

s o f t w a r e

**Florentin
Eugen
Ipate**

**Monica
Popescu**

**Dezvoltarea
aplicațiilor
de baze
de date
în**

**ORACLE 8
și
FORMS 6**

Edition
All

S o f t w a r e

Oracle Server este cel mai utilizat Sistem de Gestiune a Bazelor de Date Relaționale (SGBDR) din întreaga lume. El reprezintă nucleul liniei de produse oferite de Oracle Corporation, care include printre altele și un set de instrumente de dezvoltare a aplicațiilor din generația a patra, numit Oracle Developer. Coloana vertebrală a pachetului de programe Oracle Developer este Oracle Forms, un limbaj din generația a patra, utilizat pentru dezvoltarea interfețelor interactive ale aplicațiilor. Oracle Forms este un instrument foarte puternic, fiind îndeajuns de complet pentru a realiza singur aplicații complexe pentru baze de date.

Această carte prezintă informațiile necesare pentru realizarea de aplicații folosind SGBDR-ul Oracle și Oracle Forms. Versiunile care sunt prezentate sunt cele mai recente, și anume Oracle8 Server și respectiv Oracle Forms6.

Lucrarea se adresează programatorilor, informaticienilor, studenților și în general tuturor celor care doresc să învețe să dezvolte aplicații de baze de date utilizând SGBDR-ul Oracle și/sau Oracle Forms. Cartea poate fi extrem de folositoare pentru studenții de la facultățile de informatică sau de la alte facultăți care urmează cursuri de Oracle sau baze de date și poate fi folosită ca suport pentru astfel de cursuri. În plus, ea poate fi folosită de programatorii și informaticienii care lucrează în Oracle și doresc să-și perfectioneze cunoștințele.

ISBN 973-571-311-X



9 789735 713119

Florentin Eugen Ipate

A absolvit Facultatea de Automatizări și Calculatoare din cadrul Institutului Politehnic București (1991) și a obținut titlurile de Master of Science (1992) și Doctor of Philosophy (1995) la Universitatea din Sheffield, Anglia. În prezent este conferențiar universitar la Universitatea Româno-Americană din București și la Universitatea din Pitești și director al firmei IF Soft, firmă specializată în dezvoltarea de aplicații cu Oracle Server și Oracle Developer. Este autorul a peste 20 de articole în domeniul informaticii, publicate în jurnale internaționale sau prezentate la conferințe și a unei cărți apărută în editura Springer-Verlag.

Monica Popescu

A absolvit Facultatea de Cibernetică, Informatică Economică și Statistică din cadrul Academiei de Studii Economice (1997), și a obținut titlul de Master din cadrul Departamentului de Studii Aprofundate a aceleiași instituții (1998). A fost asistentă universitară, predând Baze de date Oracle. În prezent este consultant la firma Kepler, firmă specializată în dezvoltarea de aplicații software.

Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Florentin Eugen Ipate, Monica Popescu

Copyright © 2000, Editura BIC ALL

Toate drepturile rezervate Editurii BIC ALL

Nici o parte din acest volum nu poate fi copiată fără
permisiunea scrisă a Editurii BIC ALL

Drepturile de distribuție în străinătate aparțin în exclusivitate editurii.

Copyright © 2000 by BIC ALL

All rights reserved.

The distribution of this book outside Romania, without
the written permission of BIC ALL,
is strictly prohibited.

Descrierea CIP a Bibliotecii Naționale

IPATE, FLORENTIN EUGEN

**Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6/ Florentin Eugen Ipate,
Monica Popescu, București: BIC ALL, 2000 – (Software/Hardware)**

440 p.; 24 cm

ISBN 973-571-311-X

I. POPESCU, MONICA

004.43 ORACLE

Editura BIC ALL:

București

Bd. Timișoara nr. 58, sector 6

Tel.: 402 26 00

Fax: 402 26 10

Departamentul distribuție:

Tel.: 402 26 20

Comenzi la:

comenzi@all.ro

URL:

http://www.all.ro

Redactor:

Viorica Georgeta Murariu

Designer copertă:

Daniel Tujunel

. Printed in Romania

**Dezvoltarea
aplicațiilor
de baze
de date
în
ORACLE 8
și
FORMS 6**

**Florentin
Eugen
Ipate**

**Monica
Popescu**

2000



Cuprins

<i>Prefață</i>	xii
<i>Introducere</i>	xv
1. Baze de date: scop, istoric, utilizare	xv
2. Arhitectura unui sistem de baze de date	xv
3. Sistem de Gestiune a Bazelor de Date (SGBD).....	xvii
4. SGBDR Oracle: Concepte Generale.....	xviii
<i>Capitolul 1: Modelul relațional</i>	1
1.1. Concepte de bază ale modelului relațional	2
1.2. Constraineri de integritate	5
1.3. Operatorii sistemului relațional	6
1.3.1. Algebra relațională și limbajul SQL	6
1.4. Tabele, rânduri, coloane	16
1.5. Sisteme de Gestiune a Bazelor de Date Relaționale (SGBDR)	16
1.5.1. Regulile lui Codd.....	16
<i>Capitolul 2: Proiectarea bazelor de date relaționale</i>	21
2.1. Crearea schemei conceptuale.....	21
2.1.1. Modelul entitate-legătură (entitate-relație).....	21
2.2. Crearea design-ului logic al bazei de date	32
2.2.1. Transformarea modelului entitate legătură în modelul relațional	32
2.3. Normalizarea bazei de date	37
2.3.1. Prima formă normală (1.NF – First Normal Form).....	42
2.3.2. A doua formă normală (2.NF– Second Normal Form).....	44
2.3.3. A treia formă normală (3.NF –Third Normal Form).....	46
2.3.4. Forma normală Boyce-Codd (BCNF – Boyce-Codd Normal Form).....	49
2.3.5. A patra formă normală (4.NF – Fourth Normal Form)	50
2.3.6. A cincea formă normală (5.NF – Fifth Normal Form)	53
2.3.7. Denormalizare	56
<i>Capitolul 3: Arhitectura SGBDR Oracle</i>	59
3.1. Crearea unei baze de date.....	60
3.2. Nivelul fizic al bazei de date.....	61
3.3. Instanța unei baze de date.....	65

3.4. Legătura dintre o bază de date și o instanță	65
3.5. Pornirea unei baze de date Oracle	66
3.6. Oprirea unei baze de date Oracle	67
3.7. Crearea manuală a unei baze de date	67
Capitolul 4: Structuri logice de stocare a bazei de date	73
4.1. Spații tabel	73
4.1.1. Crearea spațiilor tabel	74
4.1.2. Modificarea spațiilor tabel	77
4.1.3. Distrugerea spațiilor tabel	78
4.2. Segmente	78
4.2.1. Segmente de date (data segments)	78
4.2.2. Segmente de indecsi (index segments)	79
4.2.3. Segmente de revenire (rollback segments)	79
4.2.4. Segmente temporare (temporary segments)	80
4.3. Extinderi	80
4.4. Blocuri de date	81
4.5. Identificatorul de rând (ROWID)	82
Capitolul 5: Securitatea bazei de date	85
5.1. Privilegii de sistem	86
5.2. Privilegii la nivel de obiect	86
5.3. Roluri	87
5.4. Utilizorii bazei de date	88
5.5. Schema	89
5.6. Crearea, modificarea și distrugerea utilizatorilor	91
5.7. Crearea, modificarea și ștergerea rolurilor	92
5.8. Acordarea rolurilor și privilegiilor	93
5.9. Revocarea rolurilor și privilegiilor	95
5.10. Activarea și dezactivarea rolurilor unui utilizator	97
Capitolul 6: Organizarea logică a bazei de date	98
6.1. Tabele	98
6.1.1. Crearea tabelelor	99
6.1.2. Tabele partiționate	101
6.1.3. Constrângeri	102
6.1.4. Crearea și popularea simultană a tabelelor	107
6.1.5. Modificarea tabelelor	108
6.1.6. Distrugerea tabelelor	110

