

ACADEMIA DE STUDII ECONOMICE A  
MOLDOVEI

---

Catedra „Cibernetică și Informatică Economică”

# **Algebra relațională și limbajul SQL**

**Vitalie COTELEA**

Chișinău-2013

CZU 004.65(075.8)

C 78

Manualul **Algebra relațională și limbajul SQL** a fost recomandat pentru editare de *Senatul Academiei de Studii Economice din Moldova* (proces-verbal nr.5 din 27 februarie 2013).

Lucrarea tratează problemele fundamentale cu privire la utilizarea algebrei relaționale și a limbajului SQL în proiectarea, menținerea și restructurarea bazelor de date. Reprezintă un ghid practic care descrie diferite aspecte ale acestor limbi, pas cu pas, cu multe exemple și expresii ale necesităților informaționale ale utilizatorilor.

Scrisă în conformitate cu versiunile recente ale standardului SQL, această carte poate fi citită, înțeleasă și practicată de toate grupurile de utilizatori: începători, studenți de la specialitățile informaticе, dar, de asemenea, de profesioniști sau profesori în domeniul tehnologiilor informației. Conceptele și mecanismele dezvoltate în manual pot fi implementate facil și cu succes în viața profesională.

Recenzenți: prof. univ., dr. hab. inf. **Ion Bolun**,  
 prof. univ., dr. hab. inf. **Anatol Popescu**

Descrierea CIP a Camerei Naționale a Cărții

**Cotelea, Vitalie**

Algebra relațională și limbajul SQL / Acad. de Studii Econ. a Moldovei. Catedra „Cibernetică și Inform. Econom.” – Chișinău: Vizual Design, 2013. – 284 p.

Referințe bibliogr.: p.275-283 - 150 ex.

ISBN 978-9975-4164-7-4.

CZU 004.65(075.8)

C 78

Autor: © Vitalie Cotelea (ASEM)

ISBN 978-9975-4164-7-4.

Vizual Design

## CUPRINS

---

<b>Introducere .....</b>	<b>9</b>
<b>1. Concepte și modele privind bazele de date .....</b>	<b>13</b>
1.1. DE LA SISTEME ORIENTATE PE PROCESE LA SISTEME ORIENTATE PE BAZE DE DATE .....	13
1.2. OBIECTIVELE BAZELOR DE DATE .....	16
1.2.1. Evitarea redundanței și inconsistenței datelor .....	16
1.2.2. Facilitarea accesului la date .....	17
1.2.3. Facilitarea elaborării programelor de prelucrare a datelor .....	18
1.2.4. Asigurarea accesului concurrent la date .....	18
1.2.5. Asigurarea accesului securizat la date .....	19
1.2.6. Păstrarea integrității datelor.....	20
1.3. CE ESTE O BAZĂ DE DATE? .....	21
1.4. COMPOENȚELE UNUI SISTEM ORIENTAT PE BAZE DE DATE .....	23
1.4.1. Hardware-ul .....	24
1.4.2. Programe-aplicații .....	24
1.4.3. Datele.....	24
1.4.4. Metadatele.....	27
1.4.5. Componența software .....	27
1.4.6. Utilizatorii bazei de date.....	29
1.5. ARHITECTURA BAZEI DE DATE CU TREI NIVELE .....	32
1.5.1. Nivelul intern .....	33
1.5.2. Nivelul logic.....	35
1.5.3. Nivelul extern.....	35
1.5.4. Independența datelor .....	37
1.5.5. Funcționarea arhitecturii cu trei nivele .....	38
1.6. STRUCTURA MODELULUI RELAȚIONAL .....	39
1.6.1. Atribute și domenii.....	40
1.6.2. Tupluri .....	42
1.6.3. Scheme și relații .....	43
1.7. CONSTRÂNGERI DE INTEGRITATE .....	45
1.7.1. Constrângerile de integritate structurale .....	46
1.7.1.1. <i>Constrângerile de unicitate și minimalitate a cheii .....</i>	46
1.7.1.2. <i>Constrângerea entității .....</i>	48
1.7.1.3. <i>Constrângerea referențială .....</i>	48
1.7.2. Constrângerile de integritate de comportament .....	50
1.7.2.1. <i>Constrângerile de comportament al domeniului .....</i>	51
1.7.2.2. <i>Constrângerile de comportament al tuplului .....</i>	52

1.7.2.3. <i>Constrângeri de comportament al relației</i> .....	52
1.7.2.4. <i>Constrângeri de comportament al bazei de date</i> .....	53
<b>2. Algebra relațională .....</b>	<b>55</b>
2.1. OPERAȚIILE TRADITIONALE PE MULTIMI .....	56
2.1.1. Scheme relaționale compatibile .....	56
2.1.2. Uniunea .....	57
2.1.3. Diferența .....	58
2.1.4. Intersecția .....	59
2.1.5. Produsul cartezian .....	61
2.1.6. Operația de redenumire .....	62
2.1.7. Complementul .....	63
2.2. OPERAȚIILE RELAȚIONALE NATIVE .....	66
2.2.1. Proiecția .....	66
2.2.2. Selectia .....	68
2.2.3. Joncțiunea naturală .....	70
2.2.4. Semijoncțiunea .....	74
2.2.5. Joncțiunea theta .....	75
2.2.6. Joncțiuni externe .....	76
2.2.7. Divizarea .....	78
2.3. INTEROGĂRI ÎN ALGEBRA RELAȚIONALĂ .....	79
2.3.1. Expresii algebrice .....	80
2.3.2. Selecții generalizate .....	82
2.3.3. Interrogări conjunctive .....	83
2.3.4. Interrogări neconjunctive .....	86
2.3.4.1. <i>Interrogări cu diferențe</i> .....	86
2.3.4.2. <i>Complementul unei multimi</i> .....	87
2.3.4.3. <i>Cuantificarea universală</i> .....	87
2.3.5. Un exemplu integrator .....	89
<b>3. SQL: Interrogarea și actualizarea bazelor de date .....</b>	<b>91</b>
3.1. PREZENTARE GENERALĂ .....	91
3.1.1. Premisele apariției limbajului SQL .....	91
3.1.2. Scurt istoric .....	92
3.1.3. Categoriile de instrucțiuni SQL .....	94
3.2. CELE MAI SIMPLE INTEROGĂRI .....	96
3.2.1. Instrucțiunea SELECT .....	96
3.2.2. Clauza DISTINCT .....	98
3.2.3. Clauza ORDER BY .....	99
3.2.4. Aliasuri de atribut .....	101
3.3. INTEROGĂRI CU CRITERII DE SELECȚIE .....	101
3.3.1. Clauza WHERE .....	102

3.3.2. Operatorii de comparație .....	103
3.3.3. Operatorii logici .....	103
3.3.3.1. Operatorul <i>AND</i> .....	104
3.3.3.2. Operatorul <i>OR</i> .....	104
3.3.3.3. Operatorul <i>NOT</i> .....	105
3.3.3.4. Ordinea de evaluare a operatorilor logici .....	105
3.3.4. Operatorul IS NULL .....	106
3.3.5. Funcții cu utilizarea valorii NULL .....	108
3.3.5.1. Funcția <i>COALESCE</i> .....	108
3.3.5.2. Funcția <i>NVL</i> .....	108
3.3.5.3. Funcția <i>NULLIF</i> .....	109
3.3.6. Operatorul IN .....	109
3.3.7. Operatorul BETWEEN .....	111
3.3.8. Operatorul LIKE .....	111
3.3.9. Limitarea numărului de tupluri returnate .....	114
3.4. INTEROGĂRI CU AGREGĂRI ȘI GRUPĂRI .....	114
3.4.1. Funcții de agregare .....	115
3.4.1.1. Funcția <i>COUNT</i> .....	115
3.4.1.2. Funcția <i>AVG</i> .....	117
3.4.1.3. Funcția <i>SUM</i> .....	117
3.4.1.4. Funcțiile <i>MAX</i> și <i>MIN</i> .....	118
3.4.2. Clauza GROUP BY .....	118
3.4.3. Clauza HAVING .....	120
3.5. INTEROGĂRI CU JONCȚIUNI .....	123
3.5.1. Produsul Cartezian .....	123
3.5.2. Joncțiuni interne .....	125
3.5.2.1. Joncțiuni interne de egalitate .....	126
3.5.2.2. Clauza USING .....	128
3.5.2.3. Joncțiuni naturale .....	129
3.5.2.4. O problemă a joncțiunii naturale .....	131
3.5.2.5. Aliasuri de relații sau variabile-tuplu .....	131
3.5.2.6. Joncțiunea unei relații cu ea însăși .....	133
3.5.2.7. Joncțiuni interne de inegalitate .....	134
3.5.3. Joncțiuni externe .....	135
3.5.3.1. Returnarea tuplurilor nejonctionabile .....	136
3.5.3.2. Joncțiunea externă de stânga .....	137
3.5.3.3. Joncțiunea externă de dreapta .....	138
3.5.3.4. Joncțiunea externă completă .....	139
3.6. SUBINTEROGĂRI .....	140
3.6.1. Tipuri de subinterrogări .....	141
3.6.2. Subinterrogări cu operatori de comparație .....	142

3.6.3. Subinterrogări cu operatorul IN .....	146
3.6.4. Subinterrogări cu operatorul ANY .....	149
3.6.5. Subinterrogări cu operatorul ALL .....	150
3.6.6. Subinterrogări corelate și necorelate .....	152
3.6.7. Subinterrogări cu operatorul EXISTS .....	155
3.6.8. Subinterrogări în clauza FROM .....	159
3.6.9. Subinterrogări în clauza HAVING și în alte clauze .....	160
3.7. INTEROGĂRI CU OPERATORI DIN TEORIA MULTIMILOR .....	162
3.7.1. Operatorii UNION și UNION ALL .....	163
3.7.2. Operatorii INTERSECT și INTERSECT ALL .....	164
3.7.3. Operatorii EXCEPT și EXCEPT ALL .....	164
3.8. INSTRUCȚIUNI DE ACTUALIZARE A BAZEI DE DATE .....	165
3.8.1. Inserarea tuplurilor .....	166
3.8.2. Modificarea tuplurilor .....	168
3.8.3. Suprimarea tuplurilor .....	169
<b>4. SQL: Limbajul de definire a datelor .....</b>	<b>171</b>
4.1. DEFINIREA DATELOR ÎN SQL .....	171
4.1.1. Tipuri de date numerice .....	171
4.1.2. Tipuri de date secvențe de caractere .....	173
4.1.3. Tipuri de date temporale .....	174
4.2. DEFINIREA SCHEMEI BAZEI DE DATE .....	174
4.2.1. Definirea constrângerilor de integritate .....	174
4.2.1.1. <i>Constrângerea NOT NULL</i> .....	176
4.2.1.2. <i>Constrângerea DEFAULT</i> .....	177
4.2.1.3. <i>Definirea cheilor primare</i> .....	177
4.2.1.4. <i>Definirea cheilor externe</i> .....	178
4.2.1.5. <i>Constrângerea UNIQUE</i> .....	179
4.2.1.6. <i>Constrângerile de comportament CHECK</i> .....	180
4.2.1.7. <i>Constrângerile de comportament CHECK la nivel de bază de date</i> .....	181
4.2.2. Definirea atributelor .....	182
4.2.3. Crearea schemei relaționale .....	183
4.2.4. Modificarea și suprimarea schemei relaționale .....	187
4.2.4.1. <i>Adăugarea, modificarea și suprimarea atributelor</i> .....	187
4.2.4.2. <i>Adăugarea, modificarea și suprimarea constrângerilor</i> .....	190
4.2.4.3. <i>Amânarea verificării constrângerilor</i> .....	192
4.2.4.4. <i>Suprimarea schemei relaționale</i> .....	194
4.3. GESTIUNEA VIZIUNILOR .....	195
4.3.1. Ce este o viziune? .....	195
4.3.2. Crearea și suprimarea viziunilor .....	197

---

4.3.3. Consultarea și actualizarea viziunilor .....	199
4.3.3.1. Consultarea viziunilor.....	200
4.3.3.2. Viziuni definite pe proiecții de atrbute .....	200
4.3.3.3. Viziuni definite pe selecții de tupluri .....	202
4.3.3.4. Viziuni definite pe o relație de interogări cu <i>DISTINCT</i> sau <i>GROUP BY</i> .....	203
4.3.3.5. Viziuni definite pe mai multe relații .....	205
4.3.4. Clasificarea viziunilor .....	206
4.3.5. Din nou despre opțiunea WITH CHECK OPTION .....	209
4.3.6. Viziuni materializabile .....	211
4.3.7. Utilitatea viziunilor .....	212
4.4. SINONIME .....	213
4.5. INDECȘI .....	215
4.5.1. Crearea și suprimarea indecșilor .....	215
4.5.2. Utilitatea indecșilor .....	217
4.5.3. Tipuri de indecși.....	218
<b>5. Protecția accesului și administrarea tranzacțiilor. SQL Integrat.</b> <b>221</b>	
5.1. CONTROLUL ACCESULUI LA BAZA DE DATE.....	222
5.1.1. Drepturi de acces.....	222
5.1.2. Acordarea și retragerea drepturilor .....	223
5.2. ADMINISTRAREA TRANZACȚIILOR .....	225
5.2.1. Proprietățile tranzacțiilor .....	225
5.2.2. Modele de tranzacții .....	226
5.2.2.1. Tranzacții în SQL .....	226
5.2.2.2. Tranzacții imbricate .....	227
5.2.2.3. Puncte de reluare .....	228
5.2.3. Anomalii de execuție concurrentă a tranzacțiilor.....	228
5.2.4. Blocarea relațiilor și gestiunea tranzacțiilor .....	230
5.2.5. Serializarea tranzacțiilor.....	231
5.2.6. Niveluri de izolare a tranzacțiilor .....	232
5.3. LIMBAJUL SQL INTEGRAT.....	234
5.3.1. Structura unui program cu SQL încorporat .....	235
5.3.1.1. Delimitatoare .....	236
5.3.1.2. Aria de comunicatie a SQL .....	237
5.3.1.3. Variabilele programului.....	238
5.3.2. Manipularea datelor fără cursoare.....	239
5.3.3. Manipularea datelor cu cursoare .....	241
5.3.4. Un exemplu integrator .....	244
<b>6. Proprietăți obiect-relaționale în standardul SQL3.....</b>	<b>247</b>
6.1. DECLANȘATOARELE ȘI STANDARDUL SQL3 .....	248

## 8 Cuprins

---

6.1.1. Definirea declanșatoarelor .....	248
6.1.2. Evenimente.....	251
<i>6.1.2.1. Momentul de activare.....</i>	<i>251</i>
<i>6.1.2.2. Granularitatea .....</i>	<i>251</i>
<i>6.1.2.3. Nume de corelație .....</i>	<i>252</i>
6.1.3. Condiții .....	253
6.1.4. Acțiuni.....	254
6.1.5. Gestionarea constrângerilor de integritate .....	255
<i>6.1.5.1. Gestionarea constrângerilor de domeniu .....</i>	<i>256</i>
<i>6.1.5.2. Gestionarea constrângerilor de tuplu .....</i>	<i>259</i>
6.2. LIMBAJUL SQL3 ȘI MODELUL OBIECT-RELATIONAL.....	261
6.2.1. Tipuri de date compuse .....	261
<i>6.2.1.1.Tipul de date ARRAY.....</i>	<i>262</i>
<i>6.2.1.2. Tipul de date ROW .....</i>	<i>262</i>
6.2.2. Tipuri colecție.....	263
6.2.3. Tipuri de date definite de utilizator .....	264
<i>6.2.3.1. DISTINCT TYPES .....</i>	<i>264</i>
<i>6.2.3.2. Tipuri de date structurate definite de utilizator.....</i>	<i>264</i>
6.2.4. Relații cu tupluri-obiecte .....	266
6.2.5. Moștenirea .....	267
6.2.6. Referire la tip ca la domeniul unui atribut .....	269
6.2.7. Relații imbricate .....	271
6.3. OBIECTE MARI .....	273
<b>Bibliografie.....</b>	<b>275</b>

## Introducere

---

Poate, cel mai important subiect, într-un curs de baze de date, este modelul relațional al datelor propus de Codd [Codd70]. Bazele de date relaționale pun în aplicare structuri logice de date, numite relații și oferă modalități de realizare a operațiilor de actualizare și de regăsire a datelor din relații. Timpul consumat pentru studierea bazelor de date este substanțial și este cheltuit pe înșurarea metodelor și instrumentelor de creare, menținere, restructurare, precum și de consultare a bazelor de date.

Discuțiile despre limbajele de interogări, în general, se axează pe limbajul SQL. Algebra relațională constituie un limbaj de interogări, căruia, de obicei, i se acordă mai puțină atenție. Acest accent pus mai mult pe SQL distorsionează istoria dezvoltării bazelor de date, dar și înțelegerea limbajului SQL, fapt care complică aplicarea metodelor de proiectare. Totuși, există multe avantaje pe care le oferă cunoașterea algebrei relaționale.

În primul rând, algebra relațională ajută utilizatorii să înțeleagă modelul relațional și, evident, obiect-relațional. Modelul relațional împreună cu algebra relațională asigură o modalitate certă și consistentă pentru înțelegerea tehniciilor de interogare a bazei de date. Algebra relațională nu reprezintă un instrument de proiectare a bazelor de date, dar acesta poate sprijini analiza bazei de date și a deciziilor de proiectare.

Într-al doilea rând, cunoașterea algebrei relaționale facilitează înțelegerea acelei părți a limbajului SQL ce ține de consultarea bazei de date. Sintaxa de bază a declarației *SELECT* oferă un mod integrat de combinare a operațiilor algebrice pentru exprimarea interogărilor.

Și, în al treilea rând, înțelegerea algebrei relaționale poate fi folosită cu succes pentru îmbunătățirea performanțelor interogărilor. Componenta motorului bazei de date, care procesează interogările, traduce codul SQL într-un plan de interogare, care include operații ale algebrei relaționale. Optimizatorul de interogări încearcă să accelereze executarea interogării, reducând timpul de procesare a fiecărei operații.

Pe de altă parte, SQL este omniprezent. Dar SQL este complicat, dificil, și predispus la erori (mult mai mult decât cred susținătorii acestui limbaj), iar testarea lui nu poate fi niciodată exhaustivă. Deci, pentru scrierea corectă a instrucțiunilor SQL, trebuie următe anumite reguli și trebuie aplicate anumite tehnici.

Desigur, SQL este limbajul-standard pentru utilizarea bazelor de date relaționale, dar acest lucru nu-l face să fie neapărat relațional. Adevărul este trist, SQL se îndepărtează, uneori, de teoria relațională prea mult. Tuplurile duplicate și valorile nule sunt două exemple evidente, dar nu singurele. Pentru a fi un profesionist profund, este nevoie de înțelegerea teoriei relaționale (ce este și ce e bine), trebuie să se cunoască distanțarea limbajului SQL de această teorie și este nevoie să fie evitate problemele pe care le poate provoca.

Volumul de față tratează problemele fundamentale cu privire la utilizarea algebrei relaționale și a limbajului SQL în proiectarea, menținerea și restructurarea bazelor de date, ținând cont de aspectele menționate.

**Capitolul 1** acoperă aspectele principale ale bazelor de date relaționale utilizate în volumul de față.

E cunoscut faptul că o interogare la baza de date poate fi tratată ca o expresie algebrică, ce se aplică asupra unei mulțimi de relații și produce o relație finală. Algebra relațională este tratată în **capitolul 2**, unde este concepută ca un limbaj de programare foarte simplu care permite formularea cererii asupra unei baze de date relaționale. Orice interogare în algebra relațională descrie, pas cu pas, procedura de calcul a răspunsului, bazându-se pe ordinea în care apar operatorii în interogare. Natura procedurală a algebrei permite folosirea expresiilor algebrice ca un plan de evaluare a cererii, fapt utilizat în sistemele relaționale.

Limbajul SQL de interogare a datelor include o singură instrucțiune, dar una foarte importantă: *SELECT*. Instrucțiunea *SELECT* este cea mai complexă și mai puternică dintre instrucțiunile SQL. Cu aceasta pot fi recuperate date dintr-una sau mai multe relații, extrase anumite tupluri și chiar pot fi obținute rezumate ale datelor stocate în baza de date, din care motiv este numită „regina” limbajului SQL.

Limbajul de manipulare a datelor este nucleul limbajului SQL, conținând instrucțiunile de menținere a unei baze de date. Dacă se dorește adăugarea, sau actualizarea, sau ștergerea datelor din baza de date, se execută comenziile acestui limbaj. **Capitolul 3** tratează aspectele privind aceste două sublimbaje.

Utilizarea limbajului SQL pentru definirea schemei bazei de date este subiectul **capitolului 4**. În acest capitol, se discută aspectele logice ale relațiilor și atributelor, și se descriu comenziile necesare pentru construirea, modificarea și suprimarea relațiilor și constrângерilor de integritate, viziunilor și sinonimelor, precum și a indecșilor. În afară de acesta, limbajul de definire a datelor administrează asertațiunile, care sunt folosite, de obicei, pentru a formula restricții asupra mai multor relații din baza de date.

În **capitolul 5**, sunt relevante instrucțiunile necesare pentru procesarea tranzacțiilor, asigurând consistența bazei de date. Procesarea tranzacțiilor constituie o caracteristică a SGBD-urilor, care permite utilizatorilor să lucreze concurrent pe baza de date, și asigură că fiecare utilizator vede o versiune consistentă de date și toate schimbările sunt aplicate în ordinea corectă.

În plus, sunt descrise instrucțiunile și clauzele normei SQL2 necesare pentru extinderea limbajelor de programare, numite limbaje-gazdă, cu clauze specifice de manipulare a datelor din baza de date. Această normă presupune că atributele relațiilor manipulate sunt, mai întâi, declarate în calitate de variabile ale limbajului de programare. Totodată, în capitol, se tratează mecanismul de securitate descentralizat, unde utilizatorii sunt ei însăși responsabili pentru acordarea drepturilor de acces pentru obiectele, pe care le dețin, celorlalți utilizatori, folosind concepția de identificator de autorizație, posesiune, privilegiu.

**Capitolul 6** este consacrat dinamizării prelucrării bazelor de date cu ajutorul declanșatoarelor. Declanșatoarele se bazează pe noțiunea de eveniment–condiție–acțiune, respectiv pe reguli ce țin de aceste trei componente. Acestea sunt reguli active, definite pentru administrarea bazelor de date active. O bază de date oferă proiectantului unei aplicații posibilitatea de a monitoriza apariția unor tipuri specifice de evenimente care se produc în baza de date și executarea unor acțiuni, după verificarea

## **12 Introducere**

---

prealabilă a condiției. În afară de aceasta, în capitol, sunt tratate problemele de implementare în bazele de date a obiectelor mari, precum și a structurilor de date definite de utilizator, caracteristice bazelor de date obiect-relaționale reglementate de standardul SQL3.

## 1. Concepte și modele privind bazele de date

Una din utilizările cele mai frecvente ale calculatorului constă în stocarea datelor. Aici este important să se facă distincție între date și informație. Când ne referim la date, se are în vedere un număr, o secvență de caractere sau o mulțime de ambele obiecte. În momentul când se vorbește de informații, se au în vedere datele care se găsesc într-o corelație de formă mai mult sau mai puțin inteligentă și exprimă niște concluzii ce ar permite luarea unor decizii importante în domeniul de interes.

Calculatorul în sine nu este inteligent - prelucră date. El este capabil să facă milioane de operații simple în puțin timp. Cu el se poate prelucra un volum enorm de date pentru a extrage informații. Omul lucrează cu informații. Cum se extrag informațiile din date? Există multe posibilități. De exemplu, obținerea unor statistici despre hotelurile preferate, obținerea traseelor și itinerarelor turistice, emiterea facturilor de către agențiile de voiaj, trimiterea cartelelor clienților etc.

### 1.1. De la sisteme orientate pe procese la sisteme orientate pe baze de date

Pentru a realiza toate aceste activități, este necesară păstrarea într-o anumită structură a datelor despre clienți, pachete de voiaj, rezerve de locuri la hotel, diverse date statistice etc., în funcție de necesitățile informaționale ale utilizatorilor din domeniul de interes.

Traditional, înainte de apariția calculatoarelor, informațiile se extrăgeau după o examinare minuțioasă a datelor scrise pe niște fișe de carton care formau fișiere păstrate în niște cutii. Deseori, costul timpului consumat pentru găsirea rezultatelor dezirabile prevalea asupra costului informației obținute.

Odată cu apariția calculatoarelor și dispozitivelor electronice de păstrare a datelor (cilindri magnetici, benzi magnetice, discuri fixe) apare și

conceptul de fișier electronic. Un fișier electronic este asemănător unui fișier de carton (numai că fișa se numește înregistrare), doar că se păstrează pe suprafețe magnetice și datele pot fi accesate cu o viteză mare de către calculator.

Bineînțeles, datele necesare nu se găseau numai într-un fișier. Astfel, pentru a extrage informațiile necesare, se cerea consultarea mai multor fișiere, precum fișierul clienților, fișierul comenzilor, fișierul vânzătorilor, fișierul ofertelor de foi de voiaj.

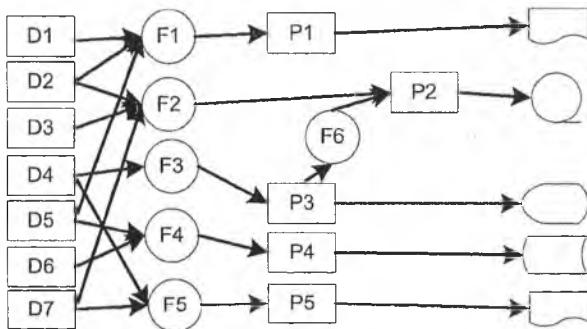
Acesta a fost un pas mare în ce privește stocarea și gestionarea datelor. Dar, odată cu trecerea timpului, se observă că informația extrasă din fișiere, deseori, este contradictorie și provoacă erori și incoerențe în gestionarea domeniului de interes. Repetarea, în diferite locuri, a unor date identice poate provoca probleme dificile după operațiile de actualizare. De exemplu, schimbarea numelui unui agent (fie după căsătorie) trebuie să fie efectuată atât în datele personale, cât și în datele contabile.

Astfel, organizarea tradițională a datelor sub formă de fișiere prezintă, pentru aplicații, serioase probleme. Multimea de programe manipulează și partajează date care nu sunt complet independente. Adică, datele și prelucrarea lor nu sunt explicit separate. Această tehnică plasează procesul în centru, iar în jurul lui se rotesc datele.

Bineînțeles, că aplicațiile care se elaborau pentru un domeniu concret de interes erau orientate să acopere necesitățile foarte specifice procesului de prelucrare a datelor. și limbajele de programare cu structurile de date utilizate, de asemenea, se concentrău pentru o realizare mai eficientă a procesului de prelucrare. De fapt, pentru fiecare problemă din domeniul de interes, se construia unul sau mai multe fișiere a căror structură era dependentă de secvența de operații care trebuiau efectuate pentru soluționarea problemei.

În figura 1.1, se poate vedea clar că procesul este elementul central în proiectarea structurii datelor. De exemplu, dacă procesul  $P1$  nu ar exista, ce sens are existența fișierului  $F1$ ? În afară de aceasta, la o analiză mai adâncă, se vede că există date ( $D2, D4, D5$  și  $D7$ ) ce se găsesc în mai multe fișiere.

Această situație este periculoasă prin inconsistenta care poate apărea, de exemplu, dacă elementul  $D_2$  în fișierul  $F_2$  are o valoare distinctă de valoarea din fișierul  $F_1$ . Care este corectă?



*Figura 1.1. Un sistem orientat pe procese*

Astfel, organizarea datelor într-un sistem orientat pe procese răspunde unor cerințe concrete în funcție de problema care se rezolvă. Diverse aplicații utilizează date distincte pentru prelucrări diferite. Prin urmare, orice echipă de utilizatori dispun de propriile fișiere. Structura în care datele sunt păstrate depinde de mulți factori: hardware-ul disponibil, bunăvoiețea și armonia din echipele de programatori (alegerea limbajului de programare, alegerea structurii de păstrare a datelor). Dar accesul la date, întotdeauna, va fi determinat de organizarea fizică a datelor.

Pentru soluționarea acestor probleme, din nou, intervin capacitateile calculatorului de a face rapid multe lucruri simple. Se propune ca calculatorul să controleze concordanța între date și pentru aceasta, datelor li se dă o structură distinctă, care să nu se mai bazeze pe conceptul clasic de fișier. Pentru a obține informațiile necesare, se leagă datele din diferite componente în structuri complexe. Așa au apărut bazele de date.

Bazele de date caută să rezolve, în principal, problema inconsistenței inerente a sistemelor orientate pe procese. În acest caz, analiza începe cu conceperea structurii datelor ca un tot întreg la care se adaugă constrângerile definite de procesele aplicate asupra lor. Adică, centrul lumii este constituit din date în jurul căror se rotesc procesele (figura 1.2). Gestionarea acestor date, de asemenea, se face în mod centralizat. și baza de date reprezintă un model al domeniului de interes.

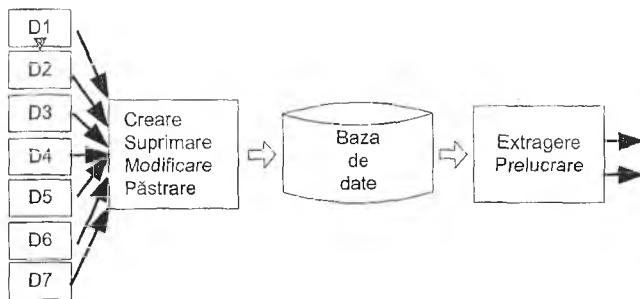


Figura 1.2. Un sistem orientat pe bază de date

Informatica, la fiecare etapă, a trecut prin numeroase forme de abordare a problemelor de programare în funcție de caracteristicile calculatoarelor și de experiența informaticienilor. Astfel, au apărut bazele de date ierarhice și de tip rețea. Mai târziu, au fost propuse bazele de date relaționale, care vor fi studiate detaliat în acest volum. Bazele de date relaționale, rapid, au substituit celelalte tipuri, deoarece se bazează pe un studiu matematic serios care asigură eficiența operațiilor și permit realizarea corectitudinii și completitudinii lor, iar problemele ce apar sunt definite și soluționate într-o formă strict științifică.

Actualmente, bazele de date relaționale sunt cele mai răspândite și utilizate la nivel profesional, cu toate că pentru aplicații avansate și medii academice, sunt destul de preferate bazele de date obiect-relaționale (în special pentru lucrul în Internet) și bazele de date deductive (pentru lucru în Inteligență Artificială).

### 1.2. Obiectivele bazelor de date

Obiectivele pe care le urmăresc bazele de date sunt determinate de dezavantajele sistemelor orientate pe procese. Principalele obiective sunt examineate în continuare.

#### 1.2.1. Evitarea redundanței și inconsistenței datelor

Capacitatele enorme ale calculatoarelor se folosesc nu numai pentru stocarea volumelor mari de date. Ele sunt folosite și pentru controlul dacă

datele existente sunt consistente, adică pentru verificarea dacă nu există date contradictorii între datele stocate.

Pe de altă parte, utilizatorul este cel care decide care date trebuie păstrate în baza de date și care nu. Evident, sunt necesare numai datele utile pentru activitățile din domeniul de interes. Utilizatorul este cel care decide, în afară de aceasta, în ce formă trebuie să se ofere datele. De exemplu, un număr de telefon să se păstreze ca un număr sau ca o secvență de caractere (care ar permite să se introducă cratima pentru a separa prefixul de însăși numărul). Evitarea redundanței (datelor repetitive) în datele stocate este un obiectiv prioritar, deoarece:

- Nu se risipesc capacitatea unităților de stocare a datelor.
- Se evită extragerea acelorași informații, dar prin căi distincte care, deseori, pot fi logic contradictorii.

### 1.2.2. Facilitarea accesului la date

În sistemele vechi de până la apariția bazelor de date, datele se păstraau în fișiere electronice, care puteau fi accesate numai prin programe speciale care realizau o sarcină specifică și destul de concretă. Pentru orice tip de informații, care se doreau a fi extrase, se elabora un program care consulta fișierele respective. Astfel, dacă un funcționar al contabilității dorea să cunoască clientii din Chișinău, care au cumpărat în anul curent foi de voiaj în sumă de 7000 lei, prezenta aceste necesități unui programator, care, la rândul său, elabora un program de soluționare a problemei formulate.

Evident, această formă de lucru este lentă, iar programul de extragere a datelor devine un factor vulnerabil, mai ales dacă se elaborează mai multe programe particulare care accesează aceleași date. În afară de aceasta, dacă se schimbă structura bazei de date (de exemplu, în loc de păstrarea împreună a numelui și prenumelui unui client, se oferă un loc aparte pentru nume și alt loc pentru prenume), unele programe în funcțiune nu vor mai putea rula, deoarece nu vor vedea structura nouă a bazei de date.

Pentru a evita toate aceste probleme, se introduce un nivel de abstracție între date și aplicații, susținut de un program auxiliar de maximă importanță, care se numește Sistem de Gestiune a Bazei de Date (SGBD). Una din aceste funcții constă în oferirea unui limbaj care permite

consultarea facilă a bazei de date fără a fi necesară elaborarea unui program special. Este suficient să se formuleze cererea în acest limbaj simplu, iar răspunsul va fi obținut mai mult sau mai puțin imediat grație gestionării datelor de SGBD.

### **1.2.3. Facilitarea elaborării programelor de prelucrare a datelor**

Precum s-a menționat anterior, datele pot fi repartizate în mai multe fișiere din baza de date. Inclusiv, pot fi chiar mai multe baze de date din care se dorește extragerea unor date. În plus, fiecare bază de date poate fi creată de un sistem distinct. Astfel, devine dificilă crearea aplicațiilor care procesează aceste date. Această problemă, de asemenea, se soluționează, utilizând SGBD-ul. Sistemul de gestiune desparte aplicațiile de procesul de căutare a datelor și oferă o vizionare uniformă a datelor, facilitând, prin urmare, programarea aplicațiilor complexe.

### **1.2.4. Asigurarea accesului concurrent la date**

Un calculator este o mașină care este capabilă să facă numai un lucru odată: citirea din memorie, scrierea în memorie, realizarea unui calcul,... în definitiv niște lucruri simple. Majoritatea lucrurilor sunt formulate de diverse persoane la diferite calculetoare (orice persoană poate avea o tastatură și un monitor distinți, adică un terminal) dar, deseori, asupra aceleiași date. De exemplu, accesul unui program de rezervare a biletelor de avion la o companie dată. În acest caz, datele referitoare la rezervări se păstrează doar pe un calculator central care și gestionează toate operațiile tastate la fiecare terminal.

Așadar, fiecare utilizator poate formula cereri calculatorului central în momentul când el consideră necesar, iar calculatorul central va trebui să răspundă în timp acceptabil. Dat fiind faptul că multe persoane pot lucra concomitent, este posibil ca calculatorul central să recepteze cereri de acces la date formulate simultan, adică este cazul unui acces concurrent.

Pentru a observa problemele ce se pot produce în cazul accesului concurrent, presupunem că calculatorului central i se dictează ce înregistrare trebuie stocată și care operații trebuie făcute: consultarea dacă

unele date există, modificarea unui câmp a înregistrării sau formarea unei înregistrări goale și inserarea datelor în ea.

Dacă nu se iau în considerație cerințele utilizatorilor în ce privește timpul de acces, nu apar probleme, deoarece calculatorul poate examina cererile într-o ordine anumită. Dar pentru a oferi senzația că fiecare utilizator este tratat la egal, se face uz de o tehnică numită *regim de timp partajat*, când orice minut al timpului se divizează în părți egale foarte mici oferite la rând fiecărui utilizator.

Dar aici pot apărea probleme legate de accesul concurrent la date. De exemplu, ce se întâmplă dacă un utilizator dorește să eliminate înregistrările clientilor care au beneficiat de serviciile hotelului Național, în timp ce altul dorește să expedieze unui client al acestui hotel o invitație de turism și pentru aceasta are nevoie de niște modificări a unei înregistrări? Dacă eliminarea se va face prima, cum va fi modificată înregistrarea care nu există?

Aceste probleme se pot agrava enorm în momentul în care se introduce conceptul de *tranzacție*. O tranzacție presupune o mulțime de operații asupra datelor care trebuie să fie tratată ca o operație unică: se face în întregime sau deloc. Un exemplu de tranzacție poate fi creșterea cu 2.8% a creditului oferit fiecărui client. Dacă sunt mai multe mii de clienți, operația poate dura și între timp poate apărea altă cerere. Conform proprietății unei tranzacții (se execută complet sau nu se execută deloc) se face uz de copia fiecărei înregistrări afectate (dacă se produce vreo problemă, se distrug toate copiile), iar cele originale rămân intacte. Ce se întâmplă dacă, în timp ce se gestionează copiile înregistrărilor, apare o cerere de schimbare a creditului maximal al clientului Tudor Petrache? Se mărește cu 2.8%, se scrie valoarea nouă sau se lasă valoarea veche? Toate aceste probleme sunt soluționate de SGBD, care trebuie să asigure un rezultat final consistent și trebuie să informeze utilizatorul interesat, dacă s-a produs vreo eroare în procesul executării tranzacției.

#### 1.2.5. Asigurarea accesului securizat la date

În sistemele vechi cu fișe de carton, fișierele mai importante se aflau în afara razei de acces a majorității utilizatorilor. La ele aveau acces numai unii utilizatori autorizați. Această metodă de securitate lăua diverse forme,

în funcție de gradul de securitate cu care erau dotate anumite date: uși cu lacăte, cartele magnetice, gardă de securitate și chiar criptarea datelor mai importante.

Calculatoarele păstrează fișiere și pot recepta cereri, dacă nu există restricții, de la terminalele situate departe, chiar pe alte continente. Aici, metodele tradiționale de securitate nu sunt suficiente (cu toate că unele sunt necesare cu scopul, de exemplu, de a evita distrugerea fizică sau furtul calculatorului central) și se apelează la mediile electronice de protecție care împiedică accesul utilizatorilor (cu excepția unor persoane autorizate) la datele considerate mai importante. De exemplu, departamentul de comerț nu are motive de acces la numele fiecărui angajat al întreprinderii, date la care au acces funcționarii contabili sau administrația întreprinderii. Pentru a atinge acest lucru, se atribuie o serie de priorități fiecărui utilizator, dar și nivele de securitate fiecărui tip de date, astfel că orice utilizator poate avea acces numai la datele cu un număr inferior sau egal priorității sale. În afară de aceasta, există un registru ce ia notă de accesele la date de nivel înalt, fixând urma înregistrărilor solicitate, ora și utilizatorul care le accesează sau motivele unui acces frustrat. Toate aceste funcții sunt suportate de SGBD.

### **1.2.6. Păstrarea integrității datelor**

Acesta este unul din aspectele cele mai importante și el face uz intensiv de capacitatea calculatorului de prelucrare a datelor.

Datele posedă unele caracteristici specifice, în funcție de operațiile ce se pretind a face asupra lor și de evoluția domeniului de interes în timp și în spațiu. Dacă aceste caracteristici sunt, de asemenea, stocate, atunci informația extrasă de utilizator este coerentă și reflectă, în orice moment, domeniul de interes reprezentat sub formă de bază de date. Această proprietate se numește *integritatea* bazei de date. De exemplu, poate fi interzis faptul ca numele unui client să fie necunoscut la momentul inserării înregistrării respective în baza de date. Pot exista relații și mai complicate care leagă diverse fișiere. De exemplu, este trimisă o comandă unui vânzător, dar acest vânzător nu există în fișierul de vânzători sau sunt rezervate călătorii pentru un traseu care nu există, sau numărul de

bilete rezervate pentru ruta Chișinău – București întrece esențial numărul maximum permis, 10%.

Toate acestea sunt constrângeri de integritate pe care SGBD-ul trebuie să le verifice la fiecare încercare de actualizare a bazei de date.

### 1.3. Ce este o bază de date?

Ce este o bază de date și ce este un SGBD? Nu există o definiție a completă și perfectă pentru acești termeni. De fapt, datorită legăturii strânse biunivoce, care există între aceste entități, deseori, acești doi termeni sunt confundați.

În sensul cel mai larg, orice colecție de date este o bază de date.

Există numeroase definiții ale conceptului bază de date, dar toate coincid în cea că ea este o mulțime de date stocate pe un suport cu acces direct și datele sunt interdependente și structurate conform unui model capabil să redea un conținut cu o pondere semantică maximală. În continuare, se prezintă definițiile unor cercetători foarte influenți în acest domeniu:

Astfel, cercetătorul A.Silberschatz [Silberschatz97] extinde conceptul de bază de date, incluzând în el și termenul program. Se au în vedere programele care accesează aceste date, cu toate că nu rămâne clar dacă se consideră programele componente ale SGBD sau include și programele aplicații elaborate de utilizatori.

În mod ideal, un SGBD susține o serie de funcții caracteristice pentru a satisface doleanțele utilizatorilor bazei de date. Astfel, SGBD-ul trebuie:

- Să posede un limbaj de definire a datelor, care ar permite ușor crearea bazei de date, dar și modificarea structurii ei.
- Să posede un limbaj de interogare și manipulare a datelor care ar permite inserarea, suprimarea, modificarea și consultarea bazei de date într-o formă eficientă și potrivită.
- Să permită stocarea unor cantități enorme de date (mii de milioane de caractere) fără ca utilizatorul să percepă vreo degradare în ce privește randamentul total al sistemului.

- Să admită o gestiune sigură a datelor, cu respectarea accesului neautorizat și să controleze afecțiunile produse de dispozitivele mecanice și electronice asupra datelor stocate.
- Să asigure accesul simultan cel puțin pentru o parte din utilizatori, adică să asigure accesul concurent la date.

Aceste caracteristici (și altele) generează altă definiție a bazei de date, poate mai orientată la moda actuală în informatică: proiectarea orientată pe obiecte, unde programele elaborate concep lumea ca o mulțime de obiecte care se leagă între ele pentru a atinge un obiectiv comun.

**Definiția 1.1.** [Date90] Baza de date este o organizare coerentă de date persistente și independente, accesibile pentru utilizatorii concurenți.

Această definiție, de asemenea, apelează la conceptul SGBD, deoarece se referă la procesele de stocare a datelor (sunt persistente), de gestionare a integrității (deoarece sunt coerente) și de controlare a multiplilor utilizatori (permittând, astfel, concurența).

Într-un sens mai strict, o definiție mai acceptabilă a fost propusă de J.Ullman în 1988:

**Definiția 1.2.** [Ullman88] În esență, o *bază de date* nu e decât o mulțime de date bine structurate și persistente, a căror structură este definită într-o schemă folosind un limbaj de definire a datelor. Datele și schema este gestionată de un soft Sistem de Gestiuare a Bazei de Date.

Deci o bază de date este o mulțime de date indispensabile de sistemul de gestiune.

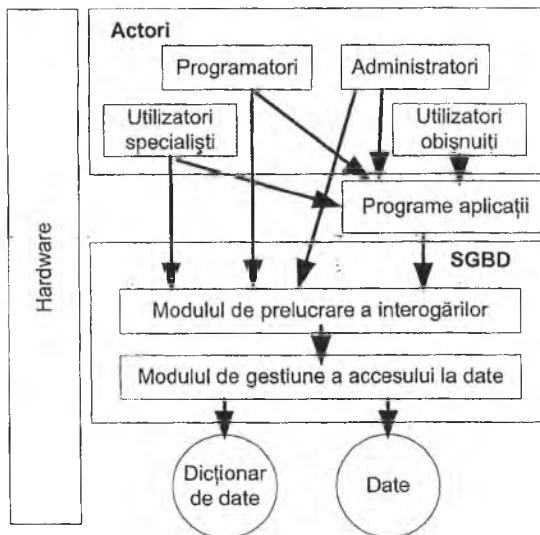
Formele, în care SGBD-uri comerciale și academice distințe abordează aceste caracteristici, diferă enorm nu numai prin tehniciile utilizate, dar și prin aproximările și paradigmile care stau la baza elaborării lor. Acest lucru se va observa când vor fi examinate diverse modele de date și tehnici de construire a schemelor logice ale bazelor de date.

## 1.4. Componentele unui sistem orientat pe baze de date

Până aici, s-a evidențiat o idee generală în definiția bazei de date - existența unui sistem de gestiune central care partajează datele între utilizatori și care facilitează executarea unei serii de acțiuni pentru asigurarea rezolvărilor indispensabile pentru o bună funcționare a sistemului în întregime.

Acum, se va examina puțin mai mult conceptul de bază de date și se vor descrie componentele sale principale, justificându-se, în afară de aceasta, necesitatea fiecărei din ele.

Un sistem orientat pe baze de date constă din următoarele componente (figura 1.3):



*Figura 1.3. Componentele unui sistem orientat pe baze de date*

- Date.
  - Dicționar de date.
  - Software-ul de sistem și sistemul de gestiune a bazei de date.

## 24 Concepte și modele privind bazele de date

---

- Actori sau utilizatori ai bazei de date.
- Hardware-ul.
- Programe aplicative.

### 1.4.1. Hardware-ul

Este vorba de echipamentele care suportă stocarea datelor, prelucrarea și distribuirea (rețele) lor. De obicei, această componentă poate fi reprezentată de un singur calculator personal, un singur calculator mainframe sau o rețea de calculatoare.

De obicei, într-o rețea de calculatoare, se aplică următoarea schemă (dacă este utilizată arhitectura client-server): Se folosește un calculator principal pe care se află programele back-end - adică partea din sistemul de gestiune al bazei de date care administrează și controlează accesul la baza de date și mai multe calculatoare aflate în diferite locații pe care se află programele front-end – adică partea din sistemul de gestiune al bazei de date ce constituie interfața cu utilizatorul.

În această schemă, numită client-server, programele back-end reprezintă serverul, iar cele front-end reprezintă clienții. Există și alte arhitecturi care împun cerințele respective asupra hardware-ului întrebuițat

### 1.4.2. Programe-aplicații

Unii utilizatori interacționează, în general, cu datele prin intermediul programelor aplicative, care oferă o interfață mai "prietenoasă" decât cea propusă de SGBD. Aceste programe sunt specifice domeniului de aplicare și rezolvă anumite probleme din acest domeniu. Elaborarea lor este simplificată grație SGBD-ului care își asumă povara operațiilor de stocare și acces la date.

### 1.4.3. Datele

Datele acționează ca o punte de legătură între componentele mașină (hardware și software) și componenta umană. Domeniul de interes, pentru care se construiește baza de date, este reprezentat de date. Ele sunt

integrate și concentrate într-un tot coerent și fără redundanțe. Aceste date sunt partajate de utilizatorii din domeniul de interes.

Efectiv, o bază de date nu ar avea niciun sens, dacă nu ar fi compusă din date. Mai puțin clare răspunsurile la întrebările: ce date trebuie păstrate, sub ce formă și cum vor fi "înțelese" aceste aspecte de către sistem.

Dislocarea sau amplasarea datelor depinde de domeniul de interes pentru care se construiește baza de date. Nu este același lucru să construiești o bază de date care stochează desenele vectoriale ale unor monumente istorice și o bază de date pentru rezervarea locurilor într-un hotel. Ceea ce deosebește fundamental primul caz de celălalt este forma în care datele se leagă între ele și tipurile de acces la date care pot fi cerute de utilizatori.

În afară de aceasta, datele trebuie stocate astfel ca interogările să fie mai eficiente, evitându-se, totodată, existența datelor duplicate care pot șterbi coerența bazei de date. Astfel, nu numai că se consideră datele care utilizatorul dorește să le stocheze, dar și toată structura în care ele vor fi organizate pentru a face cererile mai eficiente.

De exemplu, fie că baza de date dintr-un hotel conține un fișier cu toate agențiile de turism, care au contractat servicii de cazare ale hotelului dat cel puțin într-un pachet de voiaj. Și presupunem că pentru a facilita accesul la aceste agenții, datele despre ele sunt ordonate conform denumirilor lor. Ce se întâmplă, dacă se dorește, de exemplu, să se trimită tuturor agențiilor de turism din Moldova câte o felicitare cu ocazia noului an? Rezultatul este inacceptabil, din simplul motiv că pentru a răspunde la întrebare trebuie examineate toate agențiile din fișier una câte una, chiar dacă căutarea nu necesită un timp excesiv de mare.

Soluția informatică a acestei probleme constă în crearea unui fișier aparte ale cărui înregistrări conțin numai două câmpuri, *țara agenției* și *denumirea agenției*. Înregistrările acestui fișier sunt sortate după *țara agenției*, iar în cadrul fiecărei țări după câmpul *denumirea agenției*. Astfel, pentru a accesa agențiile din Moldova, nu se examinează fișierul principal, dar se trece la fișierul secundar, unde, imediat, se pot găsi agențiile de turism din Moldova. În continuare, se merge la fișierul principal pentru a accesa numai înregistrările agențiilor ale căror denumiri

au fost extrase anterior din fișierul secundar și se extrag adresele pentru a expedia cartelele de felicitare. Fișierul secundar este denumit și index al fișierului principal.

Marele avantaj al acestei metode este că asistarea fișierului auxiliar o realizează automat însuși sistemul, fapt ce scutește utilizatorul de sarcina de a scrie explicit o înregistrare auxiliară de fiecare dată când se inserează o agenție nouă în fișierul principal.

Așadar, fișierele secundare sunt create automat cu scopul de a facilita accesul la date, cu toate că unii utilizatori pot să nu cunoască nimic despre existența lor.

Este, de asemenea, importantă problema selectării datelor care trebuie stocate. În afara de aceasta, se poate întâmpla că unele date trebuie păstrate mai mulți ani, acumulându-se, astfel, date în fiecare an. Evident că aici poate deveni vulnerabilă capacitatea disponibilă de memorie secundară.

Dacă utilizarea datelor istorice nu este frecventă, poate fi aplicată varianta când datele se scriu pe dispozitive de păstrare îndelungată. Astfel, spațiul de pe dispozitivele electomagnetiche este eliberat pentru alte activități.

Dimpotrivă, după crearea bazei de date și utilizarea ei, poate apărea necesitatea introducerii unor date ce ar implica schimbarea structurii datelor. De exemplu, fie pe parcurs a apărut o agenție care propune un pachet de voiaj de aventuri în pădurea Amazonului. Acest pachet poate fi procurat de clienți mai tineri de 45 de ani. Astfel, în baza de date, trebuie introduse unele date exacte ale clientului, cum ar fi data nașterii. Cu toate că SGBD-ul permite modificarea structurii bazei de date, introducerea datelor pentru toți clienții poate fi un lucru dificil.

De asemenea, din motive de securitate și folosire eficientă a resurselor calculatorului, este importantă alegerea unei forme adecvate de păstrare a datelor. Datele pot fi criptate pentru a nu fi copiate sau comprimate pentru a ocupa mai puțin spațiu de stocare.

Există o teorie întreagă de comprimare și cifrare a datelor care poate fi făcută în funcție de caracteristicile datelor stocate și volumul de memorie necesară pentru metodele convenționale de păstrare.

#### 1.4.4. Metadatele

Evoluția bazelor de date îi face pe informaticieni să țină cont de anumite probleme și să le soluționeze pe măsură ce se perfecționează tehniciile utilizate.

Din momentul în care se creează o bază de date și până în momentul în care se decide cumpărarea unui sistem mai bun sau elaborarea unei baze noi, se schimbă structura bazei de date în măsura schimbării necesităților de date în domeniul de interes. Astfel, în afara actualizării bazei de date, poate apărea necesitatea introducerii unor câmpuri noi în înregistrări, crearea unor fișiere secundare etc.

Deoarece baza de date este utilizată în rezolvarea unor probleme specifice din domeniul de interes și poate lua diverse forme atât după conținut, cât și structură, se creează o structură specială de date pentru reflectarea tuturor schimbărilor ce au loc în baza de date. De asemenea, această structură poate conține date despre obiectivele fiecărui câmp, despre aplicațiile care le utilizează, sau dacă pot fi suprimate de cel ce le utilizează etc.

Astfel, înainte de modificarea schemei sau structurii a bazei de date personalul administrativ trebuie să consulte cu atenție aceste informații pentru a nu comite erori care se pot răsfrânge asupra bunei funcționări a întregului sistem.

**Definiția 1.3.** Datele care descriu baza de date se numesc *metadate*.

Metadatele, de asemenea, se păstrează sub forma unei baze de date și poate fi gestionată și consultată în același mod. Baza de date construită din metadate se mai numește *dicționar de date*.

#### 1.4.5. Componenta software

Această componentă este alcătuită din: sistemul de operare, programele de rețea și programele sistemului de gestiune a bazei de date. La rândul său, SGBD-ul încorporează instrumente din generația a IV-a, cum ar fi SQL, ce permit: dezvoltarea rapidă de aplicații, îmbunătățirea semnificativă a productivității, realizarea unor programe ușor de întreținut.

Unele funcții ale SGBD-ului au fost descrise anterior, iar rolul lui, în sistemul orientat pe baze de date, decurge din obiectivele bazei de date.

Urmându-l pe A.Silberschatz, SGBD-ul poate fi considerat un program care stabilește interfață dintre datele stocate pe dispozitivele electomagnetice și programele aplicații, precum și interogările formulate la baza de date.

Deja au fost specificate caracteristicile proprii unei baze de date, dintre care majoritatea rămân valabile și pentru SGBD. Pentru o mai mare explicitate, unele vor fi lărgite aici.

**Interacțiunea cu Sistemul de Operare.** Precum s-a menționat, SGBD-ul nu este decât un program. Sistemul de Operare (SO), la rândul său, este programul principal care are misiunea să supravegheze buna funcționare a calculatorului. SO asigură, printre altele, accesul la dispozitivele de intrare–ieșire, cum sunt tastatura, șoriceul, monitorul, precum și accesul la dispozitivele de stocare a datelor, de exemplu, discuri fixe, dischete, CD-ROM-uri, benzi magnetice etc.

Astfel, pentru a nu se întâmpla ceva neobișnuit, SO este singurul program care poate accesa aceste dispozitive. Dat fiind faptul că SGBD-ul solicită stocarea datelor, de exemplu, pe discul fix, el interacționează cu SO care îi oferă această posibilitate. De asemenea, pentru recuperarea unor date de pe dispozitivele de stocare întotdeauna este contactat SO.

**Menținerea integrității.** Precum s-a mai afirmat, sistemul trebuie să asigure validitatea constrângerilor de integritate. De exemplu, să evite varianta când un client al itinerarului de aventuri întrece vîrstă de 90 de ani.

**Asigurarea securității.** SGBD-ul trebuie să evite accesul fraudulos la date, cum ar fi, de exemplu, la datele cifrate.

**Asigurarea copiilor de securitate.** Deoarece calculatorul nu este un sistem infailibil și poate fi afectat atât din motive tehnice proprii (pană de circuit), cât și străine (creșterea tensiunii în rețeaua electrică), este posibilă deteriorarea sau pierderea unei părți de date.

Pentru evitarea urmărilor neplăcute și fixarea veracității conținutului bazei de date, se recurge la efectuarea copiilor datelor pe dispozitive auxiliare de stocare. Dacă datele originale sunt distruse, este suficientă transferarea copiei pe discul fix al calculatorului. Datele vor lua aceeași formă ca și datele originale. Dar, pentru a recupera datele care au fost modificate între momentul copierii și momentul distrugerii, se refac doar schimbările care au avut loc în acest interval.

**Controlul concurenței.** Precum s-a menționat anterior, SGBD-ul trebuie să asigure accesul simultan la date pentru o parte de utilizatori, sarcină însotită de numeroase probleme de coerență și coordonare. Astfel, SGBD-ul trebuie să controleze acest proces, pentru ca informațiile reprezentate de date în urma oricărei intervenții a utilizatorului să rămână consistente.

**Furnizarea mecanismelor de facilitare a interacțiunii cu baza de date.** Aceste mecanisme iau forma unor limbaje de definiție și manipulare a datelor.

În afară de aceasta, SGBD-ul **asigură independența datelor** în sensul că, în ciuda evoluției structurii datelor, aplicațiile nu suferă modificări sau modificările sunt minimale. De exemplu, fie că aplicațiile vechi au fost orientate pentru lucrul cu numere de telefon exprimate în formă numerică. Ce se întâmplă, dacă se ia hotărârea de a se trece toate telefoanele în format alfanumeric? Aici trebuie să existe un mecanism care ar ascunde de aplicațiile vechi formatul nou al numerelor de telefon. Adică aplicațiile vechi trebuie să aibă o vizionare care diferă de cea care, realmente, este aplicată pentru datele stocate.

#### 1.4.6. Utilizatorii bazei de date

Până aici, se subînțelegea că utilizatori sunt persoanele care doresc să extragă informații din baza de date. Este corect, dar utilizator poate fi considerată și orice persoană responsabilă de proiectarea și actualizarea bazei de date. Convențional, utilizatorii pot fi clasificați în patru grupuri mari: administratorii, programatorii de aplicații, utilizatorii cu pregătire specială și utilizatorii obișnuiți.

**Administratorul.** Administratorul este o persoana sau un colectiv de persoane responsabile (în majoritatea cazurilor) de controlul total al funcționării bazei de date. Între sarcinile sale intră:

- **Definirea schemei bazei de date.** Precum s-a menționat, schema bazei de date este compusă din specificațiile tipurilor de date care se stochează în baza de date și constrângerile definite asupra acestor tipuri.

Administratorul este cel care decide ce se va stoca și cum. Datele despre schema bazei de date sunt incluse în dicționarul bazei de date, acolo unde sunt păstrate metadatele.

- **Definirea structurii de păstrare și metodelor de acces.** Anterior, s-a menționat că SGBD-ul interacționează cu SO pentru a putea accesa datele păstrate în fișiere pe dispozitivele de stocare. Aceste fișiere, la rândul lor, sunt structurate în baza unor unități de date mai mici cu scopul de a facilita gestionarea lor de către SO. În sarcina administratorului intră și specificarea caracteristicilor acestor unități minime de stocare. Ele influențează viteza de recuperare și fluxul de tranzacții, de aceea, se stabilesc în funcție de caracteristicile cererilor mai frecvent formilate de utilizatori.

De asemenea, în baza cererilor mai frecvente, administratorul decide asupra necesității menținerii unor structuri fizice pilon, care fac mai ușoară extragerea datelor specificate de utilizatorii. Când se vorbește despre structuri-pilon se au în vedere indecșii. Spre satisfacția utilizatorilor, odată cu construirea indecșilor corespunzători, crește vertiginos viteza de prelucrare a cererilor de extragere a datelor. Indecșii sunt menținuți de sistem și pot fi creați chiar imediat înaintea operațiilor de inserare, suprimare sau modificare a datelor.

Trebuie menționat că indecșii ocupă un spațiu a cărui mărime depinde de fișierul original și cantitatea de memorie ce trebuie să fie proporțională cu facilitățile pe care acești indecși le oferă.

- **Modificarea schemei și organizării fizice a bazei de date.** Pe măsură ce domeniul de interes pentru care este construită baza de date evoluează pentru a corespunde cerințelor vieții, apare necesitatea reflectării acestor schimbări și în baza de date. Astfel,

trebuie să fie posibilă schimbarea atât a schemei bazei de date, cât și a structurii fizice de organizare pentru a corespunde noilor necesități.

Prin urmare, administratorul bazei de date, dar și utilizatorii sunt interesați ca aplicațiile vechi să funcționeze în continuare cu modificări minime. Fiecare aplicație veche are definită o viziune asupra bazei de date. Bineînțeles, aceste viziuni nu trebuie să afecteze coerenta datelor stocate cu schema nouă.

- **Concesiunea permiselor și privilegiilor de acces la date.** Pentru asigurarea securității datelor în bazele de date, datele dispun de o serie de nivele de importanță, astfel că pot fi accesate numai de utilizatorii cu un privilegiu superior sau egal nivelului de importanță respectiv. Administratorul este persoana indicată să atribuie datelor aceste nivele și să elibereze utilizatorilor permise în funcție de caracteristicile particulare ale fiecărui utilizator. De asemenea, el oferă fiecărui utilizator chei de acces pentru ca aceștia să poată dialoga cu sistemul, utilizând numai o parolă particulară.
- **Specificarea constrângerilor de integritate.** Metadatele descriu și constrângerile pe care trebuie să le satisfacă datele din baza de date în funcție de necesitățile pe care trebuie să le acopere, astfel ca informațiile care se extrag să fie utile și coerente. Această coerentă este determinată de o mulțime de reguli sau constrângerile pe care datele trebuie să le respecte. Administratorul trebuie să specifiche aceste constrângerile de integritate, astfel ca ele să reflecte flexibilitatea cu care evoluează necesitățile utilizatorilor din domeniul de interes.

Deseori, aceste constrângerile de integritate sunt considerate parte integrantă a schemei bazei de date și sunt incluse în dicționarul de date.

**Programatorii de aplicații.** Aceștia sunt profesioniștii ce interacționează cu sistemul, folosind instrucțiuni, scrise în limbajele de interogare a datelor și de manipulare a datelor, pe care le încorporează în cadrul unor interfețe create în alte limbaje de programare. Programatorii de aplicații sunt utilizatorii care recepționează cerințele altor utilizatori și elaborează diverse programe care satisfac aceste necesități. De obicei, programele

sunt scrise în limbaje de programare (Basic, Java, C#, etc.) în care se includ instrucțiuni speciale capabile să înțeleagă SGBD-ul. Precompilatorul convertește apelurile scrise în limbajul de interogare a datelor în proceduri specifice limbajului-gazdă. Compilatorul limbajului-gazdă generează apoi codul-obiect. Astfel, SGBD-ul furnizează datele, iar limbajul convențional, numit, deseori, limbaj-gazdă, le prelucrează.

**Utilizatorii cu pregătire specială.** Aceștia interacționează cu sistemul fără a scrie programe, dar interacționează cu baza de date, făcând uz direct de limbajele de interogați propuse de SGBD. Într-unul din capitolele ulterioare, se va examina limbajul SQL în calitate de limbaj principal de definire și manipulare a datelor în bazele de date. Administratorul trebuie să aibă grijă să acorde permisiunile respective acestor utilizatori pentru a putea satisface necesitățile informaționale.

**Utilizatorii obișnuiți.** Utilizatorii obișnuiți sunt acei care nu știu nimic despre baza de date cu excepția că ea există. Acești utilizatori sunt incapabili să folosească limbajele de acces oferite de SGBD. Ei interacționează cu baza de date prin intermediul interfețelor sau aplicațiilor elaborate de programatori.

### 1.5. Arhitectura bazei de date cu trei nivele

Unul din obiectivele bazei de date constă în asigurarea independenței dintre aplicații și datele stocate. Cu alte cuvinte, schimbările din structura datelor trebuie să aibă represență minimă asupra programelor și viceversa. Conceptul de independență presupune separarea stocurilor de date de descrierea lor în programe. Acest obiectiv a avut mare influență asupra arhitecturii SGBD-urilor.

În sistemele cu fișiere de până la apariția bazelor de date, existau doar două structuri: logică externă (viziunea utilizatorului) și fizică (forma în care se prezintau datele în memorie).

În bazele de date, apare un nivel de abstracție nou numit nivelul logic, reprezentat de structura logică sau schema logică globală. Această structură intermediară este o reprezentare globală de date independentă de echipamente și utilizatori. Celelalte două niveluri corespund nivelului intern sau fizic și nivelului extern sau viziunilor utilizatorilor. Această

arhitectură cu trei nivele (figura 1.4) este cunoscută sub numele de arhitectura ANSI/X3/SPARC [ANSI78].

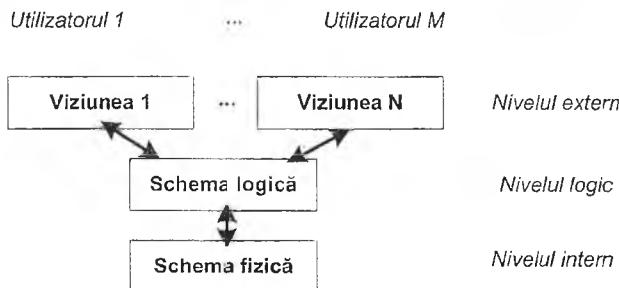


Figura 1.4. Arhitectura ANSI/X3/SPARC

Un sistem de baze de date suportă o schemă internă, o schemă conceptuală și mai multe scheme externe. Toate aceste scheme sunt descrierile diferite ale aceleiași colecții de date, care există doar în nivelul intern.

**Schema internă (fizică) a bazei de date (*internal schema*)** specifică modul de reprezentare a datelor pe suportul fizic. Modul în care SGBD-ul și SO percep datele este numit nivel intern.

**Schema logică a bazei de date (*logical schema*)** corespunde unei reprezentări unice (pentru toți utilizatorii) și abstracte a datelor, descriind ce date sunt stocate în baza de date și care sunt asociările dintre acestea. Nivelul logic realizează atât transpunerea, cât și independența dorită dintr-un nivel extern și cel intern.

**O schemă externă sau viziunea utilizatorului (*external schema, user's view*)** conține o subschemă logică a bazei de date, mai precis descrierea datelor care sunt folosite de un grup de utilizatori. Astfel, modul în care utilizatorii percep datele este numit nivel extern.

### 1.5.1. Nivelul intern

Nivelul intern descrie structura fizică, de păstrare a bazei de date. Modul de organizare a bazei de date fizice este, în mare măsură, influențat de configurația echipamentelor hardware care suportă baza de date și de sistemul de operare. Schimbarea sistemului de operare sau modificările în

configurația hardware pot atrage modificări ale bazei de date fizice. Dacă este satisfăcută condiția de independentă fizică, aceste modificări în nivelul intern al bazei de date nu vor ataca nivelele superioare ale acesteia.

**Definiția 1.3.** O descriere a detaliilor de stocare (fișierele de stocare și organizarea accesului la datele bazei de date), utilizând un model fizic de date, se numește *schemă fizică*.

Schema fizică este dependentă de SGBD-ul utilizat. Cu toate acestea, există elemente comune ce trebuie menționate:

- **Strategia de memorare.** Se stabilesc spațiul de memorare a datelor și legăturile dintre spațiile distincte de memorare. De asemenea, se specifică strategia de amplasare a datelor, care va fi utilizată pentru optimizarea timpului și spațiului memoriei secundare. În afară de aceasta, se consideră modelele de tratare a cazurilor de insuficiență a spațiilor de memorare.
- **Cările de acces.** Se specifică cheile primare, secundare, indecsii, cheile de ordonare.
- **Tehnicile de comprimare a datelor.**
- **Tehnicile de criptografie.**
- **Corelația logic-fizic.** Se specifică modelele de transformare -a unităților de date de la nivelul logic în unități la nivelul fizic. Dacă se schimbă definiția structurii de memorare, corelația logic-fizic se modifică, de asemenea, pentru a nu schimba schema logică. Este o sarcină a administratorului bazei de date să controleze aceste modificări.
- **Tehnicile de arhivare și optimizare.**
- **Structura memoriei.** Se specifică mărimea paginii, numărul de pagini atribuite fiecărei arii de memorare, mărimea tamponelor de intrare-iesire.
- **Organizarea fizică.** Pentru a îmbunătăți extragerea și timpul de acces, sistemul oferă administratorului un mecanism pentru definirea hash-ărilor, agregărilor etc. În funcție de SGBD, pot fi definite puncte între fișiere, favorizând anumite căi de acces.

- **Controlul accesului.** Se definesc reguli pentru asigurarea confidențialității și securității bazei de date.

