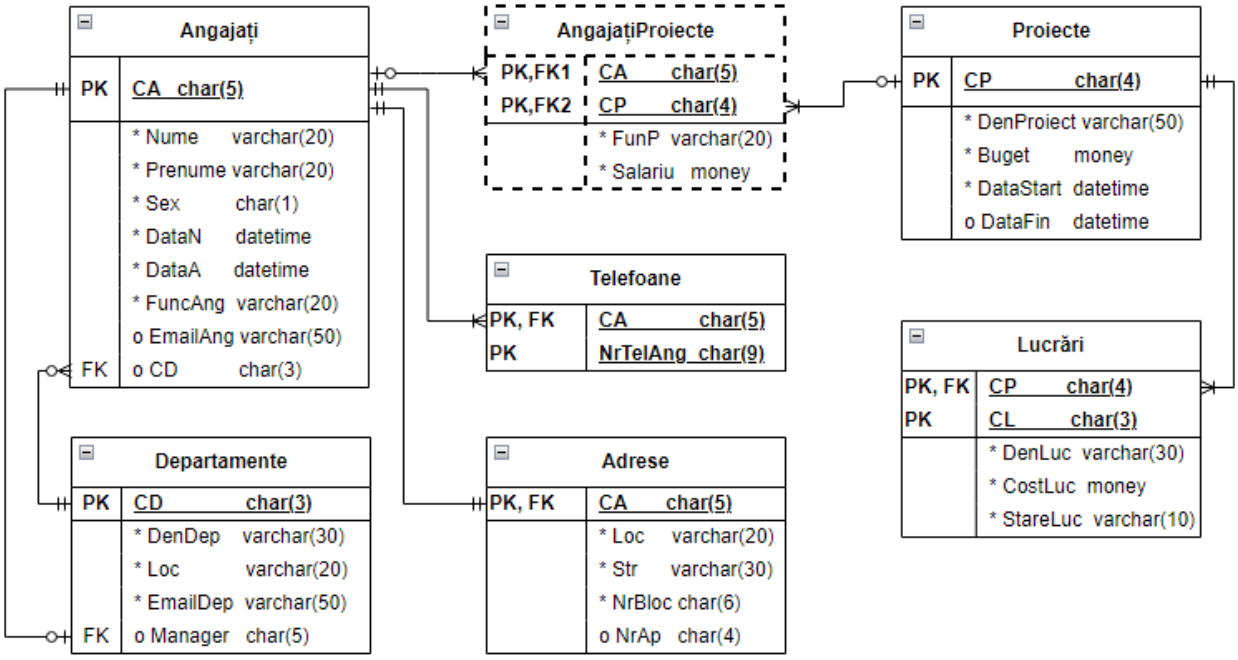


Figura 18.2. Modelul fizic al bazei de date relaționale

Securizarea bazei de date impune acordarea de **drepturi** și revocarea lor pentru diferiți utilizatori sau pentru diferite grupuri de utilizatori. Utilizatorii din același grup sunt asociați cu un anumit **rol** creat distinct în structura bazei de date pentru a specifica în mod unitar drepturile de accesare a obiectelor bazei de date și de manipulare a datelor pentru utilizatorii cu același rang. Administrarea conturilor utilizatorilor bazei de date se face mai rapid și mai eficient prin definirea

acestor roluri, decât individual, pentru fiecare utilizator în parte. Totuși, în unele cazuri, definite de regulile de afaceri, este necesar de definit anumite drepturi de acces la anumite obiecte ale bazei de date pentru utilizatori concreți. Acest lucru se realizează în aplicațiile cu baze de date prin utilizarea instrucțiunilor de gestionare a accesului (GRANT, DENY, REVOKE).

De asemenea, pot fi specificate **scheme diferite ale bazei de date** pentru diferite **categorii de utilizatori**, pentru limitarea accesului la informația confidențială. În documentația modelului fizic trebuie să fie **specificate** grupurile de utilizatori și utilizatorii distincți cu drepturile de acces la obiectele bazei de date și de efectuare a tranzacțiilor.

18.3. Analiza CRUD

Analiza CRUD (prescurtare a secvenței CREATE, RETRIEVE, UPDATE, DELETE), face referire la posibilitatea de a crea structura bazei de date, de a regăsi, de a modifica sau actualiza datele înregistrate, sau de a le șterge, atunci când își pierd valoarea sau când devin perimate. Proiectarea unei baze de date trebuie făcută astfel încât să fie posibile aceste patru operațiuni, pe serverul de baze de date.