6.2. Vederi	111
6.2.1. Crearea vederilor	112
6.2.2. Operații DML asupra vederilor	113
6.2.3. Operații DML asupra vederilor bazate pe mai multe tabele (Join-Views)	116
6.2.4. Recompilarea vederilor	119
6.2.5. Distrugerea vederilor	119
6.3. Indecși	120
6.3.1. Crearea, modificarea și distrugerea indecșilor	122
6.3.2. Tipuri de indecși	124
6.3.3. Tabele organizate pe bază de index	130
6.3.4. Crearea tabelelor organizate pe baza de index	131
6.4. Clustere	132
6.4.1. Clusterul de index	132
6.4.2. Crearea unui tabel într-un cluster de index	133
6.4.3. Clusterul hash	135
6.4.4. Crearea unui tabel într-un cluster hash	135
6.4.5. Modificarea clusterelor	137
6.4.6. Distrugerea clusterelor	137
6.5. Secvențe	137
6.5.1. Crearea, modificarea și distrugerea secvențelor	138
6.5.2. Utilizarea secvențelor	139
6.6. Sinonime	141
6.6.1. Crearea, modificarea și distrugerea sinonimelor	141
6.7. Proceduri, funcții și pachete stocate	142
6.8. Declanșatoare ale bazei de date	143
6.9. Instantane	143
6.9.1. Reîmprospătarea instantaneelor	145
6.9.2. Crearea instantaneelor	146
6.9.3. Modificarea instantaneelor	147
6.9.4. Distrugerea instantaneelor	148
6.10. Legăturile bazei de date	148
6.10.1. Crearea legăturilor bazei de date	149
6.10.2. Ștergerea legăturilor bazei de date	150
6.11. Dicționarul de date	150
Capitolul 7: Accesul concurent la date și păstrarea consistenței acestora	154
7.1. Tranzacțiile și asigurarea consistenței la scriere	154
7.1.1. Comanda AUTOCOMMIT	155
7.1.2. Puncte de salvare	156

7.2. Asigurarea consistenței cu ajutorul tranzacțiilor	157
7.2.1. Modelul multiversiune și consistența la citire	157
7.2.2. Implementarea modelului multiversiune	157
7.2.3. Tranzacții de citire și consistența la citire la nivel de tranzacție	158
7.3. Blocări	158
7.3.1. Blocări la nivel de rând	159
7.3.2. Blocările la nivel de tabel	160
7.3.3. Comanda SELECT cu clauza FOR UPDATE	162
7.3.4. Comanda LOCK TABLE	163
7.3.5. Interblocarea	164
Capitolul 8: SQL	166
8.1. Comanda SELECT	166
8.1.1. Atributele comenzi SELECT	166
8.1.2. Operatori aritmetici	167
8.1.3. Aliasuri de coloane	167
8.1.4. Operatorul de concatenare	168
8.1.5. Convertirea valorilor Null cu ajutorul funcției NVL	168
8.1.6. Prevenirea selectării înregistrărilor dupăcat	169
8.1.7. Clauza ORDER BY	170
8.1.8. Clauza WHERE	171
8.1.9. Operatori relaționali	172
8.1.10. Operatori SQL	174
8.1.11. Operatorii logici	176
8.1.12. Funcții	179
8.1.13. Funcții referitoare la o singură înregistrare	179
8.1.14. Funcții referitoare la mai multe înregistrări	184
8.1.15. Pseudo-coloana ROWNUM	185
8.1.16. Clauza GROUP BY	186
8.1.17. Clauza HAVING	187
8.1.18. Regăsirea datelor din două sau mai multe tabele	188
8.1.19. Operatorii pentru mulțimi	193
8.1.20. Subinterrogări și operatorii ANY, ALL, EXISTS	196
8.1.21. Operații pe tabele ce conțin informații de structură arborescentă	203
8.2. Comanda INSERT	204
8.3. Comanda UPDATE	206
8.2. Comanda DELETE	207
8.5. Comanda TRUNCATE	208
Capitolul 9: PL/SQL	210
9.1. Blocuri PL/SQL	210
9.2. Arhitectura PL/SQL	212

9.3. Sintaxa de bază a PL/SQL.....	213
9.4. Referirea obiectelor.....	213
9.5. Variabile și constante	214
9.5.1. Folosirea variabilelor PL/SQLpentru stocarea valorilor înregistrărilor dintr-un tabel	216
9.6. Operatori PL/SQL.....	218
9.7. Comenzile SQL și PL/SQL.....	218
9.8. Funcții predefinite în PL/SQL.....	219
9.9. Cursoare implicate.....	220
9.10. Controlul tranzacțiilor	221
9.11. Structuri de control în PL/SQL	221
9.12. Excepțiile.....	226
9.13. Utilizarea explicită a cursoarelor.....	235
9.14. Proceduri și funcții.....	241
9.14.1. Proceduri și funcții stocate	246
9.14.2. Crearea, recomplirea, distrugerea și utilizarea subprogramelor	247
9.15. Pachete	249
9.15.1. Pachete stocate	250
9.15.2. Crearea, recomplirea, distrugerea și utilizarea pachetelor stocate.....	250
9.15.3. Pachete predefinite	253
9.16. Triggere (declanșatori).....	254
9.16.1. Declanșatorii și constrângerile de integritate.....	255
9.16.2. Sintaxa declanșatorilor și modul de creare	256
9.16.3. Tipuri de declanșatori.....	259
9.16.4. Modificarea și ștergerea unui declanșator.....	265
Capitolul 10: Orientare pe obiect în Oracle8.....	267
10.1. Oracle8. și modelul relațional obiectual	267
10.2. Opțiunea obiect în Oracle 8.....	268
10.2.1. Tipul obiect	269
10.2.2. Constructor	271
10.2.3. Metode.....	271
10.2.4. Metode de comparare.....	273
10.2.5. Tabele obiect	276
10.2.6. Tipul REF (referință)	276
10.2.7. Tipurile colecție	278
10.2.8. Tipul VARRAY	278
10.2.9. Tipul tabel imbricat.....	279
10.2.10. Metode asociate colecțiilor	281
10.2.11. Exemplu.....	282

x	Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6	
10.3.	Tipuri de date LOB.....	284
10.4.	Vederi obiect	284
Capitolul 11: Oracle Forms		287
11.1.	Form Builder.....	288
11.2.	Obiecte și tipul acestora	290
11.2.1.	Forme	290
11.2.2.	Declanșatoare (Triggers)	290
11.2.3.	Fereste de avertizare (Alerts)	293
11.2.4.	Biblioteci atașate (Attached library).....	293
11.2.5.	Blocurile de date (Data Blocks).....	293
11.2.6.	Elemente (Items)	296
11.2.7.	Relații (Relations)	298
11.2.8.	Canvas-uri (Canvases)	298
11.2.9.	Editoare (Editors).....	300
11.2.10.	Liste de valori (LOVs)	300
11.2.11.	Grupuri de obiecte (Object Groups).....	302
11.2.12.	Parametri (Parameters)	302
11.2.13.	Meniuri atașate (Popup Menus)	303
11.2.14.	Unități de program (Program Units)	303
11.2.15.	Clase de proprietăți (Property Classes)	305
11.2.16.	Grupuri de înregistrări (Record Groups).....	305
11.2.17.	Rapoarte (Reports)	306
11.2.18.	Atribut vizual (Visual Attribute).....	306
11.2.19.	Fereste (Windows)	306
11.3.	Utilizarea variabilelor	306
11.4.	Crearea și utilizarea meniurilor	308
11.5.	Crearea și utilizarea bibliotecilor.....	312
11.5.1.	Biblioteci PL/SQL.....	312
11.5.2.	Biblioteci de obiecte.....	313
11.6.	Rularea unei forme	314
Capitolul 12: Aplicație în Forms.....		316
Anexa 1: Produse Oracle8.....		330
Anexa 2: Comenzi frecvent utilizate în administrarea bazelor de date și utilizările cu care se pot executa.....		338
Anexa 3: Tipuri de date PL/SQL		337
Anexa 4A: Descrierea privilegiilor de sistem		352
Anexa 4B: Comenzi Oracle și privilegii necesare		358