### 1.5.2. Nivelul logic

Fiecare SGBD are un model logic propriu prin care sunt numite și descrise toate entitățile logice din baza de date împreună cu legăturile dintre acestea. Nivelul logic este reprezentat de schema logică a bazei de date care descrie structura bazei de date și constrângerile asociate ei. Schema logică ascunde detaliile de stocare fizică a datelor și oferă, prin urmare, un nivel de abstracție mai important (pentru utilizatori) decât nivelul fizic. Astfel, nivelul logic conține schema logică a bazei de date, așa cum este ea văzută de administratorul bazei de date.

Schema logică se descrie cu ajutorul unui model logic de date de nivel înalt. De obicei, modelul logic este modelul pe care este construit SGBD-ul. Astăzi, există mai multe modele logice de date folosite în bazele de date: ierarhic, de tip rețea, relațional, obiect-relațional... Cel mai răspândit este modelul relațional care reprezintă datele sub formă de tabele interdependente și supuse unor constrângeri de integritate.

Pentru construirea schemei logice, sistemul de gestiune oferă un limbaj de definire a datelor. Trebuie reținut faptul că modelul logic este o descriere a conținutului de date din baza de date și nu cuprinde niciun fel de referire la modul de memorare a datelor sau la strategia de acces.

Trebuie menționat, că, uneori, în terminologia curentă, schema logică este numită schemă conceptuală a bazei de date, ceea ce provoacă ambiguități și se confundă, deseori, cu faza de proiectare conceptuală a bazei de date.

### 1.5.3. Nivelul extern

Nivelul extern descrie acea parte a bazei de date care este relevantă pentru fiecare utilizator. Nivelul extern este cel mai apropiat utilizatorului. Este ceea ce vede acesta din baza de date, sau modul cum vede acesta baza de date. Modelul extern este derivat din cel logic, dar poate prezenta deosebiri substanțiale față de acesta. Deseori, pentru modelul extern este folosit termenul *vedere* sau *viziune*. Prin aceste vizuri, utilizatorii au acces doar

la părți bine definite din baza de date, fiindu-le ascunse părțile care nu interesează.

Diferite viziuni pot avea reprezentări diferite ale acelorași date. Nivelul extern include numeroase viziuni externe. Fiecare viziune descrie o parte a schemei logice de care este interesat utilizatorul particular. În același timp, viziunea ascunde de utilizator toate celelalte elemente ale schemei logice. Aplicațiile, de asemenea, propun utilizatorilor viziuni parțiale și personalizate de date.

Schema externă și mecanismul viziunilor oferă următoarele facilități:

- **Adaptarea la aplicații.** Aplicațiile nu depind de schema adoptată la nivelul logic. Utilizatorul manipulează mai facil datele sub forma lor obișnuită și nu le caută sau le prelucrează în sănul unei cantități importante de date nerelavante.
- **Integrarea aplicațiilor existente.** Datele văzute de utilizator, fiind aceleași ca și datele de până la accesul la SGBD, pot fi, cu mici modificări, ușor adaptate la aplicațiile vechi.
- **Dinamica schemei logice.** Modificările schemei logice, uneori, necesare din considerente de performanță, nu implică după sine schimbarea schemelor la nivelul extern, dacă ultimele nu evoluează. Dacă această regulă este respectată, modificările la nivel logic nu provoacă rescrierea aplicațiilor.
- **Confidențialitatea și securitatea.** Administratorul bazei de date poate defini viziuni pe care le utilizează pentru specificarea exactă a dreptului de acces al utilizatorului.
- **Descentralizarea administrării bazei de date.** Fiecare utilizator sau fiecare echipă de utilizatori sunt responsabili pentru datele pe care le gestionează în limitele drepturilor oferite de administratorul bazei de date. La rândul lor, ei pot deveni administratorii propriilor viziuni și pot delega o parte din drepturile lor altor utilizatori.
- **Eterogenitatea modelelor.** Se poate prezenta un acces omogen și unificat la o mulțime de baze de date de tipuri diferite. Mecanismul viziunilor externe permite această tratare omogenă.

#### 1.5.4. Independența datelor

Arhitectura în trei nivele asigură independența datelor. Independența datelor înseamnă că există o delimitare strictă între *reprzentarea fizică* a datelor și *viziunea* pe care o are utilizatorul asupra acestor date. Modul concret, în care este realizată memorarea și organizarea datelor, este invizibil pentru utilizator. Acesta este preocupat numai de problema concretă pe care o are de rezolvat. Detaliile de implementare rămân în sarcina sistemului. Astfel, o schemă de la un nivel poate fi modificată fără a provoca schimbarea schemei de la nivelul adiacent. Problema independenței datelor prezintă două aspecte ale datelor: *independența logică* a datelor și *independența fizică* a datelor.

**Definiția 1.4.** *Independența logică* reprezintă posibilitatea de modificare a schemei logice a bazei de date fără implicarea modificării viziunilor externe.

Prin urmare, independența logică a datelor se referă la imunitatea programelor aplicative ale fiecărui utilizator față de modificările în structura logică globală a bazei de date. Astfel, pot fi adăugate noi entități logice în structura bazei de date, fapt ce asigură dezvoltarea bazei de date fără a afecta utilizatorii deja existenți. Și, viceversa, elaborarea de noi aplicații nu presupune schimbarea schemei bazei de date [Stonebraker74].

Existența independenței logice constituie un avantaj semnificativ al bazei de date în raport cu tehniciile vechi de gestiune a fișierelor, unde fiecare program aplicativ includea o descriere a datelor din fișiere. Cu toate acestea, independența logică este greu de asigurat în totalitate. De exemplu, nu este rezolvată problema eliminării de entități logice din baza de date, întrucât orice operație de acest gen se repercutează asupra utilizatorilor care fac referire la entitatea eliminată.

**Definiția 1.5.** *Independența fizică* reprezintă capacitatea de modificare a schemei fizice fără implicarea modificării viziunilor externe.

Astfel, independența fizică a datelor este o măsură a imunității datelor față de modificările care pot să apară în structura fizică de memorare a datelor. O modificare a acestei structuri nu va afecta aplicația și reciproc,

modificări ale aplicației vor lăsa nealterată structura fizică a datelor. Structura fizică a datelor este determinantă pentru strategiile de acces folosite pentru regăsirea datelor [Date71]. O aplicație, care este independentă de structura fizică a datelor, nu conține nicio referire la dispozitivele de memorare folosite sau la strategiile de acces la date.

Din punct de vedere al independenței fizice, detaliile legate de dispozitivele de memorare sau strategiile de acces nici nu trebuie să fie cunoscute de utilizator.

#### 1.5.5. Funcționarea arhitecturii cu trei nivele

Principiul de funcționare a arhitecturii ANSI/X3/SPARC se bazează pe următoarele activități.

O interogare exprimată de un utilizator, într-un limbaj acceptabil de sistem, este analizată mai întâi din punct de vedere sintactic (în conformitate cu gramatica limbajului), apoi, din punct de vedere semantic (obiectele citate trebuie să fie cunoscute în schema externă a utilizatorului).

După această validare, făcută de nivelul extern, cererea este tradusă pentru transmiterea acesteia la nivelul logic: referințele la obiectele de la nivelul extern sunt substituite cu referințele la obiectele respective din schema logică. Pentru aceasta, se utilizează descrierea regulilor de mapare a schemei externe în cea logică, stabilită, neapărat, în momentul definirii schemei externe.

La nivelul logic, se face controlul în privința confidențialității, concurenței etc. Dacă cererea este acceptată, ea este optimizată și decupată în subcereri mai elementare, care sunt transferate la nivelul intern, în caz contrar, cererea este pusă în așteptare sau refuzată.

La nivelul intern, fiecare subcerere este tradusă într-una sau mai multe cereri fizice, respectiv (în funcție de datele conținute în schema fizică), apoi SGBD-ul realizează accesul fizic la date (extragere sau modificare). Dacă este vorba de o cerere de interogare, datele extrase sunt trecute în stratul logic, apoi în stratul extern. Aici, înainte de a fi transmise

utilizatorului, ele sunt reorganizate potrivit schemei externe a utilizatorului.

## 1.6. Structura modelului relațional

Actualmente, pentru majoritatea aplicațiilor care utilizează bazele de date, modelul cel mai răspândit este cel relațional, datorită simplității, eleganței, potențialului și formalismului matematic pe care se bazează.

Acest model permite reprezentarea informației despre lumea reală într-un mod intuitiv, folosind concepte cotidiene și facilitând înțelegerea lui de orice utilizator. Acesta poate menține date despre caracteristicile proprii ale bazei de date (metadate), fapt care facilitează modificarea ei, diminuând problemele ocazionale în aplicațiile deja elaborate. Pe de altă parte, sunt introduse și mecanisme de interogare mai puternice, totalmente independente de SGBD, și, inclusiv, de organizarea fizică a datelor. Pentru SGBD, rămâne sarcina să optimizeze aceste interogări în format standard, potrivit caracteristicilor proprii de stocare.

Modelul relațional a fost propus în 1970 de E.F.Codd [Codd70] în laboratoarele din California ale companiei IBM. Acesta este tratat ca un model logic care stabilește o structură a datelor, cu toate că apoi acestea pot fi păstrate în multiple forme pentru a profita de caracteristicile fizice concrete ale mașinii pe care se implementează realmentă baza de date.

Urmează o studiere a caracteristicilor concrete ale acestui model de date, fără a intra în aspectele de păstrare fizică sau legate de un SGBD concret. Orice model de date se tratează din trei puncte de vedere: structura, constrângerile de integritate și operațiile de manipulare a elementelor structurii.

Numele de model *relational* provine de la legătura strânsă care există între elementele de bază ale acestui model și de la conceptul matematic de relație.

Una din calitățile de bază ale modelului relațional este omogenitatea lui. Toate datele sunt păstrate în tabele, ale căror linii au aceeași structură fiecare. Linia, într-un tabel, reprezintă un obiect (sau o relație între obiecte) din lumea înconjurătoare.

### 1.6.1. Atribute și domenii

În sistemele orientate pe procese, câmpul este cea mai mică unitate accesibilă de date a fișierelor. Se presupune că fiecare câmp poate conține un anumit tip de date (*INTEGER*, *REAL* etc.), pentru care se specifică numărul necesar de octeți de memorie. Câmpul, bineînțeles, are și un nume.

Prin analogie, în modelul relațional, fiecare coloană a unei linii dintr-un tabel corespunde noțiunii de câmp din fișiere. Astfel, în tabel, coloana este identificată de numele câmpului care se va numi nume de atribut. Cu alte cuvinte, orice linie este formată dintr-o mulțime de date structurate în elemente mai simple numite attribute. Numele atributului trebuie să descrie semnificația datelor ce le reprezintă. În tabelul *studenți* din figura 1.5, atributul *Facultate* indică facultatea la care își face studiile un student, pe când atributul *Nota\_Med* nu este clar la ce se referă. Evident, attributele semnifică diverse obiecte din domeniul de interes și denumirile lor au o semantică concretă într-un context dat.

**Definiția 1.6.** Fie  $A_1, A_2, \dots, A_n$ , numele de attribute sau simplu attribute care reprezintă toate obiectele domeniului de interes. Atunci mulțimea  $R = \{A_1, A_2, \dots, A_n\}$  se numește mulțime universală sau univers.

Este clar că un atribut într-o linie nu poate lua orice valoare, de exemplu, *Nota\_Med* nu poate depăși 10. Pentru a evita acest tip de situații, orice atribut trebuie să ia valori dintr-o mulțime de valori prestabilite.

Toate attributele, într-un tabel, trebuie să fie distințe și să fie luate din universul  $R$ . Attributele au un caracter global în baza de date: dacă un nume denotă două coloane în tabele distințe în aceeași bază de date, atunci el reprezintă același atribut.

**Definiția 1.7.** Domeniul unui atribut  $A_i$  din  $R$ ,  $1 \leq i \leq n$ , notat cu  $\text{dom}(A_i)$ , este o mulțime finită sau infinită de valori de același tip care le poate lua atributul  $A_i$ .

Astfel atributul *Nota\_Med* poate lua numai valori numerice, în timp ce *Facultate* poate conține fraze textuale.

Domeniul este *simplu*, dacă elementele sale sunt atomice (adică nu pot fi descompuse). Atributul care se asociază cu un domeniu de valori simplu se numește *atribut atomic*. Domeniul unei submulțimi  $R^1$  a universului  $R$ , se notează  $\text{dom}(R^1)$ , este uniunea tuturor domeniilor atributelor din  $R^1$ , adică  $\text{dom}(R^1) = \bigcup_{A_i \in R^1} \text{dom}(A_i)$ .

Trebuie menționat că deși attributele în universul  $R$  sunt distințe, domeniile acestor attribute nu trebuie neapărat să fie disjuncte. De exemplu, managerul, în același timp, poate fi funcționar. Deci, domeniile atributelor *Manager* și *Funcționar* nu sunt disjuncte, adică *Manager* și *Funcționar* sunt definite pe același domeniu (cu toate că attributele respective pot avea și valori distincte).

Domeniile cu care se poate asocia un atribut este dependent de SGBD-ul utilizat. Bineînțeles, că pot exista tipuri de domenii comune pentru majoritatea SGBD-urilor: text, numere întregi, data etc. Totodată, un domeniu de tip număr întreg poate fi infinit. Dat fiind faptul că se lucrează cu calculatorul este inevitabilă fixarea unei limite pentru valori. De exemplu, atributul *Facultate*, care este de tip text, se poate limita la 50 de caractere. Unele SGBD-uri simplifică sarcina de indicare a domeniilor atributelor, stabilind un domeniu implicit sau o ierarhie de domenii, divizând un tip de domeniu în subtipuri.

Independent de domeniul asociat, atributul poate lua o valoare specială care desemnează lipsa valorii, *NULL*. Când, din anumite circumstanțe, nu se cunoaște valoarea unui atribut sau valoarea lipsește, se poate asocia acestui atribut această valoare specială, comună tuturor domeniilor. Trebuie menționat că există diferență între valoarea *NULL* și valoarea "spațiu" sau "blanc". Calculatorul consideră un spațiu ca un caracter, deci introducerea spațiului este distinctă de conceptul de lipsă a datelor.

**Convenție.** Pentru a simplifica expresiile, cu caractere majuscule de la începutul alfabetului latin, se vor nota attributele singulare, iar cu cele de la sfârșitul alfabetului latin - mulțimi de attribute. În afară de

aceasta, uneori, se va folosi o notație mai simplă  $A_i \subseteq R$ , în loc de  $\{A_i\} \subseteq R$ . Reuniunea  $Y \cup Z$  a două mulțimi  $Y$  și  $Z$  va fi reprezentată de caracterele  $YZ$ , unde operația binară uniunea, "∪", este omisă. Parantezele figurate în notația mulțimilor pot fi suprimate, de exemplu,  $R = A_1 A_2 \dots A_n$  în loc de  $R = \{A_1, A_2, \dots, A_n\}$ .

### 1.6.2. Tupluri

În sistemele orientate pe fișiere, o mulțime de câmpuri tratată ca o unitate de salvare și căutare se numește înregistrare. Înregistrarea are un format specific și depinde de tipurile de date ale câmpurilor. O linie într-un tabel corespunde înregistrării din fișiere și în teoria relațională se numește *tuplu*.

**Definiția 1.8.** Fie  $R^1$  o submulțime a universului  $R$ ,  $R^1 \subseteq R$ , unde  $R^1 \neq \emptyset$  și fie  $dom(R^1)$  domeniul mulțimii  $R^1$ . Atunci *tuplu* se numește o funcție,  $t: R^1 \rightarrow dom(R^1)$ , din  $R^1$  în  $dom(R^1)$ , adică

$$t = \{(A_1, a_1), \dots, (A_k, a_k)\},$$

unde  $A_j$ ,  $1 \leq j \leq k$ , este un atribut în  $R^1$  și un argument al lui  $t$ , iar  $a_j$ ,  $1 \leq j \leq k$ , este o valoare în  $dom(A_j)$ .

**Definiția 1.9.** Fie  $X = B_1 \dots B_m$  o submulțime proprie a mulțimii  $R$ ,  $X \subset R$ , unde  $X \neq \emptyset$ . Atunci  $X$ -valoarea a tuplului  $t$ , notată cu  $t[X]$ , este

$$t[X] = \{(B_i, b_i) \mid b_i = t(B_i) = t(A_j),$$

$$1 \leq i \leq m, j \in \{1, \dots, k\}\}.$$

Dacă  $X = A_j$ ,  $j \in \{1, \dots, k\}$ , atunci  $A_j$ -valoarea tuplului  $t$  se numește componența  $A_j$  a tuplului  $t$ .

Din definiția 1.9 rezultă, că  $t(A_j) = t[A_j] = a_j$  pentru  $a_j \in \text{dom}(A_j)$ . Deci, în cazul unui atribut singular  $A_j$ , notațiile  $t(A_j)$  și  $t[A_j]$  pot fi utilizate deopotrivă.

Uneori, pentru tuplul  $t$  și  $X$ -valoarea tuplului  $t$  pot fi acceptate notațiile

$$t = \langle a_1, \dots, a_k \mid A_1, \dots, A_k \rangle \text{ și}$$

$$t[X] = \langle b_1, \dots, b_m \mid B_1, \dots, B_m \rangle,$$

corespunzător. Însă, dacă se ține seama de ordinea atributelor, tuplul  $t$  și  $X$ -valoarea tuplului  $t$  pot fi reprezentate, respectiv,

$$t = \langle a_1, \dots, a_k \rangle \text{ și}$$

$$t[X] = \langle b_1, \dots, b_m \rangle.$$

În tuplul  $t = \langle a_1, \dots, a_k \mid A_1, \dots, A_k \rangle$  se disting două componente – secvența de atrbute  $A_1, \dots, A_k$  care este invariantă în timp și secvența de valori  $a_1, \dots, a_k$ , care este foarte dinamică. Partea invariantă a tuplului se numește *schema tuplului* (uneori, se va nota  $\text{sch}(t)$ ). Îndată ce este definit conceptul de schemă a tuplului, expresia “tuplul  $t$  asupra  $R^1$ ” devine clară și este echivalentă expresiei “tuplul  $t$  cu schema  $R^1$ ”.

### 1.6.3. Scheme și relații

**Definiția 1.10.** Fie  $R$  mulțimea universală de atrbute. Orice submulțime  $R_i$  de atrbute a mulțimii  $R$ ,  $R_i \subseteq R$ , se numește *schemă*. Iar mulțimea  $R$  se numește *schemă universală*.

**Definiția 1.11.** Fie  $R$  schema universală. Schemă a bazei de date se numește o mulțime finită de scheme  $BD = \{R_1, \dots, R_m\}$ , unde  $R = R_1 \cup \dots \cup R_m$ .

În modelul relațional, un tabel se numește relație.

**Definiția 1.12.** Fie  $R_i$  o submulțime a schemei universale  $R$ .

Atunci o relație  $r$  asupra schemei  $R_i$ , se notează  $r(R_i)$ , este o mulțime finită de tupluri cu schema  $R_i$ .

O relație asupra unei mulțimi de atrbute pot avea un nume. Numele relației, de obicei, se scrie cu minuscule, de exemplu, relația  $r$ .

*Gradul* unei relații  $r(R_i)$  este numărul de atrbute din care este constituită schema  $R_i$ . Astfel relația *studenți*, din figura 1.5, este de gradul 4. *Cardinalitatea* unei relații constă în numărul de tupluri, din care este constituită relația. Trebuie menționat că gradul unei relații este independent de momentul în care se examinează relația. Numărul de atrbute și, propriu-zis, atrbutele sunt invariabile și rămân așa cum au fost definite în momentul creării relației. Însă, cardinalitatea depinde de situația reală, în care se reprezintă relația și poate varia în timp. De exemplu, cardinalitatea relației *studenți* este în funcție de numărul de studenți care își fac studiile la universitate.

În literatura de specialitate, există și altă definiție a relației.

**Definiția 1.13.** Fie o mulțime de atrbute  $R_i = \{A_1, \dots, A_k\}$ , unde  $R_i \subseteq R$ , și fie  $\text{dom}(A_1), \dots, \text{dom}(A_k)$  domeniile respective de valori. Atunci, o relație definită pe schema  $R_i$  este o submulțime a produsului cartezian al domeniilor atrbutelor ce constituie schema  $R_i$ ,

$$r(A_1, \dots, A_k) \subseteq \text{dom}(A_1) \times \dots \times \text{dom}(A_k).$$

Din cele expuse mai sus, se poate concluziona:

- Într-o relație, nu există atrbute cu nume duplicate, deoarece atrbutele  $A_j$ ,  $1 \leq j \leq k$ , sunt elemente ale unei mulțimi  $R_i$ .
- Relația  $r$  nu are tupluri identice, fiindcă  $r$  este o mulțime de tupluri.

- Ordinea tuplurilor în  $r$  este nesemnificativă, fiindcă  $r$  este o mulțime.
- Ordinea atributelor în schema e nesemnificativă.
- Valorile atributelor sunt atomice și domeniile lor sunt simple.

**Definiția 1.14.** O *bază de date relațională* (sau simplu *bază de date*) definită pe schema  $BD = \{R_1, \dots, R_m\}$  este o mulțime finită de relații,  $bd = \{r_1(R_1), \dots, r_m(R_m)\}$ .

Relațiile care, realmente, se păstrează și constituie baza de date se numesc *relații de bază*. Există, însă, și situații în care relațiile nu sunt elemente ale bazei de date. Este cazul așa-numitelor *relații virtuale*, cunoscute și sub numele de *relații derive sau viziuni*. Relația virtuală nu este definită explicit ca relația de bază, prin mulțimea tuplurilor constitutive, ci implicit pe baza altor relații. Relațiile de bază sunt proiectate de administratorul bazei de date, în timp ce viziunile pot fi definite și de alții utilizatori ai bazei de date.

<i>studenți</i>			
<i>Nume</i>	<i>Nota Med</i>	<i>Facultate</i>	<i>Decan</i>
Vasilache	7.8	Cibernetică	Popovici

<i>discipline</i>	
<i>Facultate</i>	<i>Disciplină</i>
Cibernetică	Baze Date

Figura 1.5. Baza de date Universitatea

## 1.7. Constrângerile 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, 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, constrângerile sunt studiate mai ales sub aspectul puterii lor de modelare și al posibilității de verificare a respectării lor.

În secțiunea anterioară, s-a menționat că orice atribut trebuie să ia valori care aparțin unui domeniu concret, asociat acestui atribut. Aceasta constituie o constrângere de integritate asupra atributelor. Alte restricții comentate au fost: imposibilitatea de a repeta tuplurile în aceeași relație sau imposibilitatea de a stabili o ordine a tuplurilor în relație, deși unele SGBD-uri relaxează un pic această constrângere.

În această secțiune, se introduc și alte constrângerile de integritate, pe care le posedă modelul relațional. Constrângerile de integritate ale modelului relațional sunt de două tipuri [Date81]: constrângerile de integritate structurale și constrângerile de integritate de comportament.

### **1.7.1. Constrângerile de integritate structurale**

Constrângerile de integritate structurale, inerente modelului relațional, se definesc prin egalitatea sau inegalitatea unor valori din cadrul relațiilor. Din categoria constrângerilor de integritate structurală, fac parte:

- Constrângerea de unicitate a cheii.
- Constrângerea de minimalitate a cheii.
- Constrângerea referențială.
- Constrângerea entității.
- Dependențe între atrbute.

#### **1.7.1.1. Constrângerile de unicitate și minimalitate a cheii**

Întrucât relația reprezintă o mulțime de tupluri, iar o mulțime nu conține elemente duplicate, într-o relație nu pot fi tupluri identice. Deci, tuplurile sunt unice și trebuie să existe posibilitatea identificării lor în cadrul unei relații. Identificarea unui tuplu, fără a consulta toate componentele acestuia, a dus la apariția noțiunii de cheie.

**Definiția 1.15.** Fie relația  $r$  definită pe schema  $R$ , unde  $R \neq \emptyset$ . Mulțimea  $K$  de atribute, unde  $K \subseteq R$ , se numește *cheie* a schemei  $R$  (sau a relației  $r$ ), dacă  $K$  satisfac următoarele condiții:

1. Pentru orice două tupluri  $t_1$  și  $t_2$  din  $r$  are loc inegalitatea  $t_1[K] \neq t_2[K]$ .
2. Nici o submulțime  $K^1$  proprie a lui  $K$  nu posedă proprietatea 1.

Proprietatea 1, reprezintă *constrângerea de unicitate a cheii*. Ea permite  $K$ -valorilor să identifice, în mod univoc, toate tuplurile dintr-o relație. Respectarea proprietății de unicitate poate fi mai complicată, dacă însăși  $K$  conține o cheie  $K^1$ , adică  $K^1 \subset K$ . În acest caz, cu toate că atributele  $K$  sunt suficiente pentru identificarea tuplurilor, unele dintre atrbute nu sunt necesare, deci pot fi eliminate din  $K$  fără a se afecta unicitatea. În cazul în care  $K$  este o submulțime proprie a unei chei, atunci utilizarea unor astfel de  $K$ -valori pentru căutarea datelor poate descoperi tupluri ce coincid pe toate atrbutele din  $K$ .

Proprietatea 2 garantează că o cheie  $K$  este constituită numai din acelle atrbute ce sunt suficiente pentru a determina univoc celelalte componente ale tuplului.

Orice SGBD, îndată ce este enunțată vreo cheie, testează, în mod automat, constrângerea de unicitate. Nu se poate spune același lucru despre constrângerea de minimalitate. Constrângerea structurală de minimalitate se bazează pe semantica atrbutelor care fac parte din cheie.

**Definiția 1.16.** [Ullman88] Fie relația  $r$  definită pe schema  $R$ . Mulțimea  $K$  de atribute, unde  $K \subseteq R$ , care posedă proprietatea 1 se numește *supercheie*.

Prin urmare, orice cheie e și supercheie. Afirmația inversă nu e corectă.

Este evident că o mulțime vidă nu poate servi drept cheie a unei relații care conține mai mult de un tuplu. Orice relație are cel puțin o cheie. Cheia poate fi constituită din orice număr de atrbute, de la un singur atrbut până la mulțimea tuturor atrbutelor.

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 alege una dintre cheile candidate să servească în mod efectiv la identificarea univocă a tuplurilor. Ea se numește *cheie primară*. Primare se numesc și domeniile atributelor ce formează o cheie primară. Celelalte chei se numesc *chei alternate* sau *secundare*. Distincția dintre cheia primară și cheile secundare constă numai în modul de acces intern la date, pentru care SGBD-ul adoptă anumite decizii asupra organizării fizice a datelor.

Cheia unei relații se numește *cheie simplă*, dacă este constituită dintr-un singur atribut, iar atunci când este formată din mai multe attribute este denumită *cheie compusă*. Unele attribute ale unei chei compuse (evidenț, secundare) pot fi definite pe domenii primare.

#### 1.7.1.2. Constrângerea entității

Conceptul de cheie este așa de important, că implică și alte restricții fundamentale asupra bazei de date. Una dintre ele este *constrângerea entității*. Constrângerea poate fi enunțată astfel:

**Definiția 1.17.** [Date81] *Constrângerea entității* reprezintă proprietatea care nu admite nici unui atribut, care face parte dintr-o cheie a relației, să ia valoarea necunoscută, *NULL*.

E cunoscut faptul că unele attribute pot avea așa-numitele valori nedefinite sau necunoscute notate cu *NULL*. Constrângerea entității impune SGBD-ului la inserarea unui tuplu să testeze dacă valoarea cheii este cunoscută. Cu valori *NULL*, cheia își pierde rolul de identificator de tuplu. În același timp, problema de unicitate a cheii nu poate fi soluționată dacă valorile atributelor-cheie sunt necunoscute. Această constrângere este automat verificată de SGBD, îndată ce este definită mulțimea de chei-candidat.

#### 1.7.1.3. Constrângerea referențială

**Definiția 1.18.** O *cheie externă* este un atribut (grup de attribute) dintr-o schemă  $R_i$  definit (definite) pe același (aceleași) domeniu (domenii) ca și cheia primară a altrei scheme  $R_j$ . Relația  $r_i$  se

numește *relație care referă*, iar  $r_j$  poartă numele de *relație referită*.

Trebuie menționat că, în unele cazuri [Chen76], cheia externă poate să se găsească în aceeași relație cu cheia primară. Cu alte cuvinte, relația referită este și relație care referă. Dar acesta este un caz excepțional și, de regulă, acesta nu reprezintă o situație în care baza de date este corect proiectată.

*Constrângerea referențială* este în legătură directă cu conceptul de cheie externă:

**Definiția 1.19.** [Date81] Constrângerea referențială reprezintă proprietatea care asigură că, dacă mulțimea de atrbute  $X$  a schemei relației  $r_i$  este definită pe domenii primare, atunci trebuie să existe o relație de bază  $r_j$  cu o cheie primară  $Y$ , încât orice  $X$ -valoare din  $r_i$  să apară în calitate de  $Y$ -valoare în  $r_j$ .

Constrângerea referențială impune valorile din domeniile primare ale relației  $r_i$  să fie printre valorile cheii primare a relației  $r_j$ .

<i>studenți</i>			<i>facultăți</i>	
<i>Nume</i>	<i>Facultate</i>	<i>An</i>	<i>Facultate</i>	<i>Decan</i>
$n_1$	$f_1$	$a_1$	$f_1$	$d_1$
$n_2$	$f_1$	$a_2$	$f_2$	$d_2$
$n_3$	$f_2$	$a_3$	$f_3$	$d_3$

Figura 1.6. Atributul *Facultate* din relația *studenți* este o cheie externă

**Exemplul 1.1.** Fie relațiile *studenți* și *facultăți* din figura 1.6.

Se presupune că, la o facultate, și fac studiile mai mulți studenți și un student poate studia doar la o singură facultate. Cheia primară a relației *studenți* este atributul *Nume*. Relația *facultăți* posedă două chei candidate: *Facultate* și *Decan*. Fie *Facultate* este cheia primară. Atunci atributul *Facultate* al relației *studenți* este cheie externă. Conform constrângerii referențiale, toate valorile atributului *Facultate* al relației care referă trebuie să se conțină în relația referită.

Constrângerile de integritate structurale, referitoare la dependența atributelor, pot fi de mai multe tipuri:

- Dependențe funcționale.
- Dependențe multivaloare.
- Dependențe joncțiune.

Aceste constrângerii nu reprezintă subiectul actualei lucrări.

### **1.7.2. Constrângerile de integritate de comportament**

Constrângerile considerate până aici sunt constrângerile impuse de chei și sunt aplicabile la toate bazele de date, care folosesc modelul relațional, independent de domeniul de aplicație sau de datele care le stochează. Constrângerile de acest tip, de asemenea, restrâng valorile pe care poate să le conțină un tuplu, fac imposibilă repetarea tuplurilor sau stabilirea unei ordini între tuplurile unei relații.

În această secțiune, pot fi prezentate constrângerile pe care administratorul le definește asupra datelor din baza de date pentru a reda și a fi un model cât mai adecvat al realității pe care o reflectă. Sunt restricții proprii ale bazei de date și ale datelor care se pretinde a fi prezentate. De exemplu, toate valorile atributului *Nota\_Med* trebuie să fie mai mari decât 0, dar nu poate depăși 10. Sau nicio persoană de vîrstă de 25 de ani nu poate avea o vechime în muncă de 37 de ani.

Sunt restricții care, în general, limitează structura datelor păstrate în baza de date, furnizând, prin urmare, informații despre caracteristicile pe care trebuie să le respecte datele păstrate.

Acest tip de constrângerii pot fi divizate în cinci grupuri principale:

- Constrângerile de comportament al domeniului.
- Constrângerile de comportament al tuplului.
- Constrângerile de comportament al relației.
- Constrângerile de comportament al bazei de date.

### 1.7.2.1. Constrângerile de comportament al domeniului

Constrângerile de domeniu sunt condiții impuse valorilor atributelor, pentru ca acestea să corespundă semnificației pe care o au în realitatea modelată. În reprezentarea unei entități printr-o relație, valorile sunt reprezentate pe atribute. Din această cauză, aceste constrângerile se mai numesc și constrângerile de atribut.

De obicei, domeniile de care dispune oricare bază de date sunt domenii generale și care cuprind valori diverse care, în majoritatea cazurilor, pot satisface necesitățile utilizatorilor. Totuși, sunt situații în care apare necesitatea restrângerii mulțimii de valori.

De exemplu, dacă schema relației *vânzări* este constituită, inclusiv, și de atributul *Reducere*, care semnifică procentajul de reducere aplicat mărfurilor, atunci, e clar că acest atribut nu poate avea o valoare mai mare de 100%. Dat fiind faptul că SGBD-ul impune doar definirea tipului atributului, în cazul dat acesta poate lua o valoare întreagă, atunci cum trebuie definit comportamentul valorii atributului *Reducere*.

Soluția vine, aplicând constrângerile de domeniu. O constrângere de comportament al domeniului reprezintă un predicat, în care pot participa numai constante și, desigur, atributul dat. În acest fel, constrângerea exemplului anterior poate fi reprezentată de inegalitatea:

$$\text{Reducere} < 100.$$

Desigur, pentru a limita domeniile de valori, pot fi utilizate predicate mai complexe. Dacă, de exemplu, baza de date conține relația *clienți* și se dorește cunoașterea sexului pentru a putea trimite scrisori personalizate, se aplică un atribut *Sex* de tipul *CHAR* de un caracter lungime. Este clar că singurele valori acceptate vor fi *M(asculin)* și *F(eminin)*. Domeniul atributului *Sex* va putea fi restrâns de următorul predicat:

$$\text{Sex} = "M" \vee \text{Sex} = "F".$$

Trebuie menționat că, în cazul proiectării schemei bazei de date, constrângerile de comportament al domeniului pot fi definite la nivel de atribut.

### 1.7.2.2. Constrângeri de comportament al tuplului

În cazul precedent, valorile ce le putea lua un atribut, depindeau exclusiv de propriul lor comportament, adică erau independente de restul atributelor pe care era definit tuplul. În multe cazuri, valorile anumitor atribute ce constituie schema tuplului trebuie să posede valori consistente, dar care nu sunt independente. De exemplu, dacă există o relație *funcționari* în care, printre altele, se cere păstrarea datelor despre vechimea în muncă și vârsta, între valorile acestor atribute există o dependență specifică. Valoarea atributului *Vârstă* nu poate fi mai mică decât valoarea atributului *Vechime\_Muncă+14*. Munca este permisă de la 14 ani.

Această constrângere nu poate fi considerată o constrângere de domeniu, deoarece restricția are un caracter dublu, determinat de cele două atribute implicate în ea. Legătura dintre valorile admisibile, produsă de o constrângere mutuală între două sau mai multe atribute ale uneia și aceleiași scheme, pe care e definit tuplul, indică faptul că are loc o constrângere de tuplu.

Predicatul care definește această constrângere are forma următoare:

$$\text{Vârstă} \geq \text{Vechime\_Muncă} + 14.$$

Constrângerile de comportament al tuplului se definesc la nivel de relație.

### 1.7.2.3. Constrângeri de comportament al relației

Sunt situații, când nu există valori ale unor atribute dependente de valorile altor atribute din același tuplu. Cu toate acestea, relația nu este consistentă. Unele domenii de interes necesită ca valorile atributelor din mai multe tupluri ale relației să se găsească într-o corelație specifică. Aceste constrângeri se numesc constrângeri de comportament al relației. Ele, de regulă, se definesc la nivel de relație.

În figura 1.7, sunt prezentate date referitoare la desfășurarea activității de practică în producție a studentilor. Durata practicii este nu mai puțin de 50 de ore și fiecare student trebuie să activeze în mai multe unități economice, prin rotație. Bineînțeles, pentru nici un student nu poate coincide data și ora practicii în două unități distincte. În relația *practica*, este reprezentată o situație ce nu poate avea loc. Tuplurile marcate cu săgeată intră în

contradicție, deoarece acestea conțin date, precum că de la 12:00 până la 18:00 studentul *Ion Popescu* își realizează practica în două locuri distincte: în hotelul *Chișinău* și în hotelul *Național*, fapt care, realmente, este imposibil.

<i>practica</i>	<i>Student</i>	<i>Data</i>	<i>Început</i>	<i>Sfârșit</i>	<i>Hotel</i>
→	<i>Ion Popescu</i>	12/05/13	12:00	18:00	<i>Chișinău</i>
	<i>Ana Cotruță</i>	12/05/13	8:00	18:00	<i>Codru</i>
→	<i>Ion Popescu</i>	12/05/13	8:00	18:00	<i>Național</i>
	<i>Ana Cotruță</i>	13/05/13	8:00	18:00	<i>Național</i>
	<i>Ana Cotruță</i>	14/05/13	8:00	18:00	<i>Chișinău</i>
	<i>Ion Popescu</i>	13/05/13	8:00	18:00	<i>Codru</i>
	<i>Ion Popescu</i>	14/05/13	8:00	18:00	<i>Național</i>
	<i>Ion Popescu</i>	17/05/13	8:00	18:00	<i>Național</i>

Figura 1.7. Un exemplu de nerespectare a constrângerii de relație

Pentru evitarea unor astfel de situații, se introduc constrângeri de relație. O constrângere de relație este un predicat care înglobează toate sau o parte din tuplurile aceleiași relații. Valoarea acestui predicat nu poate fi obținută, dacă nu sunt examineate valorile atributelor din toate tuplurile afectate.

Modul de exprimare a acestui tip de constrângeri implică utilizarea expresiilor și interogărilor care, deocamdată, nu sunt examineate în această secțiune. E de ajuns să spus că este necesară utilizarea întregului arsenal de tipuri de interogări care pot fi formulate în limbajul SQL.

#### 1.7.2.4. Constrângeri de comportament al bazei de date

Precum a fost menționat anterior, constrângările bazei de date sunt cele de relație, de tupluri și de domenii. Dacă o constrângere de domeniu poate antrena numai valorile unui atribut, iar cea de tuplu poate atrage valorile mai multor atribute (numai din același tuplu), iar cele de relație - valori din mai multe tupluri, atunci constrângările de comportament al bazei implică valori din mai multe relații (evidențial, ale aceleiași baze de date). Acest tip de constrângeri sunt cunoscute și sub denumirea de *aserțiuni*.

Fie o agenție de turism care oferă pachete de voiaj. În fiecare pachet de voiaj, numărul de locuri este limitat. Astfel, pot fi păstrate datele despre pachetele disponibile într-o relație precum cea din figura 1.8.

pachete	Tip	Destinatie	Data	Durata	Locuri
	Alpinism	Katmandu	12/07/13	25	3
	Aventuri	Nairobi	07/05/13	12	5
	Disney	Paris	14/10/13	5	15

Figura 1.8. Relația pachete din baza de date a unei agenții de turism

Pe măsura vânzării pachetelor, pot fi cunoscute datele despre clienții care au cumpărat pachete de voiaj și numărul de pachete cumpărate. Fie că, peste un timp, relația vânzări poate fi ca în figura 1.9.

vânzări	Tip	Client
	Alpinism	Ion Popescu
	Alpinism	Elena Popescu
	Aventuri	Marin Amihălăchioae
	Alpinism	Mihai Aramă
	Alpinism	Ovidiu Cătană

Figura 1.9. Relația vânzări din baza de date a unei agenții de voiaj

La o examinare atentă a relației vânzări, se poate vedea că s-au vândut patru pachete *Alpinism*, pe când există numai trei locuri disponibile. Astfel, a fost încălcată o constrângere de bază de date. Pentru a asigura integritatea bazei de date, este necesară inspectarea datelor din mai multe relații. Constrângerile de comportament pot fi verificate în mod automat, folosind predicalele respective din definițiile aserțiunilor. SGBD-urile care nu dispun de instrumentul de definire a aserțiunilor, cu scopul dc a testa constrângerile de comportament al bazei de date, pot apela la mecanismul de creare a declanșatoarelor.

## 2. Algebra relațională

---

Algebra relațională a fost propusă de E. Codd ca o colecție de operații formale care se aplică pe relații și produc relații în calitate de rezultat [Codd70]. Se poate considera că algebra relațională este, pentru relații, ceea ce este aritmetică pentru numere întregi. Această algebră, care constituie o mulțime de operații elementare asociate modelului relațional, este fără îndoială una din forțele esențiale ale modelului. Codd, inițial, a introdus opt operații, dintre care unele pot fi compuse pornind de la altele.

Aștept, algebra relațională este un limbaj teoretic, cu operații care acționează asupra uneia sau mai multor relații, pentru a defini o altă relație, fără modificarea celor inițiale. Prin urmare, atât operanții, cât și rezultatele sunt relații, așa că ieșirea unei operații poate deveni intrare pentru o alta. Această proprietate permite compozitia operațiilor. De fapt, pot fi construite expresii algebrice de complexitate arbitrară, care ar permite exprimarea operațiilor asupra unui număr mare de relații, precum și imbricarea expresiilor, la fel ca la operațiile matematice. Algebra relațională este un limbaj de tip câte-o-relație-o-dată, în care toate tuplurile sunt manipulate într-o singură instrucție, fără ciclare.

O cerere la baza de date poate fi tratată ca o expresie algebrică care se aplică asupra unei mulțimi de relații (asupra bazei de date) și produce o relație finală (rezultatul cererii). Algebra relațională, deseori, este concepută ca un limbaj de programare foarte simplu care permite a formula cererii asupra unei baze de date relaționale. Orice cerere în algebra relațională descrie pas cu pas procedura de calcul al răspunsului, bazându-se pe ordinea în care apar operatorii în cerere. Natura procedurală a algebrei permite folosirea expresiilor algebrice ca un plan de evaluare a cererii, fapt utilizat în sistemele relaționale.

În această secțiune, vor fi introduce șase operații, din care pot fi deduse celelalte și care sunt denumite aici operații de bază. Apoi, vor fi introduse unele operații suplimentare, care sunt deseori utilizate. Unii autori au sugerat că și alte operații pot fi deduse, în continuare, din operațiile de bază [Delobel83, Maier83].

Operațiile algebrei relaționale sunt clasificate în două categorii: operațiile tradiționale pe mulțimi și operațiile relaționale native. De asemenea, sunt prezentate operațiile care pot fi deduse unele din altele, principalele proprietăți ale operațiilor și exemple de expresii ale algebrei relaționale.

## 2.1. Operațiile tradiționale pe mulțimi

Trei dintre operațiile pe mulțimi – uniunea, intersecția și diferența – pot opera numai cu două relații cu scheme compatibile [Childs68].

### 2.1.1. Scheme relaționale compatibile

**Definiția 2.1.** Se spune că două relații  $r(R)$  și  $s(S)$  sunt compatibile (sau au scheme compatibile), dacă, între  $R$  și  $S$ , există o corespondență biunivocă  $f$ : pentru orice atribut  $A$  din  $R$ , există în  $S$  un atribut  $B$  încât  $\text{dom}(A) = \text{dom}(B)$ ,  $B = f(A)$  și  $A = f^{-1}(B)$ , unde  $f^{-1}$  este funcția inversă funcției  $f$ .

**Remarcă.** Două relații definite pe aceeași schemă sunt compatibile.

<i>vânzări</i>	<i>Firmă</i>	<i>Articol</i>	<i>articole</i>	<i>Articol</i>	<i>Culoare</i>
	$f_1$	$a_1$		$a_1$	$c_1$
	$f_1$	$a_2$		$a_1$	$c_2$
	$f_2$	$a_1$		$a_1$	$c_3$
	$f_3$	$a_1$		$a_3$	$c_2$
				$a_2$	$c_1$

<i>furnizori</i>	<i>Articol</i>	<i>Furnizor</i>
	$a_1$	$f_1$
	$a_1$	$f_3$
	$a_2$	$f_1$
	$a_3$	$f_3$

Figura 2.1. Relații cu scheme compatibile și incompatibile

**Exemplul 2.1.** Fie baza de date din figura 2.1, constituită din trei relații: *vânzări*(*Firmă*, *Articol*), *articole*(*Articol*, *Culoare*), *furnizori*(*Articol*, *Furnizor*). Schemele relațiilor *vânzări* și *articole* nu sunt compatibile, în timp ce schemele relațiilor *vânzări* și

furnizori sunt compatibile. Ultimele relații sunt definite pe atrbute ce primesc valori din aceeași domeniu. Valorile active sunt, totuși, diferite, fiindcă un furnizor poate să nu fie o firmă și viceversa.

### 2.1.2. Uniunea

Uniunea este o operație clasică a teoriei mulțimilor adaptată la relații. Este o operație binară, deseori, utilizată.

**Definiția 2.2.** Uniunea a două relații  $r(R)$  și  $s(S)$  se poate nota  $r \cup s$  și reprezintă mulțimea tuplurilor care se găsesc în cel puțin una din relațiile  $r(R)$  sau  $s(S)$ :

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

Această operație se poate aplica numai în cazul în care  $r(R)$  și  $s(S)$  au scheme compatibile. Dacă atrbutele au nume se cere ca, în plus, cele două liste de nume să coincidă și rezultatul va avea aceeași listă de nume pentru atrbute.

**Exemplul 2.2.** Fie relațiile  $r(A, B, C)$  și  $s(A, B, C)$  din figura 2.2. Relația  $q = r \cup s$  este prezentată în figura 2.3.

$r$	$A$	$B$	$C$	$s$	$A$	$B$	$C$
	$a_1$	$b_1$	$c_1$		$a_1$	$b_1$	$c_1$
	$a_1$	$b_2$	$c_3$		$a_1$	$b_1$	$c_2$
	$a_2$	$b_1$	$c_2$		$a_1$	$b_2$	$c_3$
					$a_3$	$b_2$	$c_3$

Figura 2.2. Relațiile  $r(A, B, C)$  și  $s(A, B, C)$

$q$	$A$	$B$	$C$
	$a_1$	$b_1$	$c_1$
	$a_1$	$b_1$	$c_2$
	$a_1$	$b_2$	$c_3$
	$a_2$	$b_1$	$c_2$
	$a_3$	$b_2$	$c_3$

Figura 2.3. Relația  $q = r \cup s$

Operația uniunea are două proprietăți. Uniunea se bucură de proprietatea comutativă, adică

$$r \cup s = s \cup r.$$

Pentru relațiile mutual compatibile  $r$ ,  $s$  și  $q$ , ea este și asociativă, adică

$$(r \cup s) \cup q = r \cup (s \cup q).$$

Astfel, în expresiile ce conțin o cascadă de operații, uniunea, parantezele pot fi omise fără a provoca ambiguități. Dacă sunt  $k$  relații compatibile  $r_1, r_2, \dots, r_k$ , uniunea acestor relații poate fi notată prin

$$\cup(r_1, r_2, \dots, r_k).$$

Operația uniunea are două cazuri speciale. Pentru orice relații  $r(R)$  și  $s(S)$  cu scheme compatibile au loc:

$$r \cup \emptyset = r;$$

$$r \cup s = s, \text{ dacă } r \subseteq s.$$

### 2.1.3. Diferență

Ca și uniunea, operația diferență se aplică asupra două relații cu scheme compatibile.

**Definiția 2.3.** *Diferența* a două relații compatibile  $r(R)$  și  $s(S)$ , notată cu  $r \setminus s$ , este o relație definită pe multimea de attribute  $R$  sau  $S$  și are, în calitate de tupluri, toate tuplurile din relația  $r$ , care nu sunt în  $s$ , adică

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

**Exemplul 2.3.** Fie relațiile  $r(A, B, C)$  și  $s(A, B, C)$  din figura 2.2. Relațiile  $q_1 = r \setminus s$  și  $q_2 = s \setminus r$  sunt prezentate în figura 2.4.

$q_1$	A	B	C
	$a_2$	$b_1$	$c_2$
$q_2$	A	B	C
	$a_1$	$b_1$	$c_2$
	$a_3$	$b_2$	$c_3$

Figura 2.4. Relațiile  $q_1 = r \setminus s$  și  $q_2 = s \setminus r$

Din exemplul de mai sus, se observă că diferența nu se bucură de proprietatea comutativă:

$$r \setminus s \neq s \setminus r.$$

Totodată, operația diferență nu este nici asociativă. Astfel,

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

deoarece

$$(r \setminus s) \setminus q = r \setminus (s \cup q)$$

pentru orice relații  $r$ ,  $s$  și  $q$  mutual compatibile.

Acet lucru se explică prin faptul că tuplurile relației-diferență sunt extrase din relația descăzut care nu se regăsește în relația scăzător. În plus, nu există restricții privind cardinalitatea (numărul de tupluri) celor două relații, adică nu este neapărat ca relația descăzut să conțină mai multe tupluri decât relația scăzător.

Pentru operația diferență pot fi menționate patru cazuri speciale. Un caz este  $\emptyset \setminus r = \emptyset$ ; altul e  $r \setminus \emptyset = r$  pentru orice relație  $r(R)$ . Celelalte cazuri vor fi examineate în următoarele secțiuni.

#### 2.1.4. Intersecția

Intersecția, ca și uniunea și diferența, poate fi aplicată numai asupra relațiilor care au tupluri similare. Pentru a face intersecția a două relații este, prin urmare, necesar ca relațiile să fie compatibile.

**Definiția 2.4.** Fiind date două relații  $r(R)$  și  $s(S)$ , intersecția lor, notată  $r \cap s$ , este o relație definită pe schema  $R$  sau  $S$  și care conține tuplurile care sunt și în  $r$  și  $s$ , adică

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

**Exemplul 2.4.** Fie relațiile  $r(A, B, C)$  și  $s(A, B, C)$  din figura 2.2. Relația  $q = r \cap s$  este prezentată în figura 2.5.

$q$	$A$	$B$	$C$
	$a_1$	$b_1$	$c_1$
	$a_1$	$b_2$	$c_3$

Figura 2.5. Relația  $q = r \cap s$

Operația intersecția este comutativă, adică

$$r \cap s = s \cap r.$$

Dar se bucură și de proprietatea asociativă. Pentru relațiile  $r$ ,  $s$  și  $q$  cu scheme compatibile se poate scrie:

$$(r \cap s) \cap q = r \cap (s \cap q).$$

Pentru orice relații  $r$ ,  $s$  și  $q$  cu scheme compatibile au loc egalitățile

$$r \cap (s \cup q) = (r \cap s) \cup (r \cap q),$$

care arată distributivitatea la stânga a intersecției față de uniune, și

$$(r \cup s) \cap q = (r \cap q) \cup (s \cap q),$$

care arată distributivitatea la dreapta a intersecției față de uniune.

În mod similar, pot fi prezentate distributivitatea la stânga și la dreapta a uniunii față de operația intersecție, respectiv:

$$r \cup (s \cap q) = (r \cup s) \cap (r \cup q),$$

$$(r \cap s) \cap q = (r \cup q) \cap (s \cup q).$$

Pot fi menționate și două cazuri speciale. Pentru orice relații  $r(R)$  și  $s(S)$  cu scheme compatibile au loc:

$$r \cap \emptyset = \emptyset;$$

$$r \cap s = r, \text{ dacă } r \subseteq s.$$

Intersecția este o operație redundantă cu operațiile de bază, deoarece este posibilă obținerea ei de la diferență cu ajutorul uneia dintre următoarele formule:

$$r \cap s = r \setminus (r \setminus s);$$

$$r \cap s = s \setminus (s \setminus r).$$

### 2.1.5. Produsul cartezian

**Definiția 2.5.** *Produsul cartezian* al două relații  $r(R)$  și  $s(S)$  (denumit de Codd și joncțiune încrucișată, CROSS JOIN) este mulțimea tuturor tuplurilor obținute prin concatenarea fiecărui tuplu din relația  $r(R)$  cu toate tuplurile relației  $s(S)$ . Formal, produsul cartezian al relațiilor  $r$  și  $s$ , este notat cu  $r \times s$  și definit astfel:

$$r \times s = \{(t_r, t_s) \mid t_r \in r(R) \& t_s \in s(S)\}.$$

Rezultă de aici următoarele:

- Numărul de attribute ale lui  $r \times s$  este egal cu suma numerelor de attribute ale schemelor  $R$  și  $S$ .
- Numărul de tupluri ale lui este  $r \times s$  egal cu produsul numerelor de tupluri ale lui  $r$  și  $s$ .
- Dacă în  $R$  și  $S$  sunt attribute cu același nume, în schema relației produsul cartezian  $r \times s$  vor fi attribute cu același nume. Pentru a le deosebi, se prefixează numele atributului cu cel al relației din care provine, ca în exemplul următor.

**Exemplul 2.5.** Fie relațiile  $r(A, B, C)$  și  $s(C, D)$ . Atunci produsul cartezian  $q = r \times s$  arată ca în figura 2.6.

<i>r</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>s</i>	<i>C</i>	<i>D</i>	<i>q</i>	<i>A</i>	<i>B</i>	<i>r.C</i>	<i>s.C</i>	<i>D</i>
<i>a</i> <sub>1</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>1</sub>		<i>c</i> <sub>1</sub>	<i>d</i> <sub>1</sub>		<i>a</i> <sub>1</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>1</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>1</sub>	
<i>a</i> <sub>1</sub>	<i>b</i> <sub>2</sub>	<i>c</i> <sub>3</sub>		<i>c</i> <sub>1</sub>	<i>d</i> <sub>2</sub>		<i>a</i> <sub>1</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>1</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>2</sub>	
<i>a</i> <sub>2</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>2</sub>					<i>a</i> <sub>1</sub>	<i>b</i> <sub>2</sub>	<i>c</i> <sub>3</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>1</sub>	
							<i>a</i> <sub>1</sub>	<i>b</i> <sub>2</sub>	<i>c</i> <sub>3</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>2</sub>	
							<i>a</i> <sub>2</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>2</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>1</sub>	
							<i>a</i> <sub>2</sub>	<i>b</i> <sub>1</sub>	<i>c</i> <sub>2</sub>	<i>c</i> <sub>1</sub>	<i>d</i> <sub>2</sub>	

Figura 2.6. Relațiile  $r(A, B, C)$ ,  $s(C, D)$  și  $q = r \times s$ 

Produsul cartezian este comutativ, adică

$$r \times s = s \times r,$$

precum și asociativ,

$$(r \times s) \times q = r \times (s \times q) = r \times s \times q.$$

Astfel, rezultatul produsului cartezian al mai multor relații nu depinde de modul de grupare a factorilor.

Numărul de tupluri în rezultat este exact  $\text{card}(r) \cdot \text{card}(s)$ , unde  $\text{card}(r)$  reprezintă cardinalitatea relației  $r$  (numărul de tupluri în  $r$ ). Din cauza că fiecare tuplu din prima relație se asociază cu fiecare tuplu din a doua relație, această operație este consumatoare de timp și trebuie utilizată numai în cazurile strict necesare.

Condițiile prefixării cu numele relației a atributelor sau  $R \cap S = \emptyset$  sunt necesare în definiția produsului cartezian. În caz contrar, rezultatul nu va fi o relație, deoarece, într-o relație, nu sunt attribute duplicate. Această problemă poate fi soluționată cu ajutorul operației redenumirea, care, de fapt, produce schimbarea numelor de attribute.

### 2.1.6. Operația de redenumire

Redenumirea atributelor poate fi utilizată la rezolvarea conflictelor de nume și pentru definirea unor operații prevăzute în algebra relatională. Redenumirea reprezintă un mecanism utilizat la schimbarea numelor relațiilor sau atributelor. Limitările impuse operatorilor standard din teoria mulțimilor pot fi restrictive în anumite situații.

Scopul operației de redenumire constă în adaptarea numelor atributelor atunci când este necesară aplicarea unui operator din teoria mulțimilor. Această operație modifică numele atributelor, lăsând intact conținutul relației.

De exemplu, uniunea relațiilor  $r(A, B)$  și  $s(A, C)$  cu rezultatul având attributele  $A$  și  $B$  se poate face, apelând la redenumirea atributului  $C$  cu  $B$ .

**Definiția 2.6.** Fie  $r(R)$  o relație,  $A \in R$  și atributul  $B \notin R \setminus \{A\}$ .

Atunci relația  $r$  cu atributul  $A$  redenumit în  $B$  este următoarea relație definită pe schema  $R \setminus \{A\} \cup \{B\}$ :

$$\delta_{A \leftarrow B}(r) = \{t_1 \mid \exists t \in r \text{ și } t_1[R \setminus \{A\}] = t[R \setminus \{A\}] \text{ și } t_1[B] = t[A]\}.$$

**Exemplul 2.6.** Fie relațiile  $r(A, B, C)$  și  $s(C, D)$  din figura 2.6.

Atunci, expresia  $r \times \delta_{C \leftarrow D, D \leftarrow E}(s)$  va reprezenta relația din figura 2.7.

$q$	$A$	$B$	$C$	$D$	$E$
$a_1$	$b_1$	$c_1$	$c_1$	$d_1$	
$a_1$	$b_1$	$c_1$	$c_1$	$d_2$	
$a_1$	$b_2$	$c_3$	$c_1$	$d_1$	
$a_1$	$b_2$	$c_3$	$c_1$	$d_2$	
$a_2$	$b_1$	$c_2$	$c_1$	$d_1$	
$a_2$	$b_1$	$c_2$	$c_1$	$d_2$	

Figura 2.7. Relația  $q = r \times \delta_{C \leftarrow D, D \leftarrow E}(s)$

### 2.1.7. Complementul

Complementul permite obținerea tuplurilor care nu aparțin unei relații. Se presupune a priori că domeniile sunt finite (în caz contrar, se obțin relații infinite). Astfel, complementul unei relații reprezintă mulțimea tuplurilor din produsul cartezian al domeniilor asociate atributelor relației, care nu figurează în extensia relației considerate.

**Definiția 2.7.** Fie relația  $r(R)$ , unde  $R = \{A_1, \dots, A_n\}$ . Complementul relației  $r$ , notat cu  $\bar{r}$ , este

$$\bar{r} = \text{tup}(R) \setminus r,$$

unde  $\text{tup}(R) = \text{dom}(A_1) \times \dots \times \text{dom}(A_n)$ , adică constă în construirea unei relații rezultat de tupluri, care nu aparțin relației  $r$ .

**Exemplul 2.7.** Fie relația  $r(A, B, C)$ , din figura 2.2 și fie  $\text{dom}(A) = \{a_1, a_2, a_3\}$ ,  $\text{dom}(B) = \{b_1, b_2\}$ ,  $\text{dom}(C) = \{c_1, c_2, c_3\}$ . Atunci  $\text{tup}(A, B, C)$  și  $\bar{r}$  sunt cele reprezentate în figura 2.8.

$\text{tup}$	$A$	$B$	$C$	$\bar{r}$	$A$	$B$	$C$
	$a_1$	$b_1$	$c_1$		$a_1$	$b_1$	$c_2$
	$a_1$	$b_1$	$c_2$		$a_1$	$b_1$	$c_3$
	$a_1$	$b_1$	$c_3$		$a_1$	$b_2$	$c_1$
	$a_1$	$b_2$	$c_1$		$a_1$	$b_2$	$c_2$
	$a_1$	$b_2$	$c_2$		$a_2$	$b_1$	$c_1$
	$a_1$	$b_2$	$c_3$		$a_2$	$b_1$	$c_3$
	$a_2$	$b_1$	$c_1$		$a_2$	$b_2$	$c_1$
	$a_2$	$b_1$	$c_2$		$a_2$	$b_2$	$c_2$
	$a_2$	$b_1$	$c_3$		$a_2$	$b_2$	$c_3$
	$a_2$	$b_2$	$c_1$		$a_3$	$b_1$	$c_1$
	$a_2$	$b_2$	$c_2$		$a_3$	$b_1$	$c_2$
	$a_2$	$b_2$	$c_3$		$a_3$	$b_1$	$c_3$
	$a_3$	$b_1$	$c_1$		$a_3$	$b_2$	$c_1$
	$a_3$	$b_1$	$c_2$		$a_3$	$b_2$	$c_2$
	$a_3$	$b_1$	$c_3$		$a_3$	$b_2$	$c_3$
	$a_3$	$b_2$	$c_1$				
	$a_3$	$b_2$	$c_2$				
	$a_3$	$b_2$	$c_3$				

Figura 2.8. Relațiile  $\text{tup}(A, B, C)$  și  $\bar{r}(A, B, C)$

Este evident că, dacă pentru vreun atribut  $A$  din  $R$  domeniul  $\text{dom}(A)$  este infinit, atunci  $\bar{r}(R)$  va conține o mulțime infinită de tupluri și, deci, nu va fi o relație conform definiției adoptate.

Este, practic, dificil să se construiască domeniul unui atribut, deseori se utilizează domenii active. De aceea, în sistemele relaționale, se introduce noțiunea de complement activ. Versiunea modificată a complementului unei relații, complementul activ, întotdeauna va fi o relație.

**Definiția 2.8.** Fie relația  $r(R)$ , unde  $R = \{A_1, \dots, A_n\}$ . Atunci

$$\text{adom}(A, r) = \{a \mid a \in \text{dom}(A) \text{ & } \exists t \in r \text{ & } t[A] = a\}.$$

se numește domeniu activ al atributului  $A$ . Fie

$$\text{atup}(R, r) = \text{adom}(A_1, r) \times \dots \times \text{adom}(A_n, r).$$

Atunci *complementul activ* al relației  $r(R)$ , notat cu  $\tilde{r}$ , va fi

$$\tilde{r} = \text{atup}(R, r) \setminus r.$$

Astfel, complementul activ este o operație unară, care constă în construirea unei relații ce conține toate tuplurile inexistente într-o relație, pornind de la valorile tuplurilor care există.

$q$	$A$	$B$	$C$	$\sim r$	$A$	$B$	$C$
	$a_1$	$b_1$	$c_1$		$a_1$	$b_1$	$c_2$
	$a_1$	$b_1$	$c_2$		$a_1$	$b_1$	$c_3$
	$a_1$	$b_1$	$c_3$		$a_1$	$b_2$	$c_1$
	$a_1$	$b_2$	$c_1$		$a_1$	$b_2$	$c_2$
	$a_1$	$b_2$	$c_2$		$a_2$	$b_1$	$c_1$
	$a_1$	$b_2$	$c_3$		$a_2$	$b_1$	$c_3$
	$a_2$	$b_1$	$c_1$		$a_2$	$b_2$	$c_1$
	$a_2$	$b_1$	$c_2$		$a_2$	$b_2$	$c_2$
	$a_2$	$b_1$	$c_3$		$a_2$	$b_2$	$c_3$
	$a_2$	$b_2$	$c_1$				
	$a_2$	$b_2$	$c_2$				
	$a_2$	$b_2$	$c_3$				

Figura 2.9. Relațiile  $q = \text{atup}(\{A, B, C\}, r)$  și  $\tilde{r}(A, B, C)$

**Exemplul 2.8.** Fie relația  $r(A, B, C)$  din figura 2.2. Se poate observa că  $\text{adom}(A, r) = \{a_1, a_2\}$ ,  $\text{adom}(B, r) = \{b_1, b_2\}$ ,

$adom(C, r) = \{c_1, c_2, c_3\}$ . Atunci, relațiile  $q = atup(\{A, B, C\}, r)$  și  $\widetilde{r}$  sunt prezentate în figura 2.9.

## 2.2. Operațiile relationale native

Operațiile relationale native proiecția, selecția, joncțiunea (joncțiunea naturală, theta joncțiunea și joncțiunea externă) și diviziunea iau în considerație compoziția tuplurilor, formate din valori ale atributelor relațiilor.

### 2.2.1. Proiecția

Operația proiecția este o operație nativă și permite suprimarea atributelor din relații. Acționează asupra unei singure relații  $r$  și definește o relație care conține o submulțime verticală a lui  $r$ , extrăgând valorile atributelor specificate.

**Definiția 2.9.** Fie dată relația  $r$  definită pe schema  $R$  și o submulțime  $X \subseteq R$ . *Proiecția* relației  $r$  asupra mulțimii de atrbute  $X$  (notată cu  $\pi_X(r)$ ) reprezintă o relație cu schema  $X$  ce constă din  $X$ -valorile tuturor tuplurilor din  $r$ :

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

Este evident că operația proiecția suprimă atrbute. În principiu, numărul de tupluri, în rezultatul obținut, este același ca și în relația inițială. Dar există o problemă: întrucât rezultatul trebuie să fie o relație, ea nu poate conține tupluri identice (în mulțime, nu există elemente dupăcat). Se poate întâmpla, că după aplicarea operației proiecția, două tupluri care, inițial, erau distințe pot deveni identice, dacă atrbutul (atrbutele) care le distingea(u) a (au) fost suprimat (suprimate). În acest caz, se păstrează numai unul din cele două (sau mai multe) tupluri identice.

**Exemplul 2.9.** În figura 2.10, sunt reprezentate relațiile  $r(A, B, C)$  și  $s = \pi_{A, C}(r)$ .

$r$	$A$	$B$	$C$	$s$	$A$	$C$
$a$	10	1		$a$	1	
$a$	20	1		$b$	1	
$b$	30	1		$b$	2	
$b$	40	2				

Figura 2.10. Relațiile  $r(A, B, C)$  și  $s = \pi_{A,C}(r)$ 

Să se observe că, din rezultat, a fost eliminat tuplul duplicat:  $\langle a, 1 \rangle$ .

Există mai multe cazuri speciale:

- În mulțimea de atrbute pentru o proiecție poate să apară toate atrbutele relației. În acest caz, se obține o relație cu același conținut cu cea inițială: dacă  $X = R$ , atunci  $\pi_X(r) = r$ .
- În cazul în care relația inițială nu conține tupluri, atunci și rezultatul proiecției este vid: dacă  $r = \emptyset$ , atunci  $\pi_X(r) = \emptyset$ .
- Dacă mulțimea de atrbute  $X = \emptyset$ , atunci proiecția  $\pi_\emptyset(r)$  este indefinită, fiindcă schema unei relații nu poate fi o mulțime vidă. Schema unei relații, produsă de operația proiecția, are cel puțin un atrbut.
- În cazul în care  $R \subset X$ , operația proiecția, de asemenea, este indefinită. Mulțimea asupra căreia se face proiecția trebuie să fie o submulțime a schemei relației inițiale.
- Fie relația  $r(R)$  și  $Y \subseteq X \subseteq R$ . Atunci are loc  $\pi_Y(\pi_X(r)) = \pi_Y(r)$ .
- Dacă relația  $r$  este definită pe o mulțime de atrbute ce include mulțimile  $X$  și  $Y$ , atunci poate fi realizată cascadarea proiecțiilor – o proiecție poate fi transformată într-o cascadă de proiecții care elimină atrbute în diverse faze:  $\pi_X(r) = \pi_X(\pi_{XY}(r))$ .
- Distributivitatea proiecției în raport cu uniunea:  $\pi_X(r \cup s) = \pi_X(r) \cup \pi_X(s)$ .

### 2.2.2. Selectia

Operația selecția are, în calitate de operand, o singură relație. Ea extrage din relația dată tuplurile care satisfac un criteriu de selecție. În consecință, se obține o relație care este constituită dintr-o submulțime de tupluri ale relației inițiale.

Pentru selectarea tuplurilor dintr-o relație, se specifică criteriul (sau condițiile) de selectare. Acest criteriu, fie că e notat prin  $F$ , reprezintă o formulă bine formată care poate fi definită recursiv.