Prin analiza CRUD se urmărește dacă modelele elaborate și implementarea bazei de date permit efectuarea tuturor operațiilor amintite, asupra tuturor datelor și în mod consistent, cu menținerea bazei de date într-o stare coerenta. Prin aceasta, analiza CRUD **validează modelul logic** creat pentru baza de date proiectată.

Prin efectuarea analizei CRUD se verifică dacă nu s-au omis din modelul bazei de date entități, atribute sau relații care sunt necesare aplicației descrise de baza de date. De exemplu, sunt multe relații între tabele, exprimate prin atributele de tip cheie străină (externă). Omiterea unei constrângeri de tip cheie străină, conduce la rezultate eronate ale unei interogări a bazei de date, care vizează datele conținute în tabelele relaționate.

Se verifică, deasemenea, dacă modelul creat nu include unele aspecte a caror prezență nu se justifică, având în vedere cerințele clientului specificate în caietul de sarcini pentru baza de date proiectată. Comunicarea permanentă dintre proiectant și client este esențială pentru crearea unui model complet și corect dimensionat al bazei de date și realizarea integrală a tuturor cerințelor funcționale identificate la etapa de analiză a cerințelor față de baza de date.

Daca transpunerea modelului conceptual al bazei de date în modelul fizic și, ulterior, implementarea bazei de date la comandatar nu sunt facute riguros, există riscul ca anumite prevederi din documentația de proiect să nu fie respectate sau îndeplinite și să se ajungă chiar la încalcări ale unor „reguli ale afacerii”, specifice activității descrise în baza de date.

Analiza CRUD trebuie făcută cu personal specializat, în mod sistematic, urmărindu-se pas cu pas fiecare cerință din documentația de proiect. În specificațiile clientului trebuie urmarite anumite cuvinte-cheie care exprimă cerințele acestuia: ***introducere, înregistrare, încărcare, importare, exportare, vizualizare, actualizare, raport, listare, tipărire, citire, căutare, modificare, schimbare, ștergere etc.*** Toate acestea se traduc în operații specifice efectuate cu datele, de tip CRUD, și trebuie să poată fi realizate pe baza modelului creat.

Analiza CRUD a modelului urmărește ca asupra **oricarei entități** din model **să se poată efectua cele patru operații** pentru a nu avea aspecte nemodelate ale aplicației. **Testarea** cu un **set complet** de date reale din domeniul proiectat, care să creeze toate situațiile reale tipice, din **perspectiva tuturor tipurilor de utilizatori** ai bazei de date și a **tuturor regulilor de afaceri**, precum și corectarea aspectelor nedorite ale bazei de date va finaliza proiectarea acesteia.

Întrebări de control:

1. Care este diferența principală dintre proiectarea logică și proiectarea fizică a unei baze de date?
2. Care sunt principalii pași de luare a deciziilor implicați în procesul de creare a modelului fizic?
3. Ce elemente suplimentare se includ în descrierea structurii unui tabel în cadrul modelului fizic?
4. De ce este recomandată specificarea indecșilor doar în cazuri bine justificate și ce impact au aceștia asupra resurselor bazei de date?

5. Cum se tratează în modelul fizic constrângerile specifice, cum sunt relațiile cu caracter netransferabil sau supertipurile/subtipurile de entități?
6. Prin ce mecanisme și instrucțiuni SQL se realizează securizarea bazei de date și gestionarea drepturilor de acces pentru diferitele grupuri de utilizatori?
7. Ce reprezintă analiza CRUD și care este rolul ei în validarea modelului bazei de date?
8. Ce cuvinte-cheie din specificațiile clientului indică necesitatea realizării operațiilor de tip CRUD asupra datelor?

Capitolul 19. GESTIONAREA TRANZACȚIILOR

19.1. Tranzacții în baza de date

Un SGBD deservește mai mulți utilizatori care accesează o bază de date. Accesul concurent al utilizatorilor este asigurat prin capacitatea de multiprogramare a sistemului de operare al calculatorului gazdă, care permite execuția concurentă a mai multor procese. Execuția concurentă a proceselor are loc fie prin partajarea timpului de execuție al procesorului între mai multe procese, fie executarea în paralel a proceselor pe mai multe procesoare ale sistemului. Accesul concurent al mai multor utilizatori la datele din baza de date necesită tehnici de menținere a consistenței și a integrității datelor memorate.

O **tranzacție** este o unitate logică de prelucrare indivizibilă (atomică) a datelor unei baze de date prin care se asigură consistența acesteia. O tranzacție trebuie să asigure consistența bazei de date indiferent dacă a fost executată individual sau concurent cu alte tranzacții, chiar și în condiții de forță majoră, când apar diferite erori logice în codul aplicației sau defecte ale sistemului hardware.