<i>Anexa 4C: Roluri predefinite și privilegii conținute de acestea</i>	361
<i>Anexa 5: Vederile dicționarului de date</i>	362
<i>Anexa 6: Funcții SQL și PL/SQL.....</i>	369
<i>Anexa 7: Pachete PL/SQL predefinite utilizate frecvent</i>	379
<i>Index.....</i>	395
<i>Bibliografie</i>	409

Prefață

Datorită imensei lor valori comerciale, bazele de date reprezintă unul dintre domeniile informaticii care au înregistrat o dezvoltare continuă în ultimii douăzeci de ani, în special ca urmarea a apariției modelului relațional. *Oracle Server* este cel mai utilizat Sistem de Gestiușa a Bazelor de Date Relaționale (SGBDR) din întreaga lume. El reprezintă nucleul liniei de produse oferite de Oracle Corporation, care include, printre altele și un set de instrumente de dezvoltare a aplicațiilor din generația a patra, numit *Oracle Developer*. Coloana vertebrală a pachetului de programe *Oracle Developer* este *Oracle Forms*, un limbaj din generația a patra, utilizat pentru dezvoltarea interfețelor interactive ale aplicațiilor. *Oracle Forms* este un instrument foarte puternic, fiind îndeajuns de complet pentru a realiza singur aplicații complexe pentru baze de date.

Această carte prezintă informațiile necesare pentru realizarea de aplicații folosind SGBDR-ul Oracle și Oracle Forms. Versiunile care sunt prezentate sunt cele mai recente, și anume Oracle8 Server și respectiv Oracle Forms 6.

Lucrarea se adresează tuturor celor care doresc să învețe să dezvolte aplicații de baze de date utilizând SGBDR-ul Oracle și/sau Oracle Forms. Spre deosebire de alte cărți existente pe piață, cartea de față nu presupune existența unor cunoștințe anterioare de Oracle și nici măcar de baze de date (în acest sens sunt incluse capitolele dedicate modelului relațional și proiectării bazelor de date relaționale). Cartea poate fi extrem de folosită pentru studenții de la facultățile de informatică sau de la alte facultăți care urmează cursuri de Oracle sau baze de date și poate fi folosită ca suport pentru astfel de cursuri. În sfârșit, deoarece cartea cuprinde ultimele nouătăți aduse de tehnologia Oracle (printre care și modelul relațional obiectual adus de către versiunea Oracle8), cartea poate fi folosită de programatorii și informaticienii care folosesc acest mediu și doresc să-și perfecționeze cunoștințele.

Cartea este structurată după cum urmează:

În introducere sunt prezentate concepțele de bază de date, sistem de gestiune a bazei de date și trăsăturile caracteristice ale acestora. În plus, sunt prezentate trăsăturile definitorii ale sistemului Oracle, limbajele SQL, PL/SQL și principalele unele Oracle.

Capitolul 1 descrie modelul relațional, componentele și caracteristicile sale de bază și algebră relațională, precum și regulile lui Codd de clasificare a unui SGBD ca relațional.

Capitolul 2 este dedicat proiectării bazelor de date relaționale și cuprinde o prezentare detaliată a modelului entitate-legătură și a modului de transformare a acestuia în modelul relațional, precum și tehnica normalizării.

Capitolul 3 prezintă arhitectura SGBDR Oracle, concepțele de bază de date și instanță a unei baze de date Oracle, fișierele unei baze de date Oracle, precum și modalitățile de pornire, oprire și creare a unei baze de date.

Capitolul 4 descrie structurile logice de stocare folosite de Oracle: spații tabel, segmente, extinderi și blocuri de date. În plus, este prezentat formatul identificatorul de rând folosit de o bază de date Oracle.

Capitolul 5 este dedicat securității bazei de date Oracle și conceptelor asociate, precum utilizator, rol, privilegii de sisteme și privilegii la nivel de obiect. Acest capitol cuprinde și instrucțiunile SQL de definire a utilizatorilor și rolurilor precum și cele privind acordarea unor privilegii sau roluri unor utilizatori sau altor roluri.

Capitolul 6 descrie organizarea logică a bazei de date, prezentând obiectele schemei unei baze de date: tabele, vederi, indecsi, secvențe, sinonime, instantane, legături ale bazei de date. Pentru fiecare dintre aceste obiecte sunt prezentate comenzi SQL de definire a datelor corespunzătoare. În plus, capitolul cuprinde și o prezentare a principalelor vederi ale dicționarului bazei de date.

Capitolul 7 prezintă modul în care Oracle permite accesul concurent la date, păstrând în același timp consistența acestora. Este prezentat conceptul de tranzacție și modul în care acestea sunt folosite de Oracle pentru asigurarea consistenței la citire. Sunt de asemenea descrise modurile de blocare folosite de Oracle precum și conceptul de interblocare.

Capitolul 8 descrie pe larg componentele de interogare și manipulare a datelor ale limbajului SQL.

Capitolul 9 prezintă limbajul PL/SQL, extinderea procedurală a limbajului SQL. În acest capitol sunt incluse procedurile, funcțiile, pachetele PL/SQL și declanșatoarele bazei de date, utilizarea și modul de definire a acestora.

Capitolul 10 este dedicat modelului relațional-obiectual, o nouitate adusă de versiunea Oracle8, acesta incluzând tipurile definite de utilizator, tabelele obiect, tipul referință, tipul colecție, precum și vederile obiect.

Capitolul 11 prezintă limbajul Oracle Forms și utilitarul Form Builder.

Capitolul 12 folosește un exemplu pentru a ilustra modul de dezvoltare a unei aplicații în Oracle Forms.