**Definiția 2.10.** O formulă  $F$  asupra unei relații  $r$ , se definește ca o expresie logică compusă din operanzi care sunt nume de atribute sau constante; operatori de comparație aritmetică  $=, \neq, <, \leq, >, \geq$ ; operatori logici  $\&, \vee$  și  $\neg$ , precum urmează:

1.  $A\theta B$  și  $A\theta a$  sunt formule bine formate (deseori, numite atomice), unde  $A$  și  $B$  sunt atribute compatibile și  $a \in \text{dom}(A)$ , iar  $\theta \in \{=, \neq, <, \leq, >, \geq\}$ .
2. Dacă  $G$  și  $H$  sunt formule bine formate, atunci conjuncția  $G \& H$ , disjuncția  $G \vee H$ , negațiile  $\neg G$  și  $\neg H$  sunt formule bine formate.
3. Nicio altă formulă nu e o formulă bine formată în calitate de criteriu.

**Definiția 2.11.** Formula bine formată  $F$  e *aplicabilă* relației  $r(R)$ , dacă orice constantă  $c$  din  $F$  este în  $\text{dom}(A)$ , unde  $A \in R$ , și orice atribut utilizat în  $F$  este în  $R$ . O relație  $r$  satisface  $F$ , dacă  $F$  e aplicabilă relației  $r$  și orice tuplu  $t \in r$  satisface formula  $F$ , în sensul că formula obținută prin substituirea oricărui atribut  $A$  din  $F$  cu  $A$ -valoarea tuplului  $t$  are valoarea adevăr.

**Definiția 2.12.** Selectia relației  $r(R)$  conform unei formule bine formate și aplicabile  $F$  este o submulțime de tupluri ale relației  $r(R)$ , notată cu  $\sigma_F(r)$ , ce constă din toate tuplurile  $t \in r$ , care satisfac  $F$ , adică

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

Cu alte cuvinte, o selecție  $\sigma_F(r)$  produce o relație care are aceleași attribute ca relația  $r$  și care conține acele tupluri din  $r$  pentru care  $F$  este adevărată.

**Exemplul 2.10.** În figura 2.11, sunt prezentate relațiile  $r(A, B, C, D)$  și  $s = \varpi_{(A=B) \& (D>5)}(r)$ .

$r$	$A$	$B$	$C$	$D$	$s$	$A$	$B$	$C$	$D$
$a$	$a$	$1$	$7$		$a$	$a$	$1$	$7$	
$b$	$b$	$5$	$5$		$b$	$b$	$23$	$10$	
$b$	$b$	$12$	$3$						
$b$	$b$	$23$	$10$						

Figura 2.11. Relațiile  $r(A, B, C, D)$  și  $s = \varpi_{(A=B) \& (D>5)}(r)$

Există cazuri speciale ale operației selecția:

- Dacă  $F$  e o formulă vidă, sau o tautologie, atunci  $\sigma_F(r) = r$ . În acest caz, asupra tuplurilor  $t \in r$  nu se impune nicio constrângere de selecție.
- Dacă  $r(R) = \emptyset$ , atunci  $\sigma_F(r) = \emptyset$  pentru orice formulă  $F$ , deoarece  $F$  e satisfăcută de orice relație vidă.

Acum se vor considera unele proprietăți ale expresiilor ce includ operația selecția:

- Compoziția a două selecții este comutativă:  
 $\sigma_G(\sigma_F(r)) = \sigma_F(\sigma_G(r))$
- Selectiile pot fi atomizate:  $\sigma_{F \& G}(r) = \sigma_F(\sigma_G(r))$ .
- Operația selecția este distributivă în raport cu operațiile binare tradiționale pe mulțimi. Astfel, dacă  $r$  și  $s$  sunt două relații compatibile și  $\chi \in \{\cup, \cap, \setminus\}$ , atunci  $\sigma_F(r \chi s) = \sigma_F(r) \chi \sigma_F(s)$ .
- Operațiile selecția și proiecția se bucură de proprietatea comutativă, dacă sunt respectate unele condiții. Fie  $r$  o relație cu

schema  $R$  și fie  $X \subseteq R$  și  $F$  o formulă bine formată definită pe  $X$ . Atunci  $\pi_X(\sigma_F(r)) = \sigma_F(\pi_X(r))$

În plus, sunt utile următoarele proprietăți:

- $\sigma_{F \vee G}(r) = \sigma_F(r) \cup \sigma_G(r)$
- $\sigma_{F \& G}(r) = \sigma_F(r) \cap \sigma_G(r)$
- $\sigma_{F \& (\neg G)}(r) = \sigma_F(r) \setminus \sigma_G(r)$

### 2.2.3. Joncțiunea naturală

Joncțiunea este una dintre operațiile esențiale ale algebrei relaționale, probabil, cea mai dificil de realizat de SGBD-uri. Joncțiunca poate compune două relații folosind un criteriu de joncțiune. Aceasta poate fi văzută ca o extensie a produsului cartezian cu o condiție de comparare a atributelor.

Joncțiunea naturală pentru două relații  $r(R)$  și  $s(S)$  (notată  $r \triangleright \triangleleft s$ ) se obține făcând joncțiunea celor două relații după condiția “atributele cu același nume au valori egale” și eliminând prin proiecție atributele dupicate (cele după care s-a făcut joncțiunea).

**Definiția 2.13.** Fie  $r(R)$  și  $s(S)$  două relații. *Joncțiunea naturală*  $r \triangleright \triangleleft s$  este o relație definită pe schema  $RS$  (adică,  $R \cup S$ ) astfel, încât:

$$r \triangleright \triangleleft s = \{t \mid t[R] \in r \& t[S] \in s\}.$$

Definiția confirmă faptul că tuplurile relației rezultat provin din combinarea tuplurilor din  $r$  cu tuplurile din  $s$  care au valori egale pentru atributele comune, adică  $(R \cap S)$ -valori egale. În practică, de obicei, atributele comune,  $R \cap S$ , constituie, într-o relație, cheia primară, iar în altă relație - cheia externă.

Joncțiunea naturală se calculează de SGBD în trei etape:

1. Mai întâi, este determinat produsul cartezian  $r \times s$ .

2. Apoi, asupra produsului cartezian obținut, este efectuată o operație de selecție, prin extragerea tuplurilor care conțin aceleași valori ale atributelor comune din schemele relațiilor incidente  $r$  și  $s$ ;
3. În final, sunt eliminate atrbutele redundante rezultate.

Pot fi observate următoarele cazuri speciale:

- Dacă schemele  $R$  și  $S$  sunt disjuncte,  $R \cap S = \emptyset$ , atunci juncțiunea relațiilor  $r$  și  $s$  este identică cu produsul cartezian al lor, adică  $r \triangleright \triangleleft s = r \times s$ .
  - Dacă  $R \cap S \neq \emptyset$  și  $|R \cap S| = k$ , atunci juncțiunea poate fi redată prin operațiile proiecția, selecția și produsul cartezian:  $r \triangleright \triangleleft s = \pi_{RS}(\sigma_F(r \times s))$ , unde
- $$F = (r.A_1 = s.A_1) \& \dots \& (r.A_k = s.A_k) \quad \text{pentru } A_i \in R \cap S, 1 \leq i \leq k.$$
- Dacă  $R = S$ , atunci  $r \triangleright \triangleleft s = r \cap s$ .

**Exemplul 2.11.** În figura 2.12, sunt reprezentate relațiile  $r(A, B, C)$ ,  $s(B, C, D)$  și  $q(A, B, C, D)$ , unde  $q = r \triangleright \triangleleft s$ .

$r$	$A$	$B$	$C$	$s$	$B$	$C$	$D$	$q$	$A$	$B$	$C$	$D$
$a_1$	$b_1$	$c_1$		$b_1$	$c_1$	$d_1$		$a_1$	$b_1$	$c_1$	$d_1$	
$a_1$	$b_2$	$c_1$		$b_1$	$c_1$	$d_2$		$a_1$	$b_1$	$c_1$	$d_2$	
$a_2$	$b_1$	$c_2$		$b_1$	$c_2$	$d_3$		$a_2$	$b_1$	$c_2$	$d_3$	
				$b_2$	$c_2$	$d_4$						

Figura 2.12. Relațiile  $r(A, B, C)$ ,  $s(B, C, D)$  și  $q = r \triangleright \triangleleft s$

Din exemplul 2.11, se vede că nu toate tuplurile participă la formarea relației rezultat. Secvența de perechi de tupluri, care participă la juncțiune, formează așa-numitele perechi de tupluri juncționabile. Astfel, tuplurile  $\langle a_1, b_1, c_1 \rangle \in r$  și  $\langle b_1, c_1, d_2 \rangle \in s$  sunt tupluri juncționabile, în timp ce tuplul  $\langle a_1, b_2, c_1 \rangle \in r$  nu e juncționabil nici cu un tuplu din relația  $s$ .

Dacă toate tuplurile din ambele relații sunt juncționabile, se spune că juncțiunea naturală este completă.

**Definiția 2.14.** Fie  $r(R)$  și  $s(S)$  două relații. *Joncțiunea naturală*  $r \triangleright \triangleleft s$  este completă, dacă

$$\forall t_r \in r, \exists t \in r \triangleright \triangleleft s, \text{ astfel încât } t[R] = t_r \text{ și}$$

$$\forall t_s \in s, \exists t_1 \in r \triangleright \triangleleft s, \text{ astfel încât } t_s[S] = t_1.$$

Una din întrebările “clasice” pentru verificarea modului în care a fost sau nu înțeleasă joncțiunea este: Câte tupluri are relația-rezultat al joncțiunii? Răspunsul este exact numai atunci când se respectă integritatea referențială. Cu alte cuvinte, atunci când atributele comune reprezintă cheile primară și cea externă în relațiile ce participă la joncțiune, respectiv.

Există posibilitatea ca, în cazul unei joncțiuni, rezultatul să nu conțină niciun tuplu. În cealaltă extremă, se află situația când fiecare tuplu al unei relații se poate combina cu fiecare tuplu din cea de-a doua relație. În acest caz, cardinalitatea relației-rezultat este dată de produsul cardinalităților celor două relații.

Rezumând cazurile expuse, se poate spune că joncțiunea naturală dintre  $r$  și  $s$  conține un număr de tupluri cuprins între 0 și  $|r| \cdot |s|$ . Mai mult decât atât:

- Dacă joncțiunea este completă, atunci  $|r \triangleright \triangleleft s| \geq \max(|r|, |s|)$ .
- Dacă  $R \cap S$  conține o cheie pentru  $s$ , atunci  $|r \triangleright \triangleleft s| \leq |r|$ ;
- Dacă  $R \cap S$  este cheie primară pentru  $s$  și există constrângerea de integritate referențială pe atributele  $R \cap S$  din  $r$  și  $s$ , atunci  $|r \triangleright \triangleleft s| = |r|$ .

Tinând cont de aceste observații asupra cardinalității rezultatului joncțiunii, pot fi prezentate următoarele legături dintre operațiile proiecția și joncțiunea naturală:

- Fie două relații  $r(R)$  și  $s(S)$ , și fie  $q = r \triangleright \triangleleft s$  (schema relației  $q$  este  $RS$ ). Dacă apoi relația  $q$  se proiectează asupra mulțimii de atribut  $R$ , adică  $r_1 = \pi_R(q)$ , atunci are loc  $r_1 \subseteq r$ .

**Exemplul 2.12.** Fie relațiile  $r(A, B, C)$ ,  $s(B, C, D)$  și  $q(A, B, C, D)$  din figura 2.12. Relația  $q$  a fost obținută în urma joncțiunii relațiilor  $r$  și  $s$ . Dacă relația  $q$  se proiectează pe schema  $\{A, B, C\}$  a relației  $r$ , adică se construiește relația  $r_1 = \pi_{ABC}(q)$ , atunci se poate observa că  $r_1 \subseteq r$  (figura 2.13).

$r_1$	$A$	$B$	$C$
	$a_1$	$b_1$	$c_1$
	$a_2$	$b_1$	$c_2$

Figura 2.13. Relația  $r_1 = \pi_{ABC}(q)$

Este evident că dintre relațiile  $r_1$  și  $r$  va fi semnul " $=$ ", dacă pentru orice tuplu  $t_r$  din  $r$  ar exista în  $s$  un tuplu jonctionabil  $t_s$ , adică, dacă tuplurile  $t_r$  și  $t_s$  ar satisface egalitatea  $t_r[R \cap S] = t_s[R \cap S]$ . Cu alte cuvinte,  $r_1 = r$ , dacă  $\pi_{R \cap S}(r) = \pi_{R \cap S}(s)$ .

Este interesantă legătura dintre proiecție și joncțiunea naturală în cazul în care se schimbă ordinea de aplicare a acestor operații:

- Fie relația  $q$  e definită pe mulțimea de attribute  $RS$  și fie se construiesc relațiile  $r = \pi_R(q)$  și  $s = \pi_S(q)$ . Apoi, se realizează joncțiunea  $q_1 = r \triangleright \triangleleft s$ . Se poate observa că  $q \subseteq q_1$ .

În cazul când  $q = q_1$ , se spune că relația  $q$  se descompune fără pierderi pe mulțimile de attribute  $R$  și  $S$ .

Să presupunem acum că procesul de descompunere a relației  $q_1$  continuă:

- Se proiectează relația  $q_1$  pe schemele  $R$  și  $S$ , respectiv,  $r_1 = \pi_R(q_1)$  și  $s_1 = \pi_S(q_1)$ , din care se construiește relația  $q_2 = r_1 \triangleright \triangleleft s_1$ . Atunci, are loc egalitatea  $q_1 = q_2$ .

Dacă procesul de descompunere și restabilire este continuat, egalitatea se va păstra.

Proprietăți ale joncțiunii naturale:

- Joncțiunea naturală este comutativă:  $r \triangleright \triangleleft s = s \triangleright \triangleleft r$ .
- Joncțiunea naturală se bucură de proprietatea asociativă:  

$$(r \triangleright \triangleleft s) \triangleright \triangleleft q = r \triangleright \triangleleft (s \triangleright \triangleleft q)$$
.
- Este valabilă distributivitatea joncțiunii în raport cu uniunea: Fie relațiile  $r(R)$ ,  $r_1(R)$  și  $s(S)$ . Atunci, are loc următoarea egalitate:  

$$(r_1 \cup r) \triangleright \triangleleft s = (r_1 \triangleright \triangleleft s) \cup (r \triangleright \triangleleft s)$$
.
- Fie relațiile  $r(R)$  și  $s(S)$ . În cazul în care criteriul de selecție  $F$  este definit pe attributele schemei  $S$ , are loc anticiparea selecției în raport cu joncțiunea:  

$$\sigma_F(r \triangleright \triangleleft s) = r \triangleright \triangleleft \sigma_F(s)$$
.
- Fie relațiile  $r(R)$  și  $s(S)$ . În cazul în care criteriul  $Y \subset S$  și  $R \cap S \subseteq Y$ , atunci, poate fi definită anticiparea proiecției în raport cu joncțiunea:  

$$\pi_{RY}(r \triangleright \triangleleft s) = r \triangleright \triangleleft \pi_Y(s)$$
.

De ce se insistă atât de mult pe importanța joncțiunii? În primul rând, pentru că permite reconstituirea relației universale inițiale. Modelul relațional se bazează pe descompunerea bazei în relații, astfel încât nivelul redundanței datelor și problemele la actualizarea tabelelor să fie reduse la minim. Cele mai multe interogări, însă, operează cu date și predicate aplicate simultan atributelor din două sau mai multe relații.

#### 2.2.4. Semijoncțiunea

În unele cazuri, în timpul executării unei joncțiuni, nu este necesar să se păstreze attributele ambelor relații în relația-rezultat. Uneori, numai attributele unuia din cei doi operanzi se cer conservate. Pentru optimizarea evaluării interogărilor la baza de date a fost introdusă o operație specifică [Berstein81] numită semijoncțiune.

Semijoncțiunea e o operație binară. Ea constă în construirea unei relații din cele două, dar e formată numai din tupluri luate dintr-o singură relație ce participă la joncțiune.

**Definiția 2.15.** Fie două relații  $r(R)$  și  $s(S)$ . Semijoncțiunea relațiilor  $r$  și  $s$ , notată cu  $r \triangleright\triangleleft s$ , este o mulțime de tupluri determinată de proiecția asupra mulțimii de atrbute  $R$  a joncțiunii naturale dintre relațiile  $r$  și  $s$ :

$$r \triangleright\triangleleft s = \pi_R(r \triangleright\triangleleft s).$$

O formulă echivalentă pentru realizarea acestei operații este  $r \triangleright\triangleleft s = r \triangleright\triangleleft \pi_{R \setminus S}(s)$ . În general, semijoncțiunea nu se bucură de proprietatea comutativă, deci  $r \triangleright\triangleleft s \neq s \triangleright\triangleleft r$ .

**Exemplul 2.13.** În figura 2.14, sunt prezentate relațiile  $r(A, B)$ ,  $s(B, C)$  și  $q(A, B)$ , unde  $q = r \triangleright\triangleleft s$ .

$r$	$A$	$B$	$s$	$B$	$C$	$q$	$A$	$B$
	$a_1$	$b_1$		$b_1$	$c_1$		$a_1$	$b_1$
	$a_1$	$b_2$					$a_2$	$b_1$
	$a_2$	$b_1$						

Figura 2.14. Relațiile  $r(A, B)$ ,  $s(B, C)$  și  $q = r \triangleright\triangleleft s$

## 2.2.5. Joncțiunea theta

În general, produsul cartezian nu prezintă interes, deoarece reprezintă o fuziune necondiționată a tuplurilor celor două relații-operanzi.

Joncțiunea theta, numită și joncțiunea generală, a două relații produce o a treia relație care conține toate combinațiile de tupluri ale celor două relații care satisfac o condiție inițială specificată. Condiția trebuie, desigur, să permită fuziunea celor două relații. Denumirea de theta-joncțiune este folosită din motive istorice, simbolul  $\theta$  (unde  $\theta \in \{=, \neq, <, \leq, >, \geq\}$ ), fiind utilizat, inițial, pentru a desemna o condiție de comparare între atrbutele celor două relații.

**Definiția 2.16.** Fie două relații  $r(R)$  și  $s(S)$ . *Joncțiunea theta* a relațiilor  $r$  și  $s$ , notată cu  $r \triangleright \triangleleft_F s$  reprezintă produsul cartezian al relațiilor  $r$  și  $s$ , urmat de o selecție după condiția  $F$  (numită și condiție de joncțiune):  $r \triangleright \triangleleft_F s = \sigma_F(r \times s)$ .

Conform definiției, joncțiunea-theta este efectuată în următoarea ordine:

1. Se calculează produsul cartezian  $r \times s$ .
2. Din relația obținută, sunt extrase, prin selecție, tuplurile care satisfac formula  $F$ .

**Exemplul 2.14.** Fie relațiile  $r(A, B, C)$  și  $s(C, D)$ . Atunci joncțiunea-theta  $q = r \triangleright \triangleleft_{r.C > s.C} s$  arată ca în figura 2.15.

$r$	$A$	$B$	$C$	$s$	$C$	$D$	$q$	$A$	$B$	$r.C$	$s.C$	$D$
$a_1$	$b_1$	4		3	$d_1$		$a_1$	$b_1$	4	1	$d_2$	
$a_1$	$b_2$	2		4	$d_1$		$a_1$	$b_1$	4	3	$d_1$	
$a_2$	$b_1$	6		1	$d_2$		$a_1$	$b_2$	2	1	$d_2$	
							$a_2$	$b_1$	6	3	$d_1$	
							$a_2$	$b_1$	6	4	$d_1$	
							$a_2$	$b_1$	6	1	$d_2$	

Figura 2.15. Relațiile  $r(A, B, C)$ ,  $s(C, D)$  și  $q = r \triangleright \triangleleft_{r.C > s.C} s$

Aici, trebuie menționată condiția necesară  $R \cap S = \emptyset$  pentru a obține în urma joncțiunii-theta o relație. Cu acest scop, poate fi utilizată operația de redenumire.

**Definiția 2.17.** O joncțiune theta în care condiția de selecție  $F$  este o conjuncție de atomi de egalitate, fiecare atom implicând câte un atribut din fiecare relație-operand se numește *echi-joncțiune*.

## 2.2.6. Joncțiuni externe

Joncțiunile definite anterior pierd tuplurile cel puțin ale unei relații, în cazul în care relațiile ce se joncționează nu posedă proiecții identice pe atributele de joncțiune. Pentru a păstra toate datele, în toate cazurile, este necesar să se definească joncțiuni care păstrează tuplurile, care nu găsesc tupluri corespunzătoare joncționabile, cu valori nule asociate. Cu acest scop,

Codd [Codd79] a introdus joncțiunile externe. Valorile nule (necunoscute) vor fi notate cu semnul  $\perp$ .

Dat fiind faptul că la joncțiune participă două relații, pot exista trei variante de joncțiune externă:

- Joncțiunea externă la stânga (left outer join), în care, ca rezultat, apar toate tuplurile relației din stânga operatorului. Notația este  $r \circ \triangleright \triangleleft s$ .
- Joncțiunea externă la dreapta (right outer join), în care, ca rezultat, apar toate tuplurile relației din dreapta operatorului. Notația este  $r \triangleright \triangleleft \circ s$ .
- Joncțiunea externă completă (full outer join), în care, ca rezultat, apar toate tuplurile relației din stânga și din dreapta operatorului. Notația este  $r \triangleright \circ \triangleleft s$ .

Dacă precedentele tipuri de joncțiune sunt comutative, în cazul joncțiunii externe, trebuie specificată relația din care se extrag tuplurile fără corespondent în cealaltă relație.

**Exemplul 2.15.** Fie relațiile  $r(A, B)$ ,  $s(B, C)$  din figura 2.16. Cele trei tipuri de joncțiune externă apar în figura 2.17:  $q_L = r \circ \triangleright \triangleleft s$ ,  $q_R = r \triangleright \triangleleft \circ s$  și  $q_F = r \triangleright \circ \triangleleft s$ , respectiv.

$r$	$A$	$B$	$s$	$B$	$C$
	$a_1$	$b_1$		$b_1$	$c_1$
	$a_1$	$b_2$		$b_4$	$c_2$
	$a_2$	$b_1$		$b_2$	$c_3$
	$a_2$	$b_3$			

Figura 2.16. Relațiile  $r(A, B)$  și  $s(B, C)$

$q_L$	$A$	$r.B$	$s.B$	$C$	$q_R$	$A$	$r.B$	$s.B$	$C$
	$a_1$	$b_1$	$b_1$	$c_1$		$a_1$	$b_1$	$b_1$	$c_1$
	$a_1$	$b_2$	$b_2$	$c_3$		$a_1$	$b_2$	$b_2$	$c_3$
	$a_2$	$b_1$	$b_1$	$c_1$		$a_2$	$b_1$	$b_1$	$c_1$
	$a_2$	$b_3$	$\perp$	$\perp$		$\perp$	$\perp$	$b_4$	$c_2$

$q_F$	$A$	$r.B$	$s.B$	$C$
	$a_1$	$b_1$	$b_1$	$c_1$
	$a_1$	$b_2$	$b_2$	$c_3$
	$a_2$	$b_1$	$b_1$	$c_1$
	$a_2$	$b_3$	$\perp$	$\perp$
	$\perp$	$\perp$	$b_4$	$c_2$

Figura 2.17. Relațiile  $q_L = r \circ \triangleright \triangleleft s$ ,  $q_R = r \triangleright \triangleleft \circ s$  și  $q_F = r \triangleright \circ \triangleleft s$ , respectiv

Trebuie menționat că joncțiunile externe pot fi definite și pentru joncțiunile-theta (joncțiunile generale) introducând o condiție  $F$  pe care joncțiunea externă trebuie să o satisfacă:  $r \circ \triangleright \triangleleft_F s$ ,  $r \triangleright \triangleleft_F \circ s$ ,  $r \triangleright \circ \triangleleft_F s$ .

## 2.2.7. Divizarea

Divizarea este cea mai complexă și mai greu de explicitat dintre operațiile prezentate în acest capitol. Ea permite căutarea, într-o relație a subtuplurilor care pot fi completate cu toate celealte din altă relație.

**Definiția 2.18.** Fie două relații  $r(R)$  și  $s(S)$ , astfel încât  $S \subseteq R$  și fie  $Q = R \setminus S$ . Relația  $q$  obținută prin operația de divizare (notația posibilă pentru divizare este  $r \div s$ ) a relației  $r$  prin relația  $s$  este definită pe mulțimea de atribute  $Q$ :

$$r \div s = \{t \mid \text{pentru } \forall t_s \in s(S) \quad \exists t_r \in r(R) \text{ & } t_r[Q] = t \text{ & } t_r[S] = t_s\}.$$

Divizarea poate fi obținută pornind de la operațiile diferență, produsul cartezian și diferență, precum urmează:

$$r \div s = \pi_Q(r) \setminus \pi_Q((\pi_Q(r) \times s) \setminus r).$$

Codd și-a imaginat operația divizării ca operațiune inversă a produsului cartezian. Fiind date relațiile  $r$  (cu schema  $R$ ) și  $s$  (cu schema  $S$ ), aplicarea operatorului de divizare va genera relația  $r \div s$  de grad  $|R \setminus S|$  ale cărei tupluri concatenate cu tuplurile relației  $s$  vor genera tupluri aparținând relației  $r$ . Cu alte cuvinte, fiind date două relații  $q(Q)$  și  $s(S)$ , dacă  $r = q \times s$ , atunci  $q = r \div s$ .

**Exemplul 2.16.** În figura 2.18, sunt prezentate relațiile  $r(A, B, C)$ ,  $s(B, C)$  și  $q(A)$ , unde  $q = r \div s$ . Să se observe că  $q \times s \subseteq r$  și că  $q$  conține un număr maximal de tupluri care posedă această proprietate.

$r$	$A$	$B$	$C$
$a_1$	$b_1$	$c_1$	
$a_2$	$b_1$	$c_1$	
$a_1$	$b_2$	$c_1$	
$a_1$	$b_2$	$c_2$	
$a_2$	$b_1$	$c_2$	
$a_1$	$b_2$	$c_3$	
$a_1$	$b_2$	$c_4$	
$a_1$	$b_1$	$c_5$	

$s$	$B$	$C$
	$b_1$	$c_1$
	$b_2$	$c_1$

$q$	$A$
	$a_1$

Figura 2.18. Relațiile  $r(A, B, C)$ ,  $s(B, C)$  și  $q = r \div s$

Deși este o operație complexă, divizarea relațională este deosebit de utilă pentru formularea interogărilor în care apare cuantorul universal,  $\forall$  (redat prin „toți”, „orice”, „oricare”, „niciun” etc.), care indică faptul că afirmația enunțată are loc pentru fiecare element din sfera subiectului, fără excepție.

### 2.3. Interogări în algebra relațională

Din secțiunea precedentă, se poate trage concluzia că uniunea, diferența, produsul cartezian, proiecția, selecția și redenumirea sunt operațiile de bază ale algebrei relaționale. Cu alte cuvinte, acestea constituie setul minimal de operații care oferă toate funcționalitățile algebrei relaționale.

Celelalte operații, precum intersecția, complementul, joncțiunea și diviziunea pot fi derivate din cele șase operații de bază, însă substituția acestora este, uneori, complexă. Prin urmare, un limbaj algebric de interogări ce ar folosi numai operațiile de bază poate fi destul de neeficient.

Operațiile algebrei relaționale nu prezintă interes doar pe plan teoretic. Contribuția practică a acestora este, de asemenea, importantă. Astfel, va fi nevoie de ele pentru optimizarea cererilor la nivel de limbaj al sistemelor relaționale. În afară de aceasta, ele își găsesc aplicarea în proiectarea calculatoarelor pentru baze de date: în loc să fie puse în aplicare în software-uri, sunt implantate direct în componentele hardware ale calculatorului.

O interogare poate fi definită ca fiind o funcție care, aplicată asupra unei instanțe a unei baze de date, produce o relație. Mai exact, fiind dată o schemă  $DB$  a unei baze de date, o interogare este o funcție care, pentru fiecare instanță  $db$  a lui  $DB$ , produce o relație definită pe o mulțime  $X$  de atribute. În algebra relațională, interogările unei scheme  $DB$  de baze de date sunt formulate cu ajutorul expresiilor, ale căror atomi sunt relații din  $db$ .

Se consideră baza de date alcătuită din relațiile următoare:

*articole(Art\_Id, Art\_Nume, Oraș, Bucăți, Pret);*  
*clienți(Cl\_Id, Cl\_Nume, Cl\_Oraș, Reducere);*  
*agenți(Ag\_Id, Ag\_Nume, Ag\_Oraș, Comision);*  
*comenzi(Lună, Cl\_Id, Ag\_Id, Art\_Id, ComBucăți, Sumă).*

### 2.3.1. Expresii algebrice

Din operațiile algebrei relaționale, este posibilă compunerea unui limbaj de interogare a bazelor de date. O întrebare poate fi apoi reprezentată de un arbore cu operatori relaționali. Parafrazarea în limba engleză a acestor expresii constituie baza limbajului SQL care va fi examinat în următorul capitol.

Folosind expresii cu operații ale algebrei relaționale pot fi elaborate răspunsuri la cele mai multe întrebări care pot fi formulate la o bază de date relațională. Mai exact, operațiile de bază ale algebrei relaționale constituie un limbaj complet, care se spune că are puterea unei logici de ordinul unu. Mai simplu spus, algebra relațională poate reda orice întrebare exprimabilă în logica de ordinul unu fără funcții, de exemplu, cu calcul de tupluri sau domenii. Cu toate acestea, algebra relațională poate fi, de asemenea, extinsă cu funcții [Zaniolo85].

**Definiția 2.19.** Fie  $U = A_1 \dots A_n$  mulțimea universală de atribute,  $\text{dom}(U) = \{\text{dom}(A_i) \mid A_i \in U \ \& \ 1 \leq i \leq n\}$  și,  $f_{\text{dom}} : U \rightarrow \text{dom}(U)$ , o funcție ce pune în corespondență fiecărui atribut din  $U$  un singur domeniu (mai multe atribute pot avea același domeniu de valori). Fie  $DB = \{R_1, \dots, R_m\}$  schema bazei de date, unde  $U = R_1 \dots R_m$  și  $db = \{r_1, \dots, r_m\}$  o bază de date cu schema dată  $DB$ . Fie  $\Theta = \{=, \neq, <, \leq, >, \geq\}$  mulțimea de operații aritmetice de comparare asupra domeniilor din  $\text{dom}(U)$  și fie  $O$  mulțimea de operații ce include cel puțin cele șase operații relaționale de bază. Atunci, septetul  $A = (U, \text{dom}(U), f_{\text{dom}}, DB, db, \Theta, O)$  se numește *algebră relațională*. *Expresie algebraică* asupra  $A$  este orice expresie bine formată (în conformitate cu restricțiile impuse operațiilor relaționale) de relații din  $db$  și relații constante cu scheme din  $DB$  asociate cu operații din  $O$ .

În expresiile algebrice, se admit paranteze și se presupune că operațiile binare nu au prioritate una față de alta, cu excepția priorității operației  $\cap$  față de  $\cup$ . Într-o consecutivitate de relații legate cu aceeași operație, parantezele pot fi omise, dacă, bineînțeles, operația este asociativă.

Numele de relații  $r_1, \dots, r_m$  servesc drept variabile, unde  $r_i$  parcurge mulțimea de relații cu schema  $R_i$ ,  $1 \leq i \leq m$ . Trebuie, însă, ținut cont de ambiguitatea că  $r_i$  denotă atât numele unei relații, cât și o instanță curentă a unei relații cu schema  $R_i$ .

Ca și orice limbaj formal, algebra relațională permite formularea expresiilor echivalente. Pentru notarea echivalenței, de obicei, se utilizează  $\equiv$ . Două expresii se numesc echivalente, dacă, în urma evaluărilor, se obține aceeași relație.

Echivalența expresiilor în algebra relațională este extrem de importantă la optimizarea interogărilor exprimate în SQL prin faptul că ele sunt translate în expresii ale algebrei relaționale, expresii pentru care se evaluatează costul satisfacerii interogării. Costul este exprimat în termenii dimensiunii rezultatelor intermediare și finale. Dacă există mai multe expresii diferite echivalente interogării, atunci se va alege acea expresie care costă cel mai puțin. Pentru a obține expresii echivalente, se utilizează transformări, precum cele reprezentate pentru fiecare operație la sfârșitul secțiunilor respective. Cu aceste transformări se substituie o expresie echivalentă cu alta.

### 2.3.2. Selecții generalizate

Exemplele examineate până în prezent selectau tupluri care satisfăceau un criteriu simplu de selecție, de exemplu, „Să se găsească toate datele despre articolele produse în Chișinău”. Criteriile de selecție pot fi mai complexe, ca în interogarea: „Să se găsească toate datele despre articolele cu prețul mai mare de 100 de lei și produse în Chișinău”. Această cerere poate fi exprimată, apelând la o compoziție de selecții:

$$\sigma_{Preț > 100}(\sigma_{Oraș = "Chișinău"}(articole)).$$

O compoziție de selecții poate fi redată printr-o selecție cu o conjuncție de criterii. Cererea precedentă este echivalentă cu următoarea:

$$\sigma_{Preț > 100 \ \& \ Oraș = "Chișinău"}(articole)).$$

Astfel, compozitia mai multor selecții permite formularea unui criteriu de căutare constituit dintr-o conjuncție de criterii atomice. La rândul său, o uniune de selecții poate fi exprimată printr-o selecție cu un criteriu disjuncție. Iată o cerere de căutare a datelor despre articolele din Chișinău sau despre articolele cu un preț mai mare de 100:

$$\sigma_{Oraș = "Chișinău"}(articole) \cup \sigma_{Preț > 100}(articole).$$

Această cerere, de asemenea, poate fi simplificată, transformând uniunica în disjuncție logică și introducând-o în criteriu:

$$\sigma_{Oraș = "Chișinău"} \vee \sigma_{Pref > 100} (\text{articole}).$$

În sfârșit, diferența poate să exprime negația și “eliminarea” tuplurilor. De exemplu, iată o cerere de selecție a articolelor cu un preț mai mare de 100, dar care nu sunt produse în Chișinău:

$$\sigma_{Pref > 100}(\text{articole}) \setminus \sigma_{Oraș = "Chișinău"}(\text{articole}).$$

Această cerere este echivalentă cu o selecție în care se aplică operatorul “ $\neq$ ”:

$$\sigma_{Pref > 100 \ \& \ Oraș \neq "Chișinău"}(\text{articole}).$$

Așadar, operațiile de uniune și diferență permit definirea unei selecții  $\sigma_F$  cu criteriul  $F$  constituit dintr-o expresie booleană. Dar trebuie menționat faptul că, deși toate selecțiile cu “sau” pot fi exprimate folosind operația uniunea, afirmația inversă nu este adevărată.

### 2.3.3. Interogări conjunctive

Interogările conjunctive sunt cele mai răspândite și constituie o categorie aparte în algebra relațională. Intuitiv, este vorba de toate căutările care se exprimă cu conjuncția “și”, spre deosebire de cele cale implică conjuncția “sau” ori negații. În algebra relațională, acestea constituie expresiile care pot fi redată numai prin trei operații:  $\pi$ ,  $\sigma$ ,  $\times$  (și, deci, indirect,  $\triangleright \triangleleft$ ).

Cele mai simple sunt cererile care nu utilizează decât operațiile  $\pi$  și  $\sigma$ .

**Exemplul 2.17.** Se consideră următoarele interogări conjunctive simple:

- Să se găsească denumirile articolelor produse în orașul Chișinău:

$$\pi_{Art\_Nume}(\sigma_{Oraș = "Chișinău"}(\text{articole}));$$

- Să se găsească numele agenților care încasează un comision de 10%:

$$\pi_{Ag\_Nume}(\sigma_{Comision=10}(agenți));$$

- Să se găsească numele și reducerile acordate clienților din Orhei:

$$\pi_{Cl\_Nume, Reducere}(\sigma_{Cl\_Oras="Orhei"}(clienți)).$$

Cererile care sunt ceva mai complexe (dar extrem de utile) implică și operația joncțiunea (produsul cartezian prezintă mai mult interes în cazul în care este asociat cu selecția). Joncțiunea se utilizează atunci când atributurile solicitate pentru evaluarea cererii se găsesc în cel puțin două relații. Aceste "atributuri necesare" reprezintă:

- Atributurile care trebuie să fie incluse în relația rezultat.
- Atributurile asupra cărora se aplică criteriul de selecție.

**Exemplul 2.18.** Se consideră următoarea cerere: "Să se găsească numele și orașele articolelor care au fost vândute în luna ianuarie". O analiză simplă este suficientă pentru a constata că e nevoie de atributurile *Art\_Nume* și *Oraș* care apar în schema relației *articole*, precum și *Lună* care apare în relația *comenzi*.

Astfel, pentru asocierea tuplurilor din relațiile *articole* și *comenzi* este aplicată operația joncțiunea. Bineînțeles, în acest caz, trebuie să fie determinat atributul (sau atributurile) de joncțiune. În cazul dat, atributul *Art\_Id* este unul care apare în ambele relații. Uneori, în expresii, pentru explicitate pot fi indicate atributurile de joncțiune:

$$\pi_{Art\_Nume, Oraș}(articole \bowtie \sigma_{articole.Art\_Id=comenzi.Art\_Id} \sigma_{Lună="ianuarie"}(comenzi)).$$

În practică, majoritatea operațiilor de joncțiune se bazează pe faptul că se efectuează asupra atributelor care constituie cheia primară într-o relație și cheia externă în celalătă relație. Nu este vorba de o regulă absolută, dar ea rezultă din faptul că joncțiunea, astfel, va permite reconstituirea tuplurilor de date care sunt natural asociate (ca denumirile articolelor și lunile de vânzare ale lor), dar care au fost repartizate în mai multe relații la momentul proiectării logice a bazei de date.

**Exemplul 2.19.** Urmează alte câteva expresii ce ilustrează această stare de fapte:

- Să se găsească agenții de la care a procurat marfă Petrescu:  
 $\pi_{Ag\_Id}(comenzi \triangleright \triangleleft \sigma_{Cl\_Nume = "Petrescu"}(clienți));$
- Să se găsească orașele în care a vândut marfă agentul 03:  
 $\pi_{Cl\_Oras}(\sigma_{Ag\_Id = "03"}(comenzi) \triangleright \triangleleft clienți);$
- Să se găsească numele clienților care au procurat televizoare:  
 $\pi_{Cl\_Nume}(\sigma_{Art\_Nume = "televizor"}(articole) \triangleright \triangleleft comenzi \triangleright \triangleleft clienți);$

Ultima expresie algebrică conține două joncțiuni, realizate pe două chei primare și/sau externe. Aceste chei definesc legăturile dintre relații și servesc drept suport pentru formularea cererilor.

Urmează acum un exemplu care arată că această regulă nu este sistematică.

**Exemplul 2.20.** Se dorește formularea unei cereri care ar căuta numele clienților și orașele lor de reședință. Clienții respectivi au procurat cel puțin un articol produs în orașele lor de reședință.

Pentru a obține răspunsul la această interogare, este nevoie de datele stocate în relațiile *clienți*, *comenzi* și *articole*. Cererea poate fi redată prin următoarea expresie algebrică:

$$\pi_{Cl\_Nume, Cl\_Oras}(clienți \triangleright \triangleleft Cl\_Id = Cl\_Id, Cl\_Oras = Oras \\ (comenzi \triangleright \triangleleft Art\_Id = Art\_Id articole)).$$

Joncțiunea dintre relațiile *comenzi* și *articole* se realizează pe perechea cheie externă-cheie primară, pe când joncțiunea dintre relația *clienți* și relația obținută din prima joncțiune - nu.

### 2.3.4. Interogări neconjunctive

În sfârșit, vor fi examinate unele exemple de interogări care implică operația diferență. Utilizarea acestei operații este mai puțin frecventă, dar ea este absolut necesară, deoarece nu poate fi exprimată cu ajutorul celor trei operații “conjunctive” menționate anterior.

#### 2.3.4.1. Interogări cu diferențe

Diferența permite exprimarea tuturor cererilor în care figurcază o negație. De exemplu, este cazul în care se dorește selectarea datelor care nu satisfac o anumită proprietate, sau selectarea tuturor datelor care trebuie să satisfacă o condiție.

Următoarea interogare asupra bazei de date este o ilustrare concretă de acest tip. Cererea “Care sunt articolele ce nu au fost comandate prin agentul 03?” poate fi exprimată în forma algebraică în modul următor:

$$\pi_{Art\_Id}(articole) \setminus \pi_{Art\_Id}(\sigma_{Ag\_Id = '03'}(comenzi))$$

Precum ne sugerează acest exemplu, procedura generală de construire a unei cereri de tipul “Toate obiectele  $O$  care nu satisfac proprietatea  $P$ ” este următoarea:

1. Se construiește prima cerere  $A$ , care găsește toate  $O$ .
2. Se construiește a doua cerere  $B$ , care găsește toate  $O$  ce satisfac  $P$ .
3. În sfârșit, se face  $A \setminus B$ .

Cererile  $A$  și  $B$  pot, bineînțeles, fi de complexitate arbitrară, care ar implica jonețuni, selecții etc. Singura constrângere impusă constă în faptul că rezultatele cererilor  $A$  și  $B$  trebuie să fie relații cu scheme compatibile.

**Exemplul 2.21.** Urmează două cereri complementare care ilustrează acest principiu.

- Să se găsească orașele, unde se produc articole, dar nu locuiesc agenți:

$$\pi_{Oraș}(articole) \setminus \pi_{Agenți}(agenți);$$

- Să se găsească articolele care nu au fost comandate de nici un client din Orhei

$$\pi_{Art\_Id}(articole) \setminus \pi_{Art\_Id}(\sigma_{Cl\_Oraș = "Orhei"}(clienți) \triangleright \triangleleft comenzi);$$

#### 2.3.4.2. Complementul unei mulțimi

Diferența poate fi aplicată și pentru calcularea complementului unei mulțimi.

**Exemplul 2.22.** Fie cererea următoare: Să se găsească articolele și clienții care nu le-au cumpărat. Cu alte cuvinte, între toate asocierile posibile articol-client, se caută acele care nu sunt reprezentate în bază!

Acesta este un caz rar, unde produsul cartezian e util. El tocmai permite constituirea tuturor asocierilor posibile articol-client. Apoi rămâne să se construiască asocierile care sunt în bază, adică care articol și de către care client a fost cumpărat, după care, cu ajutorul operației diferență, se obține răspunsul la cererea formulată:

$$(\pi_{Art\_Id}(articole) \times \pi_{Cl\_Id}(clienți)) \setminus \pi_{Art\_Id, Cl\_Id}(comenzi).$$

#### 2.3.4.3. Cuantificarea universală

În sfârșit, diferența este necesară la formularea cererilor care apelează la cuantificatorul de universalitate; cereri care necesită, de exemplu, ca o proprietate să fie întotdeauna adevărată. A priori, nu se vede de ce diferența poate fi utilă în cazul dat. Accasta rezultă, pur și simplu, din faptul că o proprietate este adevărată pentru toate elementele unei mulțimi, dacă și numai dacă nu există niciun element al acestei mulțimi pentru care proprietatea este falsă.

În practică, pentru a exprima cererile de acest tip, înțotdeauna, se apelează la partea a două a echivalenței mai sus-formulate. Astfel, înțotdeauna, în locul cuantificatorului universal, se aplică negația și cuantificatorul existențial.

**Exemplul 2.23.** Fie cererea: Care sunt clienții ale căror articolele cumpărate au un preț mai mare de 100 fiecare? Această întrebare poate fi reformulată într-o echivalentă: Care sunt clienții ce nu au cumpărat niciun articol cu un preț mai mic sau egal cu 100? Ultima întrebare poate fi formulată de expresia:

$$\pi_{Cl\_Id}(comenzi) \setminus \pi_{Cl\_Id}(\sigma_{Preț \leq 100}(articole) \triangleright \triangleleft comenzi).$$

În continuare, urmează un tip de cereri mai complexe. Cu toate că enunțul (în română) este destul de simplu, acest lucru nu se poate spune despre expresia algebraică a interogării.

**Exemplul 2.24.** Fie cererea: Să se găsească clienții ce au comandat toate articolele comandate de toți ceilalți. Traducerea acestei cereri într-o echivalentă poate fi următoarea: Să se găsească clienții ce nu au comandat niciun articol, care nu a fost comandat de cineva. Se constată o negație dublă, ceea ce implică următoarea expresie:

$$\pi_{Cl\_Id}(comenzi) \setminus \pi_{Cl\_Id}((\pi_{Cl\_Id}(comenzi) \times \pi_{Art\_Id}(comenzi)) \setminus \\ \pi_{Cl\_Id, Art\_Id}(comenzi)).$$

Trebuie menționat că expresia obținută poate fi rescrisă cu ajutorul operației divizarea:

$$\pi_{Cl\_Id \setminus Art\_Id}(comenzi) \div \pi_{Art\_Id}(comenzi).$$

### 2.3.5. Un exemplu integrator

**Exemplul 2.25.** Urmează expresiile algebrei relaționale ce corespund unor interogări formulate asupra bazei de date  $db = (\text{articole}, \text{cli}enți, \text{agen}ți, \text{comen}zi)$  prezentată la începutul secțiunii.

- Să se găsească clienții ce au plasat comenzi prin agenții care au livrat articolul *art03*:

$$\pi_{Cl\_Id}(comenzi \triangleright \triangleleft \pi_{Ag\_Id}(\sigma_{Art\_Id = "art03"}(comenzi)));$$

- Să se găsească clienții ce au comandat articolele *art01* și *art07*:

$$\pi_{Cl\_Id}(\sigma_{Art\_Id = "art01"}(comenzi)) \cap$$

$$\pi_{Cl\_Id}(\sigma_{Art\_Id = "art07"}(comenzi));$$

- Să se găsească clienții ce au comandat articole cu o reducere la preț ca a clientilor din Chișinău sau Iași:

$$\pi_{Cl\_Id}(cli\!en\!ți \triangleright \triangleleft$$

$$\pi_{Reducere}(\sigma_{Cl\_Oraș = "Chișinău" \vee Cl\_Oraș = "Iași"}(cli\!en\!ți)));$$

- Să se găsească numele clientilor ce au comandat cel puțin un articol la prețul 0.50 lei:

$$\pi_{Cl\_Nume}((\pi_{Art\_Id}(\sigma_{Preț=0.50}(articole)) \triangleright \triangleleft comenzi) \triangleright \triangleleft cli\!en\!ți);$$

- Să se găsească numele clientilor ce n-au dat nicio comandă prin agentul *a03*:

$$\pi_{Cl\_Nume}(cli\!en\!ți \triangleright \triangleleft (\pi_{Cl\_Id}(cli\!en\!ți) \setminus$$

$$\pi_{Cl\_Id}(\sigma_{Ag\_Id = "a03"}(comenzi))));$$

- Să se găsească clienții ce-și plasează comenzi numai prin agentul *a03*:

$$\pi_{Cl\_Id}(cli\!en\!ți) \setminus \pi_{Cl\_Id}(\sigma_{Ag\_Id \neq "a03"}(comenzi));$$

- Să se găsească articolele ce nu au fost comandate de vreun client din Chișinău printr-un agent cu sediul la Bălți:

$$\pi_{Art\_Id}(articole) \setminus \\ \pi_{Art\_Id}((\pi_{Cl\_Id}(\sigma_{Cl\_Oras} = "Chișinău"(clienti)) \triangleright \triangleleft comenzi) \triangleright \triangleleft \\ \sigma_{Ag\_Oras} = "Bălți"(agenți));$$

- Să se găsească articolele comandate prin agenții care au luat comenzi de la clienții ce au comandat cel puțin un articol printr-un agent, ce a servit clientul c001:

$$\pi_{Art\_Id}(comenzi) \triangleright \triangleleft (\pi_{Ag\_Id}(comenzi) \triangleright \triangleleft \\ (\pi_{Cl\_Id}(comenzi) \triangleright \triangleleft \\ (\pi_{Ag\_Id}(\sigma_{Cl\_Id} = "c001"(comenzi))))));$$

- Să se găsească agenții ce au primit comenzi de livrare a articolelor comandate și de clientul c004:

$$\pi_{Ag\_Id Art\_Id}(comenzi) \div \\ \pi_{Art\_Id}(\sigma_{Cl\_Id = "c004"(comenzi)});$$

- Să se găsească numele clienților ce au comandat toate tipurile de articole la prețul 0.50 lei:

$$\pi_{Cl\_Name}(clienți) \triangleright \triangleleft (\pi_{Cl\_Id Art\_Id}(comenzi) \div \\ \pi_{Art\_Id}(\sigma_{Pret = 0.50}(articole))));$$

## 3. SQL: Interogarea și actualizarea bazelor de date

Standardul SQL – definește o bază comună pentru definirea, accesul și folosirea datelor în modelul relațional. De fapt, standardul SQL este un limbaj cuprindător care include sublimbaje pentru: interogarea și manipularea datelor, definirea datelor și restricțiilor de integritate, definirea viziunilor, funcțiilor și procedurilor stocate, controlul tranzacțiilor, gestiunea declanșatoarelor (triggers), autorizarea utilizatorilor.

Scopul acestei secțiuni constă în prezentarea instrucțiunilor SQL caracteristice limbajului de interogare a datelor și limbajului de manipulare a datelor. Limbajul de interogare a datelor este utilizat pentru a extrage date din baza de date, iar limbajul de manipulare a datelor permite utilizatorilor să insereze, să reactualizeze și să șteargă date din baza de date.

### 3.1. Prezentare generală

#### 3.1.1. Premisele apariției limbajului SQL

Toate sistemele de gestiune a bazelor de date utilizează SQL pentru crearea, modificarea și accesul la date. Limbajul SQL (*Structured Query Language*) a apărut după lucrările matematice ale lui E.Codd, lucrări care au pus fundamentele bazelor de date relaționale. Pot fi menționate trei premise principale de răspândire și implementare a limbajului SQL în SGBD-uri și limbi de programare.

În primul rând, structurarea și manipularea datelor a devenit mult mai complexă. Pentru o aplicație de talie medie, baza de date conține, deseori, mai mult de treizeci de relații strâns interconectate. Astfel, este anevoieoașă manipularea datelor în mod algoritmic tradițional. SQL este un limbaj declarativ care permite interogarea bazei de date fără să țină cont de reprezentarea fizică a datelor, amplasarea lor, căile de acces sau de

algoritmi necesari. Drept rezultat, SQL se adresează unei comunități largi de utilizatori (nu neapărat informaticieni), fapt ce constituie atuul cel mai spectaculos (și cel mai cunoscut) al SGBD-urilor relaționale. El se poate utiliza în mod interactiv, dar, în aceeași măsură, în asociere cu interfețe grafice și, cel mai important, în limbaje de programare. Ultimul aspect este prețios, deoarece SQL nu permite scrierea programelor în sensul curent al acestui termen și deci trebuie să se asocieze cu un limbaj, cum ar fi C, COBOL sau JAVA pentru a realiza prelucrări complexe cu acces la baza de date.

În al doilea rând, arhitectura client-server este omniprezentă. În timp ce stațiile client execută codul unei aplicații girând, în particular, o interfață grafică, serverul optimizează procesul de manipulare și control al datelor. În plus, aplicațiile trebuie să fie portabile și să gireze date virtuale, cu alte cuvinte, provenite de la oricare server. Elaborarea unei aplicații într-un mediu eterogen nu este posibilă decât cu comunicarea între client și server, dacă nu este realizată de primitive SQL normalize.

Și, în sfârșit, aplicațiile ce trebuie elaborate devin din ce în ce mai complexe. Profilul programatorului frecvent se schimbă. El trebuie permanent să prelucreze date din ce în ce mai voluminoase, să integreze tehnici de manipulare a interfețelor, să stăpânească logica bazată pe evenimente și programarea orientată pe obiecte, în regim multimedia și toate acestea, în contextul client-server, unde se folosesc protocoale de rețele eterogene. Deci, se cere un mediu de elaborare performant, care întreține o mulțime de sarcini suplimentare. Astfel, au apărut software-uri de elaborare, care permit programatorului să se concentreze asupra aplicației propriu-zise: generatoare de ecrane, de rapoarte, de cereri, de ajutor pentru conceptualizarea programului, de conexiune la baza de date prin rețele. În toate aceste software-uri, simplitatea și calitățile standarde ale lui SQL fac ca ultimul să fie utilizat de fiecare dată când este necesar.

### 3.1.2. Scurt istoric

Istoria limbajului SQL începe în anul 1974 în laboratorul campaniei IBM din San Jose, California, cu o descriere, propusă de Donald Chamberlin, a unui limbaj pentru specificarea caracteristicilor unei baze de date cu o schemă logică relațională. Acest limbaj se numea SEQUEL (*Structured*

*English Query Language*), care a fost implementat într-un prototip SEQUEL-XRM, pe parcursul anilor 1974-1975. Experimentele efectuate între anii 1976 și 1977 cu acest prototip au adus la o revizuire a limbajului (SEQUEL/2) care, mai târziu, din motive înțelese, își schimbă numele, devenind SQL [Chamberlin76]. Un prototip (*System R*) bazat pe acest limbaj a fost adoptat și utilizat în campania IBM și de unii clienți selectați. Grație succesului atins de acest sistem, care încă nu era comercializat, și alte campanii au inițializat elaborarea produselor relaționale bazate pe SQL. În 1981, IBM începe lansarea produselor sale relaționale și, în 1983, începe vânzarea SGBD-ului DB2 [IBM82, Date84]. Pe parcursul anilor optzeci, numeroase companii, de exemplu, Oracle și Sybase, comercializează produse bazate pe SQL, limbaj care deja tinde a fi un standard în ce privește bazele de date relaționale [IBM87].

În 1986, ANSI (*American National Standards Institute*) adoptă SQL (în esență, dialectul SQL al companiei IBM) în calitate de standard pentru limbajele relaționale, iar, în 1987, devine standard adoptat și de ISO (*International Standards Organization*). Această versiune a limbajului este cunoscută sub numele SQL/86. Pe parcursul anilor următori, limbajul a suportat alte versiuni care s-au sfârșit cu versiunea SQL/89 [Date89, X/Open92] și succesiv cea actuală SQL/92 (prescurtat, SQL2) [ISO92].

Faptul existenței unui standard pentru limbajele bazelor de date relaționale deschide calea spre intercomunicabilitatea diverselor software-uri bazate pe el. Din punct de vedere practic, din păcate, lucrul merge puțin altfel. Realmente, fiecare producător adoptă și implementează propriul SGBD cu un SQL, extins astfel ca baza de date să susțină viziunea proprie a domeniului de interes.

În anul 1993, Microsoft lansează ODBC (*Open Database Connectivity*) ce permite aplicațiilor Windows să se conecteze la un SGBD și să execute comenzi SQL. ODBC este o bibliotecă de funcții proiectată pentru a furniza o interfață de programare a aplicațiilor (*API, Application Programming Interface*), care să asigure suportul pentru sistemele de baze de date. Comunicarea cu SGBD-ul se face printr-un driver, precum comunică sistemul de operare Windows cu o imprimantă prin intermediul unui driver.

Peste un an, Borland lansează IDAPI (*Integrated Database Application Programming Interface*), care este o bibliotecă de funcții SQL ce se pot integra într-un program-gazdă.

În ultimii ani, au avut loc procese de revizuire a limbajului SQL2 de către comitetele ANSI și ISO, care, începând cu anul 1999, au propus completări, conducând spre SQL3 [Shaw90]. Caracteristicile principale ale noii versiuni vin pentru a transforma limbajul SQL într-un limbaj stand-alone (SQL2 este un limbaj găzduit de alte limbi) și introducerea unor tipuri noi de date mai complexe cu posibilități de gestiune a datelor orientate obiect.

### 3.1.3. Categoriile de instrucțiuni SQL

Instrucțiunile limbajului SQL pot fi grupate în categorii în funcție de utilitatea lor și de entitățile manipulate. Pot fi distinse șase categorii de instrucțiuni.

**Limbajul de definire a datelor** (DDL, *Data Definition Language*) este un limbaj orientat la structura bazei de date. DDL permite crearea, modificarea și suprimarea obiectelor bazei de date (relațiilor, atributelor, constrângerilor, viziumilor, indecsilor,...). El permite, de asemenea, definirea domeniilor de date (numere, secvențe de caractere, date temporale,...) și adăugarea constrângerilor asupra valorilor datelor. și permite, în cele din urmă, autorizarea și interzicerea accesului la date, activarea și dezactivarea auditului pentru un utilizator dat. Instrucțiunile SQL care folosesc comenziile *CREATE*, *ALTER* și *DROP*, sunt considerate parte a DDL.

**Limbajul de interogare a datelor** (DQL, *Data Query Language*) include instrucțiuni SQL care permit obținerea datelor din baza de date. Este limbajul în care utilizatorii pot formula cereri la baza de date. Deși reprezintă o parte foarte importantă a limbajului SQL, DQL este format din instrucțiuni care folosesc o singură comandă: *SELECT*.

**Limbajul de manipulare a datelor** (DML, *Data Manipulation Language*) constituie setul de comenzi referitor la actualizarea conținutului bazei de date. DML permite introducerea a noi tupluri, modificarea tuplurilor, suprimarea tuplurilor nedeterminate din baza de date și,

în fine, blocarea relațiilor. Instrucțiunile principale sunt *INSERT*, *UPDATE*, *DELETE*, *LOCK*.

**Limbajul de protecție a accesului** (DCL *Data Control Language*) este preocupat de gestiunea drepturilor de acces la relații. Acesta include instrucțiuni SQL care permit administratorilor să controleze accesul la datele din baza de date și folosirea diferitelor privilegii ale sistemului DBMS, cum ar fi oferirea și revocarea accesului la baza de date. Instrucțiunile SQL care folosesc comenzi *GRANT* și *REVOKE* sunt considerate parte a DCL.

**Limbajul pentru controlul tranzacțiilor** (TCL, *Transaction Control Language*) gestionează modificările efectuate de DML, adică caracteristicile tranzacțiilor și validarea și anularea modificărilor. Comenzi TCL nu respectă cu exactitate sintaxa instrucțiunilor SQL, dar afectează puternic comportamentul instrucțiunilor SQL incluse în tranzacții. Din limbajul TCL fac parte instrucțiunile: *COMMIT*, *SAVEPOINT*, *ROLLBACK*, *SET TRANSACTION*.

**SQL integrat** (*Embedded SQL*) asigură încapsularea instrucțiunilor SQL în codul de program scris într-un limbaj de programare. SQL integrat permite declararea obiectelor sau instrucțiunilor, executarea instrucțiunilor, gestiunea variabilelor și cursoarelor, precum și tratarea erorilor. Comenzi *DECLARE*, *CONNECT*, *OPEN*, *FETCH*, *CLOSE*, *WHENEVER* etc. sunt considerate parte a SQL integrat. Limbajul SQL integrat este destinat programatorilor de aplicații.

Lista de avantaje și dezavantaje ale limbajului SQL, așa cum se prezintă astăzi, poate fi ușor schițată: la activul său se înregistrează simplitatea sa, supletea sa și funcțiile sale largi; la pasivul său, lipsa stricteții (el permite și încurajează diverse formulări procedurale), o utilizare greoaie (cuvinte-cheie, fără utilizarea mijloacelor moderne de interfețe om/mașină).

Din punct de vedere structural, limbajul SQL este compus din comenzi, clauze, operatori și funcții. Aceste elemente se combină în instrucțiuni ale componentelor SQL menționate.

Se prezintă notațiile folosite în sintaxa limbajului. Cuvintele-cheie ale limbajului vor fi scrise cu caractere majuscule, relațiile cu caractere

minuscule, iar atributele vor începe cu un caracter majuscul. Parantezele rotunde trebuie privite întotdeauna ca și cuvintele-cheie ale SQL. De asemenea:

- parantezele unghiulare „<...>” reprezintă simboluri neterminale a limbajului;
- parantezele pătrate „[...]” indică faptul că termenii delimitați sunt optionali (pot să nu apară sau să apară doar o singură dată);
- acoladele „{...}” grupează elementele într-o formulă;
- bara verticală „|” indică faptul că unul dintre termenii delimitați de aceasta trebuie să apară.

### 3.2. Cele mai simple interogări

Instrucțiunea *SELECT* este cea mai complexă și mai puternică dintre instrucțiunile SQL. Cu aceasta pot fi recuperate date dintr-o sau mai multe relații, extrase anumite tupluri și chiar pot fi obținute rezumate ale datelor stocate în baza de date, din care motiv este numită „regina” limbajului SQL.

#### 3.2.1. Instrucțiunea SELECT

Pentru a extrage datele din baza de date se utilizează instrucțiunea *SELECT*. Sintaxa instrucțiunii *SELECT* este foarte bogată. Ea va fi desfășurată puțin câte puțin, pe parcursul expunerii materiei. Prezentarea instrucțiunii va începe pentru a vedea interogările simple bazate pe un singură relație și se vor limita, deocamdată, la următoarea sintaxă:

*SELECT <listă de atrbute> FROM <nume relație>;*

Instrucțiunea începe cu clauza *SELECT*, urmată de o listă de atrbute, după care urmează clauza *FROM* și apoi, un nume de relație. În definiție, *<listă de atrbute>* este lista de atrbute care se dorește să fie schema relației-rezultat. Numele atrbutelor trebuie separate prin virgulă. Acestea vor apărea în urma interogării în ordinea în care sunt specificate. Elementul *<nume relație>* indică de unde vor fi extrase datele, iar clauza *FROM* este singura clauză obligatorie în instrucțiunea *SELECT*.

**Exemplul 3.1.** Numele și prenumele tuturor funcționarilor vor fi extrase din relația funcționari de instrucțiunea următoare:

*SELECT Nume, Prenume FROM funcționari;*

În locul elementului *<listă de atribute>* poate fi specificat simbolul asterisc, “\*”. Simbolul asterisc, se utilizează pentru desemnarea tuturor atributelor. O clauză *SELECT* de forma *SELECT \*...* indică faptul că trebuie extrase toate attributele relației care apare în clauza *FROM*.

**Exemplul 3.2.** Utilizarea asteriscului, “\*”, după clauza *SELECT* presupune că se cere selectarea tuturor datelor despre funcționari. Interogarea returnează, în consecință, toate attributele și afișează toate tuplurile relației *funcționari*:

*SELECT \* FROM funcționari;*

Clauza *SELECT* poate conține, de asemenea, expresii aritmetice care constau din operatori aritmetici care se aplică asupra constantelor sau atributelor. Operatorii aritmetici sunt de două tipuri: binari și unari. Cei binari acceptă doi operanzi, pe când cei unari acceptă un singur operand.

Operatorii aritmetici sunt utilizati în expresii și rezultatul acestor expresii este o valoare numerică. Operatorii aritmetici sunt: adunarea (+), scăderea (-), înmulțirea (\*) și împărțirea (/). Ordinea de precedență a operatorilor poate fi schimbată cu ajutorul parantezelor.

Astfel, sintaxa instrucțiunii *SELECT* poate fi dezvoltată, precum urmează:

*SELECT <expresie1>, ..., <expresieN>  
FROM <nume relație>;*

Termenul *<expresie>* se referă fie la un atribut, fie la o expresie aritmetică asupra numerelor și atributelor existente pentru generarea unui nou atribut.

**Exemplul 3.3.** Următoarea instrucțiune afișează numele, prenumele și salariul majorat cu 10% ale tuturor funcționarilor:

*SELECT Nume, Prenume, Salariu\*I.1  
FROM funcționari;*

Trebuie menționat că nu se folosesc două semne minus (-) într-o expresie aritmetică pentru a indica scăderea unui număr negativ. Semnul „-“ este rezervat în Oracle pentru a indica începutul comentariilor.

### 3.2.2. Clauza DISTINCT

Nu este greu de observat că instrucțiunile de mai sus nu reprezintă altceva decât exemple ale operației relaționale *proiecția* aplicate asupra relației specificate. Însă, spre deosebire de varianta teoretică, relația obținută poate conține tupluri duplicate.

Specificarea cheilor permite evitarea duplicatelor în relațiile de bază, stocate în baza de date, dar nu garantează că în urma cererilor nu vor apărea relații cu asemenea tupluri. Limbajele formale de interogare se bazează pe relațiile matematice. Din acest motiv, în cadrul relațiilor, nu sunt permise tupluri duplicate, dar, deoarece eliminarea acestora este extrem de costisitoare, SGBD-urile admit dupliicatele. Interrogările SQL pot extrage multimi de tupluri ce conțin aceleași valori pentru toate atributele spre deosebire de algebra relațională și calculul relațional, astfel, rezultatul unei interogări poate să nu mai fie o relație în sensul strict al acestui termen.

- Dat fiind faptul că operația de eliminare a dupliacelor este consumatoare de timp și adesea nu este necesară, executarea acestei operații este lăsată la latitudinea persoanei ce implementează interogarea. Dacă se dorește emularea comportării din algebra relațională în SQL, ar trebui eliminate toate dupliacetele la fiecare execuție a unei operații de proiecție. Cu acest scop, se poate folosi clauza *DISTINCT*, iar dacă se dorește să se obțină asigurarea că tuplurile dupliacat nu sunt eliminate, se folosește clauza *ALL*.

Syntaxa generală a interogării presupune că, între comanda *SELECT* și primul atribut din lista de attribute, se includ cuvintele-cheie *DISTINCT* sau *ALL*:

*SELECT [<DISTINCT | [ALL]>]...*

Clauza *DISTINCT* impune eliminarea tuplurilor dupliacate, iar opțiunea *ALL* indică faptul că vor fi păstrate toate tuplurile din rezultat (deci, inclusiv dupliacetele). Clauza *ALL* este opțiunea implicită.

**Exemplul 3.4.** Mulți funcționari pot locui pe adrese asociate de același oficiu poștal. În cazul în care se dorește forțarea eliminării dupliilor codului poștal, se introduce opțiunea *DISTINCT*. Prin urmare, cererea

```
SELECT DISTINCT CodPoștal
FROM funcționari;
```

nu afișează codurile poștale redundante.

**Exemplul 3.5.** Este important să se menționeze că SQL permite utilizarea explicită a cuvântului-cheie *ALL* pentru a specifica, în mod explicit, că duplicatele nu se elimină:

```
SELECT ALL CodPoștal
FROM funcționari;
```

### 3.2.3. Clauza ORDER BY

În general, rezultatul unei interogări conține tupluri, aranjate într-o ordine oarecare. Dacă se dorește impunerea unei ordonări, după un anumit criteriu asupra tuplurilor returnate de o interogare, se va utiliza clauza *ORDER BY*. Sintaxa este:

```
...ORDER BY <expresie> [ ASC | DESC ] [, ...]
```

Termenul *<expresie>* se referă fie la un atribut, fie la o serie de operații aritmetice de bază (i.e. +, -, \* și /) asupra atributelor existente pentru generarea unui nou atribut. Sortarea se poate face atât pentru valori alfabetice, cât și valori numerice. *ASC* specifică ordinea ascendentă, iar *DESC* ordinea descendentă de sortare. Absența specificării *ASC* sau *DESC* presupune ordinea ascendentă care este utilizată în mod implicit.

Clauza *ORDER BY* permite sortarea rezultatului final al interogării. Aceasta este ultima clauza în orice instrucțiune SQL și trebuie să apară doar o singură dată într-un *SELECT*, chiar dacă există interogări imbricate sau un set de interogări cu operatori pe mulțimi.