Fie avem următoarele relații [11]:
Zboruri(*#IdCursa, AeroportPlecare, AeroportSosire, Data, NrLocuriLibere*)
Pasageri(*#IdPasager, Nume, Prenume, Email, NrPaşaport, NrCreditCard*)
Rezervări(*#IdRezervare, IdPasager, IdCursa, NrBileteRezervate*)
Facturi(*#IdFactura, IdPasager, DataFacturarii, Pret*)

Atunci când se face o tranzacție de cumpărare a unui bilet la o cursă oarecare sistemul efectuează mai multe operații:

1. Se inserează o linie nouă în tabelul **Pasageri**, care conține datele pasagerului.
2. Se verifică existența locurilor libere la cursa dorită și posibilitatea rezervării numărului de bilete solicitate, și se face propriu-zis rezervarea, prin inserarea unei linii noi în tabelul **Rezervări**, linie care conține numărul de identificare al pasagerului, numărul de identificare al cursei și numărul biletelor rezervate. Se modifică valoarea atributului *NrLocuriLibere* din rândul corespunzător pentru cursa solicitată din tabelul **Zboruri** după formula *NrLocuriLibere= NrLocuriLibere-NrBileteRezervate.* Dacă locuri libere nu sunt, sau dacă se solicită mai multe bilete decât locuri libere existente, atunci rezervarea este imposibilă, se afișează anunțul respectiv și se anulează efectul tranzacția din punctul 1.
3. La achitarea biletului se inserează o linie în tabelul **Facturi**. Această înregistrare este folosită pentru a tipări o factură, care va fi folosită pentru plata în numerar sau va fi trimisă companiei de cărți de credit. Dacă pasagerul renunță să achite rezervarea sau dacă este depășită limita de timp pentru realizarea unei tranzacții de bancare, atunci sistemul afișează mesajul corespunzător și anulează efectele tranzacțiilor precedente aplicate tabelelor **Zboruri**, **Rezervări** și **Pasageri** din punctele 1 și 2.
4. Se emite biletul (pornind de la datele din rezervare şi factura corespunzătoare) și se transmite biletul pe adresa Email a pasagerului.

Dacă sistemul se defectează după ce s-a executat pasul 2, sa făcut o rezervare, dar biletul nu a fost facturat şi nici emis. Mai rău, dacă defecțiunea are loc după ce s-a executat pasul 3, atunci banii au fost decontați de pe cardul bancar și clientului i se trimite factura, dar el nu a primit biletul. Astfel de situații sunt inacceptabile. Chiar dacă nu se defectează sistemul, pot să apară probleme dacă baza de date este accesată concurent de mai mulți utilizatori. De exemplu, dacă doi clienți diferiți concomitent vor să procure ultimul bilet sau selectează același loc în avion.

Dacă toate acțiunile aferente unei rezervări ar fi grupate ca o operație indivizibilă (atomică), o parte din problemele arătate mai sus ar dispărea. O operație indivizibilă de acces la baza de date este numită tranzacție şi ea fie execută cu succes toate acțiunile şi se termină cu o validare a

modificărilor efectuate asupra bazei de date (**commit**), fie nu poate efectua (din diferite motive) toate acțiunile și este abandonată și anulată (**rollback**).

În cazul în care o tranzacție a efectuat cu succes toate acțiunile și este validată, în momentul validării toate modificările efectuate asupra bazei de date devin permanente, vor fi vizibile altor tranzacții și nu mai pot fi anulate. Până în momentul validării, modificările efectuate de tranzacție au un caracter provizoriu, nu sunt vizibile altor tranzacții și pot fi oricând revocate. În cazul abandonării unei tranzacții, execuția acesteia este oprită și efectele tuturor acțiunilor executate până în momentul abandonării sunt anulate, astfel încât baza de date este adusă în starea de dinaintea lansării tranzacției.

Pentru demonstrarea mecanismului de control al concurenței și reconstituirii datelor în timpul tranzacțiilor vom examina operațiile cu datele **read(X)** (citește articolul de date (*data item*) A într-o variabilă X) și **write(X)** (scrie valoarea variabilei X în articolul de date A). Unitatea de transfer a datelor între discul magnetic și memoria operativă a sistemului este un bloc, care corespunde unui sector de pe disc. În general, un articol de date este un câmp care memorează valoarea unui atribut dintr-o înregistrare (tuplu), dar poate fi o întreagă înregistrare sau chiar un bloc întreg.