Cărtea cuprinde și următoarele anexe:

Anexa 1: Produse Oracle8

Anexa 2: Comenzi frecvent utilizate în administrarea bazelor de date și utilitarele cu care se pot executa

Anexa 3: Tipuri de date PL/SQL

Anexa 4A: Descrierea privilegiilor de sistem

Anexa 4B: Comenzi Oracle și privilegii necesare

xiv Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Anexa 4C: Roluri predefinite și privilegii conținute de acestea

Anexa 5: Vederile dicționarului de date

Anexa 6: Funcții SQL și PL/SQL

Anexa 7: Pachete PL/SQL predefinite utilizate frecvent

Introducere

1. Baze de date: scop, istoric, utilizare

De-a lungul timpului au existat mai multe definiții ale conceptului de *bază de date*. În general, noțiunea de *date* denumește cunoștințele fixate pe un anumit suport fizic în vederea utilizării și prelucrării într-un anumit scop. Pentru a permite interpretarea și prelucrarea, datele trebuie reprezentate într-un anumit mod. O *bază de date (database)* reprezintă o colecție de date organizate care servește unui anumit scop. Spunem că o bază de date este organizată în sensul că ea conține date care sunt stocate, reprezentate și accesate într-o manieră consistentă. Spunem că o bază de date servește unui anumit scop, în sensul că ea nu conține date care nu sunt relevante pentru acesta. O agenda de telefon este un bun exemplu de bază de date. Ea conține date relevante pentru o anumită persoană (numele, adresa, numărul de telefon). Culoarea telefonului unei persoane este o informație irelevantă și ea nu este conținută în această bază de date. Foarte multe baze de date se axează pe domeniul economic, dar există și baze de date cu scopuri științifice, militare, etc. Pentru a răspunde cerințelor actuale, bazele de date conțin pe lângă date de tip text sau numeric și alte tipuri cum ar fi imaginile, sunetele și elementele multimedia.

Din punct de vedere istoric în informatică, bazele de date au apărut ca răspuns la nevoia, constatăă de programatori, ca datele să fie stocate pentru a putea fi transmise de la o rulare a aplicației la alta. Această cerință a devenit cunoscută sub numele de *persistența datelor*, adică nevoia ca datele rezultate la o rulare a programului să persiste, sau să fie salvate pentru altă rulare. De la această nevoie fundamentală a început evoluția bazelor de date. O a doua cerință care a dus la apariția bazelor de date a fost cea legată de simplitatea stocării și manipulării datelor. Cu alte cuvinte, datele trebuiau stocate în aşa fel încât să poată fi accesate și manipulate cu ușurință.

2. Arhitectura unui sistem de baze de date

Un sistem de baze de date poate fi văzut din patru puncte de vedere numite nivele: *conceptual, extern, logic și fizic*.

Nivelul conceptual

Este nivelul fundamental deoarece descrie într-un mod natural și fără ambiguități ~~materialul și urmează să fi modelat~~. Dacă o persoană este familiară cu un anumit sistem, atunci ~~de judecățea~~ realiză designul acestuia fără a mai comunica cu alte persoane. În caz contrar, ~~materialul~~ consultă mai multe persoane care sunt familiare cu sistemul respectiv. Așa cum ~~materialul~~ comunică între ei cu ajutorul cuvintelor, figurilor sau exemplelor, cel mai clar mod

de a descrie un sistem este utilizarea unui limbaj natural, a unor diagrame intuitive și bineînțeles a exemplelor. Pentru a simplifica procesul de modelare se examinează informația în cele mai mici unități posibile: un fapt la un anumit moment. Aceste principii conduc la ideea de a exprima mai întâi un sistem la nivel conceptual, folosind concepte cu care oamenii pot lucra foarte ușor. Prin urmare, la acest nivel va fi realizată *schema conceptuală* ce reprezintă design-ul general al sistemului.

Nivelul extern

La nivel extern se specifică design-ul bazei de date percepțut de un anumit utilizator sau grup de utilizatori precum și modul în care acest design este legat de schema conceptuală. Schema externă reprezintă specificarea informațiilor care pot fi văzute de către un utilizator și modul în care acestea sunt prezentate. În cele mai multe cazuri, un utilizator poate accesa doar o parte din informații. De exemplu, este mult mai ușor ca un utilizator să selecteze numai informații relevante pentru acesta sau din motive de securitate să nu poată obține anumite informații. Schema externă este realizată astfel încât grupuri diferite de utilizatori să acceseze numai anumite subschemă ale schemei conceptuale globale. Mai mult, utilizatori diferiți pot dori ca aceeași informație să fie reprezentată în moduri diferite (tabele, grafice) sau pot prefera anumite metode de operare sau navigare în funcție de nivelul de experiență acumulat. Prin urmare la acest nivel se pot proiecta diferite interfețe cu utilizatorul.

Nivelul logic

Așa cum am văzut până acum, schema conceptuală este creată pentru a comunica, adică pentru a reprezenta sistemul ce urmează a fi proiectat într-un mod cât mai clar. Pentru a realiza o implementare eficientă această schemă conceptuală trebuie convertită într-o structură de nivel inferior. Prin urmare, pentru o anumită aplicație dată, se alege un model logic adecvat de organizare a datelor (de exemplu, modelul relațional, ierarhic, rețea, etc.). Se mai spune că schema conceptuală este transformată într-o schemă logică exprimată cu ajutorul unor structuri abstrakte de date și operații furnizate de modelul de date respectiv. De exemplu, pentru modelul relațional faptele sunt depozitate în tabele, constrângările sunt exprimate cu ajutorul cheilor primare sau străine, etc.

Nivelul intern

După ce a fost realizată schema logică, aceasta trebuie proiectată prin intermediul unei scheme interne într-un anumit SGBD (Sistem de Gestire a Bazelor de Date). De exemplu, schema relațională poate fi implementată în Oracle, Access sau DB2. Schema internă include toate detaliile despre stocarea fizică și structurile de acces utilizate în sistemul respectiv (de exemplu, indecsi, clustere, etc.). Pentru același SGBD pot fi alese diferite structuri de stocare, după cum și pentru SGBD-uri diferite există structuri diferite. Prin urmare, pentru aceeași schemă logică pot fi alese mai multe scheme interne.

În concluzie, unul dintre avantajele nivelului conceptual este acela de a fi cel mai stabil dintre niveluri. El nu este practic afectat de schimbările interfețelor cu utilizatorii, de

structurile fizice de stocare sau de tehniciile de accesare a datelor. Presupunând că schema conceptuală a fost implementată într-un SGBD ierarhic, dar mai târziu se dorește utilizarea unui SGBD relațional, acest lucru nu va duce la modificarea schemei conceptuale, exceptând faptul când sistemul studiat s-a modificat.

3. Sistem de Gestiune a Bazelor de Date (SGBD)

Noțiuni despre SGBD

Un *Sistem de Gestiune a Bazelor de Date (SGBD, DataBase Management System)* este un sistem software care gestionează o bază de date și care permite utilizatorului să interacționeze cu aceasta. El acționează ca un depozit pentru toate datele și este responsabil pentru următoarele acțiuni:

- Stocarea datelor;
- Definirea structurilor de date;
- Manipularea datelor;
- Interogarea (extragerea și prelucrarea) datelor;
- Asigurarea securității datelor;
- Păstrarea integrității datelor;
- Permitea accesului concurent la date cu păstrarea consistenței acestora;
- Asigurarea unui mecanism de recuperare a datelor;
- Asigurarea unui mecanism de indexare care să permită accesul rapid la date.

Modul de organizare a datelor: modelul de date

Modelul de date folosit de un SGBD descrie modul de organizare a datelor în baza de date. Modelul de date reprezintă un tipar după care este organizată din punct de vedere ~~baza~~ baza de date. Modelul de date nu specifică datele, implementarea sau organizarea lor ~~nu~~, ci doar modul lor de organizare logică. După modelul folosit, există mai multe categorii principale de SGBD-uri:

SGBD ierarhic. Modelul ierarhic stochează datele în structuri de tip arbore. El pornește de la ideea că între date există o relație de tip părinte-copil. Nivelul cel mai de sus al arborelui, cunoscut sub numele de rădăcină, poate avea orice număr de dependențe. La rădăcină, acesteia pot avea dependențe lor și așa mai departe. În prezent, modelul ierarhic ~~nu~~ este deținut și nu se mai folosește aproape deloc.