În absența clauzei *ORDER BY*, ordinea tuplurilor este aleatorie și nu este garantată. De multe ori, plasarea cuvântului-cheie *DISTINCT* este suficientă pentru a stabili o sortare, deoarece SGBD-ul trebuie să se antreneze într-o comparație de tupluri, dar acest mecanism nu garantează

o soluție, deoarece sortarea are loc într-un mod necontrolabil care poate varia de la un server la altul.

Trebuie menționat că sortarea are un cost computerial ridicat din care motive trebuie aplicată numai când este necesară.

**Exemplul 3.6.** Instrucțiunea ce urmează extrage tuplurile din relația *funcționari*, le sortează (ascendent) după atributul *Nume* și construiește din ele o relație definită pe schema {*CodPoștal*, *Nume*, *Telefon*}:

```
SELECT CodPoștal, Nume, Telefon
      FROM funcționari ORDER BY Nume;
```

Când se utilizează atât clauza *WHERE*, cât și clauza *ORDER BY*, *WHERE* apare întotdeauna înaintea lui *ORDER BY*.

**Exemplul 3.7.** Sortarea poate fi multiplă. Astfel, pentru sortarea alfabetică (ascendentă) în funcție de nume și apoi de prenume ale funcționarilor, se folosește interogarea:

```
SELECT * FROM funcționari
          ORDER BY Nume, Prenume;
```

Atunci când mai multe atribute (sau expresii) sunt specificate, sortarea se face, în primul rând, după primul, apoi pentru următorul pentru tuplurile care sunt egale în conformitate cu primul. Numărul de atribute, după care se pot ordona tuplurile întoarse, este numărul maxim de atribute existente în relație.

**Exemplul 3.8.** Tuplurile pot fi sortate și după atribute care lipsesc în schema relației rezultat:

```
SELECT Nume, Telefon
      FROM funcționari ORDER BY CodPoștal DESC, Nume ASC;
```

În ultimul exemplu, tuplurile vor fi mai întâi sortate după atributul care nu va fi afișat, *CodPoștal* (în mod descendent), apoi, în cazul egalității valorilor atributului *CodPoștal*, după atributul *Nume* (în ordine crescătoare).

### 3.2.4. Aliasuri de atribut

Orice nume de atribut într-o schemă are o anumită semantică. Când se afișează rezultatul unei interogări, în mod obișnuit, se utilizează numele atributelor selectate ca titlu. În multe cazuri, în contextul interogării, acest nume poate fi criptic sau fără înțeles.

Luând în vedere că cererile la baza de date sunt niște vizuini ale utilizatorilor, poate apărea necesitatea atribuirii altor nume atributelor din schema relației rezultat. Pentru a schimba un nume de atribut, se utilizează un alias. Sintaxa generală a unui alias de atribut este următoarea:

...<expresie> AS <nume atribut> ...

Termenul <expresie> se referă fie la un atribut, fie la operații aritmetice de bază asupra atributelor existente pentru generarea unui nou atribut. Clauza SQL "AS" specifică un nume nou pentru un atribut sau expresie. Numele noi nu pot conține blancuri.

**Exemplul 3.9.** Numele de atribut *Nume*, în relația rezultat, va fi substituit cu numele *Nume\_Funcționar*, iar pentru a afișa salariul anual, însemnând *Salariu\*12*, se utilizează aliasul de atribut *Salariu\_Anual*:

```
SELECT Nume AS Nume_Funcționar,
       Salariu*12 AS Salariu_Anual
  FROM funcționari ORDER BY Nume_Funcționar;
```

Să se observe că aliasul de atribut se specifică după acesta în lista selectată. Numele specificat este folosit ca titlu în rapoartele SQL. Clauza "AS" este utilă în lucrul cu funcțiile și ajută să se identifice mai ușor datele de ieșire.

## 3.3. Interogări cu criterii de selecție

Până aici au fost examinate forme de cereri asupra bazei de date, care extrageau toate tuplurile (poate cu excepția celor repetate) din relația menționată după clauza *FROM*. Mai jos, se vor examina posibilitățile de filtrare a tuplurilor cu scopul de a le extrage numai pe acele care satisfac unele condiții prestabilite.

### 3.3.1. Clauza WHERE

Clauza *WHERE* este utilizată pentru a specifica tuplurile selectate dintr-o relație sau mai multe indicate de clauza *FROM*. Ea constă din cuvântul-cheie *WHERE*, urmat de o condiție de selecție numită și *predicat*:

... *WHERE* <*predicat*> ...

Un *predicat* reprezintă o expresie logică care ia valorile *adevărat* sau *fals*. Tuplurile pentru care expresia <*predicat*> ia valoarea *adevărat* vor fi returnate de interogare.

Dar, înainte de a trece la dezvoltarea subiectului, trebuie menționate trei detalii importante. Primul din ele, este că, dacă în expresia *predicat* participă date de tipul secvență de caractere, ele trebuie luate între două apostrofuri. Al doilea, în unele SGBD pentru datele te tip *BIT* (*MEMO* în Microsoft Access) nu se pot defini condiții de căutare. Al treilea, se referă la atributele temporale, de exemplu, *DATE*. Acest tip de date diferă de la un sistem la altul. Aici, pentru tipul *DATE*, se va folosi formatul *aaaa-ll-zz*, unde *aaaa* reprezintă anul, *ll* luna, *zz* și ziua. Data, de asemenea, trebuie să fie luată între două apostrofuri. De exemplu, dacă ne referim la ziua 3 a lunii septembrie a anului 2013, se vor utiliza formele: '2013-09-03'.

SQL oferă un set bogat de condiții de selecție care permit specificarea unei mari varietăți de interogări, în mod eficient și natural. Există șase condiții de selecție de bază:

1. Predicatelor care apelează la operatorii  $=$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$  și  $\neq$  pentru compararea valorilor a două expresii SQL pentru fiecare tuplu în parte. Expresiile care folosesc acești operatori pun o întrebare despre două valori pe care le compară.
2. Predicatelor care utilizează operatorii logici *AND*, *OR* și *NOT* pentru combinarea condițiilor de selecție simple având drept scop obținerea condițiilor mai complexe.
3. Predicatelor utilizate pentru testarea valorilor necunoscute (*NULL*).
4. Predicatelor care permit testarea dacă valoarea unei expresii aparține unei multimi de valori (operatorul *IN*).

5. Predicale care folosesc operatorul *BETWEEN* pentru a verifica dacă valoarea unei expresii este cuprinsă între două valori specificate.
6. Predicale utilizate pentru testarea “potrivirii” unei secvențe de caractere cu un model (operatorul *LIKE*).

### 3.3.2. Operatorii de comparație

Ca și în majoritatea limbajelor de interogări, criteriile de selecție sunt construite folosind operatorii de comparație. Operatorii de comparație permit verificarea a două expresii returnând valoarea adevărat sau fals. Aceștia pot fi utilizati pentru compararea expresiilor aritmetice, secvențe de caractere, datelor temporale. Operatorii de comparare adoptați de ISO sunt:  $=$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ,  $\neq$ .

**Exemplul 3.10.** Interogarea extrage toate datele despre funcționari care ridică un salariu lunar mai mare de 2100:

```
SELECT * FROM funcționari
WHERE Salariu > 2100.00;
```

**Exemplul 3.11.** Se afișează numele, prenumele și data de naștere ale funcționarilor angajați în serviciu nu mai devreme de 1 ianuarie 2010:

```
SELECT Nume, Prenume, Data_Naștere
FROM funcționari
WHERE Data_Angajare >= '2010-01-01';
```

**Exemplul 3.12.** Se afișează numele și prenumele funcționarilor a căror țară de origine este Republica Moldova:

```
SELECT Nume, Prenume
FROM funcționari
WHERE Țara = 'Republica Moldova';
```

### 3.3.3. Operatorii logici

Toate clauzele *WHERE* prezентate anterior filtrează datele, utilizând un criteriu simplu. Pentru un grad superior al filtrării, standardul SQL2 susține operatorii logici *AND*, *OR* și *NOT*. Operatorii *AND*, *OR* se

utilizează pentru a combina mai multe predicate. Operatorul *NOT* se plasează înaintea unui predicat pentru a-i inversa sensul.

### 3.3.3.1. Operatorul AND

Operatorul *AND* se folosește pentru adăugarea, în clauza *WHERE*, a condițiilor care trebuie să fie concomitent satisfăcute de datele selectate. Următorul exemplu demonstrează acest lucru:

**Exemplul 3.13.** Instrucțiunea regăsește numele tuturor funcționarilor departamentului 'Software' vârsta cărora a depășit 25 de ani. Clauza *WHERE*, din această instrucțiune *SELECT*, este alcătuită din două condiții, iar cuvântul-cheie *AND* este folosit pentru combinarea condițiilor specificate. Predicatul clauzei *WHERE* este adevărat, dacă ambele condiții sunt satisfăcute.

```
SELECT Nume FROM funcționari
WHERE Departament = 'Software' AND Vârstă > 25;
```

### 3.3.3.2. Operatorul OR

Operatorul *OR* este, într-un sens, opusul operatorului *AND*. El face ca instrucțiunea *SELECT* să returneze tuplurile ce corespund oricareia dintre condiții. De fapt, cele mai multe SGBD-uri nu evaluează a doua condiție legată cu *OR* din *WHERE*, dacă prima condiție a fost deja îndeplinită. Dacă prima condiție din îmbinarea *OR* este satisfăcută, tuplul oricum va fi regăsit, indiferent care ar fi a doua condiție. Astfel, cuvântul-cheie *OR* este folosit într-o clauză *WHERE* pentru a specifica regăsirea oricărora tupluri ce corespund indiferent căreia dintre condițiile specificate.

**Exemplul 3.14.** Instrucțiunea *SELECT* regăsește numele și prenumele funcționarilor, care își desfășoară activitatea în cadrul departamentelor „Software” sau „Hardware”. Operatorul *OR* cere sistemului să extragă tuplurile care satisfac oricare dintre condiții, nu neapărat amândouă. Dacă ar fi fost folosit operatorul *AND* în locul *OR*, nu ar fi fost returnată niciodată.

```
SELECT Nume, Prenume FROM funcționari
WHERE Departament = 'Software'
      OR Departament = 'Hardware';
```

### 3.3.3.3. Operatorul NOT

Operatorul *NOT* în clauza *WHERE* are o singură funcție – negarea oricărei condiții care îl urmează. Prin urmare, acesta niciodată nu este utilizat singur, ci în asociere cu alt predicat care este plasat după *NOT*. Exemplul 3.15 demonstrează folosirea operatorului *NOT*:

**Exemplul 3.15.** Cererea extrage numele funcționarilor, care nu sunt celibatari:

```
SELECT Nume AS Funcționar_căsătorit
FROM funcționari
WHERE NOT(Stare = 'Celibatar');
```

Exemplul anterior poate fi obținut, de asemenea, prin folosirea operatorului „*<>*”, aşa cum se prezintă în exemplul următor:

**Exemplul 3.16.** Operatorul *NOT* este util în clauzele mai complexe. Despre cererea care urmează, nu se poate spune că ar exista vreun avantaj real în locul utilizării operatorului *NOT*:

```
SELECT Nume AS Funcționar_căsătorit
FROM funcționari
WHERE Stare <> 'Celibatar');
```

### 3.3.3.4. Ordinea de evaluare a operatorilor logici

Clauzele *WHERE* pot conține oricât de mulți operatori *AND*, *OR* și *NOT*. Combinarea acestor trei tipuri de operatori permite efectuarea filtrărilor sofisticate și complexe. Totuși, combinarea operatorilor *AND*, *OR* și *NOT* ridică unele probleme. Aceste probleme apar datorită faptului că operatorul *AND* este prioritar față de operatorul *OR*.

Ținând cont de aceasta, ori de câte ori sunt scrise clauze *WHERE*, care folosesc simultan operatorii *AND* și *OR*, trebuie utilizate paranteze pentru a grupa operatorii în mod explicit. Nu întotdeauna este trivial să te bazezi pe ordinea de evaluare prestabilită, chiar dacă este exact ceea ce se dorește. Nu există niciun neajuns în folosirea parantezelor și, întotdeauna, este preferabil să se eliminate orice ambiguități.

**Exemplul 3.17.** Cererea

```
SELECT Nume, Prenume  
FROM funcționari  
WHERE Departament = 'Software'  
      OR Departament = 'Hardware'  
      AND Oraș = 'Chișinău';
```

este echivalentă cu cererea

```
SELECT Nume, Prenume  
FROM funcționari  
WHERE Departament = 'Software'  
      OR (Departament = 'Hardware'  
      AND Oraș = 'Chișinău');
```

dar nu produce același rezultat ca interogarea:

```
SELECT Nume, Prenume  
FROM funcționari  
WHERE (Departament = 'Software'  
      OR Departament = 'Hardware')  
      AND Oraș = 'Chișinău';
```

### 3.3.4. Operatorul IS NULL

Cuvântul-cheie *NULL* desemnează o valoare „necunoscută” (sau „inexistentă” - SQL nu face diferență). Valoarea *NULL* nu este tratată ca o constantă ordinată. Modalitățile de utilizare în interogări și testarea valorilor *NULL* sunt specifice.

Cu toate acestea, logica binară, care cunoaște doar două valori „adevărat” și „fals”, e substituită de SQL cu o logică ternară, cu trei valori „adevărat” (*TRUE*), „fals” (*FALSE*) și „necunoscut” (*NULL*). Urmează tabelele de adevară ale acestei logici:

<i>a</i>	<i>NOT a</i>
<i>FALSE</i>	<i>TRUE</i>
<i>NULL</i>	<i>NULL</i>
<i>TRUE</i>	<i>FALSE</i>

<i>a AND b</i>	<i>FALSE</i>	<i>NULL</i>	<i>TRUE</i>
<i>FALSE</i>	<i>FALSE</i>	<i>FALSE</i>	<i>FALSE</i>
<i>NULL</i>	<i>FALSE</i>	<i>NULL</i>	<i>NULL</i>
<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>	<i>TRUE</i>

<i>a OR b</i>	<i>FALSE</i>	<i>NULL</i>	<i>TRUE</i>
<i>FALSE</i>	<i>FALSE</i>	<i>NULL</i>	<i>TRUE</i>
<i>NULL</i>	<i>NULL</i>	<i>NULL</i>	<i>TRUE</i>
<i>TRUE</i>	<i>TRUE</i>	<i>TRUE</i>	<i>TRUE</i>

Figura 3.1. Tabelele de adevăr pentru logica cu trei valori

Cu alte cuvinte, valoarea *NULL* determină apariția unei logici cu trei valori de adevăr în cazul condiției de selecție. Pentru orice tuplu dat, rezultatul evaluării acestei condiții poate fi *TRUE*, *FALSE* sau *NULL*.

Prezența lui *NULL* în SQL2 poate constitui o sursă de probleme. Este important să se înțeleagă că “valoarea nulă”, de fapt, nu este o valoare, ci o absență a valorii și că asupra ei nu se poate aplica nicio operație aritmetică sau de comparație din cele de mai sus. Astfel, *NULL* este un cuvânt-cheie, și nu o constantă. Adică, comparația de tipul *Nume=NULL* este incorrectă.

Pentru testarea prezenței sau absenței valorii pentru un atribut, există un predicat special cu următoarea sintaxă:

...<atribut> *IS [NOT] NULL*...

Din sintaxa operatorului *IS NULL*, reiese că testarea unei valori nule nu poate furniza rezultatul *NULL*. Această expresie poate avea fie valoarea *TRUE*, fie valoarea *FALSE*.

**Exemplul 3.18.** Instrucțiunea afișează datele doar despre funcționarii a căror adresă este cunoscută:

```
SELECT * FROM funcționari
WHERE Adresă IS NOT NULL;
```

### 3.3.5. Funcții cu utilizarea valorii NULL

Dacă, într-o expresie aritmetică, vreun atribut are valoarea *NULL*, atunci expresia este *NULL*. De exemplu, dacă se face împărțirea prin zero, se obține o eroare. Totuși, dacă se împarte un număr prin *NULL*, rezultatul este *NULL*.

#### 3.3.5.1. Funcția COALESCE

O excepție de la această regulă constatăm funcția *COALESCE*. Funcția *COALESCE* returnează prima expresie nenulă din lista de expresii ce constituie parametrii acestei funcții. Sintaxa funcției este următoarea:

...*COALESCE* (<expresie1>, ... <expresieN>)...

unde se returnează <expresie1>, dacă aceasta este nenulă, se returnează <expresie2> în cazul în care prima expresie este nulă, dar a doua e nenulă. Expressia <expresieN> este returnată în cazul în care aceasta este nenulă, dar toate expresiile precedente sunt nule. Toate expresiile trebuie să fie de același tip. Funcția dată permite înlocuirea valorii *NULL* cu o altă valoare.

**Exemplul 3.19.** Instrucțiunea afișează salariul, dacă nu este nul. Dacă salariul este necunoscut, returnează suplimentul la salariu, dar, dacă și acesta este nul, returnează 0:

```
SELECT COALESCE(Salariu, Supliment, 0)  
FROM funcționari;
```

#### 3.3.5.2. Funcția NVL

Funcția *NVL* din Oracle este folosită pentru convertirea unei expresii nule în altă valoare. Această funcție are sintaxa:

...*NVL* (<expresie1>, <expresie2>)...

unde <expresie1> reprezintă valoarea-sursă sau o expresie care poate lua valoarea *NULL*, iar <expresie2> reprezintă valoarea-destinație pentru convertirea valorii *NULL*.

Funcția returnează <expresie1>, dacă aceasta nu este *NULL*, <expresie2> în caz contrar. Funcția dată poate fi utilizată pentru convertirea oricărui tip de date, dar tipul de date returnat este întotdeauna de același tip ca și tipul expresiei <expresie1>.

**Exemplul 3.20.** Instrucțiunea afișează salariul, dacă nu este nul. Dacă salariul este necunoscut, instrucțiunea returnează 0:

```
SELECT COALESCE(Salariu, 0) FROM funcționari;
```

Avantajul funcției *COALESCE*, în raport cu funcția *NVL* din Oracle, constă în faptul că *COALESCE* poate avea mai multe valori alternative. Începând cu Oracle 10g, această funcție este, de asemenea, disponibilă.

### 3.3.5.3. Funcția NULLIF

Din contra, funcția valabilă în SQL2 *NULLIF* (care deja există și în Oracle) compară două expresii. Dacă acestea sunt egale, funcția returnează valoarea *NULL*. Dacă nu sunt egale, funcția returnează prima expresie. Nu este posibilă specificarea literalului *NULL* pentru prima expresie. Astfel, funcția are următoarea sintaxă de utilizare:

```
...NULLIF (<expresie1>, <expresie2>) ...
```

unde *<expresie1>* este valoarea-sursă comparată cu *<expresie2>*, iar *<expresie2>*, la rândul său, este valoarea-sursă comparată cu *<expresie1>* (dacă aceasta nu este egală cu *<expresie1>*, valoarea expresiei *<expresie1>* este returnată).

**Exemplul 3.21.** Instrucțiunea afișează salariul, dacă acesta nu este egal cu -1, și returnează valoare *NULL*, în caz contrar:

```
SELECT COALESCE(Salariu, -1) FROM funcționari;
```

### 3.3.6. Operatorul IN

Operatorul *IN* implementează un predicat de apartenență la o mulțime discretă. Cu alte cuvinte, operatorul este utilizat pentru determinarea faptului dacă valoarea unei expresii este egală cu vreuna dintre valorile unei liste de expresii. Sintaxa operatorului *IN* este următoarea:

```
...<expresie> [NOT] IN (<valoare1>, <valoare2>, ...).
```

Termenul *<expresie>* identifică atributul de evaluat, iar *<valoare1>, <valoare2>, ...* reprezintă lista de expresii cu a căror valoare se compară valoarea expresiei *<expresie>*. Dacă valoarea *<expresie>* este printre cele ale listei *<valoare1>, <valoare2>, ...*, atunci *IN* are

valoarea *adevărat*, în caz contrar – *fals*. Evident, *NOT IN* are semnificația inversă lui *IN*.

Operatorul *IN* se folosește pentru a specifica un domeniu de condiții, oricare dintre ele putând fi îndeplinite. *IN* necesită o listă de valori valide, care să fie separate prin virgule și cuprinse între paranteze. Următorul exemplu demonstrează acest lucru:

**Exemplul 3.22.** Instrucțiunea SELECT regăsește toate datele despre funcționarii care își desfășoară activitatea cel puțin în cadrul unui departament specificat. Operatorul *IN* este urmat de o listă de departamente valide ('Software', 'Hardware', 'Proiectare Sisteme'), despărțite prin virgule și întreaga lista este cuprinsă între paranteze:

```
SELECT * FROM funcționari  
WHERE Departament  
      IN ('Software', 'Hardware', 'Proiectare Sisteme');
```

Următoarea interogare obține exact același rezultat, folosind doar operatorul logic *OR*:

```
SELECT * FROM funcționari  
WHERE Departament = 'Software'  
      OR Departament = 'Hardware'  
      OR Departament = 'Proiectare Sisteme';
```

Deși există și varianta echivalentă, avantajele folosirii operatorului *IN* sunt substanțiale:

- Atunci când se lucrează cu liste lungi de opțiuni valide, sintaxa operatorului *IN* este mai simplă și, deci, mai lizibilă.
- În cazul în care operatorul *IN* este folosit în asociere cu operatorii *AND* și *OR*, ordinea de evaluare este mai simplu de gestionat.
- Aproape întotdeauna, operatorii *IN* se execută mai repede decât listele de operatori *OR*.
- Avantajul cel mai mare constă în faptul că operatorul *IN* poate să conțină o altă instrucțiune *SELECT*, și astfel va permite construirea clauzelor *WHERE* foarte dinamice.

### 3.3.7. Operatorul BETWEEN

Operatorul *BETWEEN* implementează predicatul de apartenență la un interval de valori. Astfel, la recuperarea tuplurilor, acest operator determină dacă valoarea unei expresii este cuprinsă între două valori. Sintaxa operatorului *BETWEEN* diferă de a altor operatori ai clauzei *WHERE*, deoarece necesită două valori - începutul și sfârșitul intervalului:

```
...<expresie> [NOT] BETWEEN <valoare1>
    AND <valoare2>...
```

Semnificația termenelor din sintaxă este următoarea. Termenul *<expresie>* reprezintă expresia ce identifică atributul de evaluat. Expresiile *<valoare1>* și *<valoare2>* sunt evaluate pentru a obține cele două capete ale intervalului și a căror valoare se compară valoarea expresiei *<expresie>*.

Dacă valoarea *<expresie>* se află între valorile *<valoare1>* și *<valoare2>* (inclusiv), atunci operatorul *BETWEEN* are valoarea *adevărat*, în caz contrar - *fals*. Dacă oricare dintre *<expresie>*, *<valoare1>* sau *<valoare2>* are valoarea *NULL*, *BETWEEN* are tot valoarea *NULL*. Evident, dacă condiția *NOT* este prezentă, atunci sunt selectate tuplurile ce nu au valori cuprinse în intervalul respectiv.

**Exemplul 3.23.** Cererea extrage numele și prenumele funcționarilor pentru care codul poștal ia o valoare cuprinsă între 2012 și 2048:

```
SELECT Nume, Prenume FROM funcționari
WHERE CodPoștal BETWEEN 2012 AND 2048;
```

Același rezultat poate fi obținut de următoarea interogare:

```
SELECT Nume, Prenume FROM funcționari
WHERE CodPoștal >= 2012
      AND CodPoștal <= 2048;
```

### 3.3.8. Operatorul LIKE

Nu întotdeauna se cunoaște valoarea exactă pe baza căreia se va efectua căutarea. Instrucțiunea *SELECT* permite extragerea tuplurilor care corespund unui model de caractere cu ajutorul operatorului *LIKE*. Acest operator este folosit în cazul în care se dorește compararea valorii unei

expresii cu un anumit model (format). De exemplu, se cere vizualizarea tuturor funcționarilor al căror nume începe cu litera A.

Sintaxa pentru acest operator este următoarea:

```
...<expresie> [NOT] LIKE '<model>'  
[ESCAPE '<caracter escape>']...
```

Operatorul *LIKE* ia valoarea *adevărat*, dacă valoarea expresiei *<expresie>* este „asemănătoare” (sau se potrivește) cu operandul al doilea - ‘*<model>*’ și respectiv ia valoarea *fals*, în caz contrar. Pot fi determinate secvențele de caractere care nu se potrivesc cu un model, utilizând forma *NOT LIKE*.

Asemănarea constă în faptul că, în compoziția celui de-al doilea operand, pot intra, pe lângă toate celelalte caractere (privite drept constante), și metacaractere, care nu sunt întotdeauna aceleași pentru toate SGBD-urile.

Pentru construirea modelelor de căutare, SQL2 utilizează două metacaractere:

- **Liniuța de subliniere**, „\_”. Simbolul „\_” poate fi folosit drept metacaracter pozițional, ceea ce înseamnă că se potrivește cu orice caracter aflat pe poziția respectivă în secvența de caractere evaluată.
- **Procentul**, „%”. Caracterul „%” poate fi folosit drept metacaracter nepozitional. Cu alte cuvinte, acesta se potrivește cu orice număr de caractere, indiferent de lungime.

Clauza *ESCAPE* se folosește pentru a schimba interpretarea unor metacaractere, trecându-le în categoria celor obișnuite, în cadrul modelului specificat.

SGBD-ul Microsoft Access oferă o caracteristică similară, dar pentru metacaracterul pozițional este folosit semnul de întrebare, „?”, iar pentru metacaracterul nepozitional este folosit asteriscul, „\*”.

**Exemplul 3.24.** Instrucțiunea ce urmează returnează numele și prenumele angajaților din relația *funcționari* al căror nume începe cu “A”:

```
SELECT Nume, Prenume
FROM funcționari WHERE Nume LIKE 'A%';
```

Numele care încep cu "a" nu vor fi returnate.

Metacaracterele pot să apară oriunde în model și mai multe metacaractere pot fi scrise ca o singură secvență de caractere.

**Exemplul 3.25.** Pot fi combinate diferite tipuri de potriviri pe caracter:

```
SELECT Nume, Prenume
FROM funcționari WHERE Nume LIKE '_a%';
```

În anumite cazuri, operatorul *LIKE* poate fi utilizat în locul operatorului *BETWEEN*.

**Exemplul 3.26.** Instrucțiunea afișează numele, prenumele și data angajării tuturor funcționarilor a căror zi de naștere este cuprinsă între ianuarie 1990 și decembrie 1990:

```
SELECT Nume, Prenume, Data_Angajare
FROM funcționari
WHERE Data_Naștere LIKE '1990-__-__';
```

Una dintre problemele legate de potrivirea la un model constă în includerea metacaracterelor în calitate de constante de tip secvență de caractere. Pentru a testa prezența unui caracter procent (%) într-o valoare de atribut, de exemplu, nu poate fi inclus, pur și simplu, caracterul % în model, deoarece SQL îl va trata ca pe un metacaracter. Pentru a rezolva această problemă, se utilizează clauza *ESCAPE* pentru a declara un caracter „escape”. Când un caracter „escape” apare într-un model, caracterul imediat următor este tratat ca o constantă de tip secvență de caractere, și nu ca un metacaracter.

**Exemplul 3.27.** Următoarea instrucțiune *SELECT* afișează numele, prenumele și posturile tuturor funcționarilor al căror post conține semnul subliniere, „\_”:

```
SELECT Nume, Prenume, Post
FROM funcționari
WHERE Post LIKE '%^_%' ESCAPE '^';
```

În calitate de caracter „escape”, a fost nominalizat ‘^’. În acest caz, caracterul imediat următor este tratat ca o constantă și, prin urmare, vor fi afișate posturile în denumirea cărora există semnul sublinierea.

### 3.3.9. Limitarea numărului de tupluri returnate

Din păcate, nicio metodă standardizată nu permite să se limiteze numărul de tupluri returnate de către un *SELECT*. Cu toate acestea, majoritatea SGBD-urilor oferă această facilitate.

Urmează câteva exemple, care prezintă modul în care unele SGBD-uri limitează la 10 numărul de tupluri returnate.

SQL Server:

```
SELECT TOP 10 Nume, Prenume FROM funcționari
```

MySQL și Postgresql:

```
SELECT Nume, Prenume FROM funcționari  
LIMIT 10
```

Oracle:

```
SELECT Nume, Prenume FROM funcționari  
WHERE ROWNUM <= 10
```

Soluția oferită de Oracle nu este întotdeauna ușor gestionabilă, deoarece *ROWNUM* numerotează tuplurile înainte de o eventuală sortare.

## 3.4. Interogări cu agregări și grupări

Toate cererile prezentate până aici pot fi interpretate ca o secvență de operații efectuate tuplu după tuplu. La fel, rezultatul era constituit din valorile provenite din tupluri individuale. Standardul SQL2 permite exprimarea condițiilor de selecție asupra unor grupuri de tupluri, și nu neapărat asupra tuturor tuplurilor relației și constituirea rezultatului în baza agregării valorilor în cadrul fiecărui grup. Cu acest scop, limbajul conține mijloace de partiziune a relației în grupuri, conform unor criterii, mijloace de agregare și mijloace de filtrare a tuplurilor aggregate. Bineînțeles, chiar însăși relația poate fi considerată un grup de tupluri.

### 3.4.1. Funcții de agregare

Funcțiile de agregare constituie una din cele mai importante extensii ale SQL, în comparație cu algebra relațională. În algebra relațională, toate condițiile sunt evaluate pentru un singur tuplu la un moment dat. Adesea, apare necesitatea evaluării unor proprietăți ce nu depind de un tuplu, ci de o mulțime de tupluri. Prin urmare, funcțiile de agregare sunt funcții care iau un set (o mulțime sau multimulțime) de valori la intrare și produc o singură valoare la ieșire.

Norma SQL2 prezintă o mulțime de funcții de agregare principalele fiind: *COUNT*, *AVG*, *MAX*, *MIN*, *SUM*. Intrările pentru funcțiile *SUM* și *AVG* pot fi colecții de numere, în timp ce celelalte funcții pot fi aplicate și asupra datelor nenumerice, precum sunt secvențele de caractere.

#### 3.4.1.1. Funcția COUNT

Funcția *COUNT* calculează numărul de tupluri ai rezultatului interogării. Sintaxa sa este următoarea:

*COUNT* (\* | [DISTINCT | ALL] <expresie>)

unde <expresie> reprezintă o listă de atrbute, o expresie sau o constantă. Funcția *COUNT* poate calcula orice tip de date, inclusiv *DATE*.

Funcția *COUNT* se aplică asupra setului de valori obținute în urma evaluării expresiei <expresie> și nu număără, de regulă, tuplurile pentru care toate atrbutele specificate au valoarea necunoscută, *NULL*. Dacă mulțimea de valori a expresiei evaluate este vidă, atunci funcția *COUNT* dă 0.

În cazul în care se utilizează asteriscul în calitate de argument, funcția *COUNT(\*)* calculează numărul total de tupluri, incluzând și toate acele ale căror componente conțin valoarea *NULL*.

**Exemplul 3.28.** Următoarea interogare găsește numărul funcționarilor angajați în departamentul 'Software':

```
SELECT COUNT(*)
FROM funcționari
WHERE Departament='Software';
```

Nu se recomandă să se pună asteriscul între ghilimele ("\*"). Dacă se poate alege între două variante *COUNT(\*)* și *COUNT(Atribut)*, prima variantă este considerabil mai rapidă decât a doua.

Opțiunea *DISTINCT* returnează numărul valorilor distincte pentru lista de atribute din rezultat. Cuvântul *DISTINCT* nu se consideră parte a argumentului funcției. Este un cuvânt predefinit care indică faptul că, înainte de a aplica funcția asupra multimii de valori a argumentului, trebuie eliminate duplicatele, dacă este cazul. Trebuie menționat că SQL nu permite utilizarea opțiunii *DISTINCT* cu *COUNT(\*)*.

**Exemplul 3.29.** Se calculează numărul valorilor distincte pentru atributul *Salariu* pentru toți angajații din relația *funcționari*:

```
SELECT COUNT (DISTINCT Salariu ) FROM funcționari;
```

În cazul în care lipsește cuvântul *DISTINCT* sau se utilizează opțiunea *ALL*, atunci funcția *COUNT* returnează numărul de tupluri ce conțin valori diferite de *NULL* pentru lista de atribute specificate, fără ca duplicatele să fie eliminate.

**Exemplul 3.30.** Să se găsească numărul de tupluri din relația *funcționari* care au valori diferite de *NULL* pentru atributul *Salariu*.

```
SELECT COUNT (ALL Salariu ) FROM funcționari;
```

Dacă *<expresie>* reprezintă o listă de atribute, funcția *COUNT* numără un tuplu, numai dacă cel puțin un atribut nu este *NULL*. Tuplul, pentru care toate atributele specificate iau valoarea *NULL*, nu se consideră. Atributele din lista de atribute trebuie separate prin caracterul ampersand (&).

**Exemplul 3.31.** Să se găsească numărul de tupluri extrase din relația *funcționari*, care au valori diferite de *NULL* cel puțin pentru unu din atributurile *Nume* și *Prenume*.

```
SELECT COUNT(TelMobil & TelFix) FROM funcționari;
```

### 3.4.1.2. Funcția AVG

Funcția de agregare *AVG* calculează media aritmetică a unei mulțimi de valori numerice (suma valorilor divizată la numărul de valori). Funcția *AVG* ignoră toate valorile necunoscute, *NULL*. Sintaxa este următoarea:

*AVG ([DISTINCT | ALL] <expresie>),*

unde *<expresie>* reprezintă un atribut de tip numeric, o expresie de acest tip sau o constantă. Cuvintele-cheie *DISTINCT* și *ALL* au semnificațiile discutate, deja, la funcția *COUNT*.

**Exemplul 3.32.** Cererea ce urmează calculează salariul mediu al inginerilor din departamentul 'Software'.

```
SELECT AVG (Salariu) AS Salariu_Mediu
FROM funcționari
WHERE Departament= 'Software'
      AND Post= 'inginer';
```

Rezultatul acestei interogări este o relație definită pe un singur atribut, care este constituită dintr-un singur tuplu cu valoarea numerică ce corespunde salariului mediu al inginerilor care activează în departamentul 'Software'. Opțional, atributului-rezultat îl poate fi dat un nume (*Salariu\_Mediu*), utilizând clauza *AS*.

### 3.4.1.3. Funcția SUM

Funcția *SUM* calculează suma unei mulțimi de valori numerice. Sintaxa funcției este:

*SUM ([DISTINCT | ALL] <expresie>),*

unde *<expresie>* reprezintă o expresie ce identifică valori numerice, pentru care se dorește calculul sumei. Valorile *NULL* sunt ignorate.

**Exemplul 3.33.** Cu ajutorul următoarei instrucțiuni, se calculează salariul lunar total al inginerilor care activează în departamentul 'Software'.

```
SELECT SUM (Salariu) AS Salariu_Total
FROM funcționari
WHERE Departament= 'Software' AND Post= 'inginer';
```

### 3.4.1.4. Funcțiile MAX și MIN

Funcțiile *MAX* și *MIN* returnează valoarea maximă, respectiv minimă a unei mulțimi de valori. Sintaxa lor este:

{*MAX* | *MIN*} ([*DISTINCT* | *ALL*] <*expresie*>),

unde <*expresie*> este atributul asupra căruia se realizează calculul, o constantă sau o expresie. Funcțiile *MAX* și *MIN* necesită definirea unei ordini în expresia specificată, fiind aplicabile și asupra secvențelor de caractere și datelor de temporale.

**Exemplul 3.34.** Să se găsească salariul maxim și minim pentru toți angajații din relația funcționari:

```
SELECT MAX(Salariu), MIN(Salariu)  
FROM funcționari;
```

Cu toate că se permite utilizarea opțiunii *DISTINCT* cu *MAX* și *MIN*, rezultatul funcțiilor nu se schimbă. Poate fi utilizat cuvântul-cheie *ALL* în locul *DISTINCT*, pentru a specifica păstrarea duplicatelor, dar, dat fiind faptul că *ALL* se specifică în mod implicit, nu este necesară includerea acestei clauze.

Trebuie menționat că, în clauza *SELECT*, nu se pot utiliza simultan funcții de agregare și nume de atribute, dacă cererea nu conține clauza de grupare. Acest aspect este examinat în secțiunea următoare.

### 3.4.2. Clauza GROUP BY

Există situații în care ar fi de dorit să se aplice funcțiile de agregare nu numai la un singur set de tupluri, dar, de asemenea, la un grup de seturi de tupluri. Acest lucru se realizează în limbajul SQL cu ajutorul clauzei *GROUP BY*. Atributul sau attributele specificate în această clauză se utilizează pentru formarea grupurilor. Tuplurile cu aceeași valoare pentru toate attributele specificate în clauza *GROUP BY* sunt plasate într-un grup.

Clauza *GROUP BY* are ca efect gruparea tuplurilor unei relații pe baza valorilor unui atribut sau grup de attribute. Expresiile din cadrul acestei clauze furnizează criteriul de grupare al tuplurilor unei relații în submulțimi de tupluri, toate având aceeași valoare pentru expresiile respective. Aceste submulțimi sau grupuri de tupluri urmează a fi tratate

unitar în anumite operații, cum ar fi aplicarea funcțiilor de agregare care se calculează nu pe întreaga relație, ci pentru fiecare grup de înregistrări în parte.

Clausa *GROUP BY* este opțională și servește pentru crearea grupurilor menționate anterior. Dacă este specificată, ea se scrie după clauza *WHERE* sau după *FROM* în cazul în care clauza *WHERE* nu există. Sintaxa generală pentru utilizarea acestei clauze este:

... *GROUP BY* <expresie1> [,<expresie2> ...] ...

Expresiile care urmează după *GROUP BY*, deseori, sunt numite expresii de grupare. În clauza *GROUP BY*, pot apărea cel mult 255 de expresii. Tuplurile relației sunt partajate în grupuri, astfel că, în același grup, sunt incluse tuplurile care au valori egale ale expresiilor de grupare. Pot exista grupuri dint-un singur tuplu. Valorile *NULL* se consideră egale în acest sens, deci pot constitui un grup aparte. Cu toate acestea, valorile *NULL* nu se evaluatează de nicio funcție de agregare.

Fie sunt formulate grupurile. În lista după clauza *SELECT*, pot fi specificate doar expresii de două tipuri:

- Expresii care dau același rezultat pentru toate tuplurile din cadrul fiecarui grup. Acestea sunt expresiile de grupare.
- Expresii cu funcții de agregare. Pornind de la un grup de tupluri, acestea produc un singur rezultat.

În consecință, expresiile din lista *SELECT*, pentru fiecare astfel de grup, vor returna un singur tuplu de date. Prin urmare, relația ce rezultă din orice interogare, care conține clauza *GROUP BY*, caracterizează aceste grupuri de tupluri, și nu tupluri individuale.

Expresiile din clauza *SELECT* a unei interogări, care conține opțiunea *GROUP BY*, trebuie să reprezinte o proprietate unică de grup, adică fie o expresie de grupare, fie o funcție de agregare aplicată tuplurilor unui grup, fie o expresie formată pe baza primelor două.

**Exemplul 3.35.** Interogarea ce urmează calculează pentru fiecare departament salariul mediu al angajaților săi:

```
SELECT Departament, AVG (Salariu) AS Salariu_Mediu
FROM funcționari
GROUP BY Departament;
```

Să se rețină că, în lista *SELECT*, este prezentă o expresie de grupare (un atribut) și alta cu o funcție de agregare. Expresia *Departament* este un atribut care returnează aceeași valoare pentru toate tuplurile fiecărui grup și, de fapt, reprezintă o expresie de grupare. Acest lucru permite construirea, în baza fiecărui grup de tupluri, a câte unui tuplu de ieșire.

**Exemplul 3.36.** Următoarea interogare este una greșită, dat fiind faptul că *Salariu* nu este o expresie de grupare și, prin urmare, nu poate să apară în lista clauzei *SELECT*:

```
SELECT Departament, Salariu, AVG (Salariu) AS Salariu_Mediu
FROM funcționari
GROUP BY Departament;
```

Trebuie menționat că orice atribut din relația indicată de clauza *FROM* poate apărea în lista de atribute de grupare a clauzei *GROUP BY*, chiar dacă el nu intră în lista de atribute a comenzi *SELECT*. Clauza *GROUP BY* are sens, dacă comanda *SELECT* include cel puțin o funcție de agregare. În afară de aceasta, orice atribut din lista de atribute a comenzi *SELECT* trebuie să fie prezent în lista de atribute de grupare a clauzei *GROUP BY*, sau să fie în calitate de argument al unei funcții SQL de agregare.

În afară de aceasta, este interzisă utilizarea unui alias de atribut în clauza *GROUP BY*. Tuplurile sunt sortate implicit în ordinea ascendentă a atributelor incluse în lista *GROUP BY*. Este posibilă încălcarea acestei reguli folosind clauza *ORDER BY*. Atributele de tip *BIT* sau *Obiecte OLE* nu pot apărea în calitate de atribute de grupare în clauza *GROUP BY*.

### 3.4.3. Clauza HAVING

Împreună cu *GROUP BY*, se poate folosi clauza *HAVING*. Clauza *HAVING*, la fel ca și în cazul clauzei *WHERE*, restrâng numărul de tupluri returnate de interogare. Într-o interogare care utilizează clauzele

*GROUP BY* și *HAVING*, tuplurile sunt mai întâi grupate, după care sunt selectate doar tuplurile ce îndeplinesc condiția din clauza *HAVING*. Clauza *HAVING* este folosită pentru a restrânge numărul de grupuri returnate de clauza *GROUP BY*.

Clauza *HAVING* permite precizarea unei condiții asupra grupurilor de tupluri returnate de instrucțiunea *SELECT*. Condiția clauzei se referă la proprietățile globale ale unui grup de tupluri, și nu la tupluri individuale. Astfel, tuplurile unui grup sunt tratate împreună, în bloc, conform criteriului specificat de această clauză. Exprimarea acestui criteriu se face cu ajutorul atributelor de grupare și al funcțiilor de agregare aplicate grupurilor.

Clauza *HAVING* este optională în instrucțiunea *SELECT* și servește pentru specificarea grupurilor care vor fi selectate. *HAVING*, asemenea clauzei *WHERE*, este urmat de un predicat. Sintaxa acestei clauze este următoarea:

...*HAVING* <predicat> ...

Există importante deosebiri între clauzele *WHERE* și *HAVING*, cu toate că ambele permit specificarea condițiilor asupra interogărilor. Clauza *WHERE* specifică o condiție, care este aplicabilă unui singur tuplu, în timp ce *HAVING* indică o condiție care se aplică asupra oricărui grup de tupluri construit de clauza *GROUP BY*. Predicatul clauzei *HAVING* se aplică după formarea grupurilor în timp ce predicatul clauzei *WHERE* se aplică înaintea grupării.

**Exemplul 3.37.** Interogarea extrage salariul mediu pentru fiecare departament în care activează mai mult de 20 de persoane:

```
SELECT Departament, AVG(Salariu) AS Salariu_Mediu
FROM funcționari
GROUP BY Departament HAVING COUNT(*)>20;
```

Precum se poate vedea, predicatul clauzei *HAVING* poate include funcții de agregare. Se poate constata, de asemenea, că expresia *AVG(Salariu)* se evaluează fiecărui grup de tupluri ce reprezintă funcționarii unui departament, iar expresia *Departament* este una de grupare.

În această secțiune, instrucțiunea *SELECT* a devenit mai puternică și are următoarea sintaxă generală:

```
SELECT [DISTINCT | ALL]
    {* | [<expresie>[AS <nume nou>]] [...] }
FROM <nume relație>
[WHERE <predicat>]
[GROUP BY <expresie> [HAVING <predicat>] ]
[ORDER BY <expresie1> [ASC|DESC][...]];
```

Clauzele *WHERE*, *GROUP BY*, *HAVING*, *ORDER BY* sunt opționale.

Clauza *WHERE* se utilizează pentru excluderea acelor tupluri care nu se cer grupate. Clauza *HAVING* servește pentru filtrarea tuplurilor deja grupate. Criteriul de filtrare este prezentat de *<predicat>*. Dacă clauză *ORDER BY* este omisă, în prezența clauzei *GROUP BY*, rezultatul este sortat ascendent în funcție de expresia de grupare (atributele incluse în expresie).

Clauza *GROUP BY* întotdeauna precede clauza *HAVING*. Ordinea inversă nu are niciun efect asupra grupurilor de tupluri formate de expresiile de grupare. Toate atributele din lista *SELECT*, care nu sunt utilizate în cadrul funcțiilor de agregare, trebuie să apară în clauză *GROUP BY*. Valorile *NULL* pentru expresiile din *GROUP BY* se grupează și nu se omit. Toate funcțiile de agregare ignoră valorile *NULL*. Excepție face funcția *COUNT* cu argumentul \* care va considera *NULL* ca fiind o valoare distinctă.

Instrucțiunea *SELECT*, în care apar clauzele *WHERE* și *HAVING*, se execută în următoarea ordine:

- Se selectează relația specificată în clauza *FROM*.
- Primul se aplică predicatul clauzei *WHERE*.
- Tuplurile din relație, care satisfac acest predicat, se constituie în grupuri conform expresiei clauzei *GROUP BY*.
- Grupurile care nu satisfac predicatul clauzei *HAVING* se elimină.
- La sfârșit, clauza *SELECT* generează rezultatul de ieșire.

### 3.5. Interogări cu joncțiuni

Până acum, au fost formulate interogări doar asupra unei relații, dar, de cele mai multe ori, este nevoie de date extrase din mai multe relații. Acest tip de interogări se bazează pe utilizarea joncțiunii relațiilor. Joncțiunea relațiilor constă în obținerea, pornind de la cele două relații, a unui relații noi îmbinând tuplurile uneia cu tuplurile celeilalte și concatenând schemele ambelor relații. Ea constă în formarea perechilor de tupluri.

Instrucțiunea *SELECT* permite realizarea acestei joncțiuni, inclusiv două sau mai multe relații în clauza *FROM*. Este momentul să fie extinsă clauza *FROM* din secțiunile anterioare. Se va începe studierea joncțiunilor cu operația, pornind de la care sunt definite alte operații de joncțiune, produsul cartezian.

#### 3.5.1. Produsul Cartezian

Produsul cartezian (sau joncțiunea încrucișată) a două relații asociază fiecare tuplu din prima relație cu toate tuplurile din relația a doua. El conservă toate tuplurile tuturor relațiilor jonctionate, obținând toate combinațiile posibile ale acestora. Desigur, dacă nu există duplicate în relațiile originale, toate tuplurile din rezultat vor fi, de asemenea, unice.

Un produs cartezian are tendința de a genera un număr mare de tupluri, iar rezultatul, de regulă, rar este util. Acesta poate parveni accidental în cazul nespecificării criteriilor de jonctionare, în particular, atunci când se utilizează sintaxa SQL89. Prin urmare, deseori, trebuie să fie prezentă o condiție de valabilitate a joncțiunii, fapt ce transformă produsul cartezian într-un alt tip de joncțiune. Exceptie fac doar cazurile în care este nevoie de combinarea tuturor tuplurilor din toate relațiile. Produsul cartezian poate fi folosit, de asemenea, pentru executarea unor teste în cazul în care este necesară generarea unui număr mare de tupluri pentru simularea unui număr rezonabil de date.

Interogările cu joncțiuni vor fi formulate asupra bazei de date din figura 3.2:

<i>funcționari</i>			<i>departamente</i>	
<i>Nume</i>	<i>Prenume</i>	<i>DeptID</i>	<i>DeptID</i>	<i>Denumire</i>
<i>Bobu</i>	<i>Angela</i>	<i>Dpt01</i>	<i>Dpt01</i>	<i>Software</i>
<i>Mutu</i>	<i>Andrei</i>	<i>Dpt02</i>	<i>Dpt02</i>	<i>Hardware</i>
<i>Matei</i>	<i>Ion</i>	<i>Dpt02</i>		
<i>Paiu</i>	<i>Adrian</i>	<i>Dpt04</i>		
<i>Bobu</i>	<i>Matei</i>	<i>Dpt02</i>	<i>Dpt03</i>	<i>Sisteme operare</i>

Figura 3.2. Relațiile funcționari și departamente

Joncțiunea încrucisată poate fi scrisă în două moduri diferite.

Există o reprezentare cu ajutorul unei clauze *FROM* simplificate (standardul SQL89), unde se indică numele relațiilor separate prin virgulă.

**Exemplul 3.38.** Următoarea instrucțiune simplificată realizează produsul cartezian dintre tabelele *funcționari* și *departamente* și afișează (figura 3.3) toate combinările dintre numele, prenumele funcționarilor și denumirile departamentelor:

```
SELECT Nume, Prenume, Denumire
FROM funcționari, departamente;
```

<i>Nume</i>	<i>Prenume</i>	<i>Denumire</i>
<i>Bobu</i>	<i>Angela</i>	<i>Software</i>
<i>Bobu</i>	<i>Angela</i>	<i>Hardware</i>
<i>Bobu</i>	<i>Angela</i>	<i>Sisteme operare</i>
<i>Mutu</i>	<i>Andrei</i>	<i>Software</i>
<i>Mutu</i>	<i>Andrei</i>	<i>Hardware</i>
<i>Mutu</i>	<i>Andrei</i>	<i>Sisteme operare</i>
<i>Matei</i>	<i>Ion</i>	<i>Software</i>
<i>Matei</i>	<i>Ion</i>	<i>Hardware</i>
<i>Matei</i>	<i>Ion</i>	<i>Sisteme operare</i>
<i>Paiu</i>	<i>Adrian</i>	<i>Software</i>
<i>Paiu</i>	<i>Adrian</i>	<i>Hardware</i>
<i>Paiu</i>	<i>Adrian</i>	<i>Sisteme operare</i>
<i>Bobu</i>	<i>Matei</i>	<i>Software</i>
<i>Bobu</i>	<i>Matei</i>	<i>Hardware</i>
<i>Bobu</i>	<i>Matei</i>	<i>Sisteme operare</i>

Figura 3.3. Rezultatul interogării din exemplul 3.38

Norma SQL2, pentru produsul cartezian a două relații, are o sintaxă specială. Pentru realizarea unui produs cartezian, în clauza *FROM*, se indică numele relațiilor separate de operatorul *CROSS JOIN* care produce o joncțiune încrucișată a tuplurilor.

**Exemplul 3.39.** Se afișează (figura 3.3) toate modalitățile de combinare a tuplurilor relației *funcționari* cu cele ale relației *departamente*, apelând la forma nouă de prezentare a produsului cartezian :

```
SELECT Nume, Prenume, Denumire
FROM funcționari CROSS JOIN departamente;
```

La executarea acestor instrucțiuni, tuplurile din rezultat sunt formate din toate atributele relațiilor *funcționari* și *departamente*. În tupluri, apare orice funcționar asociat cu primul departament, apoi asociat cu al doilea departament și astfel până la combinarea tuturor funcționarilor cu toate departamentele.

### 3.5.2. Joncțiuni interne

Capacitatea de a realiza o joncțiune între două sau mai multe relații reprezintă una dintre cele mai puternice facilități ale sistemului relațional. Legătura dintre tuplurile relațiilor se realizează prin existența unor atrbute caracterizate prin domenii compatibile.

În cazul în care se specifică mai multe relații în clauza *FROM*, se obține produsul cartezian al relațiilor. Acest produs cartezian prezintă, în general, puțin interes. În afară de produsul cartezian, în calitate de instrument de joncțiune a relațiilor, permise de versiunile anterioare, SQL oferă și diverse mecanisme noi pentru joncțiunea relațiilor, incluzând mai multe forme de joncțiuni interne și câteva forme de joncțiuni externe.

Joncțiunile interne, în mare măsură, corespund theta-joncțiunilor din algebra relațională. Joncțiunile interne returnează tuplurile din relațiile asociate, care satisfac criteriile de asociere prin compararea valorilor atributelor respective din cele două relații. Tuplurile care nu satisfac criteriile sunt eliminate.

Există mai multe tipuri și variante de joncțiune internă a relațiilor: joncțiuni interne de egalitate, joncțiuni interne de inegalitate, joncțiunea naturală, autojoncțiunea și semijoncțiunea.

### 3.5.2.1. Joncțiuni interne de egalitate

În *SQL*, operația de joncțiune combină datele din două relații, prin formarea de perechi de tupluri asociate din acestea. Perechile de tupluri care formează relația compusă sunt cele în care anumite atributăe din cele două relații satisfac o condiție de comparare.

Operația de tip joncțiune internă de egalitate corespunde situației în care valorile atributelor ce apar în condiția de compunere trebuie să fie egale. O astfel de joncțiune se mai numește, simplu, *echi-joncțiune*. Trebuie menționat că atributele comparate apar în relația rezultat indiferent de faptul dacă au același nume sau nu.

Forma „veche”, *SQL89*, descrie echi-joncțiunea ca selecția (în clauza *WHERE*) unui produs cartezian (în clauza *FROM*). Astfel, în clauza *FROM*, care combină tuplurile din relațiile referite, se includ mai mult decât o denumire de relație, utilizând virgula ca separator, iar în clauza *WHERE* sunt specificate atributele respective care au valori egale. Când se joncționează două relații, trebuie creată o legătură de egalitate în clauza *WHERE* prin cel puțin un atribut dintr-o relație și un atribut din altă relație.

**Exemplul 3.40.** Instrucțiunea următoare, care implică relațiile *funcționari* și *departamente*, este o echi-joncțiune, în care valorile atributului *DepartamentID* din ambele relații sunt egale (figura 3.4). Operatorul de comparație folosit este “=”:

```
SELECT Nume, Prenume, Denumire
  FROM funcționari, departamente
 WHERE funcționari.DeptID = departamente.DeptID;
```

Se poate observa că acum fiecare funcționar are listat numele departamentului lui. Tuplurile din *funcționari* sunt combinate cu tuplurile din *departamente* și sunt întoarse doar tuplurile pentru care valorile *funcționari.DeptID* și *departamente.DeptID* sunt egale.

Nume	Prenume	Denumire
Bobu	Angela	Software
Mutu	Andrei	Hardware
Matei	Ion	Hardware
Bobu	Matei	Hardware

Figura 3.4. Rezultatul interogării din exemplul 3.40

Pentru a evita ambiguitățile în tratarea interogărilor cu joncțiuni, este necesară introducerea elementelor de calificare pentru attribute, cum ar fi prefixarea atributelor cu nume de relații. În cazul concret, atât relația *funcționari*, cât și relația *departamente* conțin atributul *DeptID*.

Celelalte attribute din interogare pot fi folosite fără vreun element de calificare de tip nume de relație, deoarece apar într-o singură relație și, ca atare, sunt lipsite de ambiguitate. Totuși, în momentul scrierii unei expresii *SELECT*, care conține o condiție de joncțiune, este indicat ca numele atributelor să fie precedate de numele relației de care aparțin; în acest fel, este mărită claritatea codului SQL și se îmbunătățește accesul la baza de date.

Acest mod de prezentare a joncțiunilor este frecvent utilizat, chiar de SGBD-urile care acceptă sintaxa mai nouă, SQL2. Forma nouă descrie joncțiunea ca o operație completă, numai în clauza *FROM*. Aceasta este forma privilegiată, datorită manierei clare și centralizate de definire a căii de acces la date, apelând la diferite relații:

...*FROM <relație1> [INNER] JOIN <relație2>*  
*ON <condiții echi-joncțiune> ...*

În opozиie cu joncțiunile externe, care vor fi examinate mai jos, la această joncțiune, se poate adăuga cuvântul-cheie *INNER*. Pentru realizarea unei echi-joncțiuni, în clauza *FROM*, se indică numele relațiilor separate de operatorul *INNER JOIN*. Echi-joncțiunea compune o relație în care fiecare pereche de tupluri din relațiile originale, care satisfac legătura (condițiile de egalitate), formează un singur tuplu.

Utilizarea clauzei *ON* permite specificarea condițiilor de joncțiune separat de condițiile de căutare sau filtrare din clauza *WHERE*. Clauza *ON* specifică attributele participante la joncțiune.

**Exemplul 3.41.** În următoarea interogare, atributele *DeptID* ale relațiilor *funcționari* și *departamente* sunt jonctionate, folosind clauza *ON*. Dacă un cod de departament din relația *funcționari* este același cu identifierul unui departament din relația *departamente*, tuplul este returnat. Rezultatul este același ca al interogării din exemplul 3.40:

```
SELECT Nume, Prenume, Denumire
FROM funcționari INNER JOIN departamente
    ON funcționari.DeptID = departamente.DeptID;
```

Se constată că această joncțiune corespunde operației theta-joncțiunea a modelului relațional. Clauza *INNER* poate fi omisă.

**Exemplul 3.42.** Interogarea din exemplul 3.41 poate fi formulată și fără utilizarea explicită a clauzei *INNER*:

```
SELECT Nume, Prenume, Denumire
FROM funcționari JOIN departamente
    ON funcționari.DeptID = departamente.DeptID;
```

Clauza *FROM* indică faptul că, din produsul cartezian al relațiilor *funcționari* și *departamente*, să se păstreze doar acele elemente pentru care numărul departamentului provenit din relația *departamente* este același cu numărul departamentului provenit din relația *funcționari*. Prin urmare, se va obține o echijoncțiune între relațiile *funcționari* și *departamente* în funcție de numărul departamentelor.

În joncțiunile examineate, numele atributelor de jonctionare puteau fi diferite. Pentru a ridica ambiguitățile asupra numelor de atrbute, în cazul când atrbutele au același nume în ambele relații, ele se prefixează cu numele relațiilor de origine.

### 3.5.2.2. Clauza **USING**

Folosirea clauzei *USING* permite specificarea atrbutelor folosite pentru realizarea unei joncțiuni între două relații. Numele atrbutelor implicate în joncțiune trebuie să fie aceleași și să aibă același tip de date. Folosirea clauzei *USING* este justificată în cazul în care, în cele două relații, există mai multe atrbute cu același nume.

*JOIN* și *USING* sunt specificate numai în clauza *FROM*. După *USING* urmează lista de atribute de joncțiune luate în paranteze:

...*FROM* <relație1> *JOIN* <relație2>  
          *USING* <listă atributelor> ...

**Exemplul 3.43.** Interogarea din exemplul 3.42, care extrage numele, prenumele funcționarilor și denumirile departamentelor în care aceștia activează, poate fi rescrisă în modul următor:

```
SELECT Nume, Prenume, Denumire  
FROM funcționari JOIN departamente  
          USING (DeptID);
```

Clauza *USING* poate fi utilizată pentru a specifica doar acele atribute care ar trebui să fie folosite pentru o echi-joncțiune. Atributul la care se face referire în clauza *USING* nu trebuie să aibă vreun element de calificare oriunde în instrucțiunea SQL.

**Exemplul 3.44.** Instrucțiunea următoare nu este una validă, deoarece atributul *DeptID* este prefixat în clauza *WHERE*:

```
SELECT Nume, Prenume, Denumire  
FROM funcționari JOIN departamente  
          USING (DeptID)  
    WHERE departamente.DeptID LIKE '%02%';
```

### 3.5.2.3. Joncțiuni naturale

În cazul în care are loc o echi-joncțiune (adică atunci când condiția de joncțiune este o egalitate de valori între atributele de joncțiune ale primei relații și atributele de joncțiune ale relației a doua), deseori, este inutilă prezența dublă a atributelor de joncțiune, deoarece valorile sunt egale. Este preferabilă joncțiunea naturală care păstrează un singur exemplar de atribut de joncțiune din cele două relații.

Două relații pot fi jonctionate în mod automat, în baza atributelor din aceste relații, care au aceleași tip de date și nume. Acest lucru poate fi realizat, utilizând clauza *NATURAL JOIN*, propusă de norma SQL2. În clauza *FROM*, se indică numele relațiilor separate de operatorul *NATURAL JOIN* care produce o joncțiune naturală a tuplurilor.

În cazul în care coloanele au același nume, dar diferite tipuri de date, *NATURAL JOIN* provoacă o eroare de sintaxă.

**Exemplul 3.45.** Interogarea jonctionează relația *funcționari* cu relația *departamente* în baza atributului *DeptID*, care este singurul atribut, cu același nume, prezent în ambele relații. În cazul dat, rezultatul final este același ca și al interogării din exemplul 3.41:

```
SELECT Nume, Prenume, Denumire
FROM funcționari NATURAL JOIN departamente;
```

Dacă și alte atrbute comune ar fi fost prezente, ele tot aveau să fie incluse în joncțiune. Trebuie menționat că nu este nevoie, în acest exemplu, să se indice atrbutele de joncțiune, deoarece clauza *NATURAL JOIN* jonctionează relațiile pe toate atrbutele care au același nume în cele două relații.

Odată cu joncțiunea naturală, într-o clauză *WHERE*, pot fi implementate restricții suplimentare.

**Exemplul 3.46.** Interogarea returnează (figura 3.5) doar tuplurile cu numele, prenumele funcționarilor și denumirile departamentelor al căror identificator conține **șirul '02'**:

```
SELECT Nume, Prenume, Denumire
FROM funcționari NATURAL JOIN departamente
WHERE DeptID LIKE '%02%';
```

Nume	Prenume	Denumire
Mutu	Andrei	Hardware
Matei	Ion	Hardware
Bobu	Matei	Hardware

Figura 3.5. Rezultatul interogării din exemplul 3.46

În cazul utilizării joncțiunii naturale, este interzisă prefixarea cu numele relației a atrbutelor de jonctionare.

**Exemplul 3.47.** Interogarea următoare generează o eroare:

```
SELECT Nume, Prenume, Denumire, departamente.DeptID
FROM funcționari NATURAL JOIN departamente;
```

Este necesar să se scrie:

```
SELECT Nume, Prenume, Denumire, DeptID
FROM funcționari NATURAL JOIN departamente;
```

### 3.5.2.4. O problemă a joncțiunii naturale

Joncțiunea naturală, de obicei, se realizează între cheia externă a unei relații cu cheia primară a celeilalte relații, pe atributele acestor chei care au același nume. Dar dacă alte două atribută au, de asemenea, același nume? Atunci există asociere și pe aceste două atribută și, deci, joncțiunea naturală o va lua în considerație. De exemplu, dacă există în relația funcționari atributul *Nume* (numele funcționarului) și un atribut *Nume* (denumirea departamentului) în relația departamente, joncțiunea naturală duce la o condiție de joncțiune de tipul:

```
... WHERE funcționari.DeptID = departamente.DeptID
AND funcționari.Nume = departamente.Nume ...
```

Problema apare, deoarece este o considerație convențională, și nu a modelului relațional. În realitate, joncțiunea naturală este prost concepută: ea ar trebui să se realizeze, în mod automat, între cheia externă și cheia primară conform constrângerii referențiale.

Pentru a obține o joncțiune naturală numai pe o parte din atributele care au același nume, este necesar să utilizeze clauza *JOIN...USING* (dacă sunt mai multe atribută de joncțiune, sunt separate prin virgulă).

**Exemplul 3.48.** Interogarea de mai jos este echivalentă cu cea din exemplul 3.45:

```
SELECT Nume, Prenume, Denumire
FROM funcționari JOIN departamente
USING (DeptID);
```

### 3.5.2.5. Aliasuri de relații sau variabile-tuplu

Este posibil să se utilizeze un *alias* pentru relațiile specificate în clauza *FROM*. Similar *alias-urilor* pentru atribută, un *alias* pentru relație reprezintă un nume temporar dat relației respective, valabil doar pe perioada execuției cererii.

*Alias-urile* pot fi utilizate oriunde, ca o notație mai scurtă, în locul denumirii relației. Ele pot avea lungimea de maxim 30 de caractere, dar

este recomandat să fie scurte și sugestive. Dacă este atribuit un *alias* unei relații din clauza *FROM*, atunci el trebuie să înlocuiască aparițiile numelui relației în instrucțiunea *SELECT*.

Un *alias* poate fi utilizat pentru a califica denumirea unui atribut. Calificarea unui atribut cu numele sau *alias*-ul relației se poate face opțional, pentru claritate și pentru îmbunătățirea timpului de acces la baza de date, sau obligatoriu, ori de câte ori există o ambiguitate privind sursa atributului. Ambiguitatea constă, de obicei, în existența unor attribute cu același nume în mai mult de două relații.

Însă, uneori, fără crearea aliasurilor este greu, dacă nu imposibil să se creeze diverse cereri la baza de date cu implicarea juncțiunilor. Aliasul joacă rolul nu numai de un nume de alternativă dat relației, dar și rolul de variabilă-tuplu.

Aliasurile de relație sau variabilele-tuplu se definesc în clauza *FROM* prin utilizarea clauzei *AS*. Prin definiție, o variabilă-tuplu este o variabilă care reprezintă un tuplu de o anumită lungime fixă. O variabilă-tuplu, în SQL trebuie să fie asociată cu o relație concretă.

**Exemplul 3.49.** Pentru a determina denumirile departamentelor în care activează fiecare funcționar, se compară valoarea atributului *DeptID* din tabelul *funcționari* cu valorile atributului *DeptID* din tabelul *departamente*. Legătura dintre relațiile *funcționari* și *departamente*, respectiv, se face prin intermediul aliasurilor *f* și *d*, care joacă rolul de variabile-tuplu, parcurgând tuplurile relațiilor asociate. Rezultatul acestei interogări este același ca și rezultatul interogării din exemplul 3.41:

```
SELECT Nume, Prenume, Denumire
  FROM funcționari AS f INNER JOIN departamente AS d
    ON f.DeptID = d.DeptID;
```

La scrierea notațiilor de forma <relație>.<atribut>, numele de relație este, de fapt, o variabilă-tuplu definită implicit. Variabilele-tuplu sunt foarte utile pentru compararea a două tupluri din aceeași relație. Să se remарce că, în cazuri de acest fel, se poate referi o relație de mai multe ori, într-un mod similar operatorului de redenumire din algebra relațională. Evident, juncțiunea din exemplul 3.49 putea fi realizată și fără definirea aliasurilor.

Dar nu întotdeauna acest lucru este posibil. Este, uneori, necesară compararea tuplurilor relațiilor.

SQL permite utilizarea notăției  $(v_1, v_2, \dots, v_n)$  pentru desemnarea unui tuplu de aritatea  $n$  care conține valorile  $v_1, v_2, \dots, v_n$ . Asupra tuplurilor pot fi aplicați operatorii de comparație, iar ordinea se definește în mod lexicografic. De exemplu  $(v_1, v_2) \leq (w_1, w_2)$  este adevărat dacă  $(v_1 < w_1)$  sau în cazul în care se constată că  $(v_1 = w_1) \& (v_2 \leq w_2)$ . În mod similar, două tupluri sunt egale, dacă egale sunt toate componentele respective ale tuplului.

În secțiunile ce urmează, vor fi prezentate exemple de joncțiuni care nu pot fi realizate fără definirea variabilelor-tuplu.

### 3.5.2.6. Joncțiunea unei relații cu ea însăși

Un caz special al joncțiunilor îl reprezintă joncțiunea unei relații cu ea însăși sau autojoncțiunea.

În practică, deseori, poate fi utilă îmbinarea datelor care vin dintr-un tuplu al unei relații cu datele care vin din alt tuplu al aceleiași relații. În acest caz, trebuie să se redenumească cel puțin una din cele două relații, atribuindu-i-se un alias cu posibilitatea de a prefixa fără ambiguitate fiecare nume de atribut.