Tranzacțiile lansate de diferiți utilizatori se pot executa concurent și este posibil să actualizeze aceleași articole ale bazei de date. Dacă execuția concurentă a tranzacțiilor este necontrolată, este posibil ca baza de date să ajungă într-o stare inconsistentă (incorectă), chiar dacă fiecare tranzacție în parte este un program corect, a fost executată corect și nu au apărut defecte de funcționare ale sistemului. Mai jos sunt examinate posibilele anomalii de execuție concurență a tranzacțiilor.

Actualizarea pierdută (*lost update*) este o anomalie care apare când două tranzacții T1 și T2 actualizează același articol A al bazei de date prin partajarea timpului de execuție a procesorului (Figura 19.1, a). Valoarea memorată în articolul A în baza de date este `X+M`, în locul valorii corecte `X-N+M` (`X` fiind valoarea inițială a articolului, înainte de lansarea tranzacțiilor). Actualizarea efectuată de tranzacția T1 a fost pierdută, dat fiind faptul că tranzacția T2 folosește valoarea inițială a articolului `X`, înainte ca valoarea calculată de T1, egală cu `X-N`, să fie memorată în baza de date.

Timpul	T1	T2
t1	`read(X)`	
t2	`X=X-N`	
t3		`read(X)`
t4		`X=X+M`
t5	`write(X)`	
t6	`read(Y)`	
t7		`write(X)`
t8	`Y=Y+N`	
t9	`write(Y)`	

a)

T1	T2
`read(X)`	
`X=X-N`	
`write(X)`	
	`read(X)`
	`X=X+M`
	`write(X)`
`read(Y)`	
`abort`	

b)

Figura 19.1. Apariția anomaliilor la execuția concurentă necontrolată a două tranzacții
a) actualizare pierdută; b) citire improprie

Citirea improprie (*dirty read*) este o anomalie care apare când una din tranzacții este abandonată, iar altă tranzacție concurentă a utilizat articolele modificate, înainte de readucerea acestora la valoarea inițială (Figura 19.1, b). Tranzacția T1 a modificat articolul A prin inscrierea valorii noi a variabilei X, dar apoi este abandonată dintr-o cauză oarecare. Însă tranzacția T2 a reușit să citească valoarea modificată a articolului și a utilizat articolul modificat, înainte ca acesta să fie readus la valoarea sa inițială. Rezultatul celor două tranzacții este valoarea `X-N+M`, în locul valorii corecte `X+M`.

Citirea irepetabilă (*nonrepetable read*) este anomalia care duce la inconsistența datelor în cazul când o tranzacție T1 realizează citirea unui articol de două ori, iar între cele două citiri o altă tranzacție T2 a modificat chiar acel articol. În această situație tranzacția T1 primește două valori diferite ale aceluiași articol.

Citirea fantomă (*phantom read*) este asemănătoare citirii irepetabile și se manifestă prin faptul că o tranzacție prelucrează un set de linii rezultat al unei interogări, iar în timpul acestei prelucrări o altă tranzacție a inserat sau a șters o linie care satisface condiția interogării respective. În acest caz pot apărea sau dispărea tupluri din mulțimea de linii ale rezultatului, așa numite tupluri fantome. Diferența dintre citirea fantomă și citirea irepetabilă este că citirea fantomă apare dacă tranzacțiile concurente modifică numărul de linii utilizate de tranzacția curentă (prin instrucțiunile `INSERT` sau `DELETE`), iar citirea irepetabilă apare dacă tranzacțiile concurente modifică valorile din liniile existente și prelucrate de tranzacția curentă (prin instrucțiunea `UPDATE`).

Anomaliile examinate mai sus apar la execuția concurentă necontrolată a două sau mai multor tranzacții și necesită realizarea unui controlul al concurenței în baza de date. Consistența datelor din baza de date trebuie să fie asigurată permanent, chiar și în condițiile când o tranzacție nu poate fi finalizată cu succes datorită unor erori software sau hardware.

Tranzacțiile în baza de date au un șir de proprietăți identificate prin acronimul ACID: **A**tomicitate, **C**onsistență, **I**zolare, **D**urabilitate.

Atomicitatea (*atomicity*) tranzacției reprezintă faptul că ea este *o unitate de execuție indivizibilă.* Dacă o tranzacție este întreruptă dintr-o cauză oarecare, atunci SGBD va asigura, după eliminarea cauzei care a întrerupt executarea tranzacției, fie completarea și validarea tranzacției, fie abandonarea tranzacției și anularea tuturor efectelor acțiunilor efectuate de tranzacție până în momentul întreruperii.