Modelul rețea. Modelul rețea stochează datele sub formă de înregistrări și legături între date. Acest model poate fi văzut ca o extindere a celui ierarhic, diferența dintre cele două fiind că, în timp ce în modelul ierarhic o înregistrare copil are exact un părinte, în modelul rețea el poate avea mai mulți părinți, eventual chiar nici unul. Modelul rețea permite reprezentarea de structuri de date complexe dar este extrem de puțin flexibil și necesită un ~~model~~ extrem de complicat. Din această cauză, el este în prezent puțin folosit.

SGBD relational (SGBDR). Modelul relațional reprezintă probabil cea mai simplă structură pe care o poate avea o bază de date. Într-un SGBD relațional, datele sunt organizate în tabele. Tabele sunt formate din înregistrări care, la rândul lor, sunt formate din câmpuri. Două sau mai multe tabele pot fi legate prin intermediul unuia sau mai multor câmpuri. Bazele de date relaționale sunt foarte flexibile și ușor de folosit, astfel încât au cunoscut o dezvoltare intensă în ultimii douăzeci de ani și sunt în prezent cele mai răspândite. Cele mai populare SGBD-uri relaționale sunt Oracle, Informix și Sybase. Alături de acestea, o răspândire destul de largă o au SQL Server, produs de către Microsoft și DB2, produs de către IBM. Modelul relațional, în general, și SGBD-ul Oracle în particular vor fi prezentate pe larg în această carte.

SGBD orientat pe obiect. Acesta este cel mai nou tip de SGBD, încercând să integreze principiile programării orientate pe obiect (Actor, Smalltalk, C++) și ale bazelor de date. Bazele de date relaționale, cele mai populare în ultimul deceniu, ofereau prea puțin suport pentru tipurile neconvenționale de date. Necesitatea gestiunii obiectelor complexe (texte, grafice, hărți, imagini, sunete) și a gestiunii obiectelor dinamice (programe, simulări) care nu pot fi realizate cu ajutorul sistemelor relaționale a condus la introducerea conceptului de *obiect* în cadrul sistemelor de gestiune a bazelor de date. În prezent tehnologia bazelor de date orientate pe obiect este încă la început, neexistând un model general de SGBD orientat pe obiect, astfel încât producători diferiți folosesc moduri de organizare oarecum diferite. O soluție în acest sens este și crearea unui model relațional care să susțină și majoritatea principiilor modelării orientate pe obiect.

Deși în esență Oracle este un SGBD relațional, versiunea Oracle8 integrează și trăsături specifice programării orientate pe obiect. Astfel, Oracle8 permite stocarea obiectelor în tabele într-o manieră similară cu cea a numerelor și cuvintelor. De aceea, Oracle8 poate fi numit *SGBD relațional obiectual*. O prezentare a trăsăturilor tehnologiei orientate pe obiect din Oracle 8 se găsește în capitolul 10.

Comunicarea cu baza de date

Un SGBD nu este util dacă nu se poate comunica cu el, deci o primă funcție a unui SGBD este de a asigura accesul utilizatorului la baza de date. Prin aceasta se înțelege atât definirea structurilor din baza de date cât și manipularea și interogarea (extragerea și prelucrarea) datelor din baza de date. În plus, utilizatorul trebuie să aibă la dispoziție modalități de a controla integritatea și securitatea datelor. Cu alte cuvinte, comunicarea cu baza de date presupune următoarele patru funcții principale:

Definirea structurilor de date. Un SGBD trebuie să furnizeze un *Limbaj de Definire a Datelor (Data Definition Language, DDL)* care permite definirea (crearea, modificarea și distrugerea) *schemei* bazei de date, adică a structurilor de date folosite și a legăturilor dintre acestea.

Notă: Schema bazei de date, numită și *metadata*, nu trebuie confundată cu datele propriu-zise. Schema descrie modul de structurare al bazei de date, este definită de la început și se schimbă destul de puțin în timp. Datele, pe de altă parte, reprezintă instanțe ale schemei bazei de date și se modifică frecvent în timp. Aceste două concepte sunt analoge conceptelor de tip și variabilă dintr-un limbaj de programare.

Un SGBD trebuie să cuprindă un *dicționar de date* (*data dictionary*) căreodată referit sub denumirea de *catalog al sistemului*, care stochează date despre toate obiectele pe care le definește, cum ar fi numele obiectului, tipul, structura și locația sa. Ciclul de viață al unei structuri de date, de la crearea până la distrugerea ei, este înregistrat în dicționarul de date, precum și toate informațiile logice și fizice despre aceasta. Administratorul bazei de date trebuie să studieze aprofundat acest dicționar al SGBD-ului, pentru că acesta îl va îndruma pe întreaga perioadă de viață a bazei de date. Dicționarul de date al SGBD-ului Oracle este prezentat în secțiunea 6.11.

Manipularea datelor. Odată create structurile din baza de date, trebuie permisă inserarea de date în acestea, precum și actualizarea sau ștergerea celor existente. Aceste operații sunt efectuate de către *Limbajul de Manipulare a Datelor* (*Data Manipulation Language, DML*).

Interogarea datelor. Un SGBD trebuie să permită extragerea, vizualizarea și prelucrarea (filtrarea, sortarea, etc.) datelor existente. Aceste operații se fac cu ajutorul unui limbaj numit *limbaj de interogare sau acces* (*Data Query Language, DQL*).

Controlul datelor. *Limbajul de Control al Datelor* (*Data Control Language, DCL*) permite asigurarea securității și confidențialității datelor, salvarea datelor și realizarea fizică a modificărilor în baza de date, garantarea integrității și consistenței datelor în cazul manipulării concurente.

În Oracle, comunicarea cu baza de date se face prin intermediul limbajului SQL (*Structured Query Language*), care este în prezent cel mai răspândit limbaj de interogare a bazelor de date relaționale. În pofida numelui său, SQL nu este doar un limbaj de interogare, ci poate fi folosit și pentru definirea și manipularea datelor, cât și ca limbaj de control. Cu alte cuvinte, SQL cuprinde atât o componentă de interogare cât și componente de definire, manipulare și control a datelor. Administratorul bazei de date folosește limbajul SQL pentru a construi și întreține baza de date, iar utilizatorii pentru a vizualiza sau modifica datele.

Securitatea datelor

Securitatea este o preocupare constantă în proiectarea și dezvoltarea bazelor de date. În mod ușual, nu se pun probleme legate de existența securității, ci mai degrabă de cât de mare va fi aceasta. Un SGBD are în mod caracteristic mai multe nivele de securitate, pe măsură ce oferite de sistemul de operare sau de rețea. De obicei, un SGBD definește *conturi pentru utilizatori*, care necesită o parolă de conectare ce trebuie autentificată pentru a accesa datele.

Un SGBD oferă de asemenea și alte mecanisme cum ar fi grupurile, rolurile, privilegiile și profilurile care dă securitatea mai mult rafinament. Aceste nivele de securitate nu prevăd numai conținutul, ci și stabilirea politicii de securitate. De exemplu, într-un sistem de banking o companie își poate deschide un cont la o bancă, iar accesul la contul său din cadrul companiei pentru consultarea contului va fi autorizat printr-un număr și rel puțin o parolă. În plus, accesul poate fi diferențiat între diferiți membri ai

companiei, numai unora dintre ei fiindu-le permis să facă și tranzacții asupra contului. Pe de altă parte, personalul băncii poate avea acces la informații despre toate conturile din bancă.