Sintaxa acestei operații este similară cu joncțiunea a două relații diferite. Anume, în cazul autojoncțiunilor, se poate observa puterea aliasurilor în calitate de variabile-tuplu. Când este introdus un alias, se declară o variabilă-tuplu care parcurge conținutul relației pentru care se introduce aliasul. Când o relație apare doar o singură dată în interogare, nu este nicio diferență între a interpreta aliasul ca pseudonim sau ca o variabilă-tuplu. Când relația apare de mai multe ori, este esențial să privim aliasul ca pe o nouă variabilă.

**Exemplul 3.50.** Fie că din ansamblul de atrbute pe care este construită relația *funcționari* fac parte în afară de identificatorul funcționarului (*FncțID*) și identificatorul managerului (*MngID*). În figura 3.6, este prezentat răspunsul la instrucțiunea care

jonctionează relația *funcționari* cu ea însăși pentru a extrage numele și prenumele funcționarilor al căror nume coincide cu prenumele altui funcționar:

```
SELECT DISTINCT f1.Nume, f1.Prenume
FROM funcționari AS f1 INNER JOIN funcționari AS f2
    ON (f1.Nume = f2.Prenume);
```

Clauza *ON* este utilizată pentru compararea atributelor cu nume diferite din aceeași relație.

<i>f1.Nume</i>	<i>f1.Prenume</i>
Matei	Ion

Figura 3.6. Rezultatul interogării din exemplul 3.50

Pot fi aplicate condiții adiționale la jonctiune. Pentru adăugarea condițiilor la clauza *ON*, se folosește operatorul *AND* sau, pentru alăturarea unor condiții adiționale, poate fi utilizată clauza *WHERE*. Condițiile *WHERE* trebuie să urmeze după clauza *ON*.

**Exemplul 3.51.** Varianta interogării din exemplul precedent poate fi transcrisă în norma SQL 1989 [ISO89] astfel:

```
SELECT DISTINCT f1.Nume, f1.Prenume
FROM funcționari AS f1, funcționari AS f2
WHERE f1.Nume = f2.Prenume;
```

În instrucțiune, aliasurile *f* și *m* parcă ar fi două variabile distincte care parcurg independent aceeași relație *funcționari* și permit constituirea tuturor perechilor de tupluri asupra căror se aplică constrângerea din clauza *WHERE*.

### 3.5.2.7. Jonctiuni interne de inegalitate

Criteriile de jonctiune ale jonctiunii interne de inegalitate nu se exprimă doar prin condiția de egalitate dar și condițiile în care participă și operatorii de comparație *<*, *<=*, *>*, *>=*, *≠* sau *BETWEEN* și *IN*.

**Exemplul 3.52.** Folosind norma SQL1989 [ISO89], și relația *funcționari* din figura 3.2, să se găsească toți funcționarii ce au același nume, dar prenume diferite:

```
SELECT f1.Nume, f1.Prenume
FROM funcționari AS f1, funcționari AS f2
WHERE f1.Nume = f2.Nume AND
      f1.Prenume <> f2.Prenume;
```

În formatul standardului SQL2, interogarea de mai sus arată precum urmează:

```
SELECT f1.Nume, f1.Prenume
FROM funcționari AS f1 INNER JOIN funcționari AS f2
  ON f1.Nume = f2.Nume
  AND f1.Prenume <> f2.Prenume;
```

Răspunsul la aceste interogări este prezentat în figura 3.7.

<i>f1.Nume</i>	<i>f1.Prenume</i>
<i>Bobu</i>	<i>Angela</i>
<i>Bobu</i>	<i>Matei</i>

Figura 3.7. Rezultatul interogării din exemplul 3.52

Îmbinarea dintre relația *funcționari* cu ea însăși din exemplul 3.52 este o non-echi-joncțiune, în sensul că cel puțin un atribut (*Prenume*) nu coincide în relațiile jonctionate. Legătura dintre cele două relații (realmente una și aceeași) pentru fiecare tuplu extras de interogare satisface constrângerea *f1.Prenume <> f2.Prenume*.

Să se observe importanța utilizării aliasurilor în interogările prezentate. În primul rând, se evită necesitatea scrierii întregului nume al relației ori de câte ori este cerut acest lucru. În al doilea rând, se poate face referire de mai multe ori la același tabel. și într-al treilea rând, introducerea unui alias are semnificația declarării unei variabile-tuplu, care parcurge conținutul relației a cărui alias este.

### 3.5.3. Joncțiuni externe

Mai sus, au fost prezentate exemple de joncțiune a relațiilor, utilizând clauzele *INNER JOIN*, *NATURAL JOIN* sau *USING* care, de fapt, reprezentau joncțiuni interne.

Joncțiunile interne returnează tuplurile din ambele relații care satisfac condițiile de joncțiune. Dacă sunt tupluri în prima relație, care nu se

potrivesc cu niciun tuplu din a doua relație, aceste tupluri nu vor fi prezente în rezultat. Același lucru poate fi spus și despre tuplurile din relația a doua, care nu se jonctionează cu nici un tuplu din prima relație.

### 3.5.3.1. Returnarea tuplurilor nejonctionabile

Precum s-a menționat, toate interogările de jonctionare prezentate până aici produc, în calitate de rezultat, o submulțime de tupluri ale produsului cartezian care și satisfac o condiție de jonctionare. Se spune că toate aceste jonctionuri sunt interne.

**Exemplul 3.53.** Interogarea se extrage numele, prenumele funcționarilor și denumirile departamentelor în care aceștia activează (figura 3.8):

```
SELECT Nume, Prenume, Denumire
FROM funcționari AS f INNER JOIN departamente AS d
ON f.DeptID = d.DeptID;
```

Nume	Prenume	Denumire
Bobu	Angela	Software
Mutu	Andrei	Hardware
Matei	Ion	Hardware
Bobu	Matei-	Hardware

Figura 3.8. Rezultatul interogării din exemplul 3.53

În urma acestei interogări, în particular, un departament (*Dept03*) care nu are niciun angajat nu va apărea în listă, deoarece, în produsul cartezian al relațiilor *funcționari* și *departamente*, nu se va constata un tuplu cu egalitate pe atributul *DeptID* pentru acest departament.

Cu alte cuvinte, nu toate tuplurile din relațiile respective vor fi jonctionabile conform criteriului de jonctionare. Uneori, se poate dori o listă de diverse departamente cu funcționarii lor, dacă unele îi au, fără a omite departamentele fără angajați.

Pentru a returna tuplurile care nu sunt jonctionabile, se utilizează jonctionarea externă. Există trei tipuri de jonctionuri externe. Sintaxa jonctionii externe este:

---

...*FROM <relația1> {LEFT|RIGHT|FULL}*  
*[OUTER] JOIN <relația2>*  
*ON <condiții joncțiune>...*

Spre deosebire de joncțiunile interne, unde opțional se folosea cuvântul-cheie *INNER*, joncțiunea externă utilizează opțional cuvântul-cheie *OUTER*. Pentru realizarea unei joncțiuni externe, în clauza *FROM*, se indică numele relațiilor separate de operatorul *OUTER JOIN*, cu folosirea uneia din variantele *LEFT*, *RIGHT* sau *FULL*. Utilizarea clauzei *ON* permite specificarea condițiilor de joncțiune separat de condițiile de căutare sau filtrare din clauza *WHERE*.

- *LEFT OUTER JOIN* returnează toate tuplurile din prima relație, chiar dacă nu satisfac condițiile de joncțiune cu a doua relație. Adică, dacă există așa tupluri în prima relație, care nu satisfac condițiile pe atributele de joncțiune din relația a doua, acestea sunt, de asemenea, afișate. Atributele relației din dreapta se completează cu valori nule.
- *RIGHT OUTER JOIN* returnează toate tuplurile din a doua relație, chiar dacă nu satisfac condițiile pe atributele de joncțiune cu prima relație. Adică, dacă există asemenea tupluri în prima relație, care nu satisfac condițiile pe atributele de joncțiune din relația a doua, acestea sunt, de asemenea, afișate. Atributele relației din stânga se completează cu valori nule.
- *FULL OUTER JOIN* reprezintă uniunea dintre rezultatele *LEFT* și *RIGHT JOIN*.

Trebuie menționat faptul că o condiție care specifică un *OUTER JOIN* nu poate utiliza operatorul *IN* și nu poate fi legată de altă condiție prin operatorul *OR*.

### 3.5.3.2. Joncțiunea externă de stânga

O joncțiune externă de stânga (*LEFT [OUTER] JOIN*) produce un rezultat cu toate tuplurile primei relații specificate, completate cu atributele din tuplurile corespunzătoare ale celeilalte relații sau, dacă nu există corespondență, cu valori *NULL*.

Interogarea de mai jos returnează toate tuplurile din relația *funcționari*, și reprezintă relația din stânga, chiar dacă, în relația *departamente*, nu sunt tupluri potrivite pentru a fi jonctionate cu unele tupluri din *funcționari*.

**Exemplul 3.54.** Interogarea extrage numele, prenumele funcționarilor și denumirile departamentelor în care aceștia activează. Rezultatul va conține și funcționarii pentru care nu există departamentul respectiv și va fi sortat după numele și prenumele funcționarilor în ordine descrescătoare (figura 3.9):

```
SELECT Nume, Denumire
FROM funcționari AS f LEFT OUTER JOIN departamente AS d
    ON (f.DeptID = d.DeptID)
ORDER BY Nume DESC, Prenume DESC;
```

Nume	Prenume	Denumire
Paiu	Adrian	NULL
Mutu	Andrei	Hardware
Matei	Ion	Hardware
Bobu	Matei	Hardware
Bobu	Angela	Software

Figura 3.9. Rezultatul interogării din exemplul 3.54

Astfel, relația din stânga, *funcționari*, este relația primară, adică relația pentru care se dorește returnarea tuturor datelor. Relația din dreapta este relația secundară, adică datele din ea sunt necesare doar în măsura în care se potrivesc condițiilor de joncțiune. Astfel, se explică și denumirea de joncțiune de la stânga spre exterior.

### 3.5.3.3. Joncțiunea externă de dreapta

Simetric, o joncțiune externă de dreapta (*RIGHT [OUTER] JOIN*) produce un rezultat cu toate tuplurile relației a doua, completate cu atributele din tuplurile corespunzătoare ale celeilalte relații sau, dacă nu, cu valori *NULL*.

Interogarea din exemplul 3.55 preia toate tuplurile din relația *departamente*, care reprezintă relația din dreapta, chiar dacă, în *funcționari*, nu sunt tupluri potrivite unor tupluri din *departamente*.

Sistemul va completa automat atributele care lipsesc din relația *funcționari* cu valoarea *NULL*.

**Exemplul 3.55.** Interogarea extrage numele, prenumele funcționarilor și denumirile departamentelor lor, chiar dacă există vreun departament în care nu activează niciun funcționar. Rezultatul va fi sortat după denumirea departamentelor în ordine descrescătoare (figura 3.10):

```
SELECT Nume, Denumire
FROM funcționari AS f RIGHT OUTER JOIN departamente AS d
  ON (f.DeptID = d.DeptID)
ORDER BY Denumire DESC;
```

Nume	Prenume	Denumire
Bobu	Angela	Software
NULL	NULL	Sisteme operare
Mutu	Andrei	Hardware
Matei	Ion	Hardware
Bobu	Matei	Hardware

Figura 3.10. Rezultatul interogării din exemplul 3.55

Joncțiunea externă adaugă tupluri fictive într-o relație, pentru a face corespondența cu alte tupluri ale celeilalte relații. În exemplul precedent, un tuplu fictiv (un funcționar fictiv) este adăugat în relația funcționarilor, dacă vreun departament nu are angajați. Acest tuplu va avea valoarea egală cu *NULL* pentru toate atributele relației funcționari.

Oționa *RIGHT* indică faptul că relația din care vor fi afișate toate tuplurile (relația *departamente*) este la dreapta de *RIGHT OUTER JOIN*. Relația din stânga este cea în care se adaugă tupluri fictive.

### 3.5.3.4. Joncțiunea externă completă

În cazul *FULL OUTER JOIN*, se consideră perechile de tupluri definite atât de joncțiunea exterioară de stânga, cât și cele definite de joncțiunea exterioară de dreapta. Tuplurile care lipsesc într-o relație se completează cu valori *NULL*.

Interogarea de mai jos returnează toate tuplurile din relația *funcționari*, chiar dacă nu sunt tupluri potrivite în relația *departamente*. În același timp, această interogare preia toate tuplurile din relația *departamente*, chiar dacă nu sunt tupluri potrivite în relația *funcționari*.

**Exemplul 3.56.** Să se extragă lista funcționarilor și a denumirilor departamentelor în care aceștia activează (figura 3.11). Rezultatul va conține și funcționari care nu au vreun departament, precum și departamentele ale căror funcționari sunt necunoscuți:

```
SELECT Nume, Denumire
FROM funcționari AS f FULL OUTER JOIN departamente AS d
ON (f.DeptID = d.DeptID);
```

Nume	Prenume	Denumire
Bobu	Angela	Software
Mutu	Andrei	Hardware
Matei	Ion	Hardware
Paiu	Adrian	NULL
NULL	NULL	Sisteme operare
Bobu	Matei	Hardware

Figura 3.11. Rezultatul interogării din exemplul 3.56

### 3.6. Subinterrogări

Există situații când, pentru realizarea unui obiectiv, este implicat un lanț de două sau mai multe interogări secvențiale, în care se folosește rezultatul unor interogări ca sursă pentru găsirea datelor căutate în interogarea principala. O instrucție *SELECT* inclusă în altă interogare SQL se numește subinterrogare. Astfel de construcții se folosesc în cazul în care rezultatul dorit nu se poate obține cu o singură parcurgere a datelor. O utilizare comună a subinterrogărilor constă din efectuarea următoarelor activități: compararea valorilor sau secvențelor de valori, verificarea de apartenență la o mulțime de valori, compararea mulțimilor și cardinalității mulțimilor de tupluri. Aceste utilizări sunt discutate în secțiunile următoare.

Pentru exemplele prezentate în această secțiune, va fi utilizată baza de date constituită din două relații *funcționari* și *departamente*:

*funcționari (Nume, Prenume, Post, Salariu, DeptID),  
departamente (DeptID, Denumire, Oraș).*

### 3.6.1. Tipuri de subinterrogări

Executarea unei subinterrogări constă în efectuarea unei interrogări în interiorul altei interrogări. Se mai utilizează termenii de interrogări în cascadă sau interrogări imbricate. În cazul executării interrogării, interrogările imbricate sunt evaluate primele.

Utilizarea interrogărilor imbricate necesită o atenție deosebită:

- Tipului rezultatului returnat.
- Operatorilor care pot fi aplicați în raport cu tipul rezultatului returnat.

Înțînd cont de aceste informații se poate admite că, în majoritatea cazurilor, rezultatul interrogării imbricate va fi utilizat într-o condiție a clauzei *WHERE* sau *HAVING*.

Interrogările imbricate pot returna cinci tipuri de rezultate:

- O valoare: un tuplu definit pe un singur atribut.
- $m$  valori: un tuplu definit pe  $m$  attribute.
- O mulțime de valori:  $n$  tupluri definite pe un singur atribut.
- $n$  tupluri definite pe  $m$  attribute.
- 0 sau  $n$  tupluri.

Toate aceste tipuri de subinterrogări sunt folosite în clauzele *WHERE* sau *HAVING*. În afară de aceasta, subinterrogările pot fi plasate în clauza *FROM*. În acest caz, ele sunt asimilate unor relații temporare din care se calculează rezultatul interrogării care le include. Unele SGBD-uri acceptă subinterrogări și în clauza *ORDER BY* a unei instrucțiuni *SELECT*. Valoarea returnată de subinterrogare va determina ordinea de afișare a rezultatului. De asemenea, din punct de vedere al modului de amplasare, subinterrogările pot fi prezente în lista *SELECT* (ale căror valori returnate vor fi prezente în rezultatul final), precum și în clauzele *BETWEEN* sau *LIKE*.

După modalitatea de a face legături între două sau mai multe interogări, subinterrogările pot fi corelate sau necorelate:

- O subinterrogare este corelată când valoarea produsă de ea depinde de o valoare produsă de instrucțiunea *SELECT* exterioară care o conține.
- Orice alt tip de subinterrogare este necorelată.

Trebuie menționat că, uneori, pentru a răspunde la o întrebare formulată în limba naturală, este posibilă crearea unei interogări SQL cu sau fără interogări imbricate. Este bine să se evite utilizarea cererilor corelate foarte costisitoare. În acest caz, este pe deplin posibil să se evalueze costul interogărilor, dar acest lucru nu constituie obiectul acestei secțiuni.

### 3.6.2. Subinterrogări cu operatori de comparație

Folosind subinterrogările, pot fi găsite date dintr-una sau mai multe relații, fără să se apeleze la joncțiunea lor. Subinterrogările din instrucțiunile *SELECT* permit să se realizeze mai multe acțiuni. În această secțiune, în primul rând, va fi examinat cazul în care o expresie se compară cu rezultatul altrei instrucțiuni *SELECT*. Aceste subinterrogări se numesc scalare și returnează un singur tuplu definit pe un singur atribut. Într-al doilea rând, vor fi examineate subinterrogările care returnează o listă sau o mulțime de valori, valorile reprezentând un tuplu.

Se utilizează următoarele două forme de sintaxă, respectiv, pentru aceste subinterrogări:

... {<expresie>} | (<expresie1> [, <expresie2> ...])}                    <operator comparație> (<subinterrogare>)

Să se observe că, în afară de cei doi operanzi, fiecare comparație se realizează de un operator. Operatorul ce intervene în specificarea condiției poate fi un operator de comparație (=, >, >=, <, <=, <>) care se aplică numai în cazul în care subinterrogarea returnează o singură valoare sau un singur tuplu de valori.

În cazul în care subinterrogarea returnează un scalar, se utilizează următoarea variantă:

...<expresie> <operator comparatie>  
(<subinterrogare>)

În acest predicat, subinterrogarea trebuie să întoarcă o singură valoare (un tuplu cu un atribut). Valoarea obținută de interogarea subordonată se compară cu rezultatul evaluării expresiei poziționate în partea stângă a operatorului de comparare.

Predicatul este evaluat cu valoarea *adevărat*, în cazul în care comparația (indicată de unul din operatorii de comparație) rezultatului expresiei cu cel returnat de subinterrogare, este *adevărat*. În caz contrar, se evaluează la *fals*. Dacă interogarea subordonată nu produce nicio valoare, predicatul se evaluează la valoarea *NULL*. Dacă subinterrogarea returnează mai mult de o valoare (mai multe tupluri definite pe un atribut, sau cel puțin un tuplu definit pe mai multe atribute) se produce o eroare de executare.

Următorul exemplu de cod reprezintă o subinterrogare scalară care trebuie să returneze un singur tuplu cu un singur atribut.

**Exemplul 3.57.** Să se găsească numele și prenumele funcționarilor care activează în același departament cu Adrian Paiu:

```
SELECT Nume, Prenume FROM funcționari  
WHERE DeptID = (SELECT DeptID  
                    FROM funcționari  
                    WHERE Nume = 'Paiu'  
                    AND Prenume = 'Adrian');
```

Există subinterrogări care, în mod obligatoriu, trebuie să întoarcă un singur tuplu (definit pe unu sau mai multe atribute). Acestea folosesc următoarea sintaxă:

...(<expresie1>[,<expresie2>...])  
      <operator comparație> (<subinterrogare>)

Într-un predicat de acest tip, subinterrogarea trebuie să returneze un singur tuplu și atât de multe atribute câte sunt specificate între paranteze din stânga operatorului de comparație. Expresiile <expresie1>, <expresie2>... care figurează în lista dintre paranteze sunt evaluate și se formează un tuplu de valori. Tuplul care s-a format se compară, utilizând un operator de comparație, cu tuplul întors de instrucțiunea *SELECT* subordonată.

Precum s-a menționat anterior, două tupluri sunt considerate egale, dacă atributele corespunzătoare sunt egale și nenele. Ele sunt considerate distincte, dacă vreun atribut respectiv este distinct de omologul său și ambele sunt nenele. În orice alt caz, rezultatul prediciului este nul.

Predicatul are valoarea *adevărat*, dacă rezultatul comparației este adevărat pentru tuplul generat de subinterrogare. În caz contrar, se evaluează la *fals*. Dacă interogarea subordonată nu produce niciun tuplu, predicatul se evaluează la valoarea *NULL*. În cazul în care interogarea subordonată întoarce mai mult de un tuplu, se produce o eroare de execuțare.

Urmează o interogare din care instrucțiunea *SELECT* subordonată returnează doar un tuplu:

**Exemplul 3.58.** Să se găsească numele și prenumele funcționarilor care activează în același post și au același salar ca Adrian Paiu:

```
SELECT Nume, Prenume FROM funcționari
WHERE (Post, Salariu) = (SELECT Post, Salariu
                           FROM funcționari
                           WHERE Nume = 'Paiu'
                           AND Prenume = 'Adrian');
```

În clauza *WHERE*, tuplul de valori obținut de subinterrogare se compară cu secvența de atrbute luate în paranteze și scrise în partea stângă a operatorului de comparare.

O interogare *SELECT* poate cuprinde mai multe subinterrogări, fie imbricate, fie pe același nivel în diferite predicate combinate cu *AND* sau *OR*.

Urmează o cerere care conține mai multe subinterrogări, acestea fiind pe același nivel.

**Exemplul 3.59.** Să se găsească numele, prenumele, postul și salariul funcționarilor care activează în același post ca și Adrian Paiu sau au un salar mai mare ca Bobu Matei:

```
SELECT Nume, Prenume, Post, Salariu
FROM funcționari
WHERE Post = (SELECT Post FROM funcționari
               WHERE Nume = 'Paiu'
               AND Prenume = 'Adrian')
OR Salariu > (SELECT Salariu FROM funcționari
                 WHERE Nume = 'Bobu'
                 AND Prenume = 'Matei');
```

O interogare poate conține mai multe subinterrogări incluse una în cealaltă.

**Exemplul 3.60.** Să se găsească numele și prenumele funcționarilor care au un salariu mai mare decât salariul maximal al ocupanților postului în care activează Bobu Matei:

```
SELECT Nume, Prenume FROM funcționari
WHERE Salariu > (SELECT MAX(Salariu)
                   FROM funcționari
                   WHERE Post =
                         (SELECT Post
                          FROM funcționari
                          WHERE Nume = 'Bobu'
                          AND Prenume = 'Matei'));
```

Joncțiunile și subinterrogările pot fi combinate în instrucțiunile *SELECT*.

**Exemplul 3.61.** Să se găsească numele, prenumele și postul funcționarilor care activează în orașul Chișinău în același post ca și Adrian Paiu:

```
SELECT Nume, Prenume, Post
FROM funcționari JOIN departamente
ON funcționari.DeptID = departamente.DeptID
WHERE Oraș = 'Chișinău'
AND Post = (SELECT Post FROM funcționari
            WHERE Nume = 'Paiu'
            AND Prenume = 'Adrian');
```

Deoarece atrbutele de joncțiune sunt aceleași, această interogare poate fi redată mai simplu, folosind joncțiunea naturală:

```
SELECT Nume, Prenume, Post
```

```

FROM funcționari NATURAL JOIN departamente
WHERE Oraș = 'Chișinău'
    AND Post = (SELECT Post
                  FROM funcționari
                  WHERE Nume = 'Paiu'
                  AND Prenume = 'Adrian');

```

### 3.6.3. Subinterrogări cu operatorul IN

Operatorul *IN* deja a fost utilizat anterior, în criteriile de selecție care specificau între paranteze o listă de valori. Singura diferență în utilizarea operatorului *IN* cu subinterrogări constă în faptul că lista de valori este construită de subinterrogare. Predicatul care include operatorul *IN* utilizat cu o interogare subordonată are următoarea formă generală:

$$\dots \{<\text{expresie}> | (<\text{expresie1}>, <\text{expresie2}> \dots) \} \\ [\text{NOT}] \text{ IN } (<\text{subinterrogare}>) \dots$$

Din sintaxa generală, se observă că există două forme ale predicatului cu includerea operatorului *IN*. Predicatul, în forma sa mai simplă, presupune că, în partea stângă a operatorului *IN*, se poate găsi o expresie, iar a două variantă presupune că partea stângă este constituită dintr-un tuplu de expresii. În continuare, aceste variante vor fi examinate pe rând.

SQL utilizează calculul relațional pentru operațiile care permit verificarea apartenenței unui tuplu la o relație. Operatorul *IN* verifică apartenența la o mulțime, în cazul în care mulțimea este colecție de valori obținută de o instrucție *SELECT*. Operatorul *NOT IN* verifică neapartența la o mulțime.

$$\dots <\text{expresie}> [\text{NOT}] \text{ IN } (<\text{subinterrogare}>) \dots$$

Partea dreapta este o subinterrogare în paranteze, care poate returna doar un singur atribut. Expressia din partea stângă este evaluată și comparată cu fiecare tuplu (definit pe un singur atribut) din rezultatul subinterrogării. Valoarea predicatului este *adevărat* în cazul în care este găsit, în rezultatul returnat din subinterrogare, un tuplu echivalent cu valoarea expresiei. Predicatul ia valoarea *fals*, dacă nu se găsește niciun tuplu potrivit (inclusiv cazul când subinterrogarea nu întoarce niciun tuplu).

Atunci când *IN* este negat (*NOT NULL*), predicatul este evaluat la *adevărat*, în cazul în care expresia este diferită de toate valorile atributului returnat de subinterrogare. Este evaluat la *adevărat*, de asemenea, atunci când subinterrogarea nu returnează niciun tuplu. Dacă se găsește vreo valoarea egală cu a expresiei, este evaluat la *fals*. Dacă rezultatul expresiei este *NULL* sau dacă subinterrogarea returnează unele valori nule și diferite de cea a expresiei, predicatul este evaluat la *NULL*.

**Exemplul 3.62.** Să se selecteze toți funcționarii care câștigă un salariu egal cu salariul minim la nivel de departament.

```
SELECT Nume, Prenume, Salariu, DeptID
FROM funcționari
WHERE Salariu IN (SELECT MIN(Salariu)
                    FROM funcționari
                    GROUP BY DeptID);
```

Interrogarea internă va fi prima executată, producând ca răspuns la cerere, iar interrogarea principală este apoi procesată și folosește valorile returnate de către interrogarea subordonată pentru a-și finaliza propria condiție de căutare.

O altă modalitate de specificare a listei de valori returnată de interrogarea subordonată constă în includerea în paranteze a unei subinterrogări, care întoarce doar un singur tuplu:

$$\dots(<\text{expresie}1>[, <\text{expresie}2> \dots]) \\ [\text{NOT}] \text{ IN } (<\text{subinterrogare}>) \dots$$

În acest predicat, subinterrogarea trebuie să returneze un tuplu definit pe atâtea atribute câte sunt specificate între paranteze în stânga operatorului *IN*. Expresiile din stânga *expresie1*, *expresie2*, ... sunt evaluate, iar tuplul format de acestea se compară cu tuplurile subinterrogării, unu câte unu.

Ca de obicei, valorile nule în tupluri sunt combinate în conformitate cu normele obișnuite ale expresiilor booleene SQL. Două tupluri sunt considerate egale, dacă toate componentele lor respective sunt nenule și egale. Tuplurile sunt diferite, dacă componentele lor sunt nenule și diferite; în caz contrar, comparația tuplurilor este necunoscută, adică nulă. Dacă toate rezultatele pentru tupluri sunt diferite sau nule, cu cel puțin un *NULL*, atunci rezultatul lui *IN* este nul.

Predicatul este evaluat la *adevărat*, dacă, în subinterrogare, se găsește vreun tuplu egal cu cel din stânga operatorului *IN*. În caz contrar, este evaluat la *fals* (chiar dacă subinterrogarea nu întoarce niciun tuplu). Dacă subinterrogarea întoarce unele tupluri nule, iar restul tuplurilor sunt distințe de tuplul din stânga operatorului *IN*, predicatul este evaluat la *NULL*.

În cazul predicatului cu *NOT IN*, de asemenea, sunt evaluate expresiile din stânga *expresie1*, *expresie2*,... și tuplul pe care acestea îl formează se compară cu tuplurile returnate de subinterrogare.

Predicatul este evaluat la *adevărat*, dacă nu există niciun tuplu egal în interogarea subordonată. De asemenea, este evaluat la *adevărat*, dacă subinterrogarea nu întoarce nici un tuplu. Dacă se găsește vreun tuplu egal, este evaluat la *fals*. În cazul în care subinterrogarea returnează unele tupluri cu nul și restul de tupluri sunt diferite de tuplul din partea stângă a operatorului *NOT IN*, predicatul este evaluat la *NULL*.

Trebuie menționat că, în cazul utilizării operatorului *IN*, nu se poate presupune că subinterrogarea va fi complet evaluată.

**Exemplul 3.63.** Următoarea interogare găsește funcționarii care câștigă salariul cel mai mic din departamentul lor:

```
SELECT Nume, Prenume, Salariu, DeptID
FROM funcționari
WHERE (Salariu, DeptID) IN
      (SELECT MIN(Salariu), DeptID
       FROM funcționari
       GROUP BY DeptID);
```

Cererea de mai sus compară o pereche de atribute. Atributele din partea stângă a condiției de căutare sunt între paranteze și fiecare atribut este separat printr-o virgulă. Atributele listate în clauza *SELECT* a subinterrogării trebuie să se potrivească în număr și tip de date cu atributele respective din interogarea externă cu care ele sunt comparate.

### 3.6.4. Subinterrogări cu operatorul ANY

Cuvântul rezervat *ANY* poate modifica acțiunile operatorului de comparație pentru a permite interrogării externe să accepte mai multe tupluri returnate de interrogarea subordonată. Sintaxa generală a predicatului, care conține operatorul *ANY*, este următoarea:

$$\dots \{<\text{expresie}>|(<\text{expresie}1>[, <\text{expresie}2> \dots])\} \\ \{>|<|=|=|=|<=<>\} \text{ ANY } (<\text{subinterrogare}>) \dots$$

Mai întâi, se va examina aplicarea operatorului *ANY* asupra unei subinterrogări care construiește o relație definită pe un singur atribut și compară rezultatul acestia cu valoarea de același tip a expresiei din interrogarea externă. În calitate de comparatori, apar operatorii de comparare:

$$\dots <\text{expresie}> \{>|<|=|=|=|<=<>\} \\ \text{ANY } (<\text{subinterrogare}>) \dots$$

Partea dreaptă reprezintă o subinterrogare între paranteze, iar partea stângă - un constructor de tuplu care este evaluat și comparat cu fiecare tuplu din rezultatul subinterrogării, folosind operatorul specificat, care ar trebui să conducă la un rezultat boolean. Predicatul ia valoarea *adevărat*, dacă rezultatul comparației cu cel puțin un tuplu din relația (cu un singur atribut) construită de-subcerere este *adevărat* și întoarce valoarea *fals* în caz contrar (se include și cazul special al interrogării, care nu returnează niciun tuplu). Rezultatul este *NULL*, dacă comparația nu întoarce *adevărat* pentru niciun tuplu și returnează *NULL* pentru cel puțin un tuplu.

**Exemplu 3.64.** Cum se numesc și care sunt salariile funcționarilor care câștigă mai mult decât alțineva? Rezultatul să se ordoneze descrescător după salariu:

```
SELECT Nume, Prenume, Salariu
FROM funcționari
WHERE Salariu > ANY (SELECT Salariu
                      FROM funcționari)
ORDER BY Salariu DESC;
```

În următoarea utilizare a operatorului *ANY*, subinterrogarea trebuie să întoarcă atâtea atribute câte sunt specificate în parantezele din stânga operatorului de comparare:

$$\dots(<\text{expresie}1>[, <\text{expresie}2> \dots]) \{>|<|=|>=|<=|<>\} \\ \text{ANY}(<\text{subinterrogare}>)\dots$$

Expresiile din stânga, *expresie1*, *expresie2*, ..., sunt evaluate și din acestea este constituit un tuplu care este comparat cu tuplurile returnate de subinterrogare.

Predicatul se evaluează la *adevărat*, dacă comparația stabilită de operatorul de comparare este valabilă cel puțin pentru un tuplu întors de subinterrogare. În caz contrar, se evaluează la *fals* (chiar dacă subinterrogarea nu returnează niciun tuplu). Dacă subinterrogare returnează un tuplu cu valori nule, predicatul nu poate fi *fals* (sau ia valoarea *adevărat*, sau ia *NULL*).

**Exemplul 3.65.** Următoarea interogare găsește datele despre funcționarii care câștiga un salariu mai mare decât media pe un departament, dar salariul maxim pe acest departament e mai mic de 5000:

```
SELECT * FROM funcționari
WHERE (5000, Salariu) > ANY
      (SELECT MAX(Salariu), AVG(Salariu)
       FROM funcționari GROUP DeptID);
```

În locul lui *ANY*, se poate utiliza *SOME*, acești operatori sunt sinonime. Operatorul *IN* este echivalent cu *= ANY*. Ca și în cazul lui *IN*, nu se poate presupune că subinterrogarea este complet evaluată.

### 3.6.5. Subinterrogări cu operatorul ALL

Operatorul *ALL* compară o valoare cu toate valorile returnate de o subinterrogare. Sintaxa generală de utilizare a acestui operator în predicatele de filtrare are forma:

$$\dots\{<\text{expresie}>|(<\text{expresie}1>[, <\text{expresie}2> \dots])\} \\ \{>|<|=|>=|<=|<>\} \text{ALL}(<\text{subinterrogare}>)\dots$$

Partea stângă reprezintă o expresie sau un tuplu de expresii, iar partea dreaptă este o subinterrogare în paranteze care returnează exact același număr și tip de atribute ca al expresiei sau ca al expresiilor ce constituie tuplul. Aceste două forme sunt examineate separat în continuare.

Mai întâi, va fi examinată situația când, în partea stângă, este un scalar, sau o expresie care este evaluată la un scalar:

...<expresie> {>|<|=|>=|<=|<>} ALL (<subinterrogare>)...

În această utilizare a operatorului *ALL*, subinterrogarea trebuie să returneze un singur atribut. Predicatul se evaluează la adevărat, dacă comparația stabilită de către operatorul de comparație este valabilă pentru toate valorile atributului revenit de la interogarea subordonată. Predicatul se evaluează, de asemenea, la adevărat atunci când subinterrogarea nu returnează niciun tuplu. În caz contrar, se evaluează la fals. În cazul în care subinterrogarea returnează vreun *NULL*, predicatul se evaluează la *NULL*.

**Exemplul 3.66.** Să se găsească numele, prenumele și salariul funcționarilor care câștigă cel mai mare salariu:

```
SELECT Nume, Prenume, Salariu
FROM funcționari
WHERE Salariu >= ALL (SELECT Salariu FROM funcționari);
```

În utilizarea care urmează a operatorului *ALL*, subinterrogarea trebuie să se întoarcă atâtea atribute câte sunt specificate de expresiile luate în paranteze din partea stângă a operatorului:

...(<expresie1>[,<expresie2>...]) {>|<|=|>=|<=|<>}  
ALL (<subinterrogare>)...

Expresiile din partea stângă constituie un tuplu și sunt evaluate și comparate cu fiecare din tuplurile rezultatului, folosind operatorul de comparație dat. Rezultatul predicatului cu *ALL* este *adevărat*, dacă comparația returnează *adevărat* pentru toate tuplurile subinterrogării (se include și cazul special al interogării care nu returnează niciun tuplu). Rezultatul este *fals*, dacă comparația are valoarea *fals* cel puțin pentru un tuplu returnat de interogarea subordonată. Rezultatul este *NULL*, dacă comparația nu are valoarea *fals* pentru vreun tuplu și returnează *NULL* pentru cel puțin un tuplu.

**Exemplul 3.67.** Următoarea interogare găsește datele despre funcționarii care câștiga un salariu mai mare decât media oricărui departament, dar salariul maxim pe orice departament e mai mic de 5000:

```
SELECT * FROM funcționari
WHERE (5000, Salariu) > ALL
      (SELECT MAX(Salariu), AVG(Salariu)
       FROM funcționari GROUP DeptID);
```

De asemenea, trebuie remarcat faptul că  $<> ALL$  este echivalent cu  $NOT IN$ . Ca și în cazul operatorului  $IN$ , nu se poate presupune că subinterrogarea va fi evaluată în întregime. Cu alte cuvinte, în cazul în care în predicate sunt utilizate subinterrogări, SGBD-ul nu întotdeauna obține rezultatul complet de la interogarea subordonată, dar numai atât cât este necesar. Procesul de examinare a tuplurilor returnate de către subinterrogări continue până apare posibilitatea de determinare, dacă predicalul are valoarea *adevărat*.

### 3.6.6. Subinterrogări corelate și necorelate

Până aici, au fost prezentate numeroase exemple de includere a interogărilor în alte interogări, adică de creare a interogărilor imbricate sau subinterrogărilor. Dar, în majoritatea exemplelor, interogările subordonate nu făceau trimitere la atrbute care nu existau în propria clauză *FROM*.

Acest fapt semnifică că rezultatul subinterrogării poate fi evaluat independent de rezultatele interogărilor de la toate nivelele care o conțin, inclusiv, de interogarea principală (de la nivelul unu). Prin urmare, SGBD-ul o evaluatează o singură dată și transmite rezultatul acesteia mai sus pe ierarhie, oriunde este nevoie. Acest mod de formulare a subinterrogărilor vorbește despre faptul că interogarea subordonată *nu este corelată*.

Utilizarea subinterrogărilor necorelate este echivalentă cu efectuarea a două sau mai multe interogări secvențiale și utilizarea rezultatului interogărilor interne ca valoare de filtrare sau căutare în interogările externe. O astfel de subinterrogare poate fi privită ca o realizare a unui relații temporare, cu rezultate care pot fi accesate și utilizate de către interogarea exterioară.

Modul de evaluare al unei interogări care conține o subinterrogare necorelată este următorul:

- Subinterrogarea este executată prima și determină un rezultat.
- Interrogarea externă se execută, utilizând rezultatul furnizat de interrogarea subordonată.

Următorul exemplu reprezintă o interrogare necorelată.

**Exemplul 3.68.** Să se găsească numele și prenumele funcționarilor care activează în Chișinău:

```
SELECT Nume, Prenume FROM funcționari
WHERE DeptID IN (SELECT DeptID
                   FROM departamente
                   WHERE Oraș='Chișinău');
```

Rezultatul interrogării subordonate este întotdeauna același, indiferent de tuplul care se verifică în interrogarea principală. Din acest motiv, SGBD-ul calculează rezultatul o dată pentru evaluarea condiției *WHERE* a interrogării externe cu toate tuplurile obținute de subinterrogare.

Cu toate acestea, există subinterrogări care fac referiri în propria clauză *FROM* și la atributele altrei relații. Aceste subinterrogări se numesc *corelate*.

O subinterrogare corelată nu poate fi evaluată în mod independent de interrogarea exterioară, deoarece rezultatul ei se poate schimba în funcție de evaluarea în fiecare moment a unui tuplu din interrogarea externă. SGBD-urile, prin urmare, evaluatează de multe ori rezultatul interrogării subordonate. Este necesar să se aibă grijă ca numărul de evaluări să fie cât mai mic posibil, pentru a evita procesele inutile ale SGBD-ului.

În cazul în care, într-o subinterrogare, este specificat un nume de atribut, SGBD-ul trebuie să cunoască fără ambiguitate în ce relație se întâlnește atributul specificat. De aceea, sistemul caută, în primul rând, relația care conține un atribut cu numele specificat, urmând următoarea ordine:

1. Relațiile din clauza *FROM* proprie subinterrogării corelate. Dacă numele se întâlnește în mai mult de o relație, referința la atribut este ambiguă și eronată. Pentru evitarea acestei erori se utilizează nume de atribut calificate cu numele relației.

Dacă se găsește relația – căutarea este terminată.

2. Relațiile din clauza *FROM* primei interogări anterioare. Dacă numele este întâlnit în mai multe relații, din nou este necesară calificarea acestuia.

Dacă se găsește relația – căutarea este terminată

3. Același proces este urmat consecutiv, nivel cu nivel, până se ajunge, dacă este necesar, la interogarea externă.

Dacă se dorește schimbarea ordinii de căutare, se cere calificarea atributelor astfel, ca să indice SGBD-ului relația ce corespunde atributului.

**Exemplul 3.69.** Să se găsească numele și prenumele, și salariul funcționarilor care ridică un salarior mai mare decât media salarialului în departamentul acestuia:

```
SELECT Nume, Prenume, Salariu FROM funcționari
WHERE Salariu > (SELECT AVG(Salariu) FROM funcționari);
```

Această interogare nu răspunde la enunțul formulat. Să se observe că interogarea subordonată calculează salariorul mediu pe întreprindere. Ar trebui să se modifice subinterrogarea care calculează salariorul mediu al funcționarului ca să reacționeze la rezultatul evaluat în interogarea principală.

Este evident că, pentru ca interogarea subordonată să calculeze salariorul mediu al funcționarilor unui departament concret, trebuie refăcută condiția în clauza *WHERE*. Dar nu se poate a priori specifica o condiție de tipul *DeptID=N*, unde *N* este numărul departamentului, deoarece numărul departamentului depinde de tuplul care este evaluat în interogarea exterioară.

Soluția vine de la corelare: *N* se substituie cu atributul *DeptID* a relației funcționari care se valuează în interogarea principală. În acest mod, în

orice moment, acest atribut va lua numărul departamentului funcționarului care propriu-zis este evaluat în interrogarea principală.

Urmează un exemplu de interrogare corelată.

**Exemplul 3.70.** Pentru a obține cele spuse mai sus, se utilizează calificatorii și atunci interrogarea din exemplul 3.69 ia forma:

```
SELECT Nume, Prenume, Salariu FROM functionari AS f1
WHERE Salariu > (SELECT AVG(Salariu)
                   FROM functionari AS f2
                   WHERE f2.DeptID=f1.DeptID);
```

Acum, interrogarea subordonată se recalculează (cu respectivul cost computațional) pentru fiecare tuplu evaluat de interrogarea principală în funcție de valoarea atributului *f1.DeptID* (atributul interrogării principale).

Subinterrogările corelate sunt importante în rezolvarea unor probleme care sunt mai dificil de tratat cu ajutorul operațiilor de joncțiune (*join*) sau al subinterrogărilor necorelate. O interrogare corelată se recunoaște cu ușurință, prin faptul că ea nu poate fi evaluată independent, făcând referință la instrucțiunea externă și fiind executată o dată pentru fiecare tuplu procesat de către instrucțiunea externă.

Subinterrogările corelate necesită calificarea explicită a numelui atributelor care referă la interrogarea externă, cu numele sau aliasul relației corespunzătoare. Modul de evaluare al unei interrogări corelate este următorul:

1. Interrogarea externă determină un tuplu candidat.
2. Subinterrogarea este executată utilizând valorile tuplului candidat.
3. Rezultatul subinterrogării este utilizat de către interrogarea externă pentru calificarea sau descalificarea tuplului candidat.
4. Pașii 1-3 se repetă până când nu mai există tupluri candidat.

### 3.6.7. Subinterrogări cu operatorul EXISTS

SQL oferă posibilitatea de verificare, dacă o subinterrogare nu produce niciun tuplu în calitate de rezultat. Pentru ca acțiunea subinterrogării să reprezinte un test de existență, subinterrogarea trebuie să se utilizeze cu

operatorul *EXISTS*. Operandul operatorului *EXISTS* reprezintă o instrucție *SELECT* subordonată.

Formatul general al unui predicat cu operatorul *EXISTS* și cu o subinterrogare este prezentat mai jos. Trebuie menționat că operatorul *NOT* poate fi utilizat pentru a nega rezultatul operatorului *EXISTS*. Clauza *EXISTS* este urmată de o subinterrogare între paranteze și ia valoarea *adevărat*, dacă există cel puțin un tuplu care satisface condițiile subinterrogării:

...[*NOT*] *EXISTS* (<subinterrogare>) ...

Subinterrogarea este evaluată pentru a determina dacă acesta returnează tupluri. În cazul în care se întoarce cel puțin unul, rezultatul predicatului este *adevărat*, iar în cazul că nu întoarce niciun tuplu, rezultatul predicatului *EXISTS* este *fals*.

**Exemplul 3.71.** Fie că trebuie să se găsească funcționarii al căror departament începe cu litera "S". Această interogare poate fi scrisă, folosind operatorul *EXISTS*:

```
SELECT Nume, Prenume FROM funcționari AS f
WHERE EXISTS (SELECT *
               FROM departamente AS d
               WHERE d.DeptID = f.DeptID
                     AND Denumire LIKE 'S%');
```

Trebuie menționat că interogarea cu operatorul *EXISTS* poate fi transformată într-o echivalentă care aplică operatorul *IN*.

**Exemplul 3.72.** Interogarea din exemplul anterior poate fi rescrisă, folosind operatorul *IN*:

```
SELECT Nume, Prenume FROM funcționari AS f
WHERE f.DeptID IN
      (SELECT d.DeptID
         FROM departamente AS d
         WHERE d.DeptID = f.DeptID
               AND Denumire LIKE 'S%');
```

În cazul operatorului *EXISTS*, nu este important conținutul rezultatului subinterrogării, ci doar existența sau absența sa. Subinterrogarea nu se

execută, de obicei, mai mult decât este necesar pentru a determina dacă cel puțin un tuplu este returnat. Cu alte cuvinte, interogarea subordonată nu este neapărat forțată să fie efectuată în întregime.

Să se observe că subinterrogarea din exemplul precedent este bazată pe valorile din cererea principală. Adică se poate vorbi despre interogări corelate în cazul utilizării operatorului *EXISTS*.

Deoarece operatorul *EXISTS* nu face decât să verifice existența sau inexistența liniilor în rezultatul subinterrogării, aceasta poate conține orice număr de coloane. Nefiind necesar ca interogarea subordonată să returneze o anumită valoare, se poate selecta o constantă. De altfel, din punct de vedere al performanței, selectarea unei constante asigură mai mare rapiditate decât selectarea unei coloane.

Prin urmare, rezultatul nu depinde decât de o eventuală returnare de tupluri, și nu de conținutul lor, lista de atrbute returnate de către subinterrogare, în mod normal, nu are nicio semnificație. Există și excepții de la această regulă, cum ar fi subinterrogările care folosesc *INTERSECT*. O convenție de codificare este, de obicei, cea de a scrie toate testele *EXISTS* în forma *EXISTS (SELECT 1 WHERE...)*:

**Exemplul 3.73.** Această interogare afișează denumirile departamentelor care au cel puțin un funcționar care ridică un salariu mai mare de 10000.

```
SELECT Denumire
FROM departamente AS d
WHERE EXISTS (SELECT 1
               FROM funcționari AS f
               WHERE f.DeptID = d.DeptID
                     AND Salariu > 10000);
```

Pentru fiecare tuplu al relației *departamente*, subinterrogarea corelată este executată și dacă cel puțin un tuplu se găsește în relația *funcționari*, *EXISTS* ia valoarea *adevărat* și tuplul relației *departamente* satisface criteriul interogării.

În prezența valorilor *NULL* în subinterrogare, operatorul *EXISTS* trebuie utilizat cu atenție. Deoarece predicatul cu *EXISTS* ia valoarea *adevărat* și

în cazul în care interogarea subordonată întoarce valoarea *NULL* interogarea anterioară poate fi rescrisă într-o echivalentă.

**Exemplul 3.74.** Interogarea subordonată întoarce întotdeauna valoarea *NULL*, dacă este satisfăcută condiția *WHERE*:

```
SELECT Denumire
FROM departamente AS d
WHERE EXISTS (SELECT NULL
               FROM funcționari AS f
               WHERE f.DeptID = d.DeptID
                     AND Salariu > 10000);
```

În interogările cu subinterrogări, se aplică regula de vizibilitate pentru variabila-tuplu. Într-o subinterrogare, pot fi utilizate doar variabilele-tuplu, care sunt definite în propria subinterrogare sau în orice subinterrogare care conține această subinterrogare. Dacă variabila-tuplu este definită atât local (într-o subinterrogare), cât și global (într-o interrogare care conține o subinterrogare) se aplică definiția locală. Această regulă este similară regulii de folosire a variabilelor în limbajele de programare.

Cu ajutorul operatorului *NOT EXISTS*, se poate verifica inexistența tuplurilor în urma unei subinterrogări. Predicatul *NOT EXISTS* ia valoarea *adevărat*, dacă subcererea-argument nu produce niciun tuplu și ia valoarea *fals*, în caz contrar.

**Exemplu 3.75.** Să se găsească toate departamentele în care nu activează niciun funcționar:

```
SELECT DeptID, Denumire
FROM departamente AS d
WHERE NOT EXISTS (SELECT 1
                   FROM funcționari AS f
                   WHERE f.DeptID = d.DeptID);
```

Deși, într-o subinterrogare, operatorul *NOT IN* poate fi la fel de eficient ca și *NOT EXISTS*, operatorul *NOT EXISTS* este mult mai sigur, dacă subinterrogarea întoarce niște valori *NULL*, față de *NOT IN* pentru care condiția se evaluează la *fals* când în lista de comparații sunt incluse valori *NULL*.

Considerând cererea din exemplul 3.75, care găsește departamentele ce nu au niciun angajat, se poate trage concluzia că este corectă. Construcția *NOT IN* poate fi utilizată în calitate de alternativă în locul operatorului *NOT EXISTS*, precum e prezentat în următorul exemplu.

**Exemplu 3.76.** Să se găsească toate departamentele în care nu activează niciun funcționar:

```
SELECT DeptID, Denumire
FROM departamente AS d
WHERE d.DeptID NOT IN (SELECT f.DeptID
                        FROM funcționari AS f);
```

La prima vedere, pare că interogările din exemplele 3.75 și 3.76 sunt echivalente. Cu toate acestea, *NOT IN* evaluează la *fals* în cazul în care vreun membru al mulțimii returnate de subinterrogare este o valoare *NULL*. Prin urmare, interogarea nu va returna niciun tuplu, chiar dacă există tupluri în relația *departamente* care satisfac condiția *WHERE*. Astfel, în interogarea de mai sus, nu va fi întors niciun tuplu, dacă atributul *f.DeptID* conține valoarea *NULL*.

Operatorul *EXISTS* este foarte important, deoarece, uneori, nu există nicio alternativă la utilizarea acestuia. Toate interogările care utilizează operatorul *IN* sau un operator de comparație modificat (*=*, *<*, *>*, etc. modificat prin *ANY* sau *ALL*) pot fi exprimate cu operatorul *EXISTS*. Cu toate acestea, unele interogări formulate cu *EXISTS* nu pot fi exprimate în alt mod! Atunci, apare întrebarea: Ar trebui toate interogările anterioare să fie scrise cu operatorul *EXISTS*? Răspunsul rezidă în eficiența de procesare a interogărilor. Precum s-a menționat mai sus, interogările corelate sunt procesate mai mult timp decât interogările echivalente, dar necorelate. Este evident că trebuie aleasă varianta necorelată.

### 3.6.8. Subinterogări în clauza *FROM*

Până acum, s-a văzut că, în clauza *FROM*, a fost specificată o listă de relații. Deoarece rezultatul unei instrucțiuni *SELECT* este o relație, apare întrebarea dacă o interogare poate participa după *FROM* în calitate de relație. Răspunsul este afirmativ. Formatul unei subinterogări în clauza *FROM* arată precum urmează:

(<instrucțiune SELECT>) AS <alias>  
[(<atribut1>[,<atribut2>...])]

În acest format, <alias> este aliasul care se atribuie relației construită de subinterrogare, ale căreia atribute, de asemenea, pot fi optional atribuite nume noi. Numărul de atribute ale aliasului coincide cu numărul de atribute returnat de subinterrogare. La rândul său, subinterrogarea, evident, poate conține subinterrogări (subinterrogări simple fără operații pe mulțimi).

Aceste expresii pot fi specificate într-o listă a clauzei FROM exact ca o relație. Interogarea principală poate face referire la atributele subinterrogării ca la atributele oricărei relații obișnuite.

**Exemplul 3.77.** Să se afișeze numele și prenumele angajaților, salariile, numărul departamentului și media salariilor din fiecare departament pentru toți funcționarii care câștigă mai mult decât media salariilor din departamentul în care lucrează:

```
SELECT f.Nume, f.Prenume, f.Salariu, f.DeptID, d.SalariuMediu
FROM funcționari AS f,
     (SELECT DeptID, AVG(sal) AS SalariuMediu
      FROM funcționari
      GROUP BY DeptID) AS d
 WHERE f.DeptID = d.DeptID AND f.Salariu > d.SalariuMediu;
```

În exemplul anterior, instrucțiunea *SELECT* operează pe relația *funcționari* (cu aliasul *f*) și pe relația returnată de interogarea din clauza *FROM* (cu aliasul *d*). Relația *d* e definită pe două atribute (*DeptID* și *SalariuMediu*) și conține, pentru fiecare departament, identificatorul său și media salariilor.

### 3.6.9. Subinterrogări în clauza HAVING și în alte clauze

Subinterrogările pot fi folosite nu numai în clauza *WHERE*, dar și în clauza *HAVING*. SGBD-ul execută subinterrogarea, returnând apoi rezultatul către clauza *HAVING* a interogării principale.

**Exemplul 3.78.** Să se afișeze identificatoarele tuturor departamentelor la nivelul cărora salariul minim are o valoare mai mare decât valoarea salariului minim din cadrul departamentului Dept02:

```

SELECT DeptID, MIN(Salariu)
FROM funcționari
GROUP BY DeptID
HAVING MIN(Salariu) >
    (SELECT MIN(Salariu)
     FROM funcționari WHERE DeptID = 'Dept02');

```

Clauza *WHERE* se referă la fiecare tuplu în parte, iar clauza *HAVING* la grupurile de tupluri specificate în clauza *GROUP BY*.

**Exemplul 3.79.** Să se găsească postul având cel mai scăzut salariu mediu:

```

SELECT Post, AVG(Salariu)
FROM funcționari
GROUP BY Post
HAVING AVG(Salariu) = (SELECT MIN(AVG(Salariu))
                        FROM funcționari GROUP BY Post);

```

Trebuie reținut că o clauză *HAVING* se folosește întotdeauna împreună cu clauza *GROUP BY* și într-o clauză *HAVING* se pot folosi funcții de agregare.

Unele sisteme, precum Oracle, acceptă subinterrogări prezente în lista clauzei *SELECT*.

**Exemplul 3.80.** Valorile returnate de subinterrogările din clauza *SELECT* vor fi prezente în rezultatul final:

```

SELECT
    (SELECT MAX(Salariu)
     FROM funcționari) AS SalMax,
    (SELECT MIN(Salariu)
     FROM funcționari) AS SalMin,
    (SELECT COUNT(*)
     FROM funcționari) AS FunctTotal,
    (SELECT SUM(Salariu)
     FROM funcționari) AS SalTotal
FROM ...

```

De asemenea, din punct de vedere al modului de amplasare, în Oracle, subinterrogările pot fi prezente în și în clauzele *BETWEEN* sau *LIKE*.

**Exemplul 3.81.** Valorile returnate de subinterrogările din clauza *BETWEEN* sunt folosite pentru a selecta valorile dintr-un interval de valori:

```
SELECT * FROM funcționari
WHERE Salariu BETWEEN (SELECT MIN(Salariu) FROM
funcționari) AND 2000;
```

Poate fi și o astfel de utilizare a subinterrogărilor în clauza *BETWEEN*:

```
SELECT Nume, Prenume FROM funcționari
WHERE (SELECT MAX(Salariu) FROM funcționari)
BETWEEN 5000 AND 10000
```

### 3.7. Interrogări cu operatori din teoria mulțimilor

Standardul SQL2 pune la dispoziție operatori din teoria mulțimilor, cum ar fi operatorii uniunea, intersecția și diferența care sunt foarte aproape de algebra relațională și corespund operațiilor  $\cup$ ,  $\cap$  și  $\setminus$ , respectiv. Cu ajutorul acestor operatori, se combină două sau mai multe cereri legate cu cuvintele-cheie *UNION*, *INTERSECT* sau *EXCEPT* (în unele sisteme, precum Oracle, *MINUS*). Operațiile construiesc relații cu scheme compatibile, adică cu același număr de atrbute și de același tip.

Toți operatorii pe mulțimi au aceeași precedență. Dacă o instrucție SQL conține mai mulți operatori pe mulțimi, SGBD-ul evaluează cererea de la stânga la dreapta. Pentru a schimba această ordine de evaluare, se pot utiliza paranteze.

SQL nu impune ca schemele, pe care se execută operatorii, să fie identice (spre deosebire de algebra relațională), ci doar ca atrbutele să aibă domenii compatibile. Corespondența între atrbute nu se bazează pe nume, ci pe poziția atrbutelor. Dacă atrbutele au nume diferite, rezultatul va prelua numele de atrbute din primul operand.

Sintaxa pentru utilizarea operatorilor din teoria mulțimilor este:

```
<interrogare SQL> {<UNION | INTERSECT | EXCEPT>} [ALL]
<interrogare SQL> [...]
```

Operatorii din teoria mulțimilor presupun eliminarea duplicatelor ca opțiune implicită. Dacă se dorește utilizarea acestor operatori cu menținerea duplicatelor, este suficientă specificarea opțiunii *ALL*.

Pentru această secțiune, se presupune că două relații *funcționari1* și *funcționari2* conțin date despre funcționarii ai două filiale ale unei întreprinderi.

### **3.7.1. Operatorii UNION și UNION ALL**

Operatorul *UNION* permite fuzionarea a două selecții pentru a obține o mulțime de tupluri egală cu uniunea acestor două selecții.

**Exemplul 3.82.** Să se găsească numele inginerilor din cele două filiale:

```
SELECT Nume FROM funcționari1 WHERE Post='Inginer'  
UNION
```

```
SELECT Nume FROM funcționari2 WHERE Post='Inginer';
```

Spre deosebire de clauza *SELECT*, operatorul *UNION* înălțătură în mod automat tuplurile dupicate. Astfel, în interogarea anterioară, în cazul în care un inginer, de exemplu, *Petrache*, lucrează în ambele filiale, atunci *Petrache* va apărea o singură dată în rezultat. Pentru a păstra duplicatele, se utilizează *UNION ALL*, în loc de *UNION*:

**Exemplul 3.83.** Să se găsească numele inginerilor (inclusiv duplicate) din cele două filiale:

```
SELECT Nume FROM funcționari1 WHERE Post='Inginer'  
UNION ALL
```

```
SELECT Nume FROM funcționari2 WHERE Post='Inginer';
```

Numărul de tupluri dupicate în rezultat este egal cu numărul total de duplicate care apar în primul și în al doilea *SELECT*.

### 3.7.2. Operatorii INTERSECT și INTERSECT ALL

Operatorul *INTERSECT* permite obținerea mulțimii tuplurilor comune a două interogări. Acest operator nu ignoră valorile *NULL*.

**Exemplul 3.84.** Să se găsească departamentele care au funcționari în ambele filiale specificate:

```
SELECT Departament FROM funcționari1
INTERSECT
SELECT Departament FROM funcționari2;
```

Operatorul *INTERSECT* elimină duplicatele în mod automat. Astfel, în interogarea anterioară, dacă un departament există în ambele filiale, acesta va fi afișat doar o singură dată. Pentru a conserva duplicatele se va utiliza operatorul *INTERSECT ALL*.

**Exemplul 3.85.** Să se găsească departamentele (inclusiv repetate) care au funcționari în ambele filiale specificate:

```
SELECT Departament FROM funcționari1
INTERSECT ALL
SELECT Departament FROM funcționari2;
```

### 3.7.3. Operatorii EXCEPT și EXCEPT ALL

Operatorul de diferență a două mulțimi de tupluri se poate realiza prin intermediul operatorului *EXCEPT* (sau *MINUS* pentru Oracle). Aceasta determină tuplurile returnate de prima cerere care nu sunt selectate de către a doua cerere.

**Exemplul 3.86.** Să se găsească departamentele care au funcționari în prima filială și nu în cea de a doua:

```
SELECT Departament FROM funcționari1
EXCEPT
SELECT Departament FROM funcționari2;
```

Pentru a putea fi aplicat operatorul *EXCEPT*, este necesar ca și în cazurile precedente, ambele cereri să aibă același număr de atrbute și de același tip. Numele atrbutelor nu trebuie să fie neapărat aceleiași.

**Exemplul 3.87.** Să se găsească acele departamentele care au mai mulți funcționari din prima filială decât din filiala a doua:

```
SELECT Departament FROM funcționari
EXCEPT ALL
SELECT Departament FROM funcționari2;
```

Numărul de copii duplicate ale unui tuplu din rezultat este egal cu numărul de copii duplicate ale acestui tuplu extras din prima relație minus numărul de copii ale acestui tuplu extras din relația a doua.

Standardul SQL2 specifică că duplicatele sunt eliminate (nu poate fi folosit *DISTINCT*) din rezultatul operatorilor *UNION*, *INTERSECT* și *EXCEPT*. Costul eliminării dupliacelor nu poate fi neglijat. În ce privește acești operatori, uniunea nu poate fi realizată decât prin *UNION*. Intersecția mai poate fi exprimată și prin joncțiune, iar diferența se poate reprezenta mai ușor prin cereri imbricate.

Trebuie menționate, de asemenea, următoarele reguli pentru folosirea operatorilor de mulțimi: Numele atributelor din prima interogare apar în rezultat. Clauza *ORDER BY* apare la sfârșitul ultimei instrucțiuni *SELECT*.

### 3.8. Instrucțiuni de actualizare a bazei de date

Limbajul de manipulare a datelor (DML) este un limbaj care permite actualizarea datelor conținute în baza de date. SQL propune trei instrucțiuni care corespund celor trei tipuri de instrucțiuni de actualizare a bazei de date:

- *INSERT* adăugarea tuplurilor în relații.
- *UPDATE* modificarea conținutului tuplurilor din relații.
- *DELETE* stergerea tuplurilor din relații.

Aceste instrucțiuni funcționează asupra bazei de date aşa ca la începutul executării. Actualizarea efectuată de către alți utilizatori, între debutul și finalul executării, nu sunt luate în considerație (chiar și pentru tranzacțiile validate).

### 3.8.1. Inserarea tuplurilor

Instrucțunea de inserare a datelor poate fi tratată ca o adăugare a unui sau mai multor tupluri într-o relație. Inserarea unui tuplu prin specificarea valorilor acestuia se efectuează cu ajutorul instrucției *INSERT INTO*, care are următoarea sintaxă:

```
INSERT INTO <nume relație>
VALUES (<valoare1>, ..., <valoareN>);
```

Pentru ca inserarea unui tuplu să fie validă, ea trebuie să satisfacă constrângerile impuse de chei, constrângerile de comportament și alte tipuri de constrângerile.

#### Exemplul 3.88.

```
INSERT INTO studenți
VALUES ('P.02123', 'Popescu', 'Ion', '1997-03-27');
```

În această instrucție, valorile atributelor sunt specificate în aceeași ordine în care atributele au fost declarate în instrucțiea *CREATE TABLE*. Dacă nu se cunoaște ordinea atributelor, atunci numele atributelor pot fi specificate conform sintaxei:

```
INSERT INTO <nume relație>(<atribut1>, ..., <atributN>)
VALUES (<valoare1>, ..., <valoareN>);
```

#### Exemplul 3.89.

```
INSERT INTO studenți(Nume, Prenume, Data naștere,
ID_student)
VALUES ('Popescu', 'Ion', '1997-03-27', 'P.02123');
```

Trebuie menționat că în relație se inserează un tuplu complet, adică numărul valorilor din lista de valori trebuie să fie același cu numărul de atrbute din relație. În afară de aceasta, tipurile valorilor trebuie să fie compatibil cu tipurile atributelor în care se stochează.

**Exemplul 3.90.** În cazul în care, pentru un anumit atrbut, nu se cunoaște valoarea și bineînteles dacă sunt respectate constrângerile de integritate, în lista de valori se folosește cuvântul-cheie *NULL*.

```
INSERT INTO studenți(Nume, Prenume, Data naștere,
ID_student)
VALUES ('Popescu', 'Ion', NULL, 'P.02123');
```

Dacă la crearea relației unui atribut i-a fost asociată o valoare implicită, pentru inserarea ei se folosește cuvântul-cheie *DEFAULT*.

**Exemplul 3.91.** În cazul în care se folosește *DEFAULT*, dar atributul nu are o valoare implicită definită, în atribut se va inseră o valoare nulă.

```
INSERT INTO studenți(Nume, Prenume, Data naștere,
ID_student)
VALUES ('Popescu', DEFAULT, '1997-03-27', 'P.02123');
```

Se poate, de asemenea, inseră deodată o mulțime de tupluri. Dar această mulțime de orice cardinalitate trebuie să fie rezultatul unei cereri de extragere. Bineînțeles că schema mulțimii de tupluri extrase trebuie să fie compatibilă cu schema relației în care se adaugă tuplurile. În acest caz, se vorbeste despre *inserția calculată*. Sintaxa instrucției de inserare poate fi:

```
INSERT INTO <nume relație>[(<listă atribute>)]
<cerere SELECT>;
```

Modul de execuție al acestei cereri este următorul: Mai întâi se execută cererea *SELECT*, apoi tuplurile rezultatului sunt inserate în relație.

În cazul în care un tuplu din rezultatul cererii *SELECT* nu satisface constrângerile de integritate ale relației în care se face inserarea, întreaga instrucție eșuează (nici celealte tupluri nu sunt inserate).

În acest tip de cereri, trebuie acordată atenție atributelor-contoare sau atributelor-autonumerice, deoarece inserarea unei valori pentru un atribut de acest tip poate scrie și valoarea ce conține atributul său omolog într-o altă relație de origine, adică poate să-l incrementeze incorect.

Versiunile mai recente ale Oracle permit plasarea în lista *VALUES*, în locul valorilor, a cererilor *SELECT*. Fiecare cerere trebuie să returneze doar o singură valoare.

**Exemplul 3.92.** Cererile trebuie încadrate cu paranteze. Această opțiune nu este portabilă și ar trebui, prin urmare, evitată în programe.

```
INSERT INTO studenți(Nume, Prenume, Data naștere,
ID_student)
VALUES ('Popescu2', 'Ion', '1997-03-27',
(SELECT ID_student + 1 FROM studenți
WHERE Nume = 'Popescu'));
```

### 3.8.2. Modificarea tuplurilor

În general, valorile datelor stocate în baza de date trebuie să se modifice atunci când, în lumea exteroară, se produc schimbările respective. În fiecare caz, valorile atributelor din relațiile bazei de date se actualizează pentru a menține baza de date în calitate de model precis al lumii reale.

Instrucțiunea *UPDATE* se utilizează pentru formularea cererilor de modificare a valorilor unui sau mai multor atributelor ale unui sau mai multor tupluri existente într-o relație. Sintaxa generală a instrucțiunii de modificare a datelor poate avea două forme:

```
UPDATE <nume relație>
SET <atribut1>=<expresie1>,...,<atributN>=<expresieN>
WHERE <predicat>;
```

sau

```
UPDATE <nume relație>
SET (<atribut1>,...,<atributN>) = (SELECT...)
WHERE <predicat>;
```

Valorile atributelor sunt actualizate pentru toate tuplurile care satisfac predicatul din clauza *WHERE*. Predicatul poate conține și subinterrogări. Clauza *WHERE* este facultativă. Dacă aceasta lipsește, sunt actualizate toate tuplurile din relație. Clauza *SET* specifică atributelor care vor fi actualizate și calcularea valorilor noi.