Consistența (*consistency*) tranzacției înseamnă că ea efectuează modificări corecte ale bazei de date. Cu alte cuvinte, o tranzacție transformă baza de date dintr-o stare consistentă în altă stare consistentă. În stările intermediare prin care trece o bază de date în cursul unei tranzacții, este posibil să existe unele inconsistențe, dar starea finală în care ajunge baza de date după execuția unei tranzacții va fi una consistentă. Starea unei baze de date este consistentă dacă respectă toate constrângerile de integritate implicite sau explicite. Este responsabilitatea proiectanților de aplicații cu baza de date să elaboreze tranzacții cu proceduri și instrucțiuni corecte, care să producă o stare consistentă a bazei de date.

Izolarea (*isolation*) tranzacției constă în faptul că ea face vizibile modificările efectuate în baza de date numai după ce a fost validată de sistem (*committed*). Dacă în paralel sunt executate alte tranzacții concurente, atunci acestea nu "văd" modificările parțiale efectuate de tranzacția respectivă până în momentul validării tranzacției. Proprietatea de izolare a tranzacțiilor este importantă deoarece elimină fenomenul de abandonare în cascadă a tranzacțiilor. Dacă rezultatele parțiale ale unei tranzacții ar fi vizibile altor tranzacții concurente înainte de validarea acesteia și dacă tranzacția este abandonată sau anulată (*rollback*), atunci toate tranzacțiile concurente care au accesat rezultatele parțiale ale acesteia trebuie să fie anulate, iar aceste operații de anulare pot produce, la rândul lor, alte anulări ș.a.m.d. Acest fenomen de anulare în cascadă creează un efect de "domino". Izolarea este impusă prin metode de control al concurentei pe diferite niveluri de izolare.

Durabilitatea (*durability)* sau **permanența** (*permanency*) tranzacției constă în faptul că după validarea unei tranzacții modificările efectuate de aceasta în baza de date nu vor mai fi pierdute datorită unor defectări ulterioare a sistemului. Proprietatea de durabilitate este asigurată prin metode de reconstituire a bazei de date (*database recovery*) ale SGBD.

Dacă operațiile de acces la baza de date ale unei tranzacții sunt numai operații de citire, atunci această tranzacție este o tranzacție de citire (*read-only transaction*) și controlul concurenței unor astfel de tranzacții este unul mai simplu. În cazul când se realizează tranzacții cu citire și scriere a datelor (*read-write transactions*) sunt necesare tehnici de control al concurentei mult mai complexe.

Operațiile efectuate de o tranzacție și înregistrate de administratorul de refacere (*recovery manager*) al SGBD, care este responsabil de asigurarea celor patru proprietăți importante ale tranzacțiilor (ACID), sunt următoarele (Figura 19.2):

- `begin` – începutul unei tranzacții;
- `read` – operația de citire a datelor;
- `write` – operația de scriere a datelor;
- `end` – terminarea operației de citire sau scriere a datelor;
- `commit` – validarea tuturor modificărilor efectuate în baza de date de către tranzacție și vizibilitatea lor pentru alte tranzacții; modificările efectuate nu pot fi anulate sau pierdute;
- `rollback` – abandonarea tranzacției și anularea modificărilor aplicate în baza de date;
- `undo` – anularea efectului unei singure operații de citire sau scrie din cadrul tranzacției;
- `redo` – executarea din nou a unei operații din tranzacție.