În Oracle, securitatea datelor este asigurată în primul rând prin intermediul *utilizatorilor* bazei de date (în sensul folosit de Oracle, un utilizator al bazei de date este de fapt un cont și nu se identifică cu o persoană care accesează baza de date; în general, mai multe persoane pot folosi același cont de acces). Orice obiect al bazei de date este proprietatea unui utilizator, schema unui utilizator cuprinzând toate obiectele pe care acesta le deține. Pentru a accesa un obiect, un utilizator trebuie fie să fie proprietarul acelui obiect, fie să aibă *privilegiile* necesare pentru această operație. Privilegiile pot fi acordate direct unui utilizator sau pot fi grupate în *roluri*, care la rândul lor pot fi acordate utilizatorului. Aceste concepții sunt prezentate pe larg în capitolul 5.

Menținerea și constrângerea integrității

Integritatea datelor se referă la consistența și corectitudinea acestora. Pentru ca datele să fie consistente, ele trebuie modelate și implementate în același mod pentru toate circumstanțele. Pentru ca datele să fie corecte, ele trebuie să fie exacte, precise și să aibă un înțeles. Prin integritate datele devin informații. Integritatea nu numai că îmbunătățește calitatea datelor, ci mai mult decât atât, le dă acestora valoare și semnificație. Condițiile de integritate pe care trebuie să le satisfacă o bază de date pot fi atât condiții structurale, comune tuturor SGBD-urilor care folosesc un anumit model de date (de exemplu, orice SGBD relațional trebuie să impună respectarea integrității referențiale), cât și condiții de comportament, specifice fiecărei baze de date particulare (de exemplu, o coloană "culoare" dintr-un tabel nu poate avea decât valorile roșu, albastru, verde, etc.).

Dacă în practică se folosesc metode corecte de proiectare și implementare a bazei de date, SGBD-ul va ajuta la impunerea automată a integrității acestia, prin supravegherea valorilor introduse sau modificate, astfel încât acestea sunt respinse dacă nu au specificațiile cerute (de exemplu o verificare a domeniului de valori pe care poate să le ia o dată).

În Oracle, regulile de integritate pot fi impuse folosind *constrângeri* (*constraints*) – vezi secțiunea 6.1.3 - sau *declanșatori* (*triggers*) - vezi secțiunea 9.16.

Asigurarea accesului concurrent la date

În plus, în cazul existenței mai multor utilizatori, un SGBD trebuie să gestioneze accesul concurrent al acestora, menținând în același timp integritatea bazei de date. Vom spune în acest caz că SGBD-ul gestioneză *concurența* bazei de date. Concurența poate fi definită ca simultaneitate în sensul că doi sau mai mulți utilizatori accesează aceleași date în aceeași perioadă de timp. Gestionarea de către SGBD a concurenței o întâlnim atunci când mai mult de o persoană trebuie să acceseze aceeași bază de date, și în mod special aceleași date, iar SGBD-ul trebuie să asigure într-un fel sau altul că acest acces concurrent este posibil. În plus, fiecare utilizator trebuie să aibă o "vedere" consistentă asupra bazei de date, incluzând modificările vizibile făcute de către alții utilizatori, iar pe de altă parte SGBD-ul trebuie să împiedice modificări incorecte ale datelor, care ar compromite integritatea acestora. Iată aşadar cum noțiunile de integritate și concurență sunt strâns legate între ele.

Metodele pe care un SCBD le folosește pentru a realiza acest lucru sunt în principiu destul de simple, implementarea lor fiind însă destul de complexă. În esență, un SGBD asigură concurență în modul următor. Când două sau mai multe persoane vor să vizualizeze aceleași date fără a le modifica însă, totul este în ordine și nu trebuie luate măsuri suplimentare. Când însă cel puțin o persoană dorește să facă modificări asupra unor date care în același timp sunt vizualizate de către alte persoane, atunci SGBD-ul trebuie să stocheze mai multe copii ale datelor și să transfere toate modificările copiilor într-o singură versiune atunci când întreaga operațiune este terminată.

Problema este ceva mai complicată în momentul când mai mulți utilizatori încearcă să facă modificări asupra acelorași date. Un SGBD rezolvă această problemă folosind *blocarea* (*locking*). Utilizatorul care a efectuat primul modificarea datelor le blochează, ceilalți utilizatori neputându-le modifica până ce operația efectuată de acesta nu este încheiată. În Oracle, blocarea se poate face la nivel de tabel (atunci când un întreg tabel este blocat) sau la nivel de rând (atunci când doar rândul sau rândurile modificate sunt blocate). În general, cu cât unitatea de date blocate este mai mică, cu atât concurența este mai eficientă. Acest lucru se explică simplu prin faptul că mai mulți utilizatori pot avea acces simultan la date fără să aștepte. Prin urmare, blocările la nivel de rând sunt din acest punct de vedere cele mai eficiente. În Oracle, singurele blocări implicate - adică efectuate în mod implicit de către SCBD, fără o comandă explicită din partea utilizatorului - sunt cele la nivel de rând; Oracle blochează în mod implicit orice rând asupra căruia se execută o operație de modificare. În acest mod, se asigură un grad sporit al accesului concurrent la date. Pe de altă parte, blocările la nivel de tabel pot fi efectuate în mod explicit de către orice utilizator atunci rând, de exemplu, acesta nu dorește ca datele dintr-un tabel pe care îl accesează să fie între timp modificate de către un alt utilizator. Tipurile de blocări folosite de către Oracle sunt prezentate pe larg în secțiunea 7.3.

Tranzacții

Întregul mecanism care gestionează concurența într-o bază de date are la bază conceptul de *tranzacție*. Tranzacția este cea mai mică unitate de lucru. Aceasta înseamnă că nu poate include nici o operație mai mică decât o tranzacție. Toate tranzacțiile trebuie să fie *atomic*e, adică orice tranzacție individuală fie este complet executată, fie nu se execută deloc. Cu alte cuvinte, o tranzacție nu poate fi executată parțial. Când o tranzacție este încheiată se spune că este *permanentizată* (*committed*), când o tranzacție nu poate fi încheiată și este anulată se spune că este *derulată înapoi* (*rolled back*).

Derularea înapoi a unei tranzacții poate fi făcută explicit printr-o comandă a utilizatorului sau poate surveni în mod neașteptat, în cazul unei probleme în funcționarea sistemului. În Oracle, o tranzacție poate cuprinde una sau mai multe operații de interogare și manipulare a datelor și doar cel mult o operație de definire a datelor.

Un SCBD are, ca parte componentă, un *administrator de tranzacții* (*transaction manager*) al cărui scop este acela de a gestiona concurența și de a asigura integritatea tranzacțiilor. Administratorul de tranzacții are o sarcină grea pentru că el trebuie să permită mai mulți utilizatori să acceseze în același timp aceleși date, iar apoi, pentru a asigura corectitudinea datelor, să le înuiască înapoi ca și când accesul la date ar fi fost secvențial (adică, la un moment dat, datele au

fost accesate de o singură persoană). De aici pornește răspunsul fundamental despre modul cum un SGBD trebuie să rezolve copiile multiple ale datelor. Tranzacțiile care apar în aceeași perioadă de timp pot păstra acuratețea datelor dacă și numai dacă ele pot fi aranjate într-o *secvență* de tranzacții. Cu alte cuvinte, SGBD-ul trebuie să le rearanjeze ca și când rezultatul final al tuturor modificărilor să ar fi realizat de către o singură persoană.