Varianta a doua a instrucțiunii *UPDATE* este utilă în special când se dorește modificarea unui număr de tupluri în baza datelor dintr-o relație. Cu o instrucțiune pot fi modificate mai multe atributelor. Valorile atributelor sunt returnate de instrucțiunea *SELECT*.

**Exemplul 3.93.** Prima instrucție îl transferă pe funcționarul *Popescu* în departamentul *Software*, a doua - mărește salariul tuturor funcționarilor cu 10%, iar a treia instrucție mărește cu 10% salariul funcționarilor din departamentul *Software*:

```
UPDATE funcționari SET Departament = 'Software'  
WHERE Nume = 'Popescu';  
UPDATE funcționari SET Salariu = Salariu*1.1;  
UPDATE funcționari SET Salariu = Salariu*1.1  
WHERE Departament = 'Software';
```

Evident, că instrucțunea *UPDATE* nu extrage niciun rezultat. Pentru a cunoaște ce tupluri și cum vor fi modificate, trebuie mai întâi de examinat rezultatul unei cereri de selecție care utilizează același predicat de selecție și numai după aceasta se poate apela la instrucțunea *UPDATE*.

**Exemplul 3.94.** Lui *Popescu* să i se ofere un salariu cu 10% mai mare decât salariul mediu al inginerilor din departamentul *Software*

```
UPDATE funcționari SET Salariu =  
(SELECT AVG(Salariu) *1.1 FROM funcționari  
WHERE Departament = 'Software' AND Post='inginer')  
WHERE Nume = 'Popescu';
```

Trebuie remarcat faptul că salariile vor fi calculate pentru valorile pe care le aveau salariile la începutul executării instrucției *UPDATE* și că modificările efectuate asupra bazei pe parcursul executării acestei instrucții nu se iau în considerare.

### 3.8.3. Suprimarea tuplurilor

O ștergere este exprimată în același mod ca o interogare. Pot fi șterse doar tupluri complete dintr-o relație existentă, adică nu pot fi șterse anumite valori ale atributelor. Tuplurile sunt specificate utilizând o condiție de căutare, cu excepția cazului în care se dorește ștergerea tuturor tuplurilor unei relații. Sintaxa pentru instrucția de ștergere este:

```
DELETE FROM <nume relație> WHERE <predicat>;
```

Clauza *WHERE* indică tuplurile care trebuie să fie suprimate. Predicatul de după clauza *WHERE* poate conține subinterrogări. Această clauză este

facultativă. Dacă aceasta lipsește, sunt suprimate toate tuplurile din relația specificată.

Trebuie menționat că o instrucție *DELETE* operează numai asupra unei singure relații. Dacă se dorește ștergerea tuplurilor din mai multe relații, se utilizează câte o instrucție pentru fiecare relație. Predicatul din clauza *WHERE* poate fi la fel de complicat ca și în cazul unei instrucții *SELECT*.

**Exemplul 3.95.** Să se șteargă toate datele despre funcționari angajați în postul de ingineri:

*DELETE FROM funcționari WHERE Funcție = 'Inginer';*

Deși într-un moment dat cu un *DELETE* pot fi șterse doar tuplurile dintr-o singură relație, numărul relațiilor într-un *SELECT...FROM...WHERE* imbricat în clauza *WHERE* a unui *DELETE* poate fi oricare.

Îndată ce tuplurile sunt eliminate, utilizând o cerere de suprimare, operația nu mai poate fi întoarsă. Astfel, ca și în cazul modificării, pentru vizualizarea tuplurilor spre eliminare, mai întâi se examinează rezultatele unei cereri de selecție ce utilizează același criteriu și după aceea se formulează o cerere de suprimare. Dar independent de acest fapt este necesară menținerea unor copii de siguranță a datelor.

## 4. SQL: Limbajul de definire a datelor

---

Limbajul de definire a datelor, fiind parte componentă a limbajului SQL, este utilizat, în cea mai mare măsură, de către administratorul bazei de date. Clauzele acestui limbaj sunt folosite pentru definirea structurii corespunzătoare a obiectelor bazei de date: atribute, constrângeri de integritate, scheme, viziuni, indecsi, sinonime etc. Instrucțiunile acestui limbaj, în afară de aceasta, sunt utilizate pentru modificarea și suprimarea obiectelor create.

### 4.1. Definirea datelor în SQL

Tipurile de date SQL se clasifică conform standardului ISO în 3 grupuri: date numerice, secvențe de caractere și date temporale. Este cunoscută o varietate de sinonime valide în diverse SGBD-uri. Trebuie menționat că tipurile de date diferă de la un SGBD la altul și, deseori, sistemele nu respectă norma SQL2.

#### 4.1.1. Tipuri de date numerice

Standardul ISO distinge două categorii de atribute numerice: *attribute numerice exacte* și *attribute numerice flotante*.

Tipuri de prima categorie:

- Numere întregi: *INTEGER*, numere întregi asupra 4 octeți, poate reprezenta numere între -2147483648 și 2147483648; *SMALLINT*, numere întregi asupra 2 octeți, care pot reprezenta numere între -32768 și 32767; *TINYINT*, număr asupra 1 octet, care poate reprezenta numere de la 0 la 255.

- Numere zecimale cu un număr fixat de zecimale:  $NUMERIC(M,D)$  și  $DECIMAL(M,D)$ , unde  $M$  specifică numărul total de cifre, iar  $D$  numărul de cifre zecimale. Norma SQL2 impune atributelor de tip  $NUMERIC$  să accepte numere cu un număr exact de cifre zecimale, în timp ce atributele de tip  $DECIMAL$  nu.

**Exemplul 4.1.** Salariu  $DECIMAL(6,2)$  definește un atribut numeric Salariu. Valorile maxime constau din 6 cifre cu 2 cifre zecimale (adică 4 cifre înainte de punctul zecimal).

Numere din a doua categorie, numere aproximative cu virgulă flotantă:

- $REAL$ , numere cu precizie simplă, cu cel puțin 7 cifre semnificative.
- $DOUBLE PRECISION$  sau  $FLOAT$ , reprezintă numere cu dublă precizie cu cel puțin 15 cifre semnificative.

Definiția tipurilor de numere aproximative depinde de SGBD (numărul de cifre semnificative variază). Pentru detalii trebuie consultat manualul SGBD-ului utilizat.

Oracle are un singur tip de date numerice  $NUMBER$ . Pentru motive de compatibilitate, Oracle poate utiliza tipurile SQL2, dar ele sunt aduse la tipul  $NUMBER$ . În cazul în care se definește un atribut numeric, se specifică numărul maximal de cifre și de zecimale pe care valoarea acestui atribut poate avea:

$NUMBER$   
 $NUMBER(M)$   
 $NUMBER(M,D)$

Dacă parametrul  $D$  nu este specificat, în mod implicit, este considerat 0. Valoarea absolută a numărului trebuie să fie inferioară  $10^{128}$ .  $NUMBER$  este un număr cu virgulă flotantă (nu se precizează numărul de cifre înainte de punctul zecimal) care poate avea până la 38 cifre semnificative.

#### 4.1.2. Tipuri de date secvențe de caractere

Toate SGBD-urile susțin date în formă de secvențe de caractere. Constantele în formă de secvențe de caractere sunt luate între două apostrofuri, (''). Dacă însăși secvența conține un apostrof, acesta trebuie să fie dublu. De exemplu, 'Anna ''s database'.

Există două forme de prezentarea a acestui tip de date: *CHAR* și *VARCHAR*.

Forma *CHAR* este aplicată pentru attribute asociate cu date de lungime constantă și care nu conțin mai mult de 255 caractere. Declarația acestei forme de secvențe de caractere se supune sintaxei:

*CHAR(<lungime>)*

unde *<lungime>* este lungimea maximală pe care o poate avea secvența de caractere. Lungimea se specifică în mod obligatoriu. Atribuirea unei secvențe cu lungime mai mare este refuzată. O secvență mai scurtă se completează cu spații, proprietate importantă pentru compararea secvențelor.

Forma *VARCHAR* se utilizează pentru secvențe cu lungime variabilă. Toate SGBD-urile au o lungime maximală proprie pentru secvențele sale. Sintaxa acestei forme este:

*VARCHAR(<lungime>)*

unde *<lungime>* indică lungimea maximală a secvenței de caractere.

Tipuri secvențe de caractere în Oracle sunt tipurile respective din SQL2, numai că tipul *VARCHAR* în Oracle se numește *VARCHAR2* (Dimensiunea maximală este de 2000 de caractere)

Există și date binare care permit înregistrarea unor date, cum sunt imaginile, sunetele, adică date de talie mare cu divers format. Standardul SQL2 propune tipul *BIT VARYING* pentru secvențe foarte mari. Diverse SGBD-uri furnizează tipul binar de date propriu: *LONG RAW* în Oracle, *IMAGE* în Sybase, *BYTE* în Informix etc.

#### 4.1.3. Tipuri de date temporale

Datele temporale în SQL2 au trei forme:

- *DATE* rezervează câte 2 cifre pentru lună și zi și 4 cifre pentru an.
- *TIME* pentru desemnarea orelor, minutelor și secundelor: *OO:MM:SS*.
- *TIMESTAMP* permite specificarea precisă a datei specificându-se orele, minutele și secundele (pentru secunde se permit 6 cifre după virgulă): *AAAA-LL-ZZ OO:MM:SS*.
- *INTERVAL* permite specificarea unui interval de timp

Trebuie menționat că pentru *DATA* standardul ISO nu specifică o ordine anumită a componentelor.

Oracle oferă tipul *DATE* ca în SQL2, însă pentru Oracle un atribut de tipul *DATE* mai include și ore, minute și secunde. O constantă de acest tip este o secvență de caractere între apostrofuri. Formatul depinde de opțiunea pe care administratorul a ales-o la momentul creării bazei de date.

### 4.2. Definirea schemei bazei de date

Principalele funcții ale limbajului SQL în calitate de limbaj de definire a datelor sunt, printre altele: crearea, modificarea și suprimarea schemelor relaționale, a atributelor și a constrângerilor de integritate. Toate aceste definiții care descriu baza de date se păstrează într-un loc special numit *dicționarul de date*, unde sunt accesate de către SGBD.

#### 4.2.1. Definirea constrângerilor de integritate

O caracteristică dezirabilă a unui SGBD relațional constă în faptul că specificarea constrângerilor poate fi integral realizată, utilizând limbajul de definiție a datelor, astfel constrângerile devenind o componentă a schemei bazei de date. Evident că, în acest caz, utilizatorului trebuie să fie oferite instrumente de specificare și modificare a constrângerilor de integritate.

Constrângerile sunt evaluate atunci când se inserează tupluri, când se suprimă tupluri sau când se modifică valoarea unui atribut pentru vreun tuplu. Astfel, în orice moment, consistența bazei de date este asigurată. Există un mecanism care permite gruparea mai multor operații cu scopul de a verifica constrângerile nu pentru o operație aparte, ci pentru o mulțime de operații. Evident că aceste verificări chiar dacă dau un răspuns negativ sunt costisitoare din punct de vedere computerial în funcție de dimensiunile bazei de date și de complexitatea constrângerilor.

Constrângerile de integritate pot fi clasificate după nivelul la care sunt definite în:

- Constrângerile la nivel de atribut.
- Constrângerile la nivel de relație care pot acționa asupra unei combinații de attribute.
- Constrângerile la nivel de bază de date care pot acționa asupra unei combinații de attribute din diverse relații.

Trebuie menționat că constrângerile *NOT NULL* se pot defini doar la nivel de atribut, dar cele *UNIQUE*, *CHECK*, *FOREIGN KEY* și *PRIMARY KEY* pot fi definite atât la nivel de atribut, cât și la nivel de relație. La nivel de bază de date pot fi definite constrângerile *CHECK*.

O definiție de constrângere poate avea următorul format general:

*[CONSTRAINT <nume constrângere>] <tip> <definiție>*

unde cuvântul *CONSTRAINT* și numele constrângerii sunt opționale, iar *<tip>* indică tipul constrângerii. În cazul în care se utilizează numele constrângerii, acesta trebuie să fie unic în toată baza de date (deși pot fi în relații diferite, nu pot avea același nume). În sfârșit, *<definiție>* descrie completamente constrângerea. Deoarece definițiile constrângerilor depind de tipul acestora, ele vor fi examinate separat (figura 4.1).

Constrângere	Descriere
<i>NOT NULL</i>	Specifică că un atribut nu poate avea valoare nulă.
<i>DEFAULT</i>	Specifică o valoare implicită pentru un atribut.
<i>UNIQUE</i>	Specifică un atribut sau o combinație de atribute a căror valoare trebuie să fie unică pentru toate tuplurile relației.
<i>PRIMARY KEY</i>	Identifică unic fiecare tuplu
<i>FOREIGN KEY</i>	Stabilește și forțează o legătură de tip cheie externă dintre o mulțime de atribute și o mulțime de atribute dintr-o relație referită.
<i>CHECK</i>	Specifică o condiție de comportament al valorilor datelor care trebuie să fie adevărată.

Figura 4.1. Tipuri de constrângeri de integritate a datelor

Fiecarei constrângeri i se poate da un nume, lucru util atunci când, la un moment dat (salvări, restaurări, încărcarea BD) se doresc dezactivarea uneia sau mai multora dintre acestea. Constrângerile sunt simplu de referit, dacă li se dă un nume sugestiv. Astfel, se prefigurează numele fiecărei constrângeri cu tipul său:

- *pk\_ (PRIMARY KEY)* pentru cheile primare.
- *un\_ (UNIQUE)* pentru cheile alternative.
- *nn\_ (NOT NULL)* pentru valori obligatorii ale atributelor.
- *ck\_ (CHECK)* pentru reguli de validare a constrângerilor de comportament.
- *fk\_ FOREIGN KEY*) pentru cheile externe.

#### 4.2.1.1. Constrângerea NOT NULL

Constrângerea *NULL* permite dezvoltarea de concepte folosind logica cu mai mult de 2 valori, unde *NULL* indică că obiectul în cauză "nu are valoare" sau aceasta "nu se cunoaște". Astfel, valoarea *NULL* este o valoare particulară, care nu reprezintă valoarea 0 în cazul numerelor și nu reprezintă spațiu în cazul secvențelor de caractere, ci lipsă de date.

Această valoare *NULL* poate apărea când nu se cunosc respectivele date, de exemplu, nu se cunoaște telefonul fix al unei persoane. Nu orice atribut poate avea valoarea *NULL*, de exemplu, denumirea unui departament. În astfel de situații, la definirea relațiilor, se impune atributului constrângerea *NOT NULL*, însemnând că acest atribut nu poate lua valoare *NULL* în orice tuplu din relație.

**Exemplul 4.2.** Constrângerea *NOT NULL* se aplică atributului *Nume*:

...*Nume* *VARCHAR(15)* *NOT NULL*, ...

#### 4.2.1.2. Constrângerea *DEFAULT*

Unui atribut îi se poate asigna o valoare implicită utilizând opțiunea *DEFAULT*. Această opțiune previne introducerea unor valori *NULL* în relație în cazul inserării unui tuplu care nu specifică o valoare pentru atributul în cauză. În cazul în care, la inserarea unui tuplu, nu se specifică valoarea atributului, atunci acesta primește valoarea implicită (dacă a fost definită) sau valoarea *NULL*, dacă nu a fost definită o valoare implicită pentru atributul respectiv, dar sunt admise valori *NULL*. Dacă nu a fost definită o valoare implicită și nici nu sunt admise valori *NULL* se generează eroare.

**Exemplul 4.3.** Constrângerea *NOT NULL* se aplică atributului *Nume*:

...*Nume* *VARCHAR(15)* *NOT NULL*, ...

#### 4.2.1.3. Definirea cheilor primare

Este cunoscut faptul că o relație poate avea mai multe chei, dar numai una din ele poate fi declarată *cheie primară*. Cheia primară se specifică în definiție cu opțiunea *PRIMARY KEY*. O cheie primară poate fi constituită dintr-un singur atribut sau din mai multe atribute. În orice caz, toate atributele care fac parte din cheia primară, deoarece acestea vor identifica un mod univoc un tuplu trebuie să fie declarate *NOT NULL* (se bucură de constrângerea entității), dacă nu este indicată în mod explicit, SGBD-ul face acest lucru în mod automat.

În cazul în care constrângerea *PRIMARY KEY* constă dintr-un singur atribut, acest fapt poate fi indicat direct în definiția atributului, dar și ca o definiție independentă, utilizând sintaxa următoare:

...[*CONSTRAINT < nume constrângere >*] *PRIMARY KEY (<listă de attribute>)*...

**Exemplul 4.4.** Atributul *DepartNumăr* este declarat cheie primară:

.... *CONSTRAINT pk\_dept*  
          *PRIMARY KEY(DepartNumăr)*...

De regulă, numele constrângerii cheii primare este prefixat cu *pk\_*.

#### 4.2.1.4. Definirea cheilor externe

O cheie externă stabilește o legătură de integritate referențială, de obicei, dintre două relații.

Este obligatoriu ca relația referită să aibă o cheie primară (sau, alternativ, că există o constrângere *UNIQUE*), și să fie exact această cheie primară (sau lista de attribute cu constrângerea *UNIQUE*) la care face referire cheia externă. Trebuie menționat că pentru a crea o referință la o altă relație, relația referită trebuie, deja, să existe, adică trebuie mai întâi creată.

La fel ca și în cazul cheii primare, specificarea unei chei externe poate fi realizată prin două moduri: inclusiv în definiția unui atribut:

...<*atribut*> <*tip*> *REFERENCES*  
          <*relație referită*>[(<*attribute referite*>)]...

**Exemplul 4.5.** Atributul *DepartNumăr* este declarat cheie externă în raport cu relația *departamente*:

*DepartDenumire* *VARCHAR(14)*  
                  *REFERENCES departamente*

A doua opțiune, singura utilizabilă în cazul când cheia externă constă din mai multe attribute (dar validă și în cazul unui singur atribut) definește cheia externă ca o constrângere aparte:

...[*CONSTRAINT <nume constrângere>*]  
*FOREIGN KEY (<listă de atributे>)*  
*REFERENCES <relație referită>*  
*[(<atributе referite >)]*  
*[ON DELETE CASCADE]...*

De obicei, pentru nominalizarea constrângerilor cheilor externe, se utilizează prefixul *fk\_*. Clauza opțională *ON DELETE CASCADE* indică cum reacționează relația pentru menținerea integrității în cazul eliminării unui tuplu din relația referită. Atunci când este specificată această clauză, în cazul eliminării unui tuplu „părinte” (referit), se șterg, de asemenea, și „fiilii”(de exemplu, dacă este eliminat un departament, automat sunt eliminate toți funcționarii care lucrează în acest departament, pentru menținerea integrității referențiale). Dacă această clauză lipsește, sistemul de gestiune nu va șterge tuplul-părinte.

Oferirea nume constrângerilor este importantă în cazul în care se dorește ștergerea sau modificarea constrângerii.

Trebuie menționat că, deși Oracle nu le suportă, în SQL2 se permit alte opțiuni atunci când se șterge sau se actualizează un tuplu-părinte:

*[ON {DELETE|UPDATE} SET {NULL|CASCADE}]*

Astfel, în cazul ștergerii sau actualizării, se alege una din opțiunile:

- *SET NULL* – este plasată valoarea nulă pentru cheia externă.
- *SET CASCADE* – se șterg plurile.

#### 4.2.1.5. Constrângerea UNIQUE

O mulțime de atributе (sau unul) poate fi declarată având proprietatea *UNIQUE* care denotă faptul că nu acceptă valori dupăcat. Acest lucru este util, printre altele, pentru atributele de tip cheie candidat, care impune respectarea unicității valorilor, deoarece, într-o relație, este posibilă existența unei singure chei primare. O relație poate include mai multe constrângerile de tip *UNIQUE*. Sintaxa la nivel de atribut este

*<descriere atribut> UNIQUE*

iar la nivel de relație este

*[CONSTRAINT <nume constrângere>]  
UNIQUE(<listă de atribute>)*

**Exemplul 4.6.** Pentru a informa SGBD-ul că nu sunt două nume de departamente identice la o singură locație se poate scrie:

*...DepartDenumire VARCHAR(14) UNIQUE*

pe când a doua constrângere garantează că nu sunt două nume de departamente identice:

*CONSTRAINT un\_depart\_oraș  
UNIQUE(DepartDenumire, Oraș)*

După cum s-a menționat anterior, o constrângere a cheii externe poate referi o listă de atribute afectate de o restricție *UNIQUE*. În caz general, atributele care prezintă constrângerile de unicitate pot lua valori *NULL*. În afară de aceasta, aceste atribute pot fi actualizate. În mod automat, se creează și un index pe atributele definite cu constrângerea *UNIQUE*, ceea ce duce la mărirea vitezei de interogare pe relație.

#### 4.2.1.6. Constrângerile de comportament **CHECK**

Aceste constrângerile se utilizează pentru realizarea verificărilor valorilor care se introduc într-o relație. În funcție de faptul dacă constrângerea se specifică în declarația atributului sau în mod izolat, sintaxa generală este, după cum urmează:

*<descriere atribut> CHECK (<condiție>)*

sau

*[CONSTRAINT <nume constrângere>]  
CHECK (<condiție>)*

Numele constrângerii este similar celor anterioare. Constrângerile de tipul *CHECK* pot afecta mai multe atribute. În cazul în care constrângerea antrenează un singur atribut (e constrângere de comportament al domeniului), ea poate fi specificată la finalul descrierii acestuia. Clauza *CHECK* este urmată de o condiție logică între paranteze. În condițiile, pot fi inclusi operatori de comparație, (*<*, *>*, ...), operatori logici, (*AND*, *OR*, *NOT*) etc.

**Exemplul 4.7.** Fie date două constrângeri, una de comportament al domeniului, iar alta de comportament al tuplului:

```
... Salariu DECIMAL (6,2) CHECK(Salariu>0)
...CONSTRAINT ck_vec_mun
    CHECK((Vârstă >=14)
        AND (Vechime_muncă>Vârstă +3)).
```

Precum se poate vedea, dacă constrângerea este indicată în formă independentă, ca cea de a doua din exemplu, ea poate verifica mai multe attribute. Dacă, ca în primul caz, atunci se specifică în declarația atributului și se referă doar la domeniul acestuia.

#### 4.2.1.7. Constrângeri de comportament CHECK la nivel de bază de date

Constrângerile la nivel de bază de date se numesc aserțiuni. Aserțiunile sunt predicate logice care trebuie să fie satisfăcute de baza de date considerată ca un tot (nu ca în cazul constrângerilor la nivel de tuplu sau la nivel de relație) și care pot antrena orice număr de attribute din diverse relații. Aserțiunile permit expresii condiționate complexe care se presupun că sunt satisfăcute de baza de date întotdeauna [Bernstein80]. Ele nu se specifică în instrucțiunile de creare a relațiilor, dar pot fi create cu instrucțiuni independente *CREATE ASSERTION*.

Pentru formularea aserțiunilor, SQL dă posibilitatea folosirii unei instrucțiuni specifice, cu sintaxa:

```
CREATE ASSERTION <nume aserțiune>
    CHECK <predicat>;
```

În timp ce alte tipuri de constrângeri (la nivel de atribut sau relație) sunt asociate cu elementele schemei (atribut, relație), aserțiunile sunt ele însăși, elemente ale schemei. Bineînteles că predicatele logice pot fi complexe și pot include diverse instrucțiuni de selecție dintr-un număr de relații din baza de date. Crearea unei aserțiuni de către SGBD este însotită de verificarea validității ei asupra stării bazei de date. Condiția din aserțiune trebuie să fie întotdeauna adevărată. Mai departe, se permit numai inserările, modificările și suprimările în baza de date care nu contravin predicatului. Aserțiunile sunt verificate de fiecare dată când una dintre

relațiile care le menționează este modificată. Această verificare a validității operației de actualizare necesită un cost computerial destul de înalt.

Există, totuși, o deosebire esențială în raport cu celelalte tipuri de constrângeri: orice atribut referit în aserțiune trebuie introdus menționând explicit relația din care face parte într-o instrucțiune *SELECT...FROM...*

**Exemplul 4.8.** Fie schema bazei de date:

```
funcționari(FuncționarID, FuncționarNume,
            Departament, Salariu)
departamente(DepartamentID, FuncționarID, Oraș)
```

Următoarea instrucțiune creează o constrângere de comportament al bazei de date, care impune ca salariul maximal al funcționarilor departamentului situat în Chișinău să fie mai mic decât orice salariu al funcționarilor departamentului 'Software'.

```
CREATE ASSERTION SalariuMaiMic
  CHECK ( NOT EXIST ( SELECT *
    FROM funcționari AS f, departamente AS d
    WHERE f.FuncționarID=d.FuncționarID
    AND Oraș='Chișinău' AND
    f.Salariu > (SELECT MAX(Salariu)
      FROM funcționari
      WHERE Departament='Software')
    ));
```

#### 4.2.2. Definirea atributelor

Pentru a defini un atribut, trebuie specificate numele său, tipul de bază și o constrângere (optională) simplă asupra posibilelor valori. Tipul de date indică domeniul general al atributului care, la rândul său, poate fi constrâns printr-o restricție optională. Schema generală a definiției unui atribut este:

*<nume atribut> <tip> [<constrângere>], unde*

*<nume atribut>* este numele atributului;

*<tip>* reprezintă tipul atributului, care este unul din tipurile menționate anterior;

*<constrângere>* reprezintă o restricție asupra atributului.

Constrângerea poate indica o valoare definită implicit (*DEFAULT*) sau poate indica dacă acesta acceptă (*NULL*) sau nu (*NOT NULL*) valori nule. De asemenea, constrângerea poate indica dacă atributul are o valoare unică (*UNIQUE*) sau este o cheie primară (*PRIMARY KEY*), sau este o cheie externă (*FOREIGN KEY...REFERENCES*), sau este o constrângere de comportament al domeniului (*CHECK*).

Astfel, tipul atributului specifică valorile sale valide, iar tipurile de date sunt specificate, în funcție de SGBD utilizat.

**Exemplul 4.9.** Sunt definite două atribute: *Nume* și *Prenume*. Atributul *Nume* este de tip secvență de caractere de lungime variabilă și lungimea maximă 25, și nu poate accepta valori nule. Atributul *Prenume* poate avea lungimea maximă 30.

...*Nume* *VARCHAR(25)* *NOT NULL*...

...*Prenume* *VARCHAR(30)*)...

Dacă nu se specifică nicio restricție asupra atributului, atunci poate avea orice valoare compatibilă cu tipul de date, inclusiv nulă.

Trebuie menționată o diferență importantă între practică și teorie: aici se admite că unele atribute pot să nu ia valori, ceea ce este diferit de o secvență vidă sau de 0. Când se scrie *NULL*, SGBD-ul subînțelege absența valorii.

#### 4.2.3. Crearea schemei relaționale

Instrucțiunea de creare a unei scheme relaționale conține comanda *CREATE TABLE*. Ea trebuie să includă, de asemenea, numele relației, numele de atribute care formează schema relațională, cheia primară, posibilele chei externe și restricțiile impuse asupra valorilor pe care le pot lua atributele. Orice definiție a unei componente se separă de următoarea prin virgulă. Sintaxa generală a instrucțiunii este:

```

CREATE TABLE <nume relație>(
  <definiție atribut 1>[, <definiție atribut 2>...],
  <definiție cheie primară>,
  [<definiție cheie externă 1>
    [, <definiție cheie externă 2>...],]
  [<definiție cheie secundară1>
    [, <definiție cheie secundară 2>...],]
  [<definiție constrângere 1>
    [, <definiție constrângere 2>...]]);

```

Ordinea de realizare a definiției este mai flexibilă, cu toate că este evident că înainte de a defini un atribut, ca parte componentă a unei chei primare sau externe, sau de a defini o constrângere asupra valorilor posibile, este necesară definire acestui atribut.

**Exemplul 4.10.** Se creează o schemă relațională, unde atributul *Adresă* este o secvență de caractere de lungime fixă și ia valoarea implicită ‘necunoscută’.

```

CREATE TABLE funcționari(
  Nume VARCHAR(25) NOT NULL,
  Prenume VARCHAR(30),
  Adresă CHAR(50) DEFAULT 'necunoscută');

```

In schema din exemplul 4.10, se forțează un atribut să ia o valoare prin specificarea unei valori implicate cu ajutorul opțiunii *DEFAULT*. Atunci când, în relația *funcționari*, se va insera un tuplu fără a fi indicată o adresă, sistemul va insera automat valoarea ‘necunoscută’.

Deseori, se îmbină constrângerile *UNIQUE* și *NOT NULL*. În consecință, atributul asupra căruia se aplică aceste opțiuni nu acceptă valori repetitive și nici valori nule. Astfel, atributul este definit *cheie secundară* al schemei respective.

**Exemplul 4.11.** Se creează o schemă relațională cu o constrângere de comportament al domeniului. Atributul *Data\_naștere* este de tip *DATE* și nu ia valori mai mici de '1980-01-01' și mai mari de '1990-12-31'.

```
CREATE TABLE funcționari(  
    Nume VARCHAR(25) NOT NULL,  
    Prenume VARCHAR(30),  
    Adresă CHAR(50) DEFAULT 'necunoscută',  
    Data_naștere DATE  
        CHECK(Data_naștere > '1980-01-01'  
            AND Data_naștere<'1990-12-31'));
```

Trebuie menționat că SGBD-ul verifică constrângerile definite de fiecare dată când se inserează tupluri noi în relație, se suprimă tupluri sau se modifică valorile unui tuplu. Bineînțeles, că această verificare necesită timp, din care motiv randamentul sistemului poate fi negativ afectat, dacă se definesc un număr mare de constrângerii. În secțiunea consacrată constrângerilor, au fost examinate mai detaliat metode mai eficiente de definire a constrângerilor.

**Exemplul 4.12.** Atributul *ID\_funcționar* este declarat cheie primară a relației *funcționari*:

```
CREATE TABLE funcționari(  
    ID_funcționar CHAR(6)  
    Nume VARCHAR(25) NOT NULL,  
    Prenume VARCHAR(30) NOT NULL,  
    Vârstă INTEGER CHECK(Vârstă>=18  
        AND Vârstă<30),  
    PRIMARY KEY (ID_funcționar));
```

De asemenea, se poate specifica că o mulțime de atribute iau valori unice. Aceasta permite, de fapt, declararea *cheilor secundare*. Astfel, aplicând opțiunea *UNIQUE*, se poate indica că doi funcționari nu pot avea același nume și același prenume. Menționăm că opțiunea *UNIQUE* nu se aplică asupra valorilor necunoscute, adică *NULL*.

**Exemplul 4.13.** Multimea de atribute  $\{Nume, Prenume\}$  este declarata cheie secundara.

```
CREATE TABLE funcționari (
    ID_funcționar CHAR(6)
    Nume VARCHAR(25) NOT NULL,
    Prenume VARCHAR(30) NOT NULL,
    Vârstă INTEGER CHECK(Vârstă >= 18
        AND Vârstă < 30),
    PRIMARY KEY (ID_funcționar),
    UNIQUE(Nume, Prenume));
```

Norma limbajului SQL2 permite declararea cheilor externe ale unei relații. Cu alte cuvinte se indică multimea de atribute care, univoc, determină cheia altrei relații. Cheia externă se definește similar cheii primare, indicând clauza *FOREIGN KEY* urmată de lista de atribute ce alcătuiesc cheia externă separate prin virgulă, urmată de cuvântul rezervat *REFERENCES* care indică numele relației referite.

**Exemplul 4.14.** Atributul *ID\_departament* este declarat cheie externă, care se referă la cheia primară a relației *departamente*. Atributul *Denumire* a relației *departamente* este definit cheie externă.

```
CREATE TABLE departamente (
    ID_departament CHAR(8) PRIMARY KEY,
    Denumire VARCHAR(15) UNIQUE NOT NULL);
```

```
CREATE TABLE funcționari (
    ID_departament CHAR(6)
    Nume VARCHAR(25) NOT NULL,
    Prenume VARCHAR(30) NOT NULL,
    Vârstă INTEGER CHECK(Vârstă >= 18
        AND Vârstă < 30),
    PRIMARY KEY (ID_funcționar),
    UNIQUE(Nume, Prenume),
    FOREIGN KEY (ID_departament)
        REFERENCES departamente);
```

De fiecare dată când SGBD-ul actualizează baza de date creată de exemplul 4.14, va verifica dacă nu sunt afectate legăturile dintre relațiile *departamente* și *funcționari*. Pentru ca definiția să fie corectă, atributele care compun această definiție și relația referită trebuie să existe anterior. De aceea, existența cheilor externe restrâng ordinea de creare a schemelor relaționale. Adică îndată ce relația  $r_1$  are o cheie externă a relației  $r_2$ , ultima trebuie să fie cheie primară.

#### 4.2.4. Modificarea și suprimarea schemei relaționale

Crearea schemelor relaționale nu este decât prima etapă în viața unei scheme logice a bazei de date. Structura unei relații, create cu instrucțiunea *CREATE TABLE*, poate fi modificată prin adăugarea, modificarea sau suprimarea atributelor sau constrângerilor de integritate. Cu acest scop, se utilizează instrucțiunea *ALTER TABLE*. În continuare, sunt examinate opțiunile disponibile pentru această instrucțiune.

##### 4.2.4.1. Adăugarea, modificarea și suprimarea atributelor

Standardul SQL2 permite adăugarea, suprimarea și modificarea unui atribut din schema. Cu toate acestea, Oracle, până la versiunea 8.0 inclusiv, permite numai adăugarea și modificarea atributelor, dar nu și stergerea. Pentru adăugarea unui atribut, care, de obicei, se adaugă la sfârșitul definițiilor celor existente, se utilizează următoarea sintaxă:

```
ALTER TABLE <nume relație>
    ADD <definiție atribut>;
```

**Exemplul 4.15.** În relația *funcționari*, trebuie inclusă și adresa funcționarului. Următoarea instrucțiune adaugă atributul *Adresă* în această relație:

```
ALTER TABLE funcționari
    ADD Adresă VARCHAR(50);
```

Atributul nou se introduce în schema relației, evident, pentru toate tuplurile. Tuplurile deja existente în relație vor include valoarea *NULL* pentru atributul adăugat sau valoarea implicită, dacă este utilizată clauza *DEFAULT*. Nu este posibilă adăugarea unui atribut definit cu

constrângerea *NOT NULL* și fără valoarea implicită, dacă relația conține date.

**Exemplul 4.16.** Prima instrucțiune este incorectă, a doua - corectă:

*ALTER TABLE funcționari*

*ADD Adresă VARCHAR(50) NOT NULL;*

*ALTER TABLE funcționari ADD Adresă VARCHAR(50)*

*DEFAULT 'necunoscută' NOT NULL;*

Pentru modificarea unui atribut existent (poate fi modificat doar tipul, nu și numele) se utilizează instrucțiunea:

*ALTER TABLE <nume relație>*

*MODIFY <nume atribut vechi><definiție nouă>;*

**Exemplul 4.17.** Pentru a indica că este nevoie ca dimensiunea atributului *Adresă* să se mărească până la 120 de caractere, se va utiliza următoarea instrucțiune:

*ALTER TABLE funcționari*

*MODIFY Adresă VARCHAR(120);*

Modificarea unei scheme poate crea probleme, dacă este incompatibilă cu conținutul relației. În cazul în care se modifică tipul unui atribut, trebuie să se țină cont de următoarele aspecte, care vor fi explicate luând ca bază că relația *funcționari* are un atribut *Adresă* definit

*Adresă VARCHAR(50) DEFAULT 'necunoscută'*

*NOT NULL*

- În cazul în care atributul are valori, nu poate fi redusă dimensiunea atributului, dar poate fi mărită.

**Exemplul 4.18.** Prima instrucțiune este incorectă, deoarece reduce dimensiunea atributului *Adresă*, pe când a doua instrucțiune este corectă:

*ALTER TABLE funcționari*

*MODIFY Adresă VARCHAR(10);*

*ALTER TABLE funcționari*

*MODIFY Adresă VARCHAR(150);*

- În cazul în care coloana are valori nule, nu se mai poate cere ca atributul să nu accepte valori nule.

- Dacă se dorește să se indice că acum se admit valori nule și/sau nu are valori implicite, acest lucru trebuie să se specifice în mod explicit. Astfel, dacă se cere că deja se acceptă valori nule, se va indica *NULL*, și dacă dorim să nu existe vreo valoare implicită, se va indica *DEFAULT NULL* (adică valoarea implicită este *NULL*, care este alegerea, dacă nu este indicat altceva).

**Exemplul 4.19.** Atributul *Adresă* nu acceptă valori nule și are aceeași valoare implicită:

*ALTER TABLE* *funcționari*

*MODIFY Adresă VARCHAR(120);*

O instrucțiune corectă care modifică atributul *Adresă* ce admite valori nule:

*ALTER TABLE* *funcționari*

*MODIFY Adresă VARCHAR(120) NULL;*

O instrucțiune corectă care modifică atributul *Adresă* ce nu are valoare implicită (este *NULL*):

*ALTER TABLE* *funcționari*

*MODIFY Adresă VARCHAR(120) DEFAULT NULL;*

Pentru ștergerea unui atribut (coloana) dintr-un tabel, în instrucțiunea *ALTER TABLE* se folosește cuvântul-cheie *DROP*:

*ALTER TABLE* <nume relație>

*DROP COLUMN* <nume atribut>;

**Exemplul 4.20.** Ștergerea atributului *Adresă* din relația *funcționari* se poate face cu instrucțiunea:

*ALTER TABLE* *funcționari* *DROP COLUMN* *Adresă*;

Opțiunea *RENAME COLUMN* este o extindere proprie sistemului Oracle și nu face parte din standardul ISO SQL. Oracle asigură următoarea sintaxă pentru redenumirea atributelor:

*ALTER TABLE* <nume relație>*RENAME COLUMN* <nume atribut vechi> *TO* <nume atribut nou>;

#### 4.2.4.2. Adăugarea, modificarea și suprimarea constrângerilor

Constrângerile de integritate pot fi adăugate sau suprimate cu ajutorul instrucțiunii *ALTER TABLE*. De asemenea, poate fi modificat statutul constrângerii, apelând la clauza *MODIFY CONSTRAINT*. Pot fi adăugate doar constrângerile definite la nivel de relație. Dacă se dorește adăugarea unei constrângeri la nivel de atribut, trebuie modificat atributul.

O constrângere poate fi adăugată într-o relație deja existentă, utilizând instrucțiunea *ALTER TABLE* cu clauza *ADD*.

Sintaxa generală a instrucțiunii de adăugare a constrângerilor are forma:

```
ALTER TABLE <nume relație> ADD [CONSTRAINT <nume constrângere>] <constrângere>;
```

Se recomandă să se specifice numele constrângerii, deși acest lucru nu este obligatoriu. În caz contrar, sistemul va genera un nume de constrângere implicit.

Pot fi oferite următoarele sfaturi privind adăugarea constrângerilor:

- Constrângerea poate fi adăugată, suprimită, activată sau dezactivată, dar este imposibil să se schimbe structura.
- Constrângerea *NOT NULL* la un atribut poate fi adăugată, folosind clauza *MODIFY* în loc de clauza *ADD* a instrucțiunii *ALTER TABLE*.
- Un atribut poate fi definit *NOT NULL* numai în cazul în care relația este vidă, sau în cazul în care este specificată o valoare implicită, care va fi atribuită automat tuturor tuplurilor existente ale relației.

**Exemplul 4.21.** În relația *funcționari*, este adăugată constrângerea de unicitate asupra numelui și prenumelui funcționarului:

```
ALTER TABLE funcționari ADD CONSTRAINT un_numere_prenume UNIQUE(Nume, Prenume);
```

Dacă se dorește renunțarea la constrângere, se utilizează instrucțiunea *ALTER TABLE* cu clauza *DROP*. Opțiunea *CASCADE* a clauzei *DROP* provoacă, de asemenea, eliminarea tuturor constrângerilor asociate. Sintaxa instrucțiunii este următoarea:

*ALTER TABLE <nume relație> DROP {[CONSTRAINT <nume constrângere>} | <definiție constrângere>} [CASCADE];*

Suprimarea aserțiunii, care este o constrângere de integritate la nivel de bază de date, se realizează cu ajutorul instrucțiunii:

*DROP ASSERTION <nume aserțiune>;*

În cazul renunțării la o constrângere de integritate, aceasta nu mai este controlată de SGBD și nu mai există în dicționarul de date.

**Exemplul 4.22.** Mai jos, sunt prezentate instrucțiunile de suprimare a constrângerilor cheie primară, cheie externă și constrângerii de unicitate, precum și a aserțiunii *SalariuMaiMic* creată în exemplul 4.8, respectiv:

*ALTER TABLE departamente DROP PRIMARY KEY;*

*ALTER TABLE funcționari*

*DROP CONSTRAINT fk\_funct\_depart;*

*ALTER TABLE funcționari*

*DROP CONSTRAINT u\_depart\_loc;*

*DROP ASSERTION SalariuMaiMic;*

Constrângerile de integritate sunt, uneori, supărătoare, deoarece sunt cronofoage. Este posibil să se dorească ca acestea să fie eliminate pentru a îmbunătăți performanța în timpul încărcării unei cantități mari de date din baza de date. Pentru a face acest lucru, SGBD-ul Oracle oferă posibilitatea de activare și dezactivare a constrângerilor. Sintaxa generală a instrucțiunii de activare sau dezactivare a constrângerilor poate fi:

*ALTER TABLE <nume relație> MODIFY  
    CONSTRAINT <nume constrângere>  
        ENABLE|DISABLE*

sau

*ALTER TABLE <nume relație>  
        ENABLE|DISABLE <nume constrângere>;*

Acest instrument trebuie să fie utilizat cu prudență atunci când este sigur că nu există date adăugate în timpul invalidării constrângerii care ar încalcă constrângerea. Astfel, se poate activa o constrângere fără a o sterge sau recrea, utilizând *ALTER TABLE* cu clauza *ENABLE*:

*ALTER TABLE <nume relație>  
ENABLE CONSTRAINT <nume constrângere>;*

Trebuie menționate următoarele observații cu caracter general:

- Dacă se activează o constrângere, această constrângere se aplică la toate datele din relație.
- Dacă se activează o constrângere *UNIQUE* sau *PRIMARY KEY*, se creează automat un index *UNIQUE* sau *PRIMARY KEY*.
- Clauza *ENABLE* se poate utiliza în ambele declarații *CREATE TABLE*, precum și *ALTER TABLE*.

**Exemplu 4.23.** Activarea constrângerii *fk\_funct\_depart* în relația *funcționari*:

*ALTER TABLE funcționari  
ENABLE CONSTRAINT fk\_funct\_depart;*

Precum se știe, o constrângere devine activă îndată după definirea ei. Dar o constrângere cu nume poate fi dezactivată, după dorință. Dezactivarea constrângerii se efectuează cu declarația *ALTER TABLE* însorită de clauza *DISABLE*:

*ALTER TABLE <nume relație> DISABLE CONSTRAINT <nume constrângere> [CASCADE];*

- Clauza *DISABLE* poate fi utilizată atât în declarația *CREATE TABLE*, cât și în *ALTER TABLE*.
- Clauza *CASCADE* dezactivează constrângerile de integritate dependente.

**Exemplu 4.24.** Urmează un exemplu de dezactivare a constrângerii *fk\_funct\_depart* în relația *funcționari*:

*ALTER TABLE funcționari  
DISABLE CONSTRAINT fk\_funct\_depart;*

#### 4.2.4.3. Amânarea verificării constrângerilor

De obicei, constrângerile de integritate sunt verificate la fiecare instrucțiune SQL. În unele circumstanțe particulare, se poate dori amânarea verificării unei sau mai multor constrângerii până la încheierea unei mulțimi de interogări grupate într-o tranzacție.

Verificarea validității constrângerii poate fi amânată până la sfârșitul tranzacției. Serverul SQL2 oferă posibilitatea declarării constrângerilor *DEFERRABLE* (care pot fi amânate). Verificarea constrângerilor *DEFERRABLE* poate fi amânată până la încheierea tranzacției. Dacă constrângerea amânată este încălcată, atunci tranzacția este derulată înapoi. O constrângere care nu este *DEFERRABLE* va fi verificată imediat după fiecare instrucțiune. În cazul în care aceasta este violată, instrucțiunea este imediat derulată înapoi. Opțiunea implicită este *NOT DEFERRABLE*.

Dacă o constrângere este *DEFERRABLE*, atunci ea poate fi definită ca *INITIALLY DEFERRED* ("amânabilă amânată") sau *INITIALLY IMMEDIATE* ("amânabilă imediat"), ceea ce definește momentul implicit pentru verificarea acestei constrângerii. Constrângerile *INITIALLY IMMEDIATE* sunt verificate după fiecare instrucțiune - precum constrângerile neamânabile obișnuite - în timp ce constrângerile *INITIALLY DEFERRED* sunt verificate numai la sfârșitul tranzacției. Cazul implicit este *INITIALLY IMMEDIATE*.

Momentul în care o constrângere *DEFERRABLE* va fi verificată poate fi modificat, utilizând instrucțiunea *SET CONSTRAINT*:

```
SET CONSTRAINT nume_constrangere1
    [, nume_constrangere2, ... ]
    { IMMEDIATE | DEFERRED };
```

Constrângerile referențiale *DEFERRABLE* permit datelor să fie inserate în tabelul-copil înaintea datelor din tabelul-părinte.

**Exemplu 4.25.** Fie că, într-o companie, codul 40 al unui departament trebuie să fie schimbat cu 45. Modificarea atributului *Depart\_id* afectează angajații care activează în acest departament. Prin urmare, cheia primară și cea externă pot fi definite *DEFERRABLE* și *INITIALLY DEFERRED*. Astfel, pot fi actualizate datele și despre departament și despre angajați și la momentul de validare a tranzacției, toate rândurile vor fi validate.

Urmează o instrucțiune de amânare a constrângerii la creare:

```
ALTER TABLE departamente
ADD CONSTRAINT depart_id_pk
PRIMARY KEY (Depart_id)
DEFERRABLE INITIALLY DEFERRED;
```

Momentul, în care o constrângere este verificată, poate fi modificat, ca în instrucțiune următoare:

```
SET CONSTRAINTS depart_id_pk IMMEDIATE;
```

#### 4.2.4.4. Suprimarea schemei relaționale

Comanda simetrică celei de creare a unei relații este *DROP TABLE*, care are ca efect eliminarea din catalogul sistem a relației specificate și eliberarea spațiului de memorie ocupat. După executarea acestei instrucțiuni, nu se mai pot face niciun fel de referiri la relația în cauză. Sintaxa instrucțiunii este următoarea:

```
DROP TABLE <nume relație>.
```

Trebuie menționat că relația suprimată în unele sisteme nu mai poate fi restabilită.

**Exemplul 4.26.** Următoarea instrucțiune elimină din dicționarul de date definiția relației *funcționari*:

```
DROP TABLE funcționari;
```

Nu este posibilă suprimarea unei relații, dacă relația este referită cu o constrângere de integritate referențială. O variantă Oracle (dar nu SQL2) a instrucțiunii permite suprimarea constrângerilor de integritate și a relației:

```
DROP TABLE <nume relație>
CASCADE CONSTRAINTS.
```

SGBD-ul Oracle oferă posibilitatea de redenumire a relațiilor în cadrul instrucțiunii *ALTER TABLE*. Această opțiune nu este adoptată de standardul SQL2. Sintaxa instrucțiunii de redenumire a relațiilor în Oracle este următoarea:

```
ALTER TABLE <nume relație vechi>
RENAME TO <nume relație nou>;
```

sau

*RENAME <nume relație vechi>*

*TO <nume relație nou>;*

**Exemplul 4.27.** S-ar putea redenumi relația *funcționari* în *funct*, aplicând această sintaxă:

*ALTER TABLE funcționari RENAME TO funct;*

sau

*RENAME funcționari TO funct;*

## 4.3. Gestiuinea viziunilor

Viziunile constituie un mecanism puternic al modelului relațional de date, care se referă la interacțiunea între utilizator sau aplicație și schema bazei de date. Viziunile permit să denume diverse forme ale acelorași date și constituie o specie de relații virtuale sau "ferestre", prin care se ajunge la datele stocate în relații. Viziunile asigură realizarea nivelului extern al bazei de date conform arhitecturii ANSI / SPARC. Trebuie menționat că nivelul extern oferă utilizatorului o percepție a bazei de date mai aproape de cerințele sale, în termeni de structuri și formate de date [Bancilhon81].

### 4.3.1. Ce este o viziune?

Viziunile permit să se realizeze, în lumea SGBD-urilor relaționale, nivelul extern al bazei de date conform arhitecturii ANSI / SPARC. Trebuie menționat că nivelul extern oferă utilizatorului o percepție a bazei de date mai aproape de cerințele sale, în termeni de structuri și formate de date.

În general, viziunile garantează o mai mare independență logică a programelor în raport cu datele. În consecință, programele rămân invariante la modificările schemei în cazul în care baza de date se accesează printr-o viziune care o izolează de acestea. Atunci când se petrec schimbări în schema bazei de date, administratorul trebuie să modifice doar definiția viziunii, iar programele aplicative continuă să funcționeze fără modificări.

Viziunile joacă, de asemenea, un rol important în procesul de securitate a datelor. Utilizatorul nu poate accede decât la datele viziunii la care el are drepturi de acces. Astfel, datele din afară viziunii sunt protejate.

În mod indirect, viziunile asigură și un control al integrității, atunci când acestea sunt utilizate pentru actualizarea datelor. Se spune, în acest caz, că se realizează o actualizare, folosind viziunile. Actualizările prin viziuni sunt bine supervizate, dar despre acest fapt vor fi oferite mai multe detalii într-una din secțiunile ce urmează.

În ultimii ani, viziunile și-au găsit noi aplicații. Acestea definesc relații virtuale corespunzătoare nevoilor programelor aplicative în termeni de date. În modelul client-server, viziunile constituie elementul esențial referitor la optimizarea performanțelor. De exemplu, dacă în definiția unei viziuni se jonctionează două relații sau viziunea se obține printr-un calcul de agregare, atunci viziunea protejează clienții de riscul să citească tuplurile din relații și să facă calcule de jonctionare sau de agregare pe stația-client. Numai datele care se obțin din calculul viziunii sunt exportate pe stația-client. Astfel calculele ce țin de jonctionuri sau agregări etc. se lasă pentru server, lucru pe care el știe să-l facă bine.

În mediul magaziilor de date sau decizional, viziunile pot fi concretizate. Ele permit realizarea acumulărilor sau sintezelor mai sofisticate, extrase din baza de date în funcție de mai multe dimensiuni.

Mecanismele de actualizare a viziunilor concrete apoi de actualizare a bazei de date sunt dezvoltate cu scopul de a evita recalculul acumulărilor. Prin urmare, viziunile capătă o importanță crescândă în bazele de date. În realitate, viziunile nu sunt altceva decât relații virtuale construite în baza interogărilor. Definițiile viziunilor sunt stocate în metabază (sau dicționarul de date).

Viziunile pot fi interogate ca și relațiile obișnuite. În mod ideal, acestea ar trebui să fie și actualizate, la fel ca relațiile obișnuite. Viziunile sunt calculate pe disc în timpul creării lor. Mecanismele de actualizare diferențiate permit transferul eficient al actualizărilor în relațiile de bază. Toate aceste tehnici sunt acum bine puse la punct și vor fi prezentate în continuare.

Pentru a ilustra mecanismele viziunilor, în această secțiune, se va folosi baza de date compusă din relațiile:

*funcționari (FunctID, Nume, Prenume, Post, Salariu, DeptID),  
departamente (DeptID, Denumire, Oraș).*

#### 4.3.2. Crearea și suprimarea viziunilor

În această secțiune, mai întâi, se va defini exact ce este o viziune, apoi se va introduce sintaxa SQL pentru definirea unei viziuni.

O viziune reprezintă un set de relații derivate dintr-o bază de date, prin compozitia relațiilor din bază. Schema unei viziuni este o schemă externă în sensul ANSI/SPARC. În SQL standard, noțiunea de viziune se reduce la o singură relație dedusă. În cele din urmă, o viziune este o relație virtuală calculabilă de vreo interogare.

Viziunile se definesc, utilizând instrucțiunea *CREATE VIEW*, conform sintaxei:

```
CREATE VIEW <nume viziune> [(listă atribute)]  
AS <interrogare>  
[WITH [CASCDED|LOCAL] CHECK OPTION]
```

O viziune se definește printr-o cerere (o instrucțiune *SELECT*), care se calculează și care este realizată asupra unei sau mai multe relații. Asupra unei mulțimi de relații pot fi definite mai multe viziuni diverse.

Numele viziunii este numele relației virtuale corespunzătoare viziunii, lista de atribute definește atributele relației virtuale, iar interogarea permite calcularea tuplurilor necesare pentru popularea relației virtuale. Atributele instrucțiunii *SELECT* sunt aplicate asupra celor din viziune. În cazul în care atributele viziunii nu sunt specificate, aceasta moștenește direct atributele interogării *SELECT*.

Îndată ce viziunea este definită se poate utiliza numele ei pentru a fi accesată. De fiecare dată, la fiecare accesare, interogarea care o definește este recalculată. În general, numele unei viziuni poate apărea în orice loc al unei instrucțiuni SQL, unde poate apărea un nume de relație cu condiția că această instrucțiune este de interogare, și nu de modificare.

Clauza *WITH CHECK OPTION* specifică faptul că tuplurile inserate sau modificate prin viziune trebuie să îndeplinească condițiile interogării care definește viziunea. Aceste condiții sunt verificate după actualizare. Astfel, SGBD va testa care tupluri inserate sau modificate prin viziune corespund condiției interogării. Acest fapt asigură că tuplurile inserate sau modificate prin viziune aparțin bine viziunii. În caz contrar, dacă clauza *WITH CHECK OPTION* este prezentă, actualizarea este respinsă.

De exemplu, în cazul în care viziunea posedă doar atributele unei relații, dar interogarea un criteriu de joncționare cu altă relație, tuplurile inserate în relația corespunzătoare viziunii trebuie să jonctioneze cu tuplurile din cealaltă relație. Acest fapt poate fi folosit pentru a forța verificarea unei constrângeri referențiale la inserarea datelor într-o viziune. În secțiunile respective, actualizarea prin viziuni și justificarea clauzei de testare vor fi examinate în detaliu.

Urmează două exemple de definire a viziunilor.

**Exemplul 4.28.** Viziunea implică doar numele, prenumele și departamentul funcționarilor:

```
CREATE VIEW funct2 (Nume, Prenume, DeptID)
AS SELECT Nume, Prenume, DeptID
FROM funcționari;
```

Este posibilă crearea unei viziuni asupra viziunii:

```
CREATE VIEW funct3 (Nume, Prenume, DeptID)
AS SELECT Nume, Prenume, DeptID
FROM funct2;
```

*SELECT* poate conține toate clauzele unei instrucțiuni *SELECT*, cu excepția clauzei *ORDER BY*.

Viziunile sunt suprimate (numai definiția sa, dar nu și relațiile pe care se bazează), folosind instrucțiunea *DROP VIEW*:

```
DROP VIEW <nume viziune>;
```

Această instrucțiune suprimă definiția viziunii din metabaza bazei de date. În principiu, dar după cum va vedea mai jos, în cazul viziunilor concrete, o viziune nu are nicio existență fizică. Ea este o fereastră dinamică (și

denaturată), nematerializată pe discuri, prin care utilizatorul accede la baza de date. Suprimarea viziunii nu are consecințe asupra tuplurilor.

Specificarea numelor de atribute în viziune este optională. În mod implicit, atributele viziunii sunt numele atributelor respective ale clauzei *SELECT*. Rezultatul unei vizuni conține toate atributele specificate în interogare. Aceste atribută au aceeași ordine. Dacă unele atribută din lista *SELECT* sunt reprezentate de expresii fără nume, prezentarea atributelor în viziune este obligatorie.

**Exemplul 4.29.** Viziunea următoare reprezintă o restricție a relației *funcționari* pentru funcționarii departamentului *Dept02*:

```
CREATE VIEW functDept02
AS SELECT * FROM funcționari
WHERE DeptID = 'Dept02';
```

Suprimarea viziunii se face aplicând instrucțiunea:

```
DROP VIEW functDept02;
```

În cazul în care instrucțiunea, din exemplul 4.29, este inserată într-un program-aplicație, este necesară o prudență și o flexibilitate, pentru evita folosirii „\*” și înlocuirea asteriscului cu numele atributelor din relația *funcționari*. În cazul în care definiția relației *funcționari* este schimbată, poate apărea o eroare de execuție, dacă nu se reconstruiește viziunea *functDept02*.

#### 4.3.3. Consultarea și actualizarea viziunilor

Utilizarea unei instrucțiuni de actualizare prin viziune a conținutului bazei de date poate genera unele probleme și produce niște rezultate indezirabile. Aceasta se explică prin faptul că relația exprimată de viziune este una virtuală și nu întotdeauna există o mapare biunivocă a schimbărilor din viziune în schimbări din baza de date. Acest fapt are loc, cu toate că, în orice moment, conținutul viziunii este rezultatul evaluării interogării formulate asupra bazei de date.

Consultarea viziunilor este un proces realizabil pentru orice formă de interogări care stau la baza acestora.

#### 4.3.3.1. Consultarea viziunilor

Extragerea datelor din viziuni se face în același mod ca în cazul relațiilor obișnuite. O viziune poate fi consultată cu o instrucție *SELECT* de orice complexitate.

**Exemplul 4.30.** Astfel, este posibilă interogarea viziunii *funcțDept02*. Totul se petrece ca și cum ar exista o relație de funcționari ai departamentului *Dept02*:

```
SELECT * FROM funcțDept02;
```

În continuare, vor fi prezentate mai multe tipuri de interogări care ar putea fi folosite pentru definirea unei viziuni și vor fi analizate rezultatele, în cazul în care se va încerca să se actualizeze datele din aceste viziuni. În cele ce urmează, inserarea, ștergerea sau modificarea unei viziuni vor fi examinate separat, dacă se poate.

#### 4.3.3.2. Viziuni definite pe proiecții de atrbute

Ca regulă generală, o viziune este actualizabilă atunci când poate fi identificat fiecare tuplu al relației de bază ce corespunde oricărui tuplu din viziune.

Se consideră o viziune definită de proiecția unei relații pe o mulțime de atrbute și posibilitățile de actualizare a acesteia.

**Exemplul 4.31.** Viziunea *viziuneal* este construită pe proiecția relației *funcționari* pe atrbutele *Nume*, *Prenume* și *Salariu*:

```
CREATE VIEW viziuneal  
AS SELECT Nume, Prenume, Salariu  
FROM funcționari;
```

Se vor analiza posibilitățile de actualizare a acestei viziuni. În mod evident, acestea trebuie să fie traduse asupra relației *funcționari*.

**Inserarea:** Pot să apară mai multe cazuri. Dacă unu sau mai multe dintre atrbutele relației de bază, care nu sunt în viziune, nu permit valori nule sau au valori implice, inserarea nu este posibilă. Apare mesajul de eroare „încercarea de inserare a valorilor nule”.

**Exemplul 4.32.** Acesta este cazul viziunii *viziuneal*. Atributul *FunctID*, care nu este în definiția viziunii și fiind cheie primară, nu admite valori nule:

```
INSERT INTO viziuneal
VALUES('Vasilache', 'Ion', 0);
```

De regulă, în celelalte cazuri, este posibilă inserarea datelor în viziuni. Cu alte cuvinte, este necesar ca atributele relației de bază care nu sunt incluse în definiția viziunii sau trebuie să admită valori nule sau au valori implicate. Cu toate acestea, există unele situații în care inserarea nu este posibilă.

**Ștergerea:** Este întotdeauna posibilă datorită faptului că tuplurile viziunii, în acest caz, corespund unu la unu tuplurilor din relația de bază. Prin urmare, tuplurile care se elimină din viziune, de fapt se elimină din relația de bază.

Dar, totuși, referitor la suprimarea tuplurilor, trebuie să se clarifice unele momente.

**Exemplul 4.33.** Fie definită următoarea viziune și o interogare cu privire la aceasta:

- *CREATE VIEW vsalariu* ;
- *AS SELECT Salariu*
- FROM functionari;*

```
SELECT * FROM vsalariu
ORDER BY Salariu;
```

În rezultatul returnat de interogare, pot exista două tupluri egale, fie cu salariul 5000. Dacă se dorește ștergerea lor din viziune, acestea nu pot fi distinse, deoarece se aplică următoarea instrucționare:

```
DELETE FROM vsalariu
WHERE Salariu=5000;
```

Din relația de bază, vor fi eliminate cele două tupluri corespunzătoare.

Faptul că nu se pot diferenția tuplurile nu înseamnă că SGBD-ul nu poate, pe plan intern, să realizeze ștergerea. Prin urmare, dacă interogarea de

construire a viziunii este doar o proiecție de atrbute, întotdeauna pot fi șterse datele din ea.

**Modificarea:** La fel ca în cazul ștergerii, datele dintr-o viziune de acest tip pot fi modificate oricând. Evident că se supun modificării doar acele atrbute care apar în definiția viziunii.

#### 4.3.3.3. Viziuni definite pe selecții de tupluri

Se consideră interogarea din următorul exemplu.

**Exemplul 4.34.** 44. Viziunea este creată în baza interogării „Să se găsească toate datele despre funcționari care câștigă un salariu mai mare de 5000”. Este important să se menționeze că, în cazul dat, nu se elimină tuplurile duplicate, deoarece nu se utilizează cuvântul-cheie *DISTINCT* în instrucția *SELECT*:

```
CREATE VIEW viziunea2
AS SELECT *
FROM funcționari
WHERE Salariu > 1250;
```

În continuare, sunt examinate toate operațiile de actualizare a viziunii prezentate în exemplul 4.34.

**Inserarea:** În astfel de viziuni, întotdeauna, este posibilă inserarea tuplurilor, deoarece fiecare dintre tuplurile prezente în viziune corespunde direct unui tuplu din relația de bază. și, în afară de aceasta, se cunosc toate atrbutele.

Totuși, inserțiile pot conduce la situații incomode. De exemplu, dacă, în viziunea *viziunea2*, se inserează un tuplu, cu valoarea atrbutului *Salariu* mai mică sau egală cu 5000, tuplul se va introduce în relația de bază *funcționari*, dar nu va mai apărea în viziune, fiindcă nu satisface condiția *WHERE*.

**Ștergerea:** Ștergerea este întotdeauna posibilă, din aceleași motive. Tuplurile care se elimină din viziune, de fapt, se elimină din relația de bază. În acest caz, situații ciudate precum cele din cazul inserărilor nu se întâmplă, deoarece condiția de ștergere, dacă există, se adaugă la condiția viziunii înainte de realizarea ștergerii.

**Exemplul 4.35.** Fie e dată instrucțiunea:

*DELETE FROM viziunea2;*

Instrucțiunea va șterge din relația *funcționari* toate tuplurile prezente în viziune, dar nu și alte tupluri. Oricum, în aceste cazuri, se sugerează să se verifice cu atenție comportamentul fiecărei versiuni de SGBD.

**Modificarea:** Datele într-o asemenea viziune pot fi ușor modificate din aceleași considerații ca și în cazul ștergerii.

Cu toate acestea, există alte aspecte, prezentate de următorul exemplu.

**Exemplul 4.36.** Fie sunt date următoarele instrucțiuni de modificare și interogare a viziunii *viziunea2*:

*UPDATE viziunea2 SET Salariu=4000  
WHERE FunctID=f002;*

*SELECT Nume, Prenume, FunctID, Salariu  
FROM viziunea2;*

Tuplul indicat va fi modificat în relația de bază *funcționari*, dar, în urma consultării, tuplul dispare din viziune (ca și în cazul de parcă ar fi fost șters), deoarece nu îndeplinește condiția existenței unui salarior mai mare de 5000.

Trebuie menționat că viziuni cu un potențial mare pot fi obținute în cazul combinării operațiilor de proiecție și ale celei de selecție. Acest tip de viziuni generează aceleași probleme și combină aspectele indicate aparte pentru proiecții și selecții.

#### 4.3.3.4. Viziuni definite pe o relație de interogări cu DISTINCT sau GROUP BY

Să se considere următoarea definiție de viziune.

**Exemplul 4.37.** Cu scopul eliminării duplicatelor, viziunea *viziunea3* este definită de un SELECT cu utilizarea clauzei DISTINCT:

*CREATE VIEW viziunea3  
AS SELECT DISTINCT Post, DeptID*

*FROM* *funcționari*;

Spre deosebire de viziunile definite anterior, prin *viziunea3* nu este posibilă cunoașterea tuplurilor originale ale relației de bază pe care e definită viziunea. Eșecul propriu-zis are loc din cauza eliminării duplielor.

**Inserarea:** Nu este posibilă propagarea inserării în relația de bază, nu se cunoaște cum să se realizeze această inserare. De exemplu, nu se știe câte copii ale tuplului ar putea fi introduse, deoarece atunci când se utilizează clauză *DISTINCT* în definiția viziunii se elimină duplicatele.

**Ștergerea:** Deși teoretic ar fi posibilă ștergerea tuturor copiilor tuplurilor din relația de bază, care ar corespunde fiecărui tuplu șters din viziune, practic, nu este permisă. Executarea acestei operațiuni nu va duce, în multe cazuri, la rezultatul scontat.

**Modificarea:** Din același motive, cu toate că teoretic este posibilă modificarea, nu este permisă pentru acest tip de viziuni.

În următorul exemplu, este prezentată o viziune construită în baza unei grupări și a unei funcții de agregare.

**Exemplul 4.38.** Viziunea creată construiește numărul de funcționari pentru fiecare departament:

```
CREATE VIEW funct_per_dept
AS SELECT DeptID, COUNT(*) AS NumărFunct
FROM funcționari
GROUP BY DeptID;
```

Și în acest caz, tuplurile din viziune nu corespund unu la unu cu cele din relația de bază. Motivele pentru care actualizările în asemenea viziuni se fac nerealizabile sunt considerate, în continuare, pentru fiecare operație de actualizare aparte.

**Inserarea:** Nu este posibilă, deoarece intenția de a insera un tuplu în viziune este împiedicată de necunoașterea translării în relația de bază a acestei inserări. De exemplu, executarea instrucțiunii

```
INSERT INTO funct_per_dept VALUES ('Dept02', 2);
```

presupune introducerea a două tupluri în relația *funcționari*, dar nu se cunosc ce valori trebuie atribuite, cu excepția numărului departamentului.

**Ștergerea:** Teoretic, ștergerea tuplurilor din viziune ar fi posibilă, dar, de exemplu, încercarea de a executa instrucțiunea

```
DELETE FROM funct_per_dept
WHERE DeptID= 'Dept03';
```

nu pare logică. Nu pare logică ștergerea tuturor tuplurilor referitoare la acest departament din relația *funcționari*, deoarece viziunea doar spune căți angajați are departamentul. De aceea, eliminarea tuplurilor din viziunile constituite pe grupări sau funcții de agregare nu este permisă.

**Modificarea:** Modificările în acest tip de viziuni, de asemenea, nu sunt permise. În orice caz, teoretic modificarea ar fi posibilă. De exemplu, instrucțiunea

```
UPDATE funct_per_dept SET DeptID = 'Dept05'
WHERE DeptID = 'Dept03';
```

modifică numărul departamentului al funcționarilor respectivi și, în realitate, produce un rezultat alogic. Alte modificări, cum ar fi

```
UPDATE funct_per_dept SET NumărFunct = 2
WHERE DeptID= 'Dept03';
```

realizează ștergerea din relația *funcționari* a unui tuplu referitor la acest departament (care fie ca avea până la modificare 3 angajați). Dar și aceasta nu ar fi o acțiune corectă.