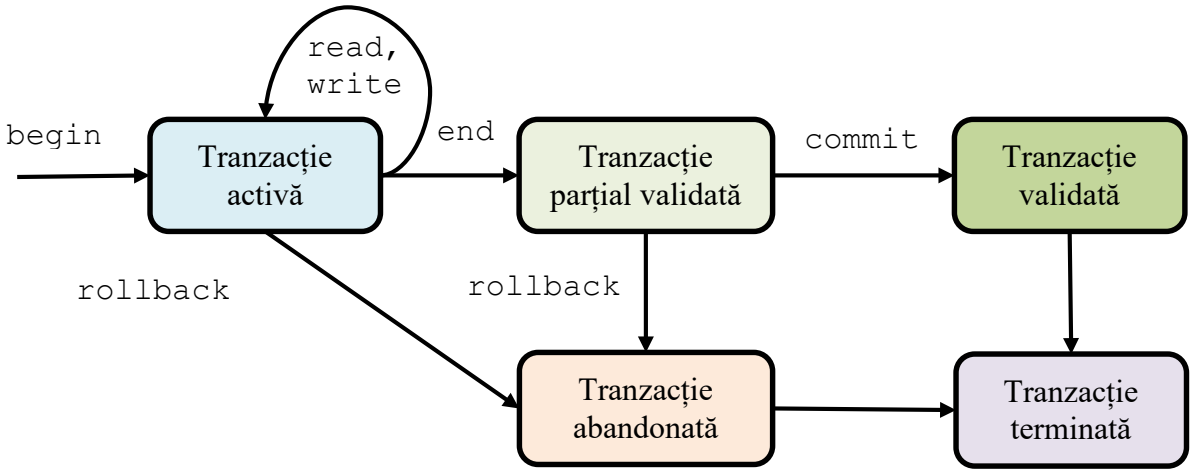


Figura 19.2 Diagrama de stare a unei tranzacții

În starea activă a tranzacției se ajunge ca urmare a lansării unei tranzacții prin operația `begin`, iar execuția corectă a operațiilor `read` sau `write` nu modifică această stare. Atunci când o tranzacție se termină, ea trece în starea de tranzacție parțial validată în care se efectuează operații de verificare impuse de tehnicile (protocoalele) de control al concurenței și de reconstituire a bazei de date. Dacă ambele verificări sunt îndeplinite cu succes, atunci tranzacția este validată prin execuția operației `commit` și este trecută în starea de tranzacție validată. Dacă una sau ambele verificări eșuează, atunci tranzacția este trecută în categoria de tranzacție abandonată prin executarea operației `rollback`. Starea de tranzacție terminată este consecința imediată a atingerii uneia din stările de tranzacție validată sau de tranzacție abandonată și nu necesită nici o altă operație pentru efectuarea acestei tranziții. În cursul stării de tranzacție activă orice tranzacție poate fi abandonată, dacă diferite verificări nu se finalizează cu succes.

Pentru refacerea bazei de date în condițiile în care unele tranzacții sunt abandonate sau în care pot apărea defecte de funcționare, SGBD creează un *fișier de jurnalizare*, în care memorează operațiile efectuate de tranzacții. Fișierul de jurnalizare este memorat pe disc și nu este afectat de diferite erori de execuție, cu excepția unei defectări fizice a discului. Fișierul de jurnalizare este salvat periodic pe un suport magnetic de rezervă ca o măsură de siguranță.

În fișierul de jurnalizare (*log file*) se înregistrează operațiile efectuate de fiecare tranzacție, care se identifică printr-un identificator unic (`T`) generat de sistem. Prima înregistrare referitoare la o tranzacție cu identificatorul `T` este înregistrarea de start a tranzacției: `[begin,T]`. La fiecare operație `write(X)` executată de o tranzacție `T` în fișierul de jurnalizare se memorează o înregistrare de tipul: `[write,T,X,vechea_valoare,noua_valoare]`, dacă pentru refacerea datelor se folosesc operații `undo`, sau o înregistrare de tipul `[write,T,X,noua_valoare]`, dacă se folosesc operații `redo`. Sistemele care nu evită abandonarea în cascadă a

tranzacțiilor înregistrează în fișierul de jurnalizare și operațiile de citire `read(X)`, printr-o înregistrare de tipul `[read,T,X,valoare]`. Pentru fiecare tranzacție `T` în fișierul de jurnalizare se mai memorează starea de terminare a tranzacției: validată (printr-o înregistrare `[commit,T]`) sau anulată (printr-o înregistrare `[rollback,T]`). După executarea validării și înregistrarea ei în fișierul de jurnalizare efectul tranzacției este permanent memorat în baza de date.

Dacă sistemul se defectează, atunci la reluarea execuției după identificarea și înlăturarea defectului SGBD va aduce baza de date într-o stare consistentă prin examinarea fișierului de jurnalizare. Datorită faptului că sunt înregistrate toate operațiile în timpul tranzacției este posibil de a anula efectul tuturor operațiilor de scriere efectuate de o tranzacție care a fost lansată, dar nu a fost validată, parcurgând în ordine inversă fișierul de jurnalizare și înlocuind valoarea existentă a articolului modificat cu vechea valoare, memorată în fișierul de jurnalizare. În cazul abandonării în cascadă, se vor abandona și tranzacțiile care au citit valori modificate de tranzacțiile abandonate.

19.2. Tehnici de control al concurenței

Controlul execuției concurente a mai multor tranzacții este necesar pentru a asigura proprietățile de atomicitate, izolare și durabilitate a tranzacțiilor, asigurând consistența datelor memorate în baza de date. Controlul concurenței poate fi realizat prin protocoale (set de reguli) impuse tranzacțiilor. Cele mai utilizate tehnici de control al concurentei sunt cele bazate pe blocarea datelor prin intermediul **zăvoarelor** (*locks*) și cele bazate pe **mărci de timp** (*timestamps*).