Un SGBD realizează *recuperarea datelor* (*data recovery*) utilizând tranzacțiile ca unități de recuperare. În funcție de modul în care se derulează o tranzacție (este încheiată normal, abandonată la cerere sau întreruptă în mod neașteptat), SGBD-ul va folosi copiile datelor afectate de tranzacție fie pentru a actualiza aceste date (dacă tranzacția a fost încheiată în mod normal), fie pentru a anula modificările făcute (dacă tranzacția a fost abandonată la cerere sau întreruptă neașteptat). SGBD-ul păstrează o *istorie a tranzacțiilor* (*transaction log*) cu scopul de a le derula înainte sau înapoi. *Derularea înapoi* (*roll back*) a unei tranzacții înseamnă anularea efectelor acesteia asupra datelor. *Derularea înainte* (*roll forward*) a unei tranzacții înseamnă operarea modificărilor tranzacției asupra datelor. Atunci când efectele unei tranzacții finalizează nu sunt transmise din memorie pe disc, de exemplu din cauza unei probleme hardware sau software, SGBD-ul repetă tranzacția respectivă. Prin urmare, cheia pentru modul de recuperare a tranzacțiilor într-un SGBD este faptul că tranzacția trebuie să fie atomică și poate fi efectuată, anulată sau repetată atunci când este necesar.

Tranzacțiile și rolul lor în gestionarea concurenței și recuperarea datelor în Oracle vor fi prezentate pe larg în capitolul 7.

Indeci

Un *index* al bazei de date are în principiu același rol cu indexul unei cărți: pentru găsirea unui termen într-o carte este posibilă căutarea acestuia pagină cu pagină, însă mai eficientă este crearea unui index de termeni pentru respectiva carte și utilizarea acestuia. Deci un index este o structură opțională a bazei de date care mărește viteza de acces la date. Pe de altă parte însă, un index îngreunează operațiile de modificare a datelor; dacă datele din baza de date se modifică, trebuie modificat și indexul, ceea ce mărește timpul de execuție a unor astfel de operații. De aceea crearea indecșilor trebuie făcută numai după o analiză atentă a datelor în cauză, fiind în principiu recomandată pentru date asupra căror operațiile de interogare sunt mult mai frecvente decât cele de modificare. În cazul folosirii judicioase a indecșilor, avantajele acestora sunt mult mai semnificative decât dezavantajele. O prezentare detaliată a indecșilor în Oracle se găsește în secțiunea 6.3.

Baze de date distribuite.

Având în vedere rolul din ce în ce mai important al rețelelor și proceselor distribuite în informatică, sistemele de gestiune a bazelor de date trebuie să poată beneficia de avantajele procesării și stocării distribuite. Intuitiv, o *bază de date distribuită* este o mulțime de date corelate logic, dar distribuite pe mai multe mașini interconectate printr-o rețea de comunicație. Din punct de vedere al utilizatorului, o bază de date distribuită reprezintă o singură bază de date. Programul de aplicație care manipulează baza de date poate avea acces la date rezidente pe mai multe mașini, fără ca programatorul să cunoască localizarea acestor date (distribuirea lor fizică).

Bazele de date distribuite oferă utilizatorului o flexibilitate și o securitate sporită față de o singură bază de date. Să luăm, de exemplu, o companie care are sedii în mai multe orașe ale lumii. Dacă se optează pentru o arhitectură distribuită, fiecare dintre aceste locații își administrează propria bază de date, pe care o accesează în mod frecvent, putând să aibă în același timp acces și la bazele de date ale celorlalte locații. Astfel distribuirea bazei de date duce la exploatarea mai eficientă a bazei de date și ușurează administrarea acesteia, care se face local. În plus, defectarea unei baze de date locale nu afectează celelalte baze de date din rețea, cu alte cuvinte bazele de date locale sunt protejate împotriva defecțiunilor din celelalte noduri ale rețelei.

Arhitectura sistemului Oracle permite cu ușurință crearea și folosirea unei baze de date distribuite, existând în plus atât modalități pentru asigurarea transparenței locației datelor față de programul aplicație, cât și modalități de replicare locală a datelor aflate în alte noduri ale rețelei, ducând astfel la reducerea traficului prin rețea.

Tipuri de utilizatori ai bazei de date

În general, o bază de date este accesată de mai mulți utilizatori, care se diferențiază după operațiile pe care aceștia le efectuează asupra bazei de date. În linii mari, utilizatorii bazei de date se împart în următoarele categorii:

- **administratorul bazei de date (DataBase Administrator, DBA)** care definește baza de date și este responsabil pentru buna funcționare a acestia, aceasta însemnând optimizarea stocării datelor și a accesului la baza de date, asigurarea integrității și securității datelor, asigurarea de măsuri pentru recuperarea datelor în caz de defectu;
- **programatorul (dezvoltatorul de aplicații)**, care creează programe ce manipulează și interoghează datele din baza de date. Programatorul trebuie să exploateze facilitățile privind accesul concurrent la date pentru a asigura integritatea și consistența acestora și trebuie să fie preocupat de probleme suplimentare precum performanța, mențenanța și portabilitatea codului;
- **utilizatorul final**, care poate interoga și manipula datele, fără a-i fi necesară cunoașterea modului de organizare a acestora sau a problemelor privind integritatea datelor sau accesul concurrent la acestea.

4. SGBDR Oracle: Concepte Generale

Nucleul Oracle, cunoscut și sub numele de Oracle Server, este cel mai utilizat SGBDR din lume și reprezintă *nucleul (back-end)* liniei de produse Oracle, care include, printre altele, un set de instrumente grafice pentru administrarea bazei de date (*Oracle Enterprise Manager*), un utilitar care permite utilizatorilor lansarea de comenzi SQL asupra bazei de date (*SQL*Plus*) și un set de instrumente de dezvoltare a aplicațiilor din generația a patra, (*Oracle Developer*). *Interfața* unui sistem (*front-end*) este software-ul pe care un utilizator îl poate vedea și utiliza; nucleul este un software aparte care este invizibil utilizatorului final. În arhitectura *client/server*, interfața se referă la *client* iar nucleul la *server*. În cazul nostru, **Nucleul** este *serverul nucleu*, iar toate aplicațiile și instrumentele software care **interacționează** cu el - printre care Oracle Enterprise Manager, SQL*Plus și Oracle Developer -

sunt considerate *interfață*. În mod ușual, de multe ori acest nucleu al bazei de date este denumit simplu "baza de date".

SGBDR Oracle, sau Oracle Server, este un depozit în care datele pot fi stocate și regăsite, prin comenzi sau interogări trimise acestuia, fără a fi necesară cunoașterea structurii sale interne. În realitate însă, modul în care datele sunt stocate sau interogate va avea un impact semnificativ asupra performanței și utilizării aplicației. Programatorul trebuie să țină seama de obiectele și structura bazei de date ce poate afecta dezvoltarea și performanța aplicației dar nu este necesar să știe cum se controlează și cum se reorganizează baza de date.