În consecință, nicio viziune definită cu ajutorul clauzelor *DISTINCT* sau *GROUP BY*, sau funcțiilor de agregare nu permit actualizări de date ale relațiilor de bază.

#### 4.3.3.5. Viziuni definite pe mai multe relații

Construirea viziunilor complexe în baza mai multor relații, apelând la operațiile de joncțiune este un lucru obișnuit în procesul de proiectare a bazelor de date care sunt conforme cu arhitectura în trei nivele propusă de ANSI/SPARC.

**Exemplul 4.39.** Viziunea următoare extrage numele angajaților și denumirile departamentelor din care fac parte:

```
CREATE VIEW funct_dept
AS SELECT Nume, Prenume, Denumire
FROM funcționari AS f, departamente AS d
WHERE f.DeptID=d.DeptID;
```

Conform standardului SQL2, atunci când, în clauza *FROM* a instrucțiunii *SELECT*, care definește viziunea, sunt specificate mai mult de o relație nu este posibilă realizarea niciunei operații de actualizare direct asupra viziunii.

Motivul constă în faptul că nu există vreo formă generică fiabilă de translare a schimbărilor dezirabile din viziune în relațiile de bază. Oracle permite, totuși, în anumite circumstanțe, unele actualizări pe acest tip de viziuni, dar acestea nu sunt în conformitate cu norma SQL și, deci, se pot schimba [Bello98]. Nu se va intra în detalii referitor la această problemă și se va presupune că pe astfel de viziuni nu pot fi realizate operații de actualizare.

#### 4.3.4. Clasificarea viziunilor

În practică, cele mai multe sisteme de gestiune rezolvă problema de actualizare a viziunilor limitându-se la posibilitățile de actualizare a viziunilor-monorelație. Numai atributurile unei relații de bază trebuie să apară în viziune. În plus, se impune să fie prezente atributele cheii relației de bază. Această constrângere face posibilă definirea unei strategii simple de actualizare. În timpul inserției, se introduc, pur și simplu, tupluri noi, cu valori nule pentru atributele neexistente. În timpul suprimării, se șterg tuplurile care răspund criteriului. Într-o modificare, se modifică tuplurile care îndeplinesc criteriul.

Definiția viziunii poate referi alte relații care permit precizarea tuplurilor viziunii. În teorie, trebuie să se verifice dacă tuplurile inserate, șterse sau modificate, într-adevăr apar în viziunii. În practică, SQL efectuează această verificare numai dacă a fost solicitată în definiția viziunii cu clauza *WITH CHECK OPTION*.

Restricționarea actualizării la viziunile monorelație este mult prea strictă. Categorii de viziuni actualizabile poate fi extinsă, pur și simplu, cel puțin cu inserarea și suprimarea viziunilor multirelație care conțin cheile relațiilor participante. Atributurile nedокументate se înlocuiesc cu valori *NULL* la inserții. Această soluție creează baze de date cu multe valori *NULL*.

Restricțiile impuse de standard SQL2 sunt relaxate de unele SGBD-uri. Acesta este cazul sistemului Oracle, care permite modificarea datelor unei relații de bază printr-o viziune care conține o joncțiune. O condiție trebuie respectată: viziunea trebuie să păstreze cheia relației. Fără a intra în detaliu, urmează un exemplu de viziune care permite SGBD-ului să modifice relația *funcționari*.

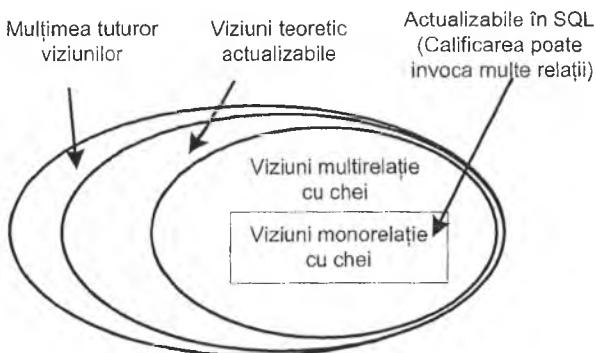
**Exemplul 4.40.** Viziunea implică numele, prenumele funcționarilor și denumirile departamentelor în care aceștia activează:

```
CREATE VIEW funct2
AS SELECT FunctID, Nume, Prenume, Denumire
FROM funcționari NATURAL JOIN departamente;
```

Următoarea instrucțiune este acceptată de Oracle:

```
UPDATE funct2
SET Nume = 'Puiu'
WHERE Nume = 'Paiu'
```

Soluția dată este adesea interzisă în sistemele comerciale. Acestea sunt departe de a permite toate actualizările teoretic posibile, după cum se vede în figura 4.2.



*Figura 4.2. Clasificarea viziunilor în funcție de posibilitățile de actualizare*

În general, viziunile în funcție de posibilitățile de actualizare pot fi divizate în următoarele categorii:

**Toate viziunile.** Toate viziunile care pot fi specificate cu o instrucțiune *SELECT*.

**Viziunile teoretic actualizabile.** Este o submulțime a celor anterioare formată din mulțimea de viziuni teoretic pot fi actualizate. Cu alte cuvinte, traducerea actualizărilor din viziuni în actualizări ale relațiilor de bază teoretic este posibilă.

**Viziunile realmente actualizabile.** Reprezintă o submulțime a viziunilor teoretic actualizabile, constituță din viziunile care, în realitate, sunt considerate actualizabile de către SGBD. Probabil, fiecare sistem de gestiune permite actualizarea datelor a unei submulțimi (reduse) a acestei mulțimi de viziuni.

În cazul în care se lansează o instrucțiune de actualizare (*INSERT*, *UPDATE*, *DELETE*) asupra unei viziuni, sistemul de gestiune transferă această actualizare asupra relației sau relațiilor din care este construită viziunea. La o tentativă de actualizare, în unele cazuri, operația poate fi realizabilă, dar în altele poate fi imposibilă și va apărea o eroare.

Următoarele condiții trebuie să fie îndeplinite pentru ca viziunea să se găsească în categoria de viziuni realmente actualizabile:

- Pentru a efectua un *DELETE*, interogarea, care definește vizuirea, nu ar trebui să includă jocuri, clauzele *GROUP BY* și *DISTINCT*, precum și funcții de agregare.
- Pentru un *UPDATE*, la condițiile de mai sus, se adaugă cerința ca atributele modificate să fie atrbute reale (și nu atrbute care sunt rezultatul unui calcul sau al unei expresii), ale unei relații de bază.
- Pentru un *INSERT*, în plus la condițiile precedente, toate atrbutele *NOT NULL* ale relației de bază trebuie să fie prezente în vizuire.

#### **4.3.5. Din nou despre opțiunea WITH CHECK OPTION**

Precum s-a menționat, atunci când se modifică, se inserează sau se sterg tupluri pot să se producă fenomene ciudate, cum ar fi inserarea unui tuplu în vizuire care nu este vizibil în această vizuire sau modificarea unui tuplu care dispără din vizuire.

Acest lucru se întâmplă din faptul că tuplurile existente într-o vizuire trebuie să îndeplinească condiția *WHERE* din definiția interogării. Dacă un tuplu este modificat astfel încât el nu mai îndeplinește condiția *WHERE*, atunci dispără din vizuire. În mod similar, tupluri noi apar în vizuire atunci când o inserție sau modificare fac ca aceste tupluri să satisfacă condiția *WHERE*. Tuplurile care intră sau care ies din vizuire se numesc tupluri-migranți.

Precum s-a menționat anterior, în standardul SQL, este introdusă o clauză optională,

*[WITH [CASCADED|LOCAL] CHECK OPTION],*

în instrucțiunea de creare a unei vizuri pentru a efectua controlul acestor tupluri.

În general, clauza *WITH CHECK OPTION* interzice migrația tuplurilor la sau de la vizuire. Calificativele *LOCAL/CASCADED* se aplică asupra ierarhiilor de vizuri, adică asupra vizurilor derivate din alte vizuri. În cazul în care, în definiția vizurii, se specifică opțiunea *WITH LOCAL CHECK OPTION*, atunci orice tuplu inserat sau modificat în vizuire și orice vizuire definită direct sau indirect pe acesta, nu trebuie să producă

migrarea tuplului în vedere, cu excepția cazului când tuplul dispare, de asemenea, din relația sau viziunea din care aceasta este creată.

Dacă se specifică *WITH CASCADED CHECK OPTION* (modul implicit), atunci inserarea sau modificarea acestei viziuni sau altei viziuni direct sau indirect definită pe viziunea în chestiune nu poate cauza migrația tuplurilor.

Această opțiune este utilă, deoarece atunci când un *INSERT* sau *UPDATE* asupra viziunii încalcă condiția *WHERE* a definiției viziunii, operația este respinsă. Clauza *WITH CHECK OPTION* poate fi specificată numai în viziunile actualizabile.

**Exemplul 4.41.** Se revine din nou la viziunea care reprezintă o restricție a relației *funcționari* la tuplurile funcționarilor departamentului *Dept02*:

```
CREATE VIEW functDept02
AS SELECT * FROM funcționari
WHERE DeptID = 'Dept02';
```

Prin intermediul viziunii *functDept02*, pot fi modificate salariile funcționarilor care activează în departamentul *Dept02*. Toate tuplurile din relația *funcționari* cu *DeptID* = 'Dept02' vor fi actualizate de instrucțiunea

```
UPDATE functDept02 SET Salariu = Salariu * 1.1;
```

Nu este greu să se observe că o viziune poate crea date care nu vor putea fi vizualizate. Se poate adăuga, de exemplu, un funcționar al departamentului *Dept01*, folosind viziunea *functDept02*.

Pentru a evita acest fenomen în instrucțiunea de creare a viziunii, după interogarea care formează viziunea, trebuie să se adauge opțiunea *WITH CHECK OPTION*. Această opțiune va interzice crearea de tupluri care nu vor mai putea fi citite de viziune.

**Exemplul 4.42.** Următoarea instrucțiune va interzice inserarea tuplurilor în relația de bază *funcționari*, care nu vor putea fi vizualizate în *functDept02*.

```
CREATE VIEW functDept02 AS
```

```
SELECT * FROM funcționari
WHERE DeptID = 'Dept02'
WITH CHECK OPTION;
```

#### 4.3.6. Viziuni materializabile

În această secțiune, se va trata problema viziunilor concrete [Colby97]. Acestea sunt deosebit de utile în magaziile de date (Data Warehouse), unde facilitează analiza datelor (OLAP) și asigură suportul decizional [Gupta95].

O viziune reprezintă, în principiu, o fereastră glisantă pe o bază de date, care este instanțiată de o întrebare. Concretizarea viziunii de către server poate fi mult mai benefică în cazul în care aceasta este frecvent folosită și în cazul în care relațiile-sursă sunt puțin modificate. Astfel, unele servere susțin viziunile concrete (în unele sisteme aceste viziuni se numesc materializabile).

O viziune concretă este calculată pornind de la relațiile bazei de date și este materializată pe disc de către SGBD.

O viziune concretă este calculată pornind de la definiția acesteia și se actualizează de fiecare dată, când o tranzacție modifică baza de date. Actualizarea se realizează, în cazul în care este posibilă, în mod diferențiat. Sunt luate în considerare doar tuplurile modificate pentru calcularea modificărilor ce trebuie aduse viziunii.

Din păcate, precum a fost menționat anterior, acest lucru nu este întotdeauna posibil. În cazul viziunilor definite de selecții, proiecții și juncțiuni (SPJ), inserțiile diferențiate sunt destul de simple, dar actualizările de tipul suprimare sau modificare sunt mult mai dificile. Pentru a reflecta suprimările și modificările, este nevoie, în general, să se găsească - cel puțin parțial – lanțuri de derivare care ar putea fi utilizate pentru calcularea tuplurilor viziunii concrete. În caz general (viziuni cu diferențe, agregări etc.), actualizările diferențiate sunt foarte dificil de realizat.

Vizualizările concrete sunt definite de instrucțiunea:

```
CREATE CONCRETE VIEW <nume viziune>
[<listă atributelor>]
AS <interrogare>;
```

Viziunile concrete sunt deosebit de utile atunci când interogările conțin agregări, deoarece acestea permit stocarea relațiilor rezumate, compacte. Viziunile concrete sunt foarte utile pentru suportul decizional în contextul unei magazii de date, în cazul în care se dorește analiza datelor multidimensionale [Gray96].

#### 4.3.7. Utilitatea viziunilor

În general, viziunile separă modul în care utilizatorii văd datele de decuparea lor din tabelele de bază. Se separă aspectul extern (ceea ce vede un utilizator particular al bazei de date) de aspectul logic (cum a fost concepută întreaga bază de date). Această separare promovează independența între programe și date. Dacă structura de date se modifică, programele nu se vor schimba, dacă au fost luate măsurile de precauție în utilizarea viziunilor. De exemplu, în cazul în care o relație este împărțită în mai multe relații după introducerea de date noi, se poate introduce o viziune nouă, care jonctionează relațiile noi și le denumește cu numele relației vechi pentru a evita rescrierea programelor care utilizau relația veche. De fapt, în cazul în care datele nu sunt modificabile prin viziuni, acestea pot fi folosite doar pentru recuperarea datelor, dar nu și pentru modificarea lor.

O viziune poate fi, de asemenea, folosită pentru a restrângă drepturile de acces la anumite atribută și tupluri ale unei relații. Un utilizator nu poate avea acces la o relație, dar are permisiunea de a utiliza o viziune care conține numai anumite atribută ale relației. La această viziune, pot fi adăugate constrângeri corespunzătoare de utilizare.

În aceeași ordine de idei, o viziune poate fi folosită pentru a pune în aplicare o constrângere de integritate cu opțiunea *WITH CHECK OPTION*.

O viziune poate simplifica, de asemenea, consultarea bazei de date, înregistrând pentru formarea ei instrucțiuni *SELECT* complexe.

Prin urmare, avantajele principale ale utilizării viziunilor sunt următoarele:

- Asigură confidențialitatea și securitatea datelor. Nu toți utilizatorii pot vedea sau cunoaște toată schema bazei de date. Pot exista date ascunse de unii utilizatori, unele cunoștințe despre date pot fi limitate în conformitate cu datele relevante fiecărui utilizator.
- Permite limitarea eventualelor operații (interrogări) ce pot avea loc asupra datelor.
- Simplificarea interfeței cu utilizatorii.
- Permite atribuirea unui nume simplu, prin care se poate accesa o operație complexă sau o interogare formulată a priori.
- Favorizează independența logică a datelor. Structura bazei de date poate fi schimbată fără a afecta esențial aplicațiile.

#### 4.4. Sinonime

Mai multe SGBD-uri (precum Oracle sau SQL Server) permit atribuirea de sinonime pentru relații și viziuni.

Sinonimele sunt niște nume de alternativă, care pot fi adăugate unor obiecte din baza de date pentru a simplifica accesul la aceste resurse. Sinonimele sunt folosite pentru cazul în care există în diferite scheme din baza de date relații cu mai multe denumiri, când denumirile obiectelor sunt foarte lungi sau greu de ținut minte de către programatorul care trebuie să le folosească și în alte cazuri. Principalul avantaj al utilizării lor constă în micșorarea timpului necesar din partea utilizatorilor pentru a-și crea instrucțiunile SQL. În felul acesta, relațiile, viziunile sau alte obiecte au un nume suplimentar pentru acces.

Crearea sinonimului se face cu instrucțiunea *CREATE SYNONYM*:

```
CREATE [PUBLIC] SYNONYM <nume sinonim>
FOR <nume obiect>;
```

Parametrul *PUBLIC* definește faptul că acest sinonim este accesibil pentru toți utilizatorii. Dacă nu este precizat, atunci sinonimul este disponibil doar pentru utilizatorul curent. Parametrul *<nume sinonim>* reprezintă numele sinonimului care va fi creat, iar *<nume obiect>* identifică obiectul pentru care se dorește crearea sinonimului.

Sinonimul poate fi *public* sau *privat*. Sinonimul public este inclus în schema unui grup de utilizatori numit *PUBLIC* și este accesibil tuturor utilizatorilor, iar cel privat aparține numai unui anumit utilizator. Sinonimele publice pot fi create doar de către utilizatorii care au drepturi administrative. Sinonimul public este accesibil fără a se specifica numele utilizatorului care a creat obiectul. Astfel, toți utilizatorii se pot adresa la un obiect al unui utilizator fără a-i cunoaște numele și nici numele obiectului.

**Exemplul 4.43.** Se creează un sinonim public și unul privat pentru relațiile funcționari și departamente, respectiv, care au nume destul de lung:

```
CREATE PUBLIC SYNONYM sin1
FOR funcționari;
CREATE SYNONYM sin2
FOR departamente;
```

În felul acesta, utilizatorii nu mai sunt obligați să acceseze în interogările SQL, de exemplu, relația *funcționari* prin scrierea numelui “*funcționari*” și este suficientă doar utilizarea denumirii nou-create, “*sin1*”:

```
SELECT * FROM sin1;
```

Un sinonim nu poate fi schimbat odată creat. El poate să fie doar suprimat. Suprimarea unui sinonim nu implică și distrugerea obiectului de care acesta face referire. Suprimarea unui sinonim este făcută, folosind o instrucțiune SQL de tipul următor:

```
DROP [PUBLIC] SYNONYM <nume sinonim>;
```

unde *<nume sinonim>* desemnează numele sinonimului care trebuie să fie distrus.

Un sinonim de tip public poate să fie suprimat doar de către administratorul bazei de date. Sinonimele private trebuie distruse de utilizatorul care le-a creat, acestea nefiind disponibile în cadrul listei numelor de obiecte ale celorlalți utilizatori.

**Exemplul 4.44.** Suprimarea unui sinonim și a unui sinonim public:

```
DROP SYNONYM sin1;
DROP PUBLIC SYNONYM sin2;
```

Sinonimele sunt utilizate din motive de securitate și comoditate, incluzând:

1. Pentru a referi un obiect fără a specifica deținătorul obiectului.
2. Pentru a furniza un alt nume pentru obiect.

Există unele restricții în folosirca sinonimelor:

- Numele sinonimului privat trebuie să fie unic în cadrul grupului de obiecte pentru care acel utilizator este proprietar.
- Din motive de performanță, nu este recomandabilă utilizarea de sinonime la referirea de obiecte în aplicații.

## 4.5. Indecși

Un index este o structură de date care permite accelerarea accesului la datele bazei de date stocate în fișiere. Accederea la diferite tupluri ale aceleiași relații se realizează printr-un atribut sau mai multe atribute (numite attribute-cheie).

Indecșii sunt actualizați în mod automat de sistem, atunci când sunt efectuate operațiunile de modificare în baza de date. Acest aspect este foarte important pentru realizarea operațiilor de scriere. Scrierea datelor în relații este asistată de scrierea datelor și în indecși. Un număr mare de indecși pot face, uneori, mai lentă modificarea datelor. Cu toate acestea, în cazuri excepționale, beneficiile obținute compensează (pe larg) această penalizare.

### 4.5.1. Crearea și suprimarea indecșilor

Să se considere, de exemplu, o selecție condiționată de o valoare a unui atribut:

```
SELECT * FROM funcționari WHERE Nume = 'Paiu';
```

O modalitate de a găsi tuplul, pentru care atributul Nume este egal cu "Paiu", constă în scanarea întregii relații *funcționari*. Pentru a obține datele, sistemul parurge secvențial toate tuplurile relației și verifică care din ele respectă egalitatea. Dacă cardinalitatea relației este mare, această parcurgere consumă timp.

Un astfel de acces, se înțelege, conduce la tempi de răspuns prohibitivi pentru relații mai mari de câteva mii de tupluri. O soluție este crearea de indecsi, care vor satisface cele mai frecvente interogări cu timp de răspuns acceptabil. Astfel, dispunerea de un index asociat atributului *Nume* ar accelera calcularea interogării, deoarece accesul la tuplurile care satisfac egalitatea va fi direct sau cu ajutorul indexului.

Un index, de fapt, este format din chei auxiliare SQL care pot accede foarte repede la date. El este întotdeauna asociat cu o mulțime de atribute ale unei relații. Crearea indecsilor se face cu instrucțiunea *CREATE INDEX* în modul următor:

```
CREATE [UNIQUE] INDEX <nume index>
ON <nume relație> (<nume atribut1> [ASC | DESC]
[,<nume atribut2> [ASC | DESC] ] ...);
```

Pentru o relație pot fi creați mai mulți indecsi și fiecare din ei va fi asociat cu o mulțime de atribute ale relației.

Existența indecsilor nu este obligatorie. De asemenea, nu este obligatoriu ca atributele care definesc un index să ia valori unice sau nerepetabile. Cu toate acestea, poate fi specificată opțiunea *UNIQUE* care impune ca fiecare valoare a indexului să fie unică în cadrul relației. Cuvintele-cheie *ASC* și *DESC* specifică criteriul de ordonare ales, ascendent sau descendent. Implicită este opțiunea *ASC*.

Un index poate conține mai multe atribute. În acest caz, cheia de acces va fi reprezentată de concatenarea tuturor acestor atribute diferite. Asupra uneia și aceleiași relații pot fi creați mai mulți indecsi independenți.

**Exemplul 4.45.** Următoarea instrucțiune creează asupra relației funcționari un index definit pe atributul *Nume*:

```
CREATE INDEX index_NumeFunct
ON funcționari (Nume DESC);
```

Doi indecsi construiți de către același utilizator nu pot avea același nume (chiar dacă aceștia sunt definiți pe două relații diferite). Un index poate fi creat imediat după crearea schemei relației, dar și după inserarea tuplurilor. Acesta va fi apoi menținut în mod automat atunci când se modifică relația.

Suprimarea unui index din baza de date se realizează cu ajutorul instrucțiunii *DROP INDEX*, cu sintaxa:

*DROP INDEX <nume index> [ON <nume relație>];*

Numele relației trebuie să fie specificat doar dacă se dorește suprimarea indexului unei relații a altui utilizator, dacă indexul are același nume. Trebuie menționat că un index este șters automat atunci când se șterge relația pe care este construit.

**Exemplul 4.46.** Indexul *index\_NumeFunct* definit pe relația *funcționari* este șters de instrucțiunea:

*DROP INDEX index\_NumeFunct;*

Indecșii pot fi creați sau suprimiti doar pentru realizarea unei interogări concrete. Dar trebuie să se țină cont că crearea indecșilor necesită niște calcule și o memorie secundară de stocare.

#### 4.5.2. Utilitatea indecșilor

Pentru utilizatorii care formulează interogări SQL faptul că există sau nu un index cu nimic nu schimbă comportamentul acestora. Doar optimizatorul de interogări al SGBD-ului verifică dacă, la momentul executării fiecărei interogări, poate sau nu beneficia de ajutorul unui index.

Scopul principal al indexului e să accelereze căutările de date în baza de date. Un tuplu este instantaneu regăsit în cazul în care căutarea poate folosi un index. În caz contrar, trebuie să fie efectuată o scanare secvențială a tuturor tuplurilor din relație. Trebuie menționat faptul că datele căutate trebuie să corespundă aproximativ la mai puțin de 20% din toate tuplurile, dacă nu, o căutare secvențială este mai preferabilă.

Indexul concatenat (cu mai multe attribute) poate, în unele cazuri, recupera toate datele căutate fără a accede la relație. Iar o joncțiune se realizează, de multe ori, mai rapid în cazul în care attributele de joncțiune sunt indexate (dacă nu există prea multe valori egale în attributele indexate). De asemenea, indecșii accelerează sortarea datelor, dacă începutul cheii de sortare corespunde unui index.

O altă utilizare a indexului constă în asigurarea unicității unei chei, folosind opțiunea *UNIQUE*. Astfel, crearea indexului va împiedica inserarea, în relația *funcționari*, a numelui unui angajat existent:

```
CREATE UNIQUE INDEX Nume  
ON funcționari(Nume);
```

Trebuie să se conștientizeze faptul că schimbările de date sunt lente în cazul în care unul sau mai mulți indecsi trebuie să fie actualizați. În plus, căutarea datelor nu va fi accelerată de index, dacă indexul conține prea multe date egale. Nu este bine, de exemplu, să se indexeze un atribut, cum ar fi *Sex* care poate lua doar două valori: „*M*” sau „*F*”.

O altă problemă este legată de faptul că modificarea de date creează o situație de blocare pe o parte a indexului, pentru a evita denaturările legate de accesul concurențial la date. În aşa cazuri, accesul la date poate fi mai încetinit.

Cu alte cuvinte, nu întotdeauna e ușor de constatat dacă un index trebuie să fie creat sau nu. În cazul în care relația este foarte rar modificată, indexul numai ocupă spațiu, iar optimizatorul de interogări este cel care trebuie să aleagă să-l utilizeze sau nu. Dezavantajele sunt minime în acest caz.

Nu este același lucru, dacă relația este frecvent supusă schimbărilor. Nu ar trebui să se creeze un index doar, dacă se crede că va îmbunătăți cu adevărat performanța pentru interogările comune sau critice. Anterior, s-au menționat regulile generale care pot ajuta să se facă o alegere. De exemplu, un index nu va produce vreo ameliorare pentru o interogare în cazul în care mai mult de 20% din tupluri din cele existente sunt preluate. Aceste reguli nu trebuie luate literal, ci de multe ori trebuie să fie adaptate la situațiile particulare. Trebuie consultate, de asemenea, notele versiunilor de SGBD-uri care furnizează norme suplimentare care, deseori, depind de versiunile de optimizatoare de interogări. În acest caz, trebuie să se țină cont și de tipurile de index.

#### 4.5.3. Tipuri de indecsi

La nivelul fizic indexul are forma unui fișier suplimentar. Există mai multe tipuri de indecsi și o varietate de tehnici de implementare, unele din ele mai

sofisticate, dar care întotdeauna urmăresc obiectivul general: evitarea accesului secvențial la tuplurile unei relații.

Principiul de lucru al tuturor indecșilor este același: un algoritm oferă o căutare rapidă în indexul unei chei a unei înregistrări dezirabile și apoi indexul oferă un pointer pentru înregistrarea căutată corespunzătoare. Acest pointer este o valoare care determină poziția înregistrării în fișierele utilizate de SGBD. Pentru Oracle această valoare se numește *ROWID*.

**Indecșii B+-arbori.** Cei mai mulți indecși sunt implementați în baza B+-arborilor. Acești arbori reprezintă o variantă a B-arborilor ale căror chei sunt în frunze și ale cărui frunze formează o listă înlănțuită. Această înlănțuire permite o parcurgere rapidă a cheilor în ordinea cheilor înregistrate în index.

B-arborii sunt un tip de arbori echilibrați: algoritmi de adăugare și stergere a datelor asigură că toate ramurile să aibă aceeași adâncime. Teoretic, este inutil ca acești indecși să fie reorganizați periodic (dar mulți administratori de SGBD-uri cred că reconstrucția indecșilor ameliorează performanța). Este cunoscut faptul că arborii dezechilibrați au o performanță slabă în efectuarea căutărilor. De exemplu, în cel mai rău caz, un arbore poate fi reprezentat de o singură ramură care, de fapt, ar avea doar performanță unei liste simple.

Principalul avantaj al acestor arbori este faptul că ei reduc considerabil numărul de accesări ale discului de către SGBD. Fiecare nod al arborelui conține un număr de chei și fiecare cheie a unui nod e asociată cu un pointer spre un nod care conține cheile ce urmează din această cheie. Cheile de căutare sunt aranjate, astfel încât să constituie noduri de mărimea respectivă pentru ca, printr-o singură lectură de disc să se recupereze un nod întreg. Așadar, pentru a localiza înregistrările care se caută, se ajunge la foarte puține lecturi de disc. Trebuie să se țină cont că accesarea unui disc este aproximativ de un milion de ori mai lentă decât accesul în memoria centrală.

Un alt avantaj al acestei structuri e că pot fi citite cu ușurință cheileordonate. Astfel, aplicarea unei clauze *ORDER BY* poate beneficia de acest index.

Există și alte tipuri de indecsi, care pot fi utilizati în circumstanțe speciale. Cei mai frecvenți sunt următorii.

**Indecșii bitmap.** Acești indecsi sunt utili, numai atunci, când datele din relații sunt foarte rar actualizate. Indecșii bitmap sunt cel mai des utilizati în aplicații decizionale OLAP (*On Line Analytical Processing*) care facilitează luarea deciziilor referitoare la analiza datelor stocate de întreprindere. Implementarea este foarte diferită de cea a B-arborilor. Pentru fiecare valoare distinctă a atributului indexat, indexul conține un tablou de biți (atâtia biți câte tupluri sunt în relație), indicând fiecare tuplu al relației cu această valoare. Este ușor să se combine diferite tablouri de biți pentru a satisface selecții de tipul „*Atr1=valoare1 AND Atr2=valoare2*”, unde atributele *Atr1* și *Atr2* sunt indexate cu un index bitmap. În cazul în care indecsi bitmap sunt utilizati de către optimizatorul de interogări, pot fi obținute performanțe foarte bune. Pentru a accelera joncțiunile între relații, sistemul Oracle posedă un tip nou de index bitmap.

**Indexul de tip tabele cu dispersie.** Acest tip de indecsi nu mai este folosit, deoarece acestia nu oferă avantaje substanțiale în comparație cu B-arborii. Ei asigură un acces aproape instantaneu la o înregistrare atunci când este dată cheia, dar nu permit parcurgerea în ordine a cheilor și nu pot fi utilizati pentru selecții de tipul „*Atr>valoare*”. Acest tip de indecsi poate fi găsit în sistemul PostgreSQL, dar nu în Oracle.

**Indecși pe valori de funcții.** Acești indecsi pot fi folosiți pentru accelerarea căutărilor de tip „*Atr>funcție(...)*”. Rezultatele sunt valori ale unei funcții pe toate tuplurile relației care sunt indexate.

## 5. Protecția accesului și administrarea tranzacțiilor. SQL Integrat

Limbajul SQL folosește instrucțiuni pentru controlul accesului la baza de date. Mecanismul de securitate, în acest caz, se bazează pe concepțele de identifier de autorizație, posesiune, privilegiu. Sistemul de securitate poate fi descentralizat, unde utilizatorii sunt ei însăși responsabili pentru acordarea drepturilor de acces pentru obiectele pe care le dețin celorlalți utilizatori. Aceste aspecte vor fi examinate în prima secțiune.

Un obiectiv major al SGBD este de a permite mai multor utilizatori să acceseze concurențial datele partajate. Dar, la sistemele în care o BD este accesată simultan de mai mulți utilizatori, apar situații de conflict datorate accesului concurențial la datele care constituie resursă comună [Bernstein87]. Modul de rezolvare a conflictului depinde de natura cererilor de acces la date. Dacă interogările de acces sunt de tip consultare, atunci sevențialitatea accesului la mediul de memorare este suficientă și nu mai este nevoie de precauții suplimentare. În cazul în care interogările sunt de tip actualizare, este necesară aplicarea unor strategii adecvate de tratare a cererilor de acces. Strategiile sunt direct legate de conceptual de tranzacții și administrarea acestora. Acesta va fi subiectul secțiunii a doua, 5.2.

Există mai multe limbi de programare care acceptă folosirea limbajului SQL încorporat. În aceste limbi de programare, fiecare instrucțiune SQL trebuie precedată de anumite cuvinte-cheie. În același timp, integrarea limbajului SQL în limbajele procedurale pune mai multe probleme. În primul rând, explorarea tuplu cu tuplu a rezultatului returnat de interogările *SELECT* necesită utilizarea cursoarelor. În al doilea rând, variabilele limbajului-gazdă trebuie să fie trecute SGBD-ului pentru intermedierea cererilor SQL. Aceste probleme vor fi discutate în secțiunea „Limbajul SQL integrat”

## 5.1. Controlul accesului la baza de date

Mecanismele de protecție a datelor reprezintă un aspect important al aplicațiilor moderne ce lucrează cu baze de date. Pentru utilizatorii cunoscuți de SGBD și identificați printr-un nume și o parolă, din motive de securitate, accesul la o bază de date este restrictionat. În acest sens, administratorul bazei de date are posibilitatea de a alege și de a implementa politici adecvate de control al accesului la baza de date. Fiecare utilizator îi sunt atribuite anumite drepturi asupra schemelor și relațiilor fiecărei scheme.

Utilizatorul, la solicitarea sistemului, inserează numele și parola. Este deschisă o sesiune pentru care SGBD-ul cunoaște identificatorul utilizatorului curent.

### 5.1.1. Drepturi de acces

Ca regulă generală, utilizatorul care creează un obiect este proprietarul lui și este autorizat să efectueze orice operație asupra acestui obiect. În momentul creării unui obiect, sistemul acordă, în mod automat, toate privilegiile asupra obiectului creatorului său. Din cauza acestei limitări, SQL2 pune la dispoziție mecanisme de organizare ce permit administratorului să specifice acele obiecte la care utilizatorii au acces și cele la care nu au acces. Prin intermediul acestor mecanisme, utilizatorii disponu de *drepturi (privilegii) de acces* la obiectele sistemului.

Fiecare privilegiu de acces este caracterizat de:

- Obiectul la care face referire.
- Utilizatorul ce acordă privilegiul.
- Utilizatorul ce primește privilegiul.
- Operația permisă asupra obiectului.
- Posibilitatea acordării privilegiului altor utilizatori.

Proprietarul unei scheme, poate acorda drepturi altor utilizatori asupra acestei scheme sau asupra elementelor schemei. SQL2 definește șase tipuri de drepturi. Primele patru sunt cu privire la conținutul unei relații sau vizuini și sunt ușor de înțeles:

- *SELECT* permite citirea conținutului unei relații sau viziuni cu scopul de a-l utiliza în interogări.
- *INSERT* permite inserarea tuplurilor de valori într-o relație sau viziune.
- *UPDATE* permite modificarea valorilor din relații și viziuni.
- *DELETE* permite eliminarea tuplurilor din relații sau viziuni.

Există și alte două drepturi:

- *REFERENCES* oferă dreptul unui utilizator neproprietar al schemei să facă referință la o relație într-o constrângere de integritate. Acordarea acestui privilegiu asupra unei resurse poate conduce la limitarea posibilității de modificare a resursei.
- *USAGE* oferă dreptul unui utilizator neproprietar al schemei să utilizeze domeniile schemei.

Privilegiul de a efectua operațiile *DROP* sau *ALTER* nu poate fi acordat. Acest tip de privilegiu este deținut doar de proprietarul resursei.

Privilegiile se acordă sau se retrag cu ajutorul instrucțiunilor *GRANT* și *REVOKE*.

### 5.1.2. Acordarea și retragerea drepturilor

Gestiunea drepturilor de acces la relații și viziuni este descentralizat. Nu există vreun administrator central, care ar atribui drepturi. Fiecare creator de relație obține toate drepturile de acces la această relație, în particular, dreptul de a efectua acțiuni de selecție (*SELECT*), de inserție (*INSERT*), de suprimare (*DELETE*), de modificare (*UPDATE*), precum și de referire a relației într-o constrângere (*REFERENCES*). Apoi el poate trece selectiv aceste drepturi altor utilizatori sau tuturor (*PUBLIC*). Un drept poate fi trecut cu dreptul de a-l transmite (*WITH GRANT OPTION*) sau nu.

SQL propune, astfel, o instrucție de acordare a drepturilor cu următoarea sintaxă:

*GRANT <privilegii> ON <nume obiect> TO <receptor>  
[WITH GRANT OPTION]*

cu:

```

<privilegii> ::= ALL PRIVILEGES | <acțiune>[,<acțiune>...]
<acțiune> ::= SELECT | INSERT | DELETE
           | UPDATE [(<nume atribut> [,<nume atribut>...])]
           | REFERENCE [(<nume atribut> [,<nume atribut>...])]

<receptor> ::= PUBLIC | [(<nume utilizator>
                           [,<nume utilizator>...])]

```

Clauza *WITH GRANT OPTION* indică posibilitatea propagării privilegiului către alți utilizatori. Se pot utiliza cuvintele-cheie *ALL PRIVILEGES* pentru acordarea tuturor privilegiilor. Multimea de privilegii (*ALL PRIVILEGES*) include drepturi de administrare (modificarea schemei și distrugerea relației). Receptorul poate fi un utilizator sau un grup de utilizatori, potrivit identificatorului de autorizare acordat.

Cuvântul-cheie *PUBLIC* în locul numelui utilizatorului precizează că toți utilizatorii cunoscuți sistemului sunt vizitați.

**Exemplul 5.1.** Transferarea drepturilor de consultare și modificare a relației *funcționari* utilizatorului *Petrache* se efectuează în modul prezentat mai jos:

*GRANT SELECT, UPDATE ON funcționari TO Petrache  
WITH GRANT OPTION*

Prezența opțiunii de transmitere (*WITH GRANT OPTION*) permite lui *Petrache* să acorde aceste drepturi altor utilizatori.

În cazul privilegiului de modificare, acesta poate fi limitat la un atribut sau mai multe, care sunt specificate între paranteze.

**Exemplul 5.2.** Utilizatorului *Petrache* i se oferă dreptul de modificare a atributelor *Nume* și *Prenume* ale relației *funcționari*:

*GRANT UPDATE (Nume, Prenume) ON funcționari TO Petrache;*

Deși nu este specificat în standardul 1989, instrucțiunea *REVOKE* retrage dreptul de la utilizator. Sintaxa acestei instrucțiuni este următoarea:

*REVOKE <privilegii> ON <nom de table>  
FROM <receptor>.*

**Exemplul 5.3.** Instructiunea ce urmeaza retrage dreptul acordat mai sus, precum si toate drepturile care depind (adică cele de citire si modificare a relatiei functionari transmisse de Vasilache):

*REVOKE SELECT, UPDATE ON functionari  
FROM Vasilache.*

## 5.2. Administrarea tranzactiilor

O tranzactie este o unitate logica de prelucrare a bazei de date care include una sau mai multe operatii de acces la baza de date. Aceste operatii se realizeaza in baza de date sau toate, sau niciuna. In caz contrar, baza de date ajunge intr-o stare inconsistenta.

O tranzactie este finalizata:

- Printr-o validare care confirmă modificările operațiilor.
- Printr-o anulare care reduce baza de date în starea sa inițială.

În modelul plat al tranzactiilor (cel folosit de SQL), intr-o sesiune de lucru, două operațiuni nu se pot suprapune și nu pot fi imbricate una în interiorul celeilalte. Tranzacția trebuie să fie finalizată înainte ca o nouă tranzacție să poată începe.

Acest mecanism este, de asemenea, folosit pentru asigurarea integrității bazei de date în caz de terminare anormală a unei sarcini. Tranzacțiile nefinalizate sunt anulate în mod automat.

### 5.2.1. Proprietățile tranzactiilor

Pentru a rămâne o oglindire corectă a lumii reale, orice tranzacție trebuie să demonstreze proprietățile de atomicitate, consistentă, izolare și durabilitate (uneori abreviate ca ACID).

**Atomicitatea.** Din moment ce o tranzacție constă dintr-un set de operații, SGBD-ul trebuie să asigure fie că toate operațiile sunt realizate, fie niciuna.

**Consistența.** Tranzacțiile trebuie să părăsească baza de date într-o stare consistentă și nu trebuie să contravină constrângerilor de integritate sau să modifice datele într-o stare anormală.

**Izolarea.** În timp ce o tranzacție actualizează date partajate, datele pot fi temporar inconsistente. Asemenea date nu trebuie să fie puse la dispoziția altor tranzacții până când tranzacția în cauză nu și-a terminat operațiile ce foloseau aceste date. Gestionarul de tranzacții trebuie, deci, să ofere iluzia că o tranzacție rulează izolat față de alte tranzacții.

**Durabilitatea.** SGBD-ul trebuie să garanteze că modificările realizate de o tranzacție validată sunt conservate, chiar și în situație de pană.

Respectarea proprietăților AID este responsabilitatea gestionarului de tranzacții al SGBD-ului, în timp ce proprietatea C este în responsabilitatea utilizatorului (sau programatorului). Dar C este susținută prin I, deoarece utilizatorul nu ia în considerare interacțiunile cu alte tranzacții, precum și verificarea automată a constrângerilor de integritate de către SGBD. Proprietatea I este efectuată de sistemul de control al concurenței, iar AD sunt susținute de procedurile de recuperare după eșecuri.

### 5.2.2. Modele de tranzacții

#### 5.2.2.1. Tranzacții în SQL

În standardul SQL, o tranzacție începe cu debutul unei sesiuni de lucru sau imediat după încheierea tranzacției precedente. Ea se termină cu o instrucțiune de validare explicită (*COMMIT*) sau de anulare (*ROLLBACK*). Unele SGBD-uri nu îndeplinesc standardul și, pentru a demara o tranzacție, cer o comandă explicită.

Utilizatorul poate, în orice moment, să valideze (să finalizeze) tranzacția, apelând la comanda *COMMIT*. Modificările devin permanente și vizibile pentru toate celelalte tranzacții.

Utilizatorul poate anula (și finaliza) tranzacția curentă, utilizând comanda *ROLLBACK*. Toate modificările de la debutul tranzacției vor fi anulate.

Unele comenzi SQL, inclusiv definițiile de date (*CREATE TABLE...*), provoacă o validare automată a tranzacției.

Structura tranzacțiilor în standardul SQL este plată și înlănțuită:

- Două tranzacții nu se pot suprapune.

- Tranzacția începe atunci când precedenta se termină.

Acest model nu este întotdeauna corespunzător unei situații concrete, în special, pentru tranzacțiile de lungă durată și tranzacțiile în bazele de date stabilite pe mai multe stații:

- Tranzacțiile pot fi o sursă de frustrare pentru utilizator, dacă toate lucrările realizate de la începutul acesteia sunt anulate.
- Tranzacțiile păstrează până la sfârșitul lor lacătele, fapt ce afectează accesul concurențial.

Sunt propuse și alte modele de tranzacții pentru atenuarea acestui model.

#### 5.2.2.2. Tranzacții imbricate

Modelul de tranzacții imbricate permite unei tranzacții să posede subtranzacții-fifice, care, la rândul lor, pot, de asemenea, avea subtranzacții.

Anularea unei tranzacții-părinte derulează înapoi toate subtranzacțiile-fifice, însă anularea unei tranzacții-fifice nu anulează neapărat tranzacția-părinte. Dacă tranzacția-părinte nu este anulată, restul tranzacțiilor-fifice nu sunt anulate.

La anularea unei tranzacții-fifice, tranzacția-părinte poate:

- Stabili un tratament de substituție.
- Relua tranzacția anulată.
- Să se anuleze (în cazul în care nu se poate trece tranzacția anulată).
- Ignora anularea (în cazul în care tratamentul ce trebuia să-l efectueze tranzacția anulată nu este necesar).

Astfel, un tratament efectuat într-o tranzacție-fifice poate fi reluat sau poate fi înlocuit cu un tratament alternativ pentru a ajunge la sfârșitul tratamentului complet. Acest model este foarte potrivit pentru tranzacțiile cu termen lung și care activează pe multe stații (o tranzacție-fifice - pe o stație în rețea), care trebuie să facă față problemelor în caz de eșec de rețea.

Limbajul SQL nu suportă acest model de tranzacții.

#### 5.2.2.3. Puncte de reluare

Fără a trece la modelul de tranzacții imbricate, cele mai recente versiuni ale principalelor SGBD-uri (și standardul SQL3) au îmbunătățit modelul tranzacțiilor plate, cu puncte de control (numite, de asemenea, puncte de salvare; *savepoint* în limba engleză).

Punctele de control, într-o tranzacție, pot fi desemnate cu instrucțiunea:

*SAVEPOINT <nume punct>*.

Există posibilitatea de anulare a tuturor modificărilor efectuate, de la un punct de control, în cazul în care au existat probleme:

*ROLLBACK <nume punct>*.

Astfel, se evită anularea întregii tranzacții și se poate încerca remedierea problemei în locul anulării tranzacției globale.

#### 5.2.3. Anomalii de execuție concurrentă a tranzacțiilor

Tranzacțiile se pot executa în mod concurențial și este posibil să actualizeze aceleasi date ale bazei de date. Dacă execuția concurrentă a tranzacțiilor este necontrolată, baza de date poate să ajungă într-o stare inconsistentă, chiar dacă fiecare tranzacție în parte constituie un program corect.

**Actualizarea pierdută** (*lost update*). Actualizarea unui articol de date efectuată, de exemplu, de tranzacția  $T_1$  poate fi pierdută, dat fiind că tranzacția  $T_2$  folosește valoarea acestui articol de date, o modifică, înainte ca valoarea calculată de  $T_1$  să fie memorată în baza de date.

Pentru evitarea unor astfel de situații, SGBD-urile blochează accesul la relații (sau la părți de relații), atunci când alte accesări pot cauza probleme. Procesele care doresc să acceseze relațiile sunt blocate și puse în așteptare până la momentul când relațiile vor fi deblocate.

**Citirea inconsistentă sau improprie** (*dirty read*). Această anomalie poate să apară atunci când una dintre tranzacții este abandonată, iar altă

tranzacție concurentă a utilizat articolele modificate de prima, înainte de readucerea acestora la valoarea inițială. Fie că tranzacția  $T_2$  citește valoarea  $V$ , creată de altă tranzacție  $T_1$ . Mai târziu, însă, tranzacția  $T_1$  anulează afecțiunea sa asupra valorii  $V$ . Prin urmare, valoarea citită de  $T_2$  este falsă.

Acest caz poate apărea în cazul în care modificările efectuate de o tranzacție sunt vizibile pentru alte tranzacții înainte de validare (*COMMIT*). Aceasta este cel mai frecvent caz care are loc în prelucrarea concurențială a tranzacțiilor.

**Citirea irepetabilă** (*nonrepeatable read*) apare dacă o tranzacție  $T_1$  citește un articol de două ori, iar între cele două citiri, o altă tranzacție ( $T_2$ ) a modificat chiar acel articol. În această situație, tranzacția  $T_1$  primește două valori diferite ale aceluiași articol.

Pentru a evita această anomalie,  $T_1$  trebuie să blocheze, între două momente de timp, datele pe care dorește să le citească pentru a împiedica modificarea lor de către alte tranzacții.

**Citirea fantomă** (*phantom read*). Anomalia este asemănătoare problemei citirii irepetabile. Diferența dintre citirea fantomă și citirea irepetabilă constă în faptul că citirea fantomă apare, dacă tranzacțiile concurente modifică numărul de tupluri utilizate de tranzacția curentă (prin instrucțiuni *INSERT* sau *DELETE*), pe când citirea irepetabilă apare, dacă tranzacțiile concurente modifică valorile din tuplurile existente și prelucrate de tranzacția curentă (prin instrucțiuni *UPDATE*).

Citirea fantomă apare atunci când o tranzacție prelucreză un set de tupluri-rezultat al unei interogări. Dacă, în timpul acestei prelucrări, o altă tranzacție a inserat sau a șters un tuplu care satisface condiția interogării respective, atunci pot să apară sau să dispară tupluri din mulțimea de tupluri-rezultat, comportare asemănătoare cu apariția sau disparația fantomelor.

Blocarea tuplurilor nu este o soluție pentru acest fenomen, fiindcă aceste tupluri nu există în prima lectură. Este nevoie de multe ori să se efectueze

blocarea explicită a relației sau să se aleagă gradul de izolare *SERIALIZABLE*.

#### 5.2.4. Blocarea relațiilor și gestiunea tranzacțiilor

Dacă drepturile de inserție, suprimare sau modificare ale unei relații le posedă mai mulți utilizatori sau dacă un utilizator inițiază mai multe sesiuni de lucru asupra bazei de date, pot apărea operații simultane asupra unuia și acelorași articole de date. Aceste modificări simultane pot duce la apariția unor fenomene nedorite, la inconsistență datelor. SGBD-urile avansate au posibilitatea de a realiza operații asupra datelor în mod concurrent, cu un grad clar de paralelism. Dacă operațiile sunt de tipul *SELECT*, adică nu se modifică conținutul relației, ele se pot realiza fără probleme.

Drept normă, se consideră starea în care o operație de inserare, modificare sau suprimare a tuplurilor blochează relația, împiedicând accesul altor utilizatori asupra ei până operația nu se termină. Blocările pot fi predefinite, fixate automat de sistem sau definite explicit, declarate de utilizator.

Blocarea poate fi realizată la nivel de tupluri sau la nivel de relații în două moduri: exclusiv (*EXCLUSIVE*) sau partajat (*SHARE*).

Folosind instrucțiunea *LOCK TABLE*, un utilizator poate bloca o relație (apelând la ea pe nume sau sinonim). În afară de aceasta, instrucțiunea trebuie să specifică modul de blocare. Blocarea se elimină, folosind instrucțiunea *UNLOCK TABLE*.

*LOCK TABLE <nume relație> IN <mod> MODE;*  
*UNLOCK TABLE <nume relație>;*

În limbajul SQL, este posibilă, în mod explicit, alegerea nivelului de protecție contra incoerenței ce poate rezulta în urma accesului concurențial la date. Comportamentul predefinit este cel de asigurare a serializabilității și restabilirii stricte, dar acest mod poate provoca o încetinire a fluxului tranzacțional pentru aplicațiile care nu au nevoie de un control atât de strict.

Cea mai simplă opțiune este cea de specificare a tranzacției că este numai de citire. În aceste condiții, se poate garanta că ea nu va provoca nicio problemă de concurență și SGBD-ul poate economisi efortul de fixare a blocajelor. Instrucțiunea SQL este:

*SET TRANSACTION READ ONLY;*

Astfel, sunt interzise comenziile *INSERT*, *UPDATE*, *DELETE* până la imediat următorul *COMMIT* sau *ROLLBACK*. Sistemul va respinge aceste comenzi. Dar există și o consecință neplăcută: două lecturi succesive ale aceleiași instrucțiuni *SELECT* pot genera rezultate diferite, deoarece între cele două lecturi datele pot fi actualizate de alte tranzacții.

Opțiunea predefinită este cea de citire și scriere. Această opțiune poate fi specificată, în mod explicit, de instrucțiunea:

*SET TRANSACTION READ WRITE;*

### 5.2.5. Serializarea tranzacțiilor

Pentru a evita toate problemele descrise în secțiunea anterioară, gestiunea tranzacțiilor ar trebui, dacă este posibil, să facă tranzacțiile serializabile: executarea simultană a mai multor tranzacții ar trebui să producă același rezultat ca și executarea secvențială a tranzacțiilor.

Există mai multe moduri de a forța executarea tranzacțiilor concurente, astfel încât acestea să fie serializabile. Strategia cea mai utilizabilă este determinată de protocolul de blocare în două faze:

- Un obiect trebuie să fie blocat înainte de acționa asupra lui (citire sau scriere).
- Nu trebuie să fie obținută o blocare după eliberarea blocării.

Aceasta include două faze:

1. Achiziționarea tuturor blocărilor pentru obiectele pe care se va acționa.
2. Abandonarea tuturor blocărilor (în practică, de multe ori, pe *COMMIT* sau *ROLLBACK*).

Cu acest protocol, situațiile care ar putea apărea pot duce la probleme de concurență, provocând blocaje sau interblocaje (cu posibile redemarări ale tranzacțiilor invalidate pentru a desface interblocajele).

Pot fi folosite și alte strategii pentru a forța o executare serializabilă. Se pot, de exemplu, ștampila tranzacțiile. Toate tranzacțiile primesc o ștampilă, care indică momentul când au demarat. În cazul accesului concurențial, citirile sau scrierile pot fi puse în așteptare pe baza acestor ștampile. Acest tip de tratament este, de multe ori, prea restrictiv, deoarece se efectuează, în unele cazuri, blocaje chiar dacă situația nu duce la probleme, și, astfel, provoacă daune concurenței.

### 5.2.6. Niveluri de izolare a tranzacțiilor

Realizarea execuțiilor serializabile ale tranzacțiilor, adesea, este prea strictă și, prin limitarea accesului simultan la date, performanța sistemului poate degrada grav. SGBD-urile pot alege alte niveluri de izolare, care nu dau o garanție la fel de puternică ca serializarea completă, dar asigură, în același timp, o protecție bună împotriva problemelor de acces multiplu, oferind, totodată, performanțe bune.

În cazul în care unele aplicații nu cer o securitate foarte puternică, norma SQL2 permite devieri de la serializabilitatea strictă asigurată de SGBD. Cu acest scop, sunt propuse opțiuni mai puțin restrictive incluse în instrucțiunea:

```
SET TRANSACTION  
ISOLATION LEVEL <nivel izolare>;
```

unde *<nivel izolare>* poate fi una din opțiunile următoare.

**SERIALIZABLE.** Tranzacția se execută ca în cazul în care ar fi fost singura tranzacție în curs. Aceasta poate fi anulată în cazul în care SGBD-ul vede că o altă tranzacție a modificat datele utilizate de tranzacția curentă. Astfel, se evită problema numită citirea irepetabilă și a tuplurilor fantomă. Această metodă, totuși, este costisitoare, deoarece limitează funcționarea în paralel a tranzacțiilor.

**REPEATABLE READ.** Această opțiune previne citirea irepetabilă, dar nu și tuplurile fantomă.

**READ COMMITTED.** Acesta este modul de funcționare a majorității SGBD-urilor (în particular, Oracle). Modificările efectuate de către o tranzacție nu sunt cunoscute pentru alte tranzacții până când tranzacția nu a fost validată (*COMMIT*). Optiunea previne problemele principale de concurență, dar nu și lecturile irepetabile.

**READ UNCOMMITTED.** Alte tranzacții văd modificările unei tranzacții înainte de *COMMIT*. Aceasta este nivelul inferior de izolare, care nu se opune niciunei din problemele de concurență menționate mai sus și, prin urmare, este folosită numai în cazuri exceptionale.

Următoarea instrucțiune SQL stabilește nivelul de izolare al tranzacției care abia a început (trebuie să fie prima instrucțiune a tranzacției):

*SET TRANSACTION*

*ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED}*

Nivelurile de izolare determină modul în care SGBD-ul introduce diferitele mecanisme de control al concurenței. Pe orice nivel de izolare, inclusiv pe cel mai slab (*READ UNCOMMITTED*), se folosesc mecanisme de control al concurenței tranzacțiilor care previn pierderea actualizărilor.

Astfel de anomalii sunt foarte grave, baza de date nu reflectă operațiile care s-au efectuat asupra datelor și nici nu există vreo posibilitate de refacere a acestor pierderi. De aceea, nu este prevăzut niciun nivel de izolare care să permită pierderea actualizării datelor.

Nivel de izolare	Citire impropriie	Citire irepetabilă	Citire fantomă
SERIALIZABLE	Nu	Nu	Nu
REPEATABLE READ	Nu	Nu	Da
READ COMMITTED	Nu	Da	Da
READ UNCOMMITTED	Da	Da	Da

Figura 5.1. Niveluri de izolare a tranzacțiilor în SQL2

Pe toate nivelurile de izolare, cu excepția nivelului *SERIALIZABLE*, pot să apară diferite anomalii (cele date în figura 5.1), dar aceste anomalii sunt anomalii de citire, care pot fi gestionate de tranzacții și nu anomalii memorate permanent în baza de date. Cu cât nivelul de izolare a tranzacțiilor este mai scăzut, cu atât pot să apară mai multe anomalii de

actualizare, dar crește gradul de concurență a execuției și scade probabilitatea de apariție a impasului.

Nu toate SGBD-urile oferă toate posibilitățile. Oracle, de exemplu, permite doar nivelurile *SERIALIZABLE* și *READ COMMITTED*. Sistemul DB2 oferă nivelurile *UNCOMMITTED READ* (coresponde nivelului *READ UNCOMMITTED*), *CURSOR STABILITY* (coresponde nivelului *READ COMMITTED*, nivel implicit), *READ STABILITY* (coresponde nivelului *REPEATABLE READ*) și *REPEATABLE READ* (coresponde nivelului *SERIALIZABLE*; nu permite tupluri fantome).

Sistemul Oracle poate simula „*REPEATABLE READ*” blocând tuplurile citite din un „*SELECT FOR UPDATE*” din relațiile de date (citește adesea cu o pierdere semnificativă de performanță).

### 5.3. Limbajul SQL integrat

Problema integrării limbajului SQL într-un limbaj de programare a apărut odată cu bazele de date relaționale. Această problemă este soluționată prin două metode. Prima, constă în includerea limbajului de manipulare a datelor în limbajul de programare respectiv. A doua, mai ambițioasă, constă în extinderea limbajelor de programare cu clauze specifice de manipulare a datelor din baza de date. Alegerea uneia din aceste metode ține, până în prezent, de domeniul cercetărilor. Dar coexistența acestor două componente este complicată din motivul că, de regulă, limbajul de programare este unul procedural, în timp ce limbajul de manipulare a datelor SQL este unul declarativ.

Standardul SQL2 propune o normă specială de integrare cu un limbaj de programare numită SQL încorporat (*Embedded SQL*). Această normă presupune că atributele relațiilor manipulate, mai întâi, sunt declarate variabile ale limbajului de programare. Programul care accede la o bază de date, de regulă, conține instrucțiuni standard ale limbajului-gazdă, un compartiment de declarare a variabilelor, o mulțime de definiții ale cursoarelor și o mulțime de instrucțiuni SQL.

Integrarea limbajului SQL încorporat este, în general, făcută cu ajutorul unei tehnici de precompilare. Precompilatorul analizează comenzi SQL și apelează subprogramele limbajului-gazdă destinate SGBD-ului. Acesta

presupune, desigur, existența unei biblioteci de primitive care leagă limbajul-gazdă cu SGBD-ul. Precompilatorul construiește un program în limbajul-gazdă.

În limbajele SQL încorporate, o instrucție SQL este anunțată de *EXEC SQL* și se termină cu punct și virgulă. Toate variabilele sunt definite într-o secțiune specială a limbajului încorporat delimitată de *BEGIN DECLARE SECTION* și *END DECLARE SECTION*. Variabilele declarate trebuie să fie compatibile cu atributele respective din schema bazei de date. În afară de aceasta, fiecare program trebuie să includă o variabilă *SQLCODE*. La fiecare execuție a unei instrucții SQL, variabila *SQLCODE* indică dacă instrucția SQL a fost executată sau nu. Cursoarele sunt utilizate pentru prelucrarea tuplurilor în limbajul-gazdă. Mecanismul cursorului asigură un tratament câte-un-tuplu al multimii de tupluri furnizate de SGBD ca răspuns la o cerere SQL. Cursoarele pot fi deschise (*OPEN*), închise (*CLOSE*), poziționate pe un tuplu pentru citire (instrucția *FETCH*) și utilizate în calitate de poziție de referință (*CURRENT OF*).

Trebuie menționat, că tehnica utilizării limbajului SQL încorporat este bine definită și asigură bogate posibilități. Ea este, totuși, relativ greu de implementat și ne putem aștepta, în anii apropiati, la o abandonare treptată în folosul limbajelor de altă generație pentru manipularea datelor stocate în baze de date.

### 5.3.1. Structura unui program cu SQL încorporat

Structura tipică a unui program de manipulare a bazei de date este următoarea:

*Programul ...*

*Declarare variabile*

...  
*Declarare variabile-gazdă pentru accesul la baza de date*

...  
*Sfârșit declarare variabile-gazdă pentru accesul la baza de date*

*Sfârșit declarare variabile*

*Inceput cod program*

*... Instrucțiuni limbaj-gazdă*

*... Conectarea la baze de date*

*... Instrucțiuni SQL*

*... Deconectarea de la baza de date*

*... Instrucțiuni limbaj-gazdă*

*Sfărșit cod program*

Evident că includerea instrucțiunilor SQL într-un program necesită ca acesta să fie dotat cu unele elemente noi de programare. Elementele incluse trebuie să admită în programul-aplicație instrucțiuni scrise într-un limbaj (SQL) străin de programare și trebuie să susțină comunicarea între programul-aplicație și SGBD.

Între aceste elemente pot fi numite următoarele:

- **Delimitatoarele.** Delimitatoarele sunt particule adăugate la sintaxa SQL pentru a-l deosebi de codul din programul-sursă.
- **Aria de comunicație.** Este zona de date definită în programul-aplicație, care este utilizată de SGBD pentru a raporta dacă ultima cerere s-a executat corect sau nu.
- **Variabilele limbajului-gazdă.** O variabilă a limbajului-gazdă reprezintă o variabilă din program, declarată în limbajul-gazdă. Ea servește, în principal, pentru păstrarea datelor pe care programul le citește sau le scrie în baza de date.

#### 5.3.1.1. Delimitatoarele

Compilatorul limbajului-gazdă nu înțelege instrucțiunile SQL ca pe instrucțiunile propriului limbaj. De aceea, instrucțiunile SQL trebuie scrise între delimitatoare pentru a fi distinse în textul programului-sursă și a permite precompilatorului să fie prelucrate.

Toate instrucțiunile limbajului SQL încorporat sunt precedate de cuvintele rezervate *EXEC SQL* și finalizează cu un simbol special, astfel fiind ușor identificate de compilator. În limbajul C, acest simbol este punct și virgulă, „;”.

### 5.3.1.2. Aria de comunicație a SQL

Atunci când se execută un program-aplicație care conține instrucțiuni ale limbajului SQL încorporat, acestea sunt executate de SGBD și apoi, în program, sunt returnate datele cerute, dacă ele, bineînțeles, există în baza de date. În afară de aceasta, programului scris în limbajul-gazdă trebuie furnizate date care ar indica dacă interogarea sa s-a executat corect sau nu. Pentru aceasta, se utilizează o zonă definită în program care se numește *aria de comunicație* a limbajului SQL (*SQLCA*, *SQL Communication Area*). Programul-aplicație poate examina zona *SQLCA*, pentru a determina succesul sau eșecul fiecărei instrucțiuni SQL.

*SQLCA* constă dintr-o mulțime de variabile care păstrează date asupra stării (și erorile, dacă se produc) a ultimei instrucțiuni executate asupra bazei de date.

Pentru a putea face uz de aceste variabile există o declarație care are următoarea formă și care, de obicei, se include la începutul programului:

*EXEC SQL INCLUDE SQLCA;*

Aceasta îi va comunica precompilatorului să includă în program structura de date *SQLCA*. Cea mai importantă parte a acestei structuri o reprezintă variabila *SQLCODE*, care este utilizată pentru verificarea existenței erorilor. Într-un program, orice instrucțiune SQL trebuie să fie urmată de un cod care controlează valorile variabilelor *SQLCODE* și *SQLSTATE*.

Variabila *SQLCODE* specifică exact eroarea apărută. Dacă variabila *SQLCODE* ia o valoare negativă, aceasta indică că a apărut o eroare. Valoare egală cu zero semnalează că instrucțiunea s-a executat cu succes, iar o valoare pozitivă denotă faptul că instrucțiunea a fost executată cu succes, dar a apărut o situație excepțională, cum ar fi încetarea returnării tuplurilor de către instrucțiunea *SELECT*.

În ce privește executarea instrucțiunilor, pentru a simplifica această sarcină, limbajul încorporat oferă o instrucțiune pentru controlul general al erorilor. Instrucțiunea dată se numește *WHENEVER* și reprezintă o directivă adresată precompilatorului, cu scopul generării automate a unui cod pentru tratarea erorilor:

*EXEC SQL WHENEVER <condiție><acțiune>;*

unde

```
<condiție> ::= {SQLERROR | SQLWARNING
                  | NOT FOUND}
<acțiune> ::= {CONTINUE | GO TO <etichetă>}
```

- *NOT FOUND* se evaluează la adevărat, dacă nu s-au găsit date ce satisfac condițiile specificate în instrucțiune (*SQLCODE* =+100).
- *SQLERROR* se evaluează la adevărat, dacă a apărut o eroare în executarea instrucțiunii (*SQLCODE* < 0).
- *SQLWARNING* îi comunică precompilatorului să trateze avertizările (*SQLCODE* > 0).

Instrucțiunea *WHENEVER* nu este o instrucțiune executabilă, este o directivă la prelucrarea limbajului SQL. Astfel:

- *WHENEVER <condiție> GO TO <etichetă>* semnifică că procesorul de SQL va include în orice instrucțiune SQL al programului instrucțiunea *IF <condiție> THEN GO TO <etichetă>*, adică controlul va fi transferat către eticheta specificată.
- *WHENEVER <condiție> CONTINUE* semnifică că procesorul de SQL nu realizează nicio acțiune, lăsând responsabilitatea de control al fluxului programului programatorului, adică se ignoră condiția și se trece la următoarea instrucțiune.

### 5.3.1.3. Variabilele programului

Variabilele limbajului-gazdă sunt utilizate în cadrul instrucțiunilor SQL încorporat, pentru transferarea datelor din baza de date în program și invers. Variabilele limbajului utilizate în instrucțiunile SQL (variabile-gazdă) trebuie să fie declarate într-o secțiune specială, care are un început și un sfârșit clar definite:

```
EXEC SQL BEGIN DECLARE SECTION;
... declararea tipurilor și variabilelor în C
EXEC SQL END DECLARE SECTION;
```

Acest bloc trebuie să apară înainte de utilizarea vreunei variabile într-o instrucțiune SQL încorporat.

**Exemplul 5.4.** Declarațiile sunt similare celor dintr-un program C și, ca în C, sunt separate de punct și virgulă. De exemplu, în continuare, sunt declarate variabilele *cd*, *aux*, *cpx*, *cpx*:

```
EXEC SQL BEGIN DECLARE SECTION;
char cd[6];
int aux;
char cpx[4];
char cpy[4];
EXEC SQL END DECLARE SECTION;
```

Variabilele trebuie să posede tipuri ce corespund utilizării. Totodată, ele trebuie să fie compatibile cu valorile SQL pe care le reprezintă. Compatibilitatea tipurilor între limbajele de programare și SGBD depinde de fiecare sistem particular. Variabilele din limbajele-gazdă pot fi singulare sau structuri. Din punct de vedere sintactic, variabilele pot apărea într-o instrucție SQL, în orice loc unde poate apărea un literal și trebuie să fie precedate de simbolul două puncte, “::”.

Dintre instrucțiunile SQL care sunt utilizate, prezintă interes două: conectarea la baza de date și deconectarea. Aceste instrucții au următoarea sintaxă:

```
EXEC SQL CONNECT 'baza_de_date';
EXEC SQL DISCONNECT;
```

Trebuie menționat că, înainte de efectuarea oricărei operații asupra bazei de date, este necesară conectarea la aceasta, utilizând prima instrucție. La fel, pentru o funcționare corectă a sesiunii bazei de date pe server, este obligatorie deconectarea de la baza de date, înainte ca programul să finalizeze.

### 5.3.2. Manipularea datelor fără cursoare

Dat fiind faptul că SQL interactiv, deseori, prelucreză multimi de tupluri în loc de tupluri izolate, iar limbajele de programare manipulează o înregistrare odată, în SQL încorporat au fost introduse cursoarele. Cursorul este mecanismul care permite navigația într-o mulțime de tupluri pentru a fi manipulate unu câte unu.

Cu toate acestea, există o serie de operații care nu implică cursoare. Acestea sunt: *SELECT*, *INSERT*, *UPDATE* și *DELETE*.