Protocoalele de control al concurenței sunt implementate de sistemele de gestiune a bazelor de date astfel încât programatorii de aplicații nu operează în mod explicit cu zăvoare sau mărci de timp, ci stabilesc opțiunile prin care SGBD adoptă anumite tehnici de control al concurenței.

Controlul concurenței tranzacțiilor prin blocare (*concurrency control using locking technique*) se realizează prin utilizare zăvoarelor. Un zăvor (*lock*) este o variabilă asociată cu un articol al unei baze de date, care descrie starea articolului în raport cu operațiile care se pot aplica acestui articol. Pentru un articol `X` zăvorul care controlează accesul la el se notează `L(X)`. În sistemele de gestiune a bazelor de date se pot utiliza două tipuri de zăvoare: **zăvoare binare** și **zăvoare cu stări multiple**.

Un zăvor binar (*binary lock*) poate avea două stări: liber (*free*, *unlocked*) și blocat (*busy, locked*). Asupra unui zăvor `L(X)` se pot executa două operații: operația de blocare `lock(X)` și operația de deblocare `unlock(X)`. Dacă zăvorul articolului `X` este deja blocat (`L(X)=0`), atunci operația de blocare `lock` pune în așteptare tranzacția până când zăvorul se eliberează. Dacă zăvorul este liber (`L(X)=1`), atunci tranzacția achiziționează zăvorul, trecându-l în starea ocupat, după care execută operațiile necesare asupra articolului `X`. După terminarea operațiilor asupra articolului X tranzacția eliberează zăvorul prin execuția operației `unlock`. Dacă zăvorul a fost ocupat, atunci tranzacția așteaptă până ce acesta este eliberat (de o altă tranzacție, care și-a terminat operațiile de acces), iar apoi execută aceeași secvență de operații: blocarea zăvorului, execuția operațiilor de acces și eliberarea zăvorului.

Operația de blocare a zăvorului trebuie să fie executată ca o operație indivizibilă. Această cerință este implementată prin instrucțiunile speciale ale procesorului (de tip `TestAndSet(TS)`). Se pot preciza următoarele reguli (protocoale) pe care trebuie să le urmeze fiecare tranzactie care folosește zăvoare binare:

- O tranzacție `T` trebuie să lanseze o operație de blocare a zăvorului asignat articolului `X` (`lock(X)`) înainte de a efectua orice operație de citire (`read(X)`) sau de scriere (`write(X)`) a articolului `X`.
- O tranzacție `T` trebuie să elibereze zăvorul unui articol `X` prin operația `unlock(X)` după ce a efectuat toate operațiile de citire (`read(X)`) sau de scriere (`write(X)`) a articolului `X`.
- O tranzacție `T` nu poate cere un zăvor pe care îl deține deja.
- O tranzacție `T` nu poate elibera un zăvor pe care nu-l deține.

Între operația de blocare (`lock(X)`) lansată de o tranzacție `T` și operația de eliberare (`unlock(X)`) a unui zăvor `L(X)` tranzacția `T` *deține* zăvorul respectiv. În cazul zăvoarelor binare, cel mult o tranzacție poate deține un zăvor la un moment dat și nici o altă tranzacție nu poate accesa articolul respectiv. Se poate spune că zăvoarele binare realizează un *mecanism de excludere mutuală*, prin care accesul unei tranzacții la un articol (tranzacția care deține zăvorul acelui articol) exclude accesul altor tranzacții la acel articol. Fiind o tehnică de control al concurenței foarte generală și relativ simplu de utilizat, tehnica zăvoarelor binare este prea restrictivă și limitează în mod nejustificat concurența în execuția tranzacțiilor. De exemplu, mai multe tranzacții pot efectua operații de citire în mod concurent, fără ca acest lucru să afecteze consistența bazei de date, dar acest lucru este interzis în tehnica zăvoarelor binare. De aceea, multe sisteme de gestiune a bazelor de date utilizează zăvoare cu stări multiple.

O marcă de timp *(timestamp*), este un identificator unic al unei tranzacții, creat de sistemul de gestiune a bazei de date, care se bazează pe timpul de start al tranzacției. O marcă de timp se poate crea fie folosind valoarea curentă a ceasului sistemului de operare, fie folosind un contor care este incrementat la fiecare asignare a unei noi mărci, în ordinea de lansare a tranzacțiilor. Unei tranzacții `T` i se asociază o marcă de timp unică, notată `TS(T)`. Pentru fiecare articol `X` al bazei de date se definesc două mărci de timp:

- `read_TS(X)` - marca de timp de citire a articolului `X`; aceasta este cea mai mare marcă de timp dintre toate mărcile de timp ale tranzacțiilor care au citit articolul `X`.
- `write_TS(X)` - marca de timp de scriere a articolului `X`; aceasta este cea mai mare marcă de timp dintre toate mărcile de timp ale tranzacțiilor care au scris în articolul `X`.