Structura instrucțiunii *SELECT* în SQL încorporat este:

```
EXEC SQL SELECT [ALL | DISTINCT]
    <listă elemente selecție>
    INTO <listă variabile>
    FROM <listă relații>
    [WHERE <condiție căutare>]
    [GROUP BY <listă atribute>]
    [HAVING <condiție căutare>]
```

**Exemplul 5.5.** Între elementele listei *SELECT* și variabilele din limbajul-gazdă trebuie să existe o corespondență biunivocă. Astfel, obținerea codului departamentului, unde lucrează profesorul cu codul 'JRF' poate fi solicitată, apelând la instrucțiunea:

```
EXEC SQL SELECT Cod_departament INTO :vector
FROM profesori WHERE Cod_Pro = 'JRF';
```

Aici variabila *vector* ia valoarea atributului *Cod\_departament* din relația *profesori* care satisface condiția din clauza *WHERE*. În acest context, instrucțiunea *SELECT* trebuie să întoarcă un singur tuplu, în caz contrar, se va produce o eroare.

Precum se poate observa, varianta instrucțiunii este dictată de clauza *INTO*, care specifică lista de variabile ale programului care vor fi utilizate pentru recepționarea datelor selectate.

Instrucțiunea de modificare *UPDATE* permite modificarea unuia sau mai multor tupluri ale unei relații; *INSERT* permite adăugarea unui tuplu la o relație, iar *DELETE* permite eliminarea unuia sau mai multor tupluri, având aceeași sintaxă ca în SQL interactiv cu o singură diferență că poate include variabile-gazdă.

**Exemplul 5.6.** În acest exemplu de modificare a relației *invatamant*, se schimbă valoarea codului unui profesor care este egală cu cea citită în variabila *cpx* cu codul profesorului păstrat în variabila *cpy* (*cpx* și *cpy* sunt declarate în exemplul 5.4).

```
EXEC SQL UPDATE invatamant
```

```
SET Cod_Pro = :cdy
WHERE Cod_Pro = :cdx;
```

**Exemplul 5.7.** Un exemplu implementat în C cu SQL incorporat poate arăta precum urmează:

```
#include <stdio.h>
EXEC SQL include sqlca;
main() {
    EXEC SQL BEGIN DECLARE SECTION;
    int cant;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT 'dbvanzari';
    EXEC SQL SELECT COUNT(*)
        INTO :cant FROM clienti;
    printf(„Numărul de tupluri este %d \n”, cant);
}
```

### 5.3.3. Manipularea datelor cu cursoare

În secțiunea precedentă, instrucțiunea *SELECT* a fost utilizată pentru găsirea datelor solicitate, dar prelucrarea lor poate fi mai complicată, dacă procesul de interogare întoarce în calitate de rezultat mai mult decât un tuplu. Complicațiile rezultă din faptul că majoritatea limbajelor de programare de nivel înalt pot prelucra tuplurile în mod individual. Intuitiv cursorul poate fi conceput ca un pointer la tuplurile unei relații obținute în urma interogării.

Un cursor întotdeauna este asociat unei relații care este specificată în instrucțiunea de declarare. Deoarece obiectivul utilizării unui cursor constă în navigarea prin tuplurile unei relației și manipularea lor ulterioră, aceste tupluri trebuie să fie ordonate. Ordinea poate fi definită implicit de sistem sau definită de programator în declarația cursorului.

Îndată ce cursorul este activat (deschis), el întotdeauna ia o poziție într-o mulțime ordonată de tupluri, numită *poziție curentă*. Această poziție poate fi: *înaintea* unui tuplu, *într-un* tuplu sau *după* un tuplu. Poziția curentă este relevantă pentru instrucțiunile de manipulare a bazei de date care utilizează cursoarele.

Sintaxa generală a definiției unui cursor este:

```
EXEC SQL DECLARE <cursor> CURSOR
FOR <cursor specificare>
<cursor specificare> ::= <expresie relație>
[ORDER BY <listă element ordine>]
[FOR {READ ONLY | UPDATE
      [OF <listă nume atribut>]}
<element ordine> ::= {<nume atribut>
| <întreg pozitiv>} [ASC | DESC]
```

Aici *<expresie relație>* definește relația asociată cursorului, care este o instrucție *SELECT*. Sintaxa ei este aceeași, ca în SQL interactiv, cu deosebirea că poate include variabile-gazdă.

Clauza *ORDER BY* permite specificarea unei ordini pentru tuplurile relației definite de *<expresie relație>*. Numele de atrbute sau întregii pozitivi (ordinea poziției unui atribut) se referă la atrbutele relației definite de *<expresie relație>*. Întregii pozitivi se utilizează atunci când atributul respectiv al relației nu are un nume, adică dacă reprezintă o dată calculată. Dacă nu se include clauza *ORDER BY*, sistemul stabilește ordinea sa implicită.

Clauza *FOR READ ONLY* (sau *UPDATE [OF <listă nume atrbuit>]*) indică că relația asociată cursorului nu poate (sau poate) fi actualizată cu aplicarea instrucțiunilor *UPDATE* sau *DELETE*.

Instrucția *OPEN* este utilizată pentru deschiderea unui anumit cursor. Formatul instrucției *OPEN* este:

```
EXEC SQL OPEN <cursor>[FOR READONLY];
```

Cuvântul-cheie optional *[FOR READONLY]* indică faptul că datele nu vor fi actualizate, ci vor fi extrase tuplurile din baza de date. Deschiderea cursorului într-un program semnifică evaluarea *<expresie relație>* asociate. Când un cursor se deschide, poziția lui curentă este în dreptul primului tuplu.

Manipularea cursorului cu scopul de a găsi următorul tuplu în relația returnată de interogare se efectuează cu instrucția *FETCH*. Formatul instrucției *FETCH* este:

*EXEC SQL FETCH [[<selector de tuplu>] FROM] <cursor> INTO <listă variabile>;*

Instrucțiunea *FETCH* permite mișcarea cursorului în mulțimea ordonată de tupluri și selectarea (citirea) tuplului în care a rămas plasat. Poziția lui curentă, după ce se execută instrucțiunea, rămâne în tuplul selectat.

Aici *<selector de tuplu>* poate fi: *NEXT*, *PRIOR*, *FIRST*, *LAST*, *ABSOLUTE N*, *RELATIVE N*. Pentru cazurile *NEXT*, *PRIOR* și *RELATIVE* mișcarea cursorului este relativă poziției curente înainte de executarea instrucțiunii *FETCH*. În opțiunile *ABSOLUTE N* și *RELATIVE N* un număr pozitiv (respectiv, negativ) indică mișcarea făcută înainte (respectiv, înapoi), urmând secvența ordonată de tupluri.

Lista *<listă variabile>* trebuie să conțină câte o variabilă-gazdă pentru fiecare atribut al relației asociate cursorului. Aceste variabile recepționează valorile tuplului selectat.

Dacă instrucțiunea *FETCH* nu selectează niciun tuplu, adică nu există tuplul următor (*NEXT*) sau tuplul anterior (*PRIOR*) în raport cu tuplul din poziția curentă a cursorului, sau nu există un asemenea tuplu, pornind de la poziția curentă (absolută sau relativă) înainte (*N* pozitiv) sau înapoi (*N* negativ), atunci poziția curentă a cursorului, după ce se execută instrucțiunea, este sau după ultimul tuplu, sau înaintea primului tuplu.

Formatul instrucțiunii *CLOSE* este foarte asemănător cu cel al instrucțiunii *OPEN*:

*EXEC SQL CLOSE <cursor>*

unde *<cursor>* reprezintă denumirea cursorului care este deschis în momentul respectiv. Îndată ce cursorul este închis, relația întoarsă de interogare nu mai este accesibilă. Toate cursoarele sunt automat închise, la încheierea tranzacției care le conține.

Cu ajutorul unui cursor este posibilă ștergerea tuplurilor din relația returnată de interogare. Formatul instrucțiunii *DELETE* bazate pe cursor este:

*EXEC SQL DELETE FROM <relație>  
WHERE CURRENT OF <cursor>;*

Această instrucție permite suprimarea tuplului în poziția curentă a cursorului, iar *<relație>* este relația care va fi actualizată (cursorul trebuie să fie asociat ei). După ce se execută instrucția, poziția curentă a cursorului trece înaintea tuplului următor, după tuplul care a fost suprimit sau înaintea ultimului tuplu, dacă tuplul eliminat era ultimul în relație.

De asemenea, poate fi definită instrucția *UPDATE* bazată pe cursor:

```
EXEC SQL UPDATE <relație>
SET <listă atribuire>
WHERE CURRENT OF <cursor>;
```

Această instrucție permite actualizarea tuplului în poziția curentă a cursorului. În *<listă atribuire>* se indică actualizările ce vor fi aplicate asupra atributelor tuplului curent. Sintaxa acestei clauze este similară celei din SQL interactiv cu excepția că se pot include referințe la variabilele gazdă.

Executarea acestei instrucții nu modifică poziția curentă a cursorului.

#### 5.3.4. Un exemplu integrator

Urmează un exemplu de program în limbajul C (utilizând SQL încorporat), care imprimă un raport. Acest gen de programe trebuie să fie precompilate, ținând cont de instrucțiunile SQL, înainte de a fi compilat normal. Cu toate acestea, programul de față nu este executabil, el poate doar servi în calitate de exemplu.

```
#include <stdio.h>
/* Această secțiune declară variabilele locale */
EXEC SQL BEGIN DECLARE SECTION;
    int ID_Cumpărător;
    char Nume[100], Prenume[100], Produs[100];
EXEC SQL END DECLARE SECTION;
/* Această instrucție include variabila SQLCA */
EXEC SQL INCLUDE SQLCA;
main() {
```

```

/* Se indică o cale posibilă pentru conectarea la baza de date */
EXEC SQL CONNECT UserID/Password;
/* Acest cod informează dacă s-a produs conectarea la baza de date
sau dacă a apărut vreo eroare pe durata conectării */
if(sqlca.sqlcode)
{
    printf(Printer, "Eroare conectare server.\n");
    exit();
}
printf("Conectat la serverul bazei de date.\n");
/* Urmează declarația unui cursor. Acesta este utilizat în cazul în
care interogarea înfoarce mai mult de un tuplu și va fi realizată o
operăție pentru fiecare tuplu rezultat din interogare. Apoi, pentru a
extrage fiecare tuplu, este utilizată instrucțunea FETCH. Dar
pentru interogarea care este actualmente executată, se va utiliza
statutul OPEN. Aici DECLARE, pur și simplu, stabilește
interogarea.*/
EXEC SQL DECLARE ProdusCursor CURSOR FOR
    SELECT Produs, ID_Cumpărător
        FROM antichități
        ORDER BY Produs;
EXEC SQL OPEN ProdusCursor;
/* FETCH pune valorile următorului tuplu al interogării în
variabilele locale.*/
EXEC SQL FETCH ProdusCursor
    INTO :Produs, :ID_Cumpărător;
while(!sqlca.sqlcode)
{
    /* Cu fiecare tuplu, în afară de aceasta, se realizează două activități.
Se ridică pretul cu $5 și se extrage numele cumpărătorului. Cu acest
scop, se utilizează UPDATE și SELECT, înainte de imprimarea
tuplului pe ecran. În afară de aceasta, să se observe prezența celor
două puncte înaintea numelor de variabile locale, atunci când se
utilizează înăuntrul instrucțiunii SQL.*/

```

```
EXEC SQL UPDATE antichitati
SET Preț = Preț + 5
WHERE Produs = :Produs
      AND ID_Cumpărător = :ID_Cumpărător;
EXEC SQL SELECT NumeProprietar, PrenumeProprietar
INTO :Nume, :Prenume
FROM proprietari_antichitati
WHERE ID_Cumpărător = :ID_Cumpărător;
printf("%25s %25s %25s", Nume, Prenume, Produs);
EXEC SQL FETCH ProdusCursor
      INTO :Produs, :ID_Cumpărător;
}
/* Închiderea cursorului */
EXEC SQL CLOSE ProdusCursor;
EXEC SQL DISCONNECT;
exit();
}
```

## 6. Proprietăți obiect-relaționale în standardul SQL3

---

Prin sistemele de baze de date active, se constituie un nou nivel de independentă a datelor: independentă cunoștințelor [Dittrich95]. Cunoștințele care provoacă o reacție sunt eliminate din programele aplicații și sunt codificate în formă de reguli active. Astfel, există reguli definite ca parte componentă a schemei bazei de date, care sunt partajate tuturor utilizatorilor și nu replicate în toate programele aplicații. Orice schimbare a comportamentului de reacție poate fi realizat, schimbând numai regulile active, fără a fi necesară modificarea aplicațiilor.

Cu ajutorul sistemelor de baze de date active, este posibilă integrarea diferitelor subsisteme (controlul accesului, gestiunea viziunilor etc.) și se extinde raza de aplicare a tehnologiilor bazelor de date cu alte tipuri de aplicații. La baza sistemelor active, sunt puse declanșatoarele (triggers) [Cochrane96].

În afară de aceasta, odată cu apariția primelor baze de date orientate pe obiecte, numeroase voci au anunțat apropiata moarte a sistemelor relaționale care dominau atunci pe piața sistemelor de gestiune a bazelor de date, începând cu anii șaptezeci și care păreau, din ce în ce mai mult, depășite de complexitatea datelor pe care trebuie să le gestioneze [Melton96].

În realitate, istoria nu s-a derulat tocmai în acest mod: bazele de date orientate pe obiecte posedă, de asemenea, unele limitări, iar modelul relațional posedă unele calități, precum integritatea referențială, care îl face greu substituibil.

Marii producători de SGBD-uri, conștienți de succesul paradigmei orientarea pe obiecte, au început să evite problema, oferind posibilități de stocare a datelor non-structurate în câmpuri speciale numite BLOB (*Binary Large Object*). Ei au extins apoi modelul relațional spre un număr

de concepte orientate pe obiecte. Această extindere este, de obicei, numită obiect-relațională și va fi examinată în această secțiune.

În decenile următoare, SGBD-urile obiect relaționale (SGBDOR) vor constitui una din tehnologiile cele mai mari în domeniul bazelor de date, în principal, pentru că este construită în baza tehnologiei relaționale, care a trecut verificarea mai mult de treizeci de ani. Această tehnologie nouă este încă la începutul dezvoltării sale și, de aceea, lipsește o normă internațională pentru a favoriza adoptarea ei pe scară largă. Prima normă nu a fost mult așteptată: ea a venit sub forma unui nou standard, SQL3, care conține numeroase clauze consacrate tehnologiei obiect-relaționale.

## 6.1. Declanșatoarele și standardul SQL3

În multe aplicații, bazele de date trebuie să evolueze, independent de intervenția utilizatorului, ca răspuns la un eveniment sau la o situație determinată [Ceri90]. În sistemele de gestiune a bazelor de date tradiționale (pasive), evoluția bazei de date se programează în codul aplicațiilor, în timp ce în sistemele de gestiune a bazelor de date active această evoluție este autonomă și se definește în schema bazei de date.

Poziibilitatea de a specifica reguli cu o serie de acțiuni care se execută, în mod automat, când se produc anumite evenimente, este una din cele mai bune performanțe, care au atins SGBD-urile în ultimul timp. Prin reguli, se pot impune respectarea constrângerilor de integritate, generarea datelor derivate, controlarea securității sau implementarea unor reguli de business. De fapt, majoritatea sistemelor relaționale comerciale dispun de declanșatoare. De la apariția primelor declanșatoare, s-au făcut multe investigații asupra problemei cum trebuie să arate un model general de funcționare a bazelor de date active. Modelul utilizat pentru descrierea declanșatoarelor se bazează pe noțiunea *eveniment–condiție–acțiune* (ECA) [Widom96].

### 6.1.1. Definirea declanșatoarelor

Majoritatea SGBD-urilor comerciale utilizează declanșatoarele pentru activarea automată a altor programe care trebuie să ruleze la efectuarea unor modificări. Însă, funcționalitatea activă a acestor sisteme este

foarte limitată în comparație cu funcționalitatea prototipurilor de cercetare existente. Cu toate acestea, capacitatea acestor sisteme este suficientă pentru a reda bazelor de date un comportament relativ complex.

Utilizarea SGBD-urilor cu caracteristici active este însosită de unele probleme, cum ar fi:

- **Absența unui standard.** Diverse SGBD-uri posedă o gamă variată de caracteristici în termeni de sintaxă și comportament de executare a regulilor în baza modelului ECA.
- **Absența unei semantici de executare bine definite.** Sunt posibile construcții alternative (declanșare la nivel de tuplu sau instrucțiune; executare de tip imediat sau întârziat), dar, în general, nu este specificat exact cum declanșatoarele trebuie să se comporte în cazul în care sunt definite mai multe declanșatoare cu diferite acțiuni.
- **Absența caracteristicilor avansate utile prezente în multe prototipuri de cercetare,** cum ar fi adresarea aplicației la evenimente, tehnicele de compozitie a evenimentelor, legătura evenimentelor cu condițiile și legătura condițiilor cu acțiunile.
- **Constrângerile.** Sistemele de gestiune limitează, în general, numărul de declanșatoare sau de interacțiuni între declanșatoare.

Standardul SQL3, bineînțeles, include în componență un elementul activ, declanșatorul. Declanșatoarele în SQL3 se conformează paradigmiei de reguli ECA acceptată în bazele de date active. Evenimentul, în acest standard, reprezintă o activitate a bazei de date care este monitorizată de SGBD, condiția reprezintă un predicat arbitrar SQL, iar acțiunea e redată de o secvență de instrucțiuni SQL. Instrucțiunile acțiunilor sunt executate secvențial. Executarea acțiunilor are loc în cazul în care, dacă apare evenimentul, iar condiția este evaluată la *adevărat*.

Sintaxa instrucțiunii de definire a declanșatorului (*TRIGGER*) este următoarea:

*CREATE TRIGGER < nume declanșator >*

{*BEFORE | AFTER | INSTEAD OF*}  
 {*INSERT | DELETE | UPDATE [OF <atribute>]*}  
*ON <nume relație>*  
 [*ORDER <ordine valoare>*]

*Eveniment*

[*REFERENCING OLD [ROW]*  
 [*AS*] <tuplu vechi> |  
*NEW [ROW] [AS]* <tuplu nou> |  
*OLD TABLE [AS]* <relație veche> |  
*NEW TABLE [AS]* <relație nouă>]  
 [{*FOR EACH ROW | FOR EACH STATEMENT*}]

*Nume de corelare*

*Granularitate*

[*WHEN (<condiție>)*]

*Condiție*

(<instrucție SQL> | *BEGIN ATOMIC*  
 <instrucție1 SQL>  
 ...  
 <instrucțieN SQL>  
*END*);

*ACTION*

Aici sunt necesare unele comentarii:

- Un declanșator este definit pe o relație concretă (la nivel de eveniment și condiție, deși acțiunea se poate realiza pe o altă relație).
- Un declanșator poate controla numai un eveniment (asupra relației asociate) și evenimentul poate fi de inserare, suprimare și modificare a datelor în relații.
- Un declanșator poate fi executat înainte, după sau în locul unei operații care îl lansează.
- Clauza *ORDER* poate fi utilizată pentru specificarea priorităților.
- Un declanșator se poate referi la valorile anterioare sau posterioare execuțării operației de declanșare. Clauza *REFERENCING* este utilizată pentru asocierea numerelor cu valorile anterioare și posterioare modificării făcute de operație.
- Clauza *WHEN* specifică condiția declanșatorului.
- *FOR EACH ROW* indică faptul că declanșatorul este executat o dată pentru fiecare tuplu modificat.

- *FOR EACH STATEMENT* specifică faptul că declanșatorul este executat o dată pentru toată instrucțiunea SQL.

### 6.1.2. Evenimente

Precum s-a menționat, un declanșator poate controla un singur eveniment asupra unei relații. Acest eveniment poate fi o inserare (*INSERT*), o suprimare (*DELETE*) sau o modificare (*UPDATE*). Dat fiind faptul că o inserare sau o suprimare într-o relație întotdeauna afectează tupluri complete, nu se admite specificarea detaliată (pe attribute) a evenimentului, în aceste cazuri. O modificare, însă, poate să se refere nu neapărat la toate attributele relației, ci la o parte doar. Astfel, dacă se dorește controlul modificării unor attribute specificate, se utilizează sintaxa:

*UPDATE OF <atribut1>[,<atribut2>,...] ON <nume relație>.*

În cazul în care modificării sunt supuse toate attributele schemei relației, sintaxa are o formă mai simplă:

*UPDATE ON <nume relație>.*

#### 6.1.2.1. Momentul de activare

Fiecare declanșator are un moment de activare și acesta poate fi până (*BEFORE*), după (*AFTER*) apariția evenimentului, sau poate fi indicat că este executat în locul (*INSTEAD OF*) evenimentului (bineînțeles, în cazul satisfacerii condiției). În fiecare caz, valorile relației înainte de actualizare și după sunt cunoscute declanșatorului.

Trebuie precizat că numai în declanșatoarele de tipul *AFTER* (și *INSTEAD OF*) se permite includerea, în partea acțiune a declanșatorului, a instrucțiunilor *INSERT*, *DELETE* sau *UPDATE*.

#### 6.1.2.2. Granularitatea

Executarea unei instrucțiuni DML (o inserție, eliminare sau modificare) se poate referi la unu sau mai multe tupluri. Granularitatea unui declanșator specifică dacă declanșatorul se activează pentru fiecare tuplu (*FOR EACH ROW*) sau numai o dată la nivel de instrucțiune DML (*FOR EACH STATEMENT*). Granularitatea implicită este *FOR EACH STATEMENT*.

### 6.1.2.3. Nume de corelație

Precum s-a menționat, atât în declanșatoarele de tip *BEFORE*, cât și în cele de tip *AFTER* valorile tuplurilor sunt cunoscute înainte și după executarea instrucției care activează declanșatorul. Numele de corelare servesc pentru a da nume mai semnificative acestor valori.

Dacă granularitatea declanșatorului este la nivel de tuplu, valoarea tuplului care a fost modificată e cunoscută sub numele “*OLD ROW*”, iar valoarea după modificare “*NEW ROW*”. Aceste valori nu se cunosc dacă granularitatea este la nivel de instrucție, deoarece, la nivel de instrucție, are loc numai o execuție a declanșatorului, iar aceasta poate afecta unu sau mai multe tupluri.

Dacă granularitatea declanșatorului este la nivel de instrucție, sunt cunoscute stările relației înainte de modificare și după modificare. Aceste valori sunt “*OLD TABLE*” și “*NEW TABLE*”, respectiv. Deși, pentru acest caz, nu se cunoaște conținutul specific al unui tuplu concret.

Dacă, într-un declanșator cu granularitatea de tuplu, se utilizează nume de corelație, ca, de exemplu,

*REFERENCING OLD ROW AS <tuplu vechi>*  
                  *NEW ROW AS <tuplu nou>,*

atunci valorile vechi și noi ale tuplurilor pot fi indicate cu numele respective. Pentru a utiliza un nume de corelație pentru tupluri, cuvântul *ROW* este optional. Pentru relații, cuvântul *TABLE* este obligatoriu.

Trebuie menționat că granularitatea declanșatorului, specificată în clauza *FOR EACH...* este indicată întotdeauna înainte de numele de corelație și că, dacă se cere accesarea la valorile vechi sau noi, utilizarea numelor de corelație este obligatorie. În sfârșit, trebuie să se indice anumite cazuri ce pot exista pentru valorile cunoscute pe parcursul executării declanșatorului:

- Valoarea *NEW ROW* nu există în declanșatoarele de suprimare, deoarece tuplul își va pierde existența.
- Valoarea *OLD ROW* nu există în declanșatoarele de inserare, din motivul că înainte de executare nu există tuplul.

- Valorile *OLD ROW* și *NEW ROW* sunt aplicabile numai în declanșatoarele cu granularitatea la nivel de tuplu, deoarece dacă granularitatea ar fi la nivel de instrucție, declanșatorul se va executa o dată pentru instrucție și nu vor fi cunoscute tuplurile afectate de aceasta. Dacă granularitatea este la nivel de instrucție, sunt vizibile numai *OLD TABLE* și *NEW TABLE*.

Astfel, dacă se dorește verificarea unei condiții asupra valorilor tuplurilor care se inserează, modifică sau suprimă, granularitatea trebuie să fie *FOR EACH ROW*.

**Exemplul 6.1.** Fie relația funcționari și instrucția SQL din figura 6.1. În această figură, sunt prezente valorile vechi (înainte de actualizare) și cele noi (după actualizare). Dacă se cere definirea unui declanșator care ar controla modificarea, ambele valori ar fi fost vizibile, deoarece instrucția este un *UPDATE*. Această instrucție de actualizare afectează numai trei tupluri rezultate, de aceea, dacă granularitatea ar fi la nivel de tuplu (*FOR EACH ROW*), în fiecare din execuții, vor putea fi văzute valorile vechi și noi pentru fiecare tuplu. În schimb, dacă granularitatea ar fi la nivel de instrucție (*FOR EACH STATEMENT*), se va putea accede numai la versiunile vechi și nouă ale relației în întregime.

UPDATE funcționari		
SET Salariu = 3100		
WHERE Salariu > 1540		
⇒		Relația nouă
Cod	Şef	Salariu
0001	1000	1500
0002	1000	1600
0010	1300	1550
1000	1100	3000
Cod	Şef	Salariu
0001	1000	1500
0002	1000	3100
0010	1300	3100
1000	1100	3100

Figura 6.1. Valori vechi și noi într-o actualizare

### 6.1.3. Condiții

Condițiile, care trebuie verificate pentru ca declanșatorul să fie activat, se specifică după eveniment, folosind clauza

*WHEN (<condiție>).*

Această clauză admite orice expresie condițională validă în SQL, adică, orice tip de condiții care pot fi incluse în clauza *WHERE* a instrucțiunii *SELECT*. În afară de aceasta, în declanșatoarele cu granularitatea *FOR EACH ROW* se poate face referință la valorile noi sau vechi ale atributelor tuplului considerat.

#### 6.1.4. Acțiuni

Acțiunea descrisă în definiția unui declanșator, care este executată atunci când se produce evenimentul specificat și condiția este verificată la adevărat, poate fi o singură instrucție (de obicei, o instrucție SQL) sau un bloc delimitat de

```
BEGIN ATOMIC  
    <instrucționi ce sfârșesc cu ";">  
END ATOMIC
```

Instrucțiunile obișnuite sunt cele de manipulare a datelor (*INSERT*, *DELETE* sau *UPDATE*), care pot fi utilizate numai în declanșatoarele de tipul *AFTER*, sau comanda *SET* care este folosită, de obicei, în declanșatoarele de tipul *BEFORE* pentru atribuirea de valori atributelor, de exemplu,

```
SET tuplu_nou.Atribut=tuplu_vechi.Atribut;
```

Referitor la declanșatoare există unele observații importante:

- Declanșatorul nu evită să se activeze dacă evenimentul s-a declanșat. Așadar, pentru un declanșator de tipul *BEFORE UPDATE*, declanșatorul este activat, apoi este executată comanda *UPDATE*. În declanșatorul de tipul *AFTER*, în primul rând, se execută comanda, apoi declanșatorul.
- Instrucția de definire a declanșatorului creează declanșatorul, dar nici într-un caz nu-l execută. În afară de aceasta, declanșatoarele nu pot fi executate direct, dar sunt activate întotdeauna, dacă are loc un eveniment și condiția ia valoarea adevărat.

### 6.1.5. Gestionarea constrângerilor de integritate

Aplicațiile clasice ale regulilor active sunt cele interne, în baza de date. Sistemul de tratare a regulilor active lucrează asemenea unui subsistem al SGBD-ului, implementând unele funcții ale acestuia. Declanșatoarele sunt generate de sistem și nu sunt vizibile de o parte de utilizatori. Caracteristica tipică a aplicațiilor interne reprezintă posibilitatea de specificare declarativă a funcțiilor, ca fiind cele ce deduc reguli active. Exemple de acest fel sunt menținerea integrității referențiale (*FOREIGN KEY*) și menținerea constrângerilor de integritate comportamentale (*CHECK*). Dar, mai întâi, vor fi examinate unele exemple de creare a declanșatoarelor.

Există două forme de definire a constrângerilor de integritate: declarativă și operațională. Cea declarativă se realizează prin SQL în instrucțiunea de creare a unei relații a bazei de date (*CREATE TABLE*). În acest caz, verificarea constrângerilor este o funcție a SGBD-ului, care analizează operațiile relevante și alege strategia de verificare. Pentru definirea constrângerilor în formă operațională se utilizează reguli active. În acest caz, de verificarea integrității este responsabil proiectantul care definește regulile, analizează operațiile relevante (evenimentele) și decide strategia verificării (condițiile și acțiunile).

Gestiunea constrângerilor de integritate cu utilizarea regulilor active presupune, în primul rând, că constrângerile sunt exprimate în predicate SQL. Predicatul corespunde unei părți a condiției unei sau mai multor reguli active asociate constrângerii. Cu toate acestea, trebuie menționat că predicatul trebuie să apară negat în regulă, în sensul că se consideră că ia valoarea *adevărat*, dacă nu este satisfăcută constrângerea. După aceasta, proiectantul trebuie să se concentreze asupra evenimentelor ce pot provoca o violare a constrângerii. Aceste evenimente sunt incluse în reguli active. și în sfârșit, proiectantul decide ce acțiune trebuie realizată, dacă nu este satisfăcută constrângerea. De exemplu, acțiunea poate fi forțarea unui *ROLLBACK* parțial al sentinelii care a cauzat nesatisfacerea, sau poate fi realizarea unei acțiuni compensatoare care ar corecta nesatisfacerea constrângerii.

Pentru construirea declanșatoarelor este folosită o relație care conține date despre funcționari.

**Exemplul 6.2.** Se creează relația *funcționari* definită de următoarea instrucțiune SQL:

```
CREATE TABLE funcționari(  
    Cod CHAR(6) NOT NULL,  
    Sef CHAR(6),  
    Salariu NUMERIC(7,2),  
    Vârstă TINYINT(3),  
    PRIMARY KEY (Cod),  
    FOREIGN KEY (Sef) REFERENCES șefi(Cod));
```

#### 6.1.5.1. Gestionarea constrângerilor de domeniu

Evident că domeniul atributului *Vârstă* poate fi constrâns în ceea ce privește mulțimea de valori pe care le poate lua acest atribut. Astfel, asupra relației din exemplul 6.2, poate fi considerată următoarea constrângere: “Vârstă este un număr pozitiv mai mic de 150”.

Cu toate că această constrângere poate fi testată cu ajutorul clauzei *CHECK*, aplicând, de exemplu, instrucția *ALTER*

```
ALTER TABLE funcționari  
ADD CONSTRAINT vârstă_validă CHECK((Vârstă>0)  
                                AND (Vârstă<150));
```

este examinată, în continuare, posibilitatea de implementare a ei, folosind un declanșator care realizează aceeași funcție.

În primul rând, trebuie determinate evenimentele care pot cauza actualizarea atributului *Vârstă*. Acestea sunt:

- Adăugarea unui nou funcționar.
- Modificarea vîrstei unui funcționar existent.

Deoarece un declanșator în SQL3 poate controla numai un eveniment, pentru verificarea acestei constrângerii, sunt necesare două declanșatoare. Crearea ambelor declanșatoare este prezentată în continuare, unde *vârstă\_validă\_ins* este un declanșator de tipul *AFTER* care controlează regula de inserare, iar *vârstă\_validă\_upd* – un declanșator de tipul *BEFORE* care verifică regula de modificare.

**Exemplul 6.3.** Următoarea instrucțiune SQL creează declanșatorul *vârstă\_validă\_ins* de tipul *AFTER*. El controlează satisfacerea constrângerii de integritate definită asupra atributului *Vârstă* în cazul inserării, în relația *funcționari*, a unui tuplu cu datele despre un funcționar nou:

```
CREATE TRIGGER vârstă_validă_ins
AFTER INSERT ON funcționari
REFERENCING NEW ROW AS tuplu_nou
FOR EACH ROW
WHEN ((tuplu_nou.Vârstă<=0) OR (tuplu_nou.Vârstă>150))
DELETE FROM funcționari
WHERE Cod=tuplu_nou.Cod;
```

Pot fi menționate unele aspecte importante ale declanșatorului din exemplul 6.3:

- Declanșatorul controlează numai un eveniment, inserarea, și, pentru a putea face referință la noile valori introduse, este de tipul *AFTER*. De aceea, acțiunea declanșatorului va suprima tuplul recent inserat, aplicând instrucțiunea *DELETE*.
- Pentru a putea accede la valorile tuplului nou, granularitatea declanșatorului este fixată la nivel de tuplu, *FOR EACH ROW*. În afară de aceasta, pentru accesarea la aceste valori este obligatorie utilizarea numelui de corelație.
- Condiția stabilită în clauza *WHEN* este una de invaliditate a atributului *Vârstă* (vârstă mai mică sau egală cu 0, sau mai mare sau egală cu 150), deoarece declanșatorul trebuie să activeze în cazul în care valoarea atributului *Vârstă* este invalidă. Pentru a accede la valorile care au fost inserate, în această clauză, se utilizează numele de corelație *tuplu\_nou*.
- Acțiunea este reprezentată de o singură instrucțiune SQL care șterge tuplul recent inserat. Aici, de asemenea, este utilizat numele de corelație pentru a suprima tuplul ce corespunde codului nou (care este cheia relației).

**Exemplul 6.4.** Următoarea instrucțiune creează declanșatorul *vârstă\_validă\_upd* de tipul *BEFORE* care verifică validitatea

atributului *Vârsta* în cazul modificării datelor despre un funcționar în relația *funcționari*.

```
CREATE TRIGGER vârsta_validă_upd
  BEFORE UPDATE OF Vârsta ON funcționari
  REFERENCING NEW AS tuplu_nou
                                OLD AS tuplu_vechi
  FOR EACH ROW
  WHEN ((tuplu_nou.Vârsta<=0)
        OR (tuplu_nou.Vârsta>150))
  SET tuplu_nou.Vârsta=tuplu_vechi.Vârsta;
```

Despre declanșatorul *vârsta\_validă\_upd* se pot menționa următoarele lucruri:

- Din nou este controlat numai un eveniment, în acest caz modificarea. Deoarece se verifică doar modificarea vârstei, în mod explicit, se indică: UPDATE OF Vârsta ON funcționari. În afara de acesta, deoarece acțiunea trebuie să modifice valoarea care se va utiliza pentru modificarea tuplului, declanșatorul este de tip *BEFORE* (se cere ca execuția declanșatorului să nu evite execuția evenimentului de declanșare).
- Pentru a putea accede la valorile, în acest caz, veche și nouă, ale tuplului care a fost modificat, granularitatea este din nou definită la nivel de tuplu (*FOR EACH ROW*). Se utilizează, de asemenea, numele de corelație și, în acest caz, în declarația acestor nume este omis cuvântul-cheie optional *ROW*.
- Condiția clauzei *WHEN* este una de invaliditate, care face ca declanșatorul să acționeze când *Vârsta* ia o valoare invalidă. Aici, iarăși, se utilizează numele de corelații.
- În acțiunea declanșatorului, atributului *Vârsta* i se atrbuie valoarea veche (care este validă). Astfel, la modificarea relației, instrucția *UPDATE* utilizează valoarea veche, prin urmare, realmente, relația nu este modificată, ci se conservă valoarea validă pentru *Vârsta*.

### 6.1.5.2. Gestionarea constrângerilor de tuplu

În continuare, se va examina construirea declanșatoarelor pentru gestiunea constrângerilor de comportament al tuplului, adică a constrângerilor care leagă mai multe attribute ale schemei relaționale. O astfel de constrângere poate fi, de exemplu, “Un funcționar nu poate câștiga un salariu mai mare decât șeful lui”.

Ca și în cazul anterior, se vor crea două declanșatoare, unul pentru verificarea inserării unui nou funcționar, iar altul pentru verificarea posibilelor modificări.

În primul caz, se cere verificarea inserării unui nou subordonat, testând, totodată, dacă el nu câștigă mai mult decât șeful său (dacă, evident, are șef). Nu trebuie verificată inserarea unui șef nou, deoarece (datorită integrității referențiale) nu poate fi cazul de inserare a unui funcționar care ar fi un șef nou.

**Exemplul 6.5.** Declanșatorul *funcționar\_valid\_ins* de tip *AFTER* controlează satisfacerea constrângerii de integritate definită asupra atributelor *Salariu* și *Sef* în cazul inserării unui funcționar nou în relația *funcționari*.

```
CREATE TRIGGER funcționar_valid_ins
AFTER INSERT ON funcționari
REFERENCING NEW ROW AS tuplu_nou
FOR EACH ROW
WHEN (EXISTS(SELECT Cod FROM funcționari
WHERE Cod=tuplu_nou.Sef)
AND
(tuplu_nou.Salariu>(SELECT Salariu
FROM funcționari
WHERE Cod=tuplu_nou.Sef)))
DELETE FROM funcționari
WHERE Cod=tuplu_nou.Cod;
```

În partea *WHEN*, se verifică dacă funcționarul inserat are un șef și dacă acest șef câștigă mai puțin decât noul funcționar. Dacă este așa, datele despre noul funcționar sunt suprimate. Trebuie menționat că clauza *WHEN* este evaluată cu scurtcircuit (dacă funcționarul nu are un șef,

condiția este falsă și nu mai este verificat salariul). Dacă aceasta nu are loc, condiția întreagă nu va fi validă, de aceea, instrucțiunea *SELECT* care obține salariul șefului trebuie să obțină exact un salar pentru a fi validă.

O condiție alternativă, care este întotdeauna validă și nu depinde de modul de evaluare, poate fi

```
... WHEN (I=(SELECT COUNT(*) FROM funcționari
          WHERE Cod=tuplu_nou.Şef
            AND Salariu< tuplu_nou.Salariu))
```

Adică, se verifică dacă noul funcționar are exact un șef (nu poate avea mai mulți) și câștigă mai puțin decât acesta.

Cazul de modificare este mai complex, deoarece există mai multe situații care nu satisfac constrângerea specificată:

- Dacă sunt modificate datele unui funcționar prin amendarea salariului lui, el nu trebuie să câștige mai mult decât șeful său. În afară de aceasta, dacă funcționarul își schimbă șeful, trebuie verificată aceeași condiție.
- Dacă se modifică datele unui șef (se micșorează salariul), atunci trebuie verificat dacă funcționarii subalterni nu câștigă mai mult ca el. Nu este necesară verificarea schimbării codului unui șef, deoarece codul este controlat de integritatea referențială.

**Exemplul 6.6.** Se construiește declanșatorul *funcționar\_valid\_upd* de tipul *BEFORE* care verifică satisfacerea constrângerii de integritate definită asupra atributelor *Salariu* și *Sef* în cazul modificării unui funcționar (fie șef sau nu) în relația *funcționari*.

```
CREATE TRIGGER funcționar_valid_upd
  BEFORE UPDATE OF Salariu, Şef ON funcționari
    REFERENCING NEW AS tuplu_nou
                           OLD AS tuplu_vechi
  FOR EACH ROW
    WHEN ((EXISTS(SELECT Cod FROM funcționari
                  WHERE Şef=tuplu_nou.Cod
                    AND Salariu > tuplu_nou.Salariu))
          OR
```

```

 $(I=(SELECT COUNT(*) FROM \text{funcționari}$ 
 $\text{WHERE Cod}=\text{tuplu\_nou.Şef}$ 
 $\text{AND Salariu}<\text{tuplu\_nou.Salariu}))$ 
BEGIN ATOMIC
  SET tuplu_nou.Salariu=tuplu_vechi.Salariu;
  SET tuplu_nou.Şef=tuplu_vechi.Şef;
END;

```

Se poate vedea că declanșatorul este activat, dacă una din situațiile de invalidare indicate are loc. Pentru a restabili situația anterioară, evitând modificarea atât a șefului, cât și a salariului funcționarului, în acțiune se utilizează instrucțiuni *SET*.

## 6.2. Limbajul SQL3 și modelul obiect-relațional

Limbajul SQL3 este caracterizat drept "limbaj orientat pe obiecte", înzestrat cu posibilitatea utilizării tipurilor noi de date abstrakte și a declanșatoarelor. El este implementat în majoritatea SGBD-urilor, inclusiv *Oracle*, *Universal Server* de Informix, "*Universal Database*" de IBM și altele.

Sistemele obiect-relaționale, care se bazează pe ideea standardului SQL3, susțin un sistem de tipuri mai bogate (prin includerea caracteristicilor orientate pe obiecte) și construcții suplimentare pentru manipularea tipurilor noi de date. Aceste tipuri noi, încearcă să păstreze fundamentele relaționale, dar în același timp extind puterea de modelare a datelor. Cu toate acestea, astăzi, nu se poate afirma că există un model obiect-relațional acceptat în calitate de standard. SGBD-urile obiect-relaționale suferă de aceleași probleme ca și SGBD-urile orientate pe obiecte, adică, sunt multe deosebiri de acest ordin între software-uri disponibile pe piață.

### 6.2.1. Tipuri de date compuse

Standardul SQL3 oferă două tipuri de date compuse *ARRAY* și *ROW*. Aceste tipuri de date acceptă și păstrează mai multe valori.

### 6.2.1.1. Tipul de date ARRAY

Un atribut al unei relații poate fi definit ca un tablou, adăugând la tipul de date al atributului cuvântul cheie *ARRAY*. La rândul său, acest cuvânt-cheie este urmat de numărul maxim de elemente pe care le poate conține, înscris între paranteze pătrate. Accesul la prima poziție dintr-un tablou este realizat cu o valoare de index de unu.

Astfel, tipul *ARRAY* reprezintă o structură de date compusă dintr-un număr finit de elemente de același tip. Datele într-un *ARRAY* sunt ordonate și pot conține valori duplicate. Această structură de date poate caracteriza un atribut, adică mulțimea de valori respectivă poate fi atribuită unui atribut.

**Exemplul 6.7.** Expresia ce urmează definește un atribut *Săptămâna* cu 7 poziții, unde fiecare poziție poate păstra o secvență de caractere de dimensiunea 10:

*Săptămâna VARCHAR(10) ARRAY[7]*

### 6.2.1.2. Tipul de date ROW

În acceptiunea obișnuită folosită în contextul modelului relațional, un tuplu este o colecție nevidă de valori, în care tipul fiecărei valori corespunde unei definiții de atribut dintr-o relație. O relație convențională este formată din tupluri cu proprietatea că fiecare valoare de atribut, în fiecare tuplu, trebuie să fie atomică, aceasta fiind definiția formei normale unu.

Standardul SQL3 lărgeste cerințele formei normale unu, prin introducerea tipului *ROW*, care admite ca un tuplu să conțină o linie, aceasta fiind introdusă ca o valoare de atribut.

Astfel, un tip *ROW* este format dintr-o secvență de atrbute și permite păstrarea valorilor structurate ca o valoare a unui singur atribut. Două atrbute de tip *ROW* sunt considerate echivalente, dacă ambele au același număr de atrbute și fiecare pereche de atrbute de pe aceleasi poziții sunt de același tip.

**Exemplul 6.8.** Atributul *Nume* este de tip *ROW* și este constituit din două componente *NumeFamilie* și *Prenume*:

```

CREATE TABLE persoane (
    CPF CHAR(13),
    Nume ROW (NumeFamilie VARCHAR (30),
    Prenume VARCHAR (30)),
    Vârstă TINYINT,
    CONSTRAINT pk_persoane PRIMARY KEY (CPF));

SELECT P.Nume, Vârstă FROM persoane AS P;

```

### 6.2.2. Tipuri colecție

Valorile neutomice din schemele relaționale pot fi reprezentate și prin tipul colecție. În general, o colecție poate fi o structură de date din categoria mulțimilor, multmulțimilor sau listelor. Tipurile *colecție* (*collection types*) definesc structuri de date care permit manipularea mulțimilor de elemente de același tip. Formal, dacă  $T$  este un tip de date, atunci  $SET(T)$ ,  $MULTISET(T)$  și  $LIST(T)$ , de asemenea, sunt tipuri de date, unde:

- $SET$  – elemente care nu posedă ordine și nu permit duplicate.
- $MULTISET$  – elementele nu posedă ordine și sunt permise valori duplicate
- $LIST$  – elementele posedă ordine și sunt permise valori duplicate.

Folosind aceste tipuri, atributele relațiilor pot conține, în plus, față de valori individuale, liste, mulțimi sau multmulțimi.

**Exemplul 6.9.** În relația administratori, atributul *Rapoarte* reprezintă o mulțime de denumiri de rapoarte, iar atributul *Proiecte* – o listă de secvențe constituită din denumirea proiectului și o mulțime de membri ai proiectului:

```

CREATE TABLE administratori (
    Nume VARCHAR (30),
    Departament VARCHAR (12),
    Rapoarte SET (VARCHAR(30) NOT NULL),
    Proiecte LIST (ROW (Denumire_Pr VARCHAR (15),
    Membri_Proiect SET
    (VARCHAR (20) NOT NULL)) NOT NULL));

```

Utilizatorul trebuie să fie capabil să selecteze și să actualizeze orice element sau submulțime de elemente ale unui atribut de tip colecție. Construirea “relațiilor derivate” cu ajutorul limbajului SQL este o bază bună pentru susținerea acestor tipuri de structuri de date.

### 6.2.3. Tipuri de date definite de utilizator

Cele două forme principale de tipuri definite de utilizatori care sunt, de obicei, furnizate de SGBD-urile obiect-relaționale cuprind *DISTINCT TYPES* și *tipurile structurate*.

#### 6.2.3.1. DISTINCT TYPES

Folosesc pentru asocierea unei înțelegeri speciale cu un tip existent. De fapt, definesc o formă nouă a unui tip. Astfel, *DISTINCT TYPES* sunt utilizate pentru crearea unui tip nou, bazat pe alt tip deja susținut de sistem (built-in) sau care definește structura lui internă. Această facilitate permite utilizatorilor să declare că două tipuri echivalente sunt tratate ca tipuri de date distințe. Mai departe, tipurile pot fi utilizate simplu prin chemarea unui tip existent, apelând la alt nume. *DISTINCT TYPES*, de asemenea, poate fi utilizat pentru definirea tipurilor noi, care au un comportament sau o mulțime de operații valide, diferite de cele din tipul său de bază.

**Exemplul 6.10.** Instrucțiunea ce urmează, definește un *DISTINCT TYPE* *t\_IdFuncionar* bazat pe tipul *CHAR(10)*:

```
CREATE DISTINCT TYPE t_IdFuncionar AS CHAR(10);
```

#### 6.2.3.2. Tipuri de date structurate definite de utilizator

*Tipurile de date structurate* definesc structuri din mai multe atrbute de orice tip. O instanță a unui tip de date structurat definit de utilizator posedă un singur și imutabil identificator, numit *OID (Object Identifier)*. Se spune că aceste instanțe reprezintă obiecte.

Un tip de date structurate definit de utilizator posedă un nume, atrbute și metode. Atributele modelează structura și starea obiectelor tipului. Metodele sunt funcții sau proceduri care se aplică asupra fiecărei instanțe a tipului, făcând calcule bazate pe valorile atrbutoarelor acestuia. Metodele

corespond componentelor obiectelor și implementează operațiile asociate tipului. Metodele pot fi scrise într-un limbaj de manipulare a datelor (de exemplu, SQL) sau într-un limbaj de programare (de exemplu, C sau Java).

**Exemplul 6.11.** Instrucțiunea, ce urmează, definește un tip *tds\_adresa*, format din atributele *Stradă*, *Oraș* și *Tară*:

```
CREATE TYPE tds_adresa AS (
    Strada VARCHAR (30),
    Oras VARCHAR (30),
    Tara VARCHAR (20));
```

Odată definit, tipul de date structurate, poate fi folosit în calitate de domeniu pentru atrbute de alt tip.

**Exemplul 6.12.** Instrucțiunea ce urmează definește tipul *tds\_persoana* cu un atrbut *Adresa* de tipul *tds\_adresa* și metoda *varsta* care calculează vârsta persoanei reprezentate de acest tip:

```
CREATE TYPE tds_persoana AS (
    Nume VARCHAR (30),
    Adresa tds_adresa,
    Telefon VARCHAR (15),
    DfNast DATE),
    INSTANCE METHOD vârsta() RETURN TINYINT;
```

Tipurile de date structurate pot fi utilizate pentru definirea domeniilor atrbutelor unei relații. În acest caz, se spune că instanțele acestor tipuri reprezintă "obiectele atrbutului".

**Exemplul 6.13.** Se poate defini relația *contracte* pentru păstrarea datelor tuturor persoanelor ce au făcut contract cu o campanie. Se utilizează tipul de date *tds\_persoana* în calitate de domeniu al atrbutului *Persoana*:

```
CREATE TABLE contacte (
    Persoana tds_persoana,
    Data DATE);
```

Mai departe, tipurile de date structurate se vor utiliza pentru definirea domeniilor atrbutelor schemelor relaționale sau în alte tipuri. De

asemenea, ele se pot utiliza pentru definirea structurii unei relații, fapt despre care se va discuta în secțiunea următoare.

#### 6.2.4. Relații cu tupluri-obiecte

O relație cu tupluri-obiecte este un tip special de relație ale cărei tupluri sunt obiecte. Acestea, de fapt, sunt niște relații-tipizate, care posedă o structură definită de un tip de date structurat. În acest mod, tuplurile relațiilor devin instanțe sau obiecte de tipul definit în structura sa. Aceste obiecte se vor numi “tupluri- obiecte”.

**Exemplul 6.14.** Fie tipul de date structurat *tds\_funcționar* definit de instrucțiunea:

```
CREATE TYPE tds_funcționar AS (
    Nume VARCHAR(30),
    Adresa tds_adresa,
    Telefon VARCHAR(15),
    CodPersonal CHAR(10),
    Salariu NUMERIC(7,2));
```

O relație care conține obiecte de tipul *tds\_funcționar* poate fi definită, utilizând instrucțiunea:

```
CREATE TABLE obiect_funcționar OF tds_funcționar;
```

Precum poate fi observat, identificarea obiectelor, în modelul obiect-relațional, depinde de tipul de relație de care aparține. În cazul unei relații de obiecte, în care obiectele sunt deriveate dintr-un tip structurat, fiecare obiect posedă un identificator de obiect (*OID*), și sunt identificate în baza existenței sale, în conformitate cu definiția modelului orientat pe obiecte.

În acest fel, o relație de obiecte reprezintă o mulțime de obiecte. Dar, în cazul relațiilor tradiționale, tuplurile relației nu posedă *OID-uri* și, astfel, existența lor se stabilește numai în baza valorilor atributelor acestora, conform cerințelor modelului relațional. Un identificator de obiect este unic numai în contextul unei relații tipizate specifice. Deci două relații tipizate pot avea tupluri cu valori ale *OID* identice. În acest caz, o relație reprezintă o multimediuțime de tupluri, adică poate poseda mai mult de un tuplu cu valorile tuturor atributelor identice. Prin urmare, aceste două

mecanisme de identificare a obiectelor, coexistă într-o schemă obiect-relațională.

### 6.2.5. Moștenirea

La fel ca și limbajele de programare orientate pe obiecte, modelul obiect-relațional susține un concept de moștenire de tipuri de date structurate definite de utilizator, care permite crearea subtipurilor de unu sau mai multe tipuri existente. Astfel, un subtip moștenește specificațiile atributelor și metodele tuturor supertipurilor asociate.

Moștenirea, specificată de clauza *UNDER*, grupează tipurile într-o ierarhie, permitând modularitatea și reutilizarea în definirea tipurilor.

**Exemplul 6.15.** Fie tipul de date structurat definit de utilizator, *tds\_persoana*, este construit din attributele *Id*, *Nume*, *Adresa*, *Telefon*:

```
CREATE TYPE tds_persoana AS (
    Id CHAR(10),
    Nume VARCHAR(30),
    Adresa tds_adresa,
    Telefon VARCHAR(13));
```

Fie că este nevoie de date suplimentare despre persoanele care sunt studenți și profesori. Întrucât studenții și profesorii, de asemenea, sunt persoane, pentru a defini tipurile *tds\_student* și *tds\_profesor* se poate utiliza moștenirea precum urmează:

```
CREATE TYPE tds_student AS (
    CodPersonal CHAR(10),
    Facultate VARCHAR(30))
UNDER tds_persoana;
```

```
CREATE TYPE tds_profesor AS (
    Salariu NUMERIC(7,2),
    Facultate VARCHAR(30))
UNDER tds_persoana;
```

În afară de moștenirea simplă, exemplificată mai sus, standardul SQL3 susține și moștenirea multiplă. Moștenirea multiplă permite ca un tip de date să fie derivat din mai multe tipuri de date.

**Exemplul 6.16.** Fie că trebuie păstrate datele despre profesorii asistenți care, în afară de faptul că sunt persoane, sunt simultan și studenți, și profesori (posibil la facultăți diferite). Atunci, se poate defini tipul structurat *tds\_profesor\_asistent* care moștenește atributele tipurilor *tds\_student* și *tds\_profesor* și, respectiv, a tipului *tds\_persoana*:

```
CREATE TYPE tds_profesor_asistent  
UNDER tds_student, tds_profesor;
```

Un tip moștenește toate atributele superclaselor sale. Un conflict poate apărea atunci când atribut cu același nume sunt moștenite de mai multe supertipuri. Pentru a evita acest conflict, aceste atribut se pot redenumi, utilizând clauza *AS*.

**Exemplul 6.17.** Atributul *Facultate* din definiția de mai sus este redenumit cu ajutorul clauzei *AS*:

```
CREATE TYPE tds_profesor_asistent UNDER  
tds_student WITH (Facultate AS Facult_student),  
tds_profesor WITH (Facultate AS Facult_profesor);
```

Se poate observa că nu există o structură de păstrare asociată tipurilor care aparțin ierarhiei descrise. Dar, pentru manipularea instanțelor de tipuri pot fi create relații de obiecte, care formează o ierarhie de relații, precum ilustrează exemplul următor.

**Exemplul 6.18.** Instrucțiunile ce urmează creează o ierarhie de relații:

```
CREATE TABLE tobiect_persoana OF tds_persoana;  
CREATE TABLE tobiect_student OF tds_student  
UNDER tobiect_persoana;  
CREATE TABLE tobiect_profesor OF tds_profesor  
UNDER tobiect_persoana;  
CREATE TABLE tobiect_profesor_asistent  
OF tds_profesor_asistent  
UNDER tobiect_student, tobiect_profesor;
```

### 6.2.6. Referire la tip ca la domeniul unui atribut

Modelul obiect-relațional permite ca domeniul unui atribut să fie referit de alt obiect de un tip specificat. Astfel, valoarea componentei unui tuplu, într-o relație, poate fi un obiect, iar obiectele pot referi alte obiecte.

În modelul relațional, cheile externe exprimă asocieri *M:1*. SGBD-urile obiect-relaționale pot exprima astfel de asocieri, utilizând OID-ul legat cu instanță a unui tip de date structurat. OID-ul permite obiectului respectiv să fie referit de alte obiecte. Aceasta se face prin tipul de date *REF*, care încapsulează o referință la un obiect specificat.

**Exemplul 6.19.** Fie tipul de date structurat *tds\_depart* și relația de obiecte *tobiect\_departament* definite, respectiv, de instrucțiunile:

```
CREATE TYPE tds_depart AS (
    NumDepart CHAR(8),
    Denumire VARCHAR(30),
    Adresa tds_adresa);
```

```
CREATE TABLE tobiect_departament OF tds_depart;
și relația functionari definită de instrucțiunea:
```

```
CREATE TABLE functionari (
    Funct_ID CHAR(10),
    Nume VARCHAR(30),
    Adresa tds_adresa,
    Telefon VARCHAR(13),
    Salariu NUMERIC(7,2),
    Depart REF tds_depart);
```

În acest caz, atributul *Depart* al schemei relației *funcționari* conține o referință (un identificator unic) la obiectele de tipul *tds\_depart*, care, în exemplul 6.19, reprezintă obiectele relației *tobiect\_departament*. Valoarea fiecărei referințe este generată de SGBD. Acest exemplu reprezintă o asociere *M:1*.

**Exemplul 6.20.** Pentru a selecta numele funcționarilor cu un salarid mai mare de 1000 și departamentele în care aceștia muncesc, se definește:

```
SELECT Nume, F.Depart -> Denumire  
FROM funcționari AS F  
WHERE Salariu > 1000;
```

În general, dacă o relație are un atribut de tipul *REF*, atunci fiecare valoare a atributului poate referi orice obiect de tipul referit, adică pot fi referite obiecte în diferite relații. Clauza *SCOPE* restrângerea aria referințelor la o singură relație, precum ilustrează următorul exemplu:

**Exemplul 6.21.** Clauza *SCOPE* restrângerea aria de referință la relația de obiect *tobiect\_departament*

```
CREATE TABLE funcționari (  
    Nume VARCHAR (30),  
    Adresa tds_adresa,  
    Telefon VARCHAR (13),  
    Funct_ID CHAR(10),  
    Salariu NUMERIC(7,2),  
    Depart REF tds_depart SCOPE tobiect_departament);
```

O caracteristică importantă ce trebuie menționată ține de utilizarea tipului *REF* în calitate de restricție de integritate referențială. În unele SGBD obiect-relationale ca, de exemplu, Oracle, cu toate că se consideră că o instanță a *REF* (adică, *OID*-ul) este validă când se găsește într-o relație, acest fapt nu este adevărat întotdeauna. E cazul când un obiect referit de un tip *REF* este exclus din relația respectivă de obiecte și, în acest fel, referința la acest obiect devine invalidă, dacă nu este verificată de SGBD. O astfel de verificare poate fi realizată, utilizând declanșatoarele care ar preveni excluderea obiectelor referite de *REF*-uri sau de aplicațiile utilizatorilor.

Altfel spus, în acord cu standardul SQL, o restricție de integritate referențială poate specifica legătura între valorile cheii externe într-una sau mai multe relații. Adică se poate defini mulțimea de atrbute *X* ale schemei relației *r*<sub>1</sub> în calitate de cheie externă a relațiilor *r*<sub>2</sub> și *r*<sub>3</sub>. Spre deosebire, o referință este asociată numai unui obiect, adică, referă numai o relație, precum este reprezentat în figura 6.2.

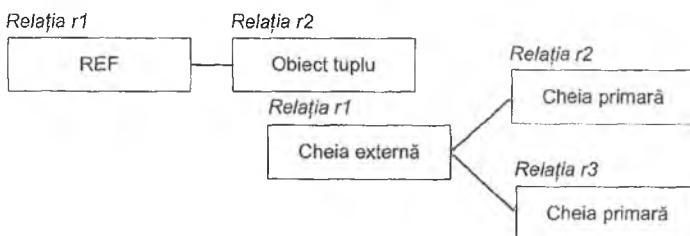


Figura 6.2. Integritatea referențială și referințele

Trebuie menționat că referințele pot fi utilizate numai în relații de obiecte, adică relații bazate pe un tip de date structurat definit de utilizator. Pe de altă parte, restricția de integritate referențială poate fi definită asupra oricărui tip de date. Precum se poate observa, cu toate că referințele pot fi utilizate pentru reprezentarea integrității referențiale, aceste concepte nu reprezintă exact același lucru.

Conceptul de referire se definește la tuplurile unei relații. Există ideea că acesta poate fi implementat, utilizând cheia primară. Adică, orice tuplu al relației poate avea un identificator în calitate de valoare a atributului implicit, iar o referință pentru un tuplu este, pur și simplu, acest identificator. Subrelațiile moștenesc implicit atributul identificator al tuplului.

### 6.2.7. Relații imbricate

Modelul obiect-relațional permite crearea relațiilor cu atrbute ale căror domenii sunt relații. Adică, relațiile pot fi imbricate în alte relații în calitate de valori ale atributelor. Acest concept a fost aplicat în SGBD-ul Oracle. O relație imbricată este o mulțime de elemente de același tip. Ea are un atribut al cărui domeniu este un tip structurat definit de utilizator. Astfel, asocierile  $M:N$  pot fi reprezentate de relațiile imbricate (*Nested Relations*).

**Exemplul 6.22.** Fie că un funcționar al unei companii lucrează în mai multe proiecte și în fiecare proiect participă mai mulți funcționari. Această situație într-o schemă conceptuală entitate-asociere reprezintă o asociere  $M:N$ . Asocierea (*lucru\_funcț*) conține un atribut (*Ore*) care indică numărul de ore lucrare într-un proiect de fiecare angajat. Atunci, tipurile de date structurate

*tds\_functionar* și *tds\_proiect* pentru reprezentarea funcționarilor și proiectelor întreprinderii pot fi definite, respectiv:

```
CREATE TYPE tds_functionar AS (
    Funct_ID CHAR(10),
    Nume VARCHAR(30),
    Adresa tds_adresa,
    Telefon VARCHAR(15),
    Salariu NUMERIC(7,2));
```

```
CREATE TYPE tds_proiect AS (
    NumProiect INTEGER,
    Descriere VARCHAR(30));
```

Relațiile-obiecte pentru manipularea instanțelor de acest tip sunt, corespunzător, definite de instrucțiunile:

```
CREATE TABLE tobiect_functionar OF tds_functionar;
CREATE TABLE tobiect_proiect OF tds_proiect;
```

Asocierea *M:N* este reprezentată de o relație imbricată, definită de un tip de date structurat. Acest tip este definit pentru toate atributele ce fac parte din asociere, adăugându-se un atribut care este o referință la tipul ce definește structura unei din relații care participă în asociere.

**Exemplul 6.23.** Tipul de relație imbricată pentru reprezentarea asocierii *lucru\_funct* este construit din atributul *Ore* și de o referință la obiectele relației *tobiect\_proiect*:

```
CREATE TYPE tds_lucru_funct AS (
    Ref_Proiect REF tds_proiect
        SCOPE tobiect_proiect, Ore INTEGER);
```

Instrucțiunea care urmează definește tipul de relație imbricată ce manipulează obiecte de tipul *tds\_lucru\_funct*.

```
CREATE TYPE t_RellImbr_lucru_funct OF tds_lucru_funct;
```

În sfârșit, tipul de date structurate *tds\_functionar* este redefinit: este adăugat atributul care reprezintă funcționarii ce lucrează în fiecare proiect.

```
CREATE TYPE tds_functionar AS (
    Funct_ID CHAR(10),
```

*Nume VARCHAR (30),  
 Adresa tds\_adresa,  
 Telefon VARCHAR (15),  
 Salariu NUMERIC(7,2),  
 Proiecte t\_RelImbr\_lucru\_funct);*

Relația care manipulează obiecte de tipul *tds\_functionar*, este atunci definită de:

```
CREATE TABLE tobiect_functionar OF tds_functionar
NESTED TABLE Proiecte STORE AS tImbr_Proiecte;
```

Trebuie menționat că, în SGBD-ul Oracle, o relație poate conține mai mult de o relație imbricată, dar nicio relație imbricată nu poate conține o altă relație imbricată.

### 6.3. Obiecte mari

Obiectele mari sunt un tip nou de date care permit stocarea unor volume mari de date – giga octeți. Ele se folosesc pentru păstrarea imaginilor, sunetelor, textelor formatare și altor necesități multimedia ale aplicațiilor actuale.

În SQL3 pentru obiectele mari sunt definite trei tipuri diferite de date:

- Obiecte binare mari, BLOB (*Binary Large Object*), un sir binar.
- Obiecte mari de tip caracter, CLOB (*Character Large Object*).
- Obiecte de talie mare de caractere naționale, NCLOB (*National Character Large Object*).

Obiectele mari sunt păstrate direct în baza de date, nu în fișiere externe. Definițiile fac parte din definiția relațiilor bazei de date. În aceeași schemă relațională pot coexista diverse atrbute de aceste tipuri, iar dimensiunile lor, exprimate în Ko, Mo sau Go, sunt incluse în definiția schemei.

**Exemplul 6.24.** Urmează definiția unei scheme relationale, care conține obiecte mari:

```
CREATE TABLE carte(
    Titlu VARCHAR(200),
    Id_carte INTEGER,
```

*Rezumat CLOB (32K),  
Text\_carte CLOB (20M),  
Pelicula BLOB (2G);*

Obiectele mari din limbajul SQL3 sunt puțin diferite de tipul de obiecte BLOB care a apărut inițial în multe SGBD-uri. În aceste sisteme, obiectul BLOB este un flux de octeți neinterpretat și SGBD-ul nu cunoaște nimic despre conținutul obiectului BLOB sau structura acestuia. Prin urmare, sistemul nu poate face interogări și operații asupra acestor tipuri de date inerent bogate și structurate, cum ar fi imaginile, imaginile video, sunetele, textele, paginile Web.

Tipurile BLOB, CLOB și NCLOB se comportă ca niște secvențe de caractere, dar posedă unele restricții fapt ce împiedică utilizarea lor în calitate de *PRIMARY KEY* sau în clauzele *UNIQUE*, *FOREIGN KEY* și în comparații, cu excepția celor de egalitate și inegalitate. Ca urmare a acestor restricții, un atribut de tip CLOB nu poate fi folosit în clauzele *GROUP BY*, *ORDER BY* sau în operațiile cu mulțimi (*UNION*, *INTERSECT* și *EXCEPT*).

În general, pentru a fi prelucrate, obiectele mari trebuie transferate de pe serverul SGBD la client. Însă standardul SQL3 permite efectuarea unor operații și pe serverul SGBD. Aceste operații sunt reprezentate de operatorii standard de tip secvențe, care se aplică asupra secvențelor de caractere și returnează, de asemenea, secvențe de caractere. Printre acestea, sunt operațiile de concatenare sau subsecvențe (*SUBSTRING*).

**Exemplul 6.25** Relația *personal* se extinde pentru a păstra un rezumat și o fotografie ale fiecărui membru de personal:

*ALTER TABLE personal ADD rezumat CLOB(50K);  
ALTER TABLE personal ADD imagine BLOB(12M);*

## Bibliografie

---

[ANSI78] ANSI/X3/SPARC Study Group on Data Base Management Systems, "Framework Report on Database Management Systems", *Information Systems*, vol.3, nr.3, 1978.

*Acesta este documentul final în care este prezentată arhitectura în trei nivele a bazelor de date propusă de ANSI.*

[Bancilhon81] Bancilhon F., Spyros N., „Update Semantics and Relational Views”, *ACM TODS*, vol.4, nr.6, 1981, p. 557-575.

*Un articol de referință în materie de actualizare a viziunilor. Autorii pun problema în termeni de invariантă a viziunilor în urma actualizărilor directe sau reflectate în baza de date.*

[Bello98] Bello R.G, Dias K., Downing A., Freenan J., Norcott D., Dun H., Witkowski A., Ziauddin M., „Materialized Views in Oracle”, *Int. Conf. on Very Large Databases*, Morgan & Kauffman Ed., New York, USA, 1998, p. 659-664.

*Acest articol explică modul în care sunt gestionate viziunile materializate în sistemul Oracle. Viziunile respective sunt folosite în depozitele de date și în replicare. Ele pot fi reînnoite la sfârșitul tranzacției, la cerere sau periodic. Actualizările pe loturi sunt optimizate. Optimizarea interogărilor ține cont de viziunile materializate, folosind o tehnică de rescriere bazată pe un model de cost.*

[Bernstein80] Bernstein P., Blaustein B., Clarke E..M., „Fast Maintenance of semantic Integrity Assertions Using Redundant Aggregate Data”, *Proc. 6th Int. Conf. on Very Large Data Bases*, Montreal, Canada, Morgan Kaufman Ed., 1991, p. 126–136.

*Autorii studiului au fost primii care au propus menținerea clusterelor redundante pentru a facilita verificarea constrângerilor de integritate în timpul actualizărilor. De atunci, aceste tehnici sunt utilizate pe scară largă sub formă de viziuni concrete.*