Serializabilitatea planificărilor se obține dacă se impun anumite condiții ordinii de accesare a articolelor de mai multe tranzacții concurente, în funcție de mărcile de timp ale acestora.

Refacerea unei baze de date după producerea unui defect (*database recovery*) înseamnă aducerea bazei de date într-o stare precedentă corectă, din care, eventual, se poate reconstrui o nouă stare corectă și cât mai apropiată de momentul apariției defectului. Pentru operațiile de refacere se folosește fișierul de jurnalizare sau o copie de rezervă a bazei de date (*database backup*). Dacă baza de date nu este distrusă fizic, dar a devenit inconsistentă datorită unui defect, atunci strategia de refacere constă în a anula modificările care au produs inconsistența (prin operații `undo`) sau de a executa din nou anumite modificări care s-au pierdut (prin operații `redo`). În acest caz nu este necesară copia de rezervă și se folosește starea actuală a bazei de date și fișierul de jurnalizare. Dacă baza de date a fost puternic distrusă, datorită unei defectări serioase a discului, atunci se restaurează starea bazei de date din copia de rezervă, după care se reconstruiește o stare cât mai actuală prin reaplicarea tuturor tranzacțiilor validate existente în fișierul de jurnalizare, dacă acesta nu a fost deteriorat, sau din ultima copie salvată a fișierului de jurnalizare.

Întrebări de control:

1. Ce este o tranzacție într-o bază de date?
2. Care este diferența dintre operațiile commit și rollback în cadrul procesului de gestionare a unei tranzacții?
3. Care sunt anomaliile principale ce pot apărea în cazul execuției concurente necontrolate a tranzacțiilor?
4. Ce reprezintă acronimul ACID și care sunt proprietățile fundamentale ale tranzacțiilor definite de acesta?
5. Care este rolul fișierului de jurnalizare (*log file*) și ce tipuri de înregistrări sunt memorate în acesta pentru a permite refacerea datelor?
6. Prin ce stări succesive trece o tranzacție de la lansare (begin) până la finalizare, conform diagramei sale de stare?
7. Cum funcționează tehnicile de control al concurenței bazate pe zăvoare (*locks*) și care este diferența de aplicare dintre zăvoarele binare și cele cu stări multiple?
8. În ce constă tehnica de control al concurenței bazată pe mărci de timp (*timestamps*) și cum sunt definite mărcile de citire (read_TS) și scriere (write_TS) pentru un articol de date?

BIBLIOGRAFIE

1. Entity Relationship Diagram Symbols. Accesibil la: https://www.conceptdraw.com/How-To-Guide/erd-entity-relationship-diagram-symbols
2. Brăescu F.-C. Baze de date. Disponibil: http://www.bd.ac.tuiasi.ro/Doc/curs/
3. Cotelea V., Cotelea M. Baze de date. – Chișinău: Tipografia Centrală, 2016.
4. Cotelea V. Algebra relațională şi limbajul SQL. - Chişinău:Vizual Design, 2013. - 284 p
5. Cotelea V. Baze de date relaționale: Proiectare logică – Chișinău, ASEM, 1997.
6. Lupu V. Baze de date. Curs pentru învățământ la distanță. – Suceava, Universitatea ”Ștefan cel Mare”, 2010.
7. Oprea S., Oprea D. Limbajul SQL. Suport de curs. – Chișinău: SEP ASEM, 2023. – 239 p.
8. Oprea S. Proiectarea bazelor de date. Suport de curs. – Chișinău: SEP ASEM, 2023. –162 p.
9. Rădulescu F., Boicea A. Baze de date online. - București : Editura Academiei Oamenilor de Știință din România, 2011, 92 p., ISBN 978-606-8371-13-9
10. Rădulescu F. Baze de date. Note de curs. Disponibil: http://bdfr.cs.pub.ro/
11. Rațiu I.G. Baze de date. Disponibil: http://rs.unitbv.ro/Carti on-line/Ratiu/BD/
12. Transact-SQL reference (Database Engine). Accesibil la: https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16

În redacția autorului

Bun de tipar 02.11.2026
Coli editoriale 10,55. Coli de autor 10,35. Coli de tipar 22,0.
Comanda nr. 74.

Serviciul Editorial-Poligrafic
al Academiei de Studii Economice din Moldova
Chişinău, MD-2005, str.Bănulescu-Bodoni 59.
Tel.: 022-402-910