Principalele caracteristici ale acestui sistem sunt următoarele:

- Aplicațiile pot fi dezvoltate în mod independent de caracteristicile fizice ale calculatorului și ale sistemului de operare folosit de către serverul bazei de date. Aceasta înseamnă că serverul bazei de date poate rula pe *orice platformă* - de la Windows NT la UNIX și Macintosh - și permite schimbarea bazei de date de pe o platformă pe alta, fără ca aplicația să simtă această schimbare. Independența față de platformă este una dintre principalele trăsături ale SGBDR Oracle.
- *Manipularea directă* a fișierelor și a structurilor pe care motorul bazei de date le gestionează trebuie evitată, chiar atunci când este posibilă. Orice manipulare a datelor gestionate de SGBD trebuie realizată prin utilizarea software-ului client al bazei de date, care trimite comenzi corespunzătoare serverului bazei de date.
- Serverul bazei de date decide cel mai bun mod de a *optimiza procesarea* în timpul răspunsurilor la interogări. Cu alte cuvinte, o aplicație nu are nevoie să cunoască informații despre alte aplicații care operează concurrent sau despre caracteristicile fizice ale datelor stocate deoarece serverul bazei de date gestionează accesul la date și optimizează viteza de regăsire a acestora în baza de date.
- *Integritatea* bazei de date poate fi garantată. Aceasta înseamnă că serverul bazei de date poate fi folosit pentru a evita stocarea datelor invalide sau pentru a împiedica extragerea datelor eronate sau ilogice. Cu alte cuvinte, dezvoltatorul aplicației este scutit de un proces complex de verificare a eventualelor erori ale datelor.
- Este asigurată *securitatea* datelor. Acesta înseamnă că un utilizator al bazei de date trebuie să aibă autorizația să folosească atât baza de date cât și aplicația client. Accesul la aplicație și la datele acesteia pot fi controlate și menținute în mod separat. Accesul la aplicație și la sistemul de operare este primul nivel de securitate al bazei de date, SGBDR-ul este al doilea nivel.
- Este asigurată *concurența* datelor, adică aceleași date pot fi accesate de către utilizatori diferiți în mod simultan, nefiind necesar ca un utilizator să aștepte decât dacă încercă modificarea acelorași rânduri care sunt în curs de modificare de către alt utilizator. În plus, fiecare comandă SQL are o "vedere" consistentă asupra bazei de date, nefiind afectată de modificările efectuate în timpul executării comenzi.

- Arhitectura Client/Server a sistemului Oracle permite realizarea cu ușurință a unei baze de date *distribuite*, adică a unei rețele de baze de date, fiecare gestionată de către propriul SGBDR Oracle, care, din punct de vedere logic, apare utilizatorului ca o singură bază de date. Datele din fiecare bază de date pot fi accesate și modificate simultan. Fiecare calculator care gestionează o bază de date poartă numele de *nod*. Baza de date la care este conectat utilizatorul în mod direct poartă numele de *bază de date locală*. Orice altă bază de date accesată de acest utilizator poartă numele de *bază de date la distanță*. Când o bază de date locală acceseașă pentru informații o bază de date la distanță, baza de date locală este un client al bazei de date la distanță (conform arhitecturii client/server).

Comunicarea cu serverul bazei de date

Pentru a putea funcționa, software-ul client al bazei de date are nevoie de un mod de comunicare eficient cu serverul bazei de date. Baza de date Oracle permite ca acest lucru să se realizeze în două moduri. Prima metodă implică un produs numit *Net8* (în cazul versiunii Oracle8) sau *SQL*Net* (pentru versiunile anterioare) care realizează comunicarea cu ajutorul romenziilor, interogărilor și datelor între software-ul client și baza de date Oracle prin conexiunea rețelei. La clientul terminal, codul aplicației client transferă mesajele către Net8 local, iar acesta le transferă către un alt Net8 îndepărtat folosind protocolul de transfer. Aceste mesaje sunt recepționate de Net8 în server, care le transmite serverului bazei de date pentru a fi executate. Serverul execuță cererea și răspunde clientului urmând aceeași cale.

A doua metodă este disponibilă numai pentru *implementări monolitice* ale bazei de date Oracle și ale instrumentelor sale, unde software-ul client și software-ul server sunt rezidente pe același calculator. Instrumentele monolitice Oracle pot specifica valoarea identificatorului bazei de date (valorarea curentă a parametrului ORACLE_SID, vezi capitolul 3) să permită software-ului client să comunice direct cu serverul, pe aceeași mașină.

Argimica benzii de comunicare între client și server este mai mare în cazul implementării monolitice. Oricum comunicarea între procesele client/server rămâne în continuare unul dintre punctele nevrăgice ale bazelor de date. De aceea, se tinde spre reducerea cât mai mult cu puțință a cantității de date transferate între procesele client/server.

SQL

SQL (*Structured Query Language, Limbaj structurat de interogare*) este limbajul utilizat în comunicarea între software-ul client și serverul bazei de date. Cu alte cuvinte, SQL este interfața între utilizatori și SGBDR-ul Oracle. Toate celelalte aplicații și utilitare de către utilizatori pentru a interacționa cu acesta acționează ca interpretoare (interpretori); ele generează comenzi SQL bazate pe cererile utilizatorilor și le transmit într-o formă de comenzi SQL.

Limbajul SQL a fost standardizat de către ANSI (American National Standards Institute). Această standardizare ca numitor comun pentru motoarele bazelor de date care folosesc SQL.

Pentru a spori funcționalitatea limbajului sau pentru a accepta structurile fizice sau trăsăturile care sunt specifice anumitor motoare de baze de date, baza de date Oracle (și multe alte motoare de baze de date care folosesc SQL) furnizează comenzi adiționale limbajului SQL.

Cel mai important aspect al limbajului SQL, și deseori cel mai greu de înțeles, este faptul că SQL este un limbaj *non-procedural*. Aceasta înseamnă că instrucțiunile SQL comunică server-ului bazei de date *ce să facă* și nu *cum să facă*. Programatorii în limbajele COBOL, C, sau alte 3GL (limbaje de generația a treia) sunt obișnuiți cu împărțirea unui program prin pași algoritmici. Aceasta este o mare piedică în scrierea eficientă a unui cod SQL, unde rezultatul final nu trebuie focalizat pe pași intermediari. Multe operații pot fi scrise în SQL utilizând o singură comandă.

Așa cum am menționat mai înainte, limbajul SQL include un *limbaj de definire a datelor (DDL)*, un *limbaj de manipulare a datelor (DML)*, un *limbaj de interrogare a datelor (DQL)* și un *limbaj de control al datelor (DCL)*. În continuare trecem în revistă pe scurt fiecare dintre aceste componente.

Limbajul de definire a datelor (a schemei bazei de date) permite descrierea structurii logice a bazei de date, cât și modul ei de stocare. El include instrucțiuni ce permit crearea (CREATE), modificarea (ALTER) și distrugerea (DROP) unei baze de date, a utilizatorilor și rolurilor acestora precum și a obiectelor care alcătuiesc schema unui utilizator: tabele de bază, tabele virtuale (vederi), indecsi, grupuri de tabele, sevențe, declanșatori, proceduri, funcții, etc. În plus, odată cu crearea tabelelor bazei de date, se pot impune restricții de integritate pentru acestea (integritatea entității, integritatea referențială, unele restricții de comportament). Limbajul de definire a datelor include și instrucțiuni care permit acordarea (GRANT) și revocarea (REVOKE) de privilegii și roluri unor utilizatori sau altor roluri. Toate aceste comenzi sunt prezentate pe larg în capitolele 4, 5 și 6.

Limbajul de manipulare a datelor permite inserarea (INSERT) de noi date în tabele, actualizarea (UPDATE) sau ștergerea (DELETE) a celor existente. Aceste instrucțiuni sunt prezentate în secțiunile 8.2, 8.3 și 8.4.

Comanda fundamentală a limbajului SQL care permite interrogarea unei baze de date este comanda SELECT. Prin utilizarea acestei comenzi, asupra tabelelor de bază și tabelelor virtuale pot fi efectuate operațiile algebrei relaționale, cât și operații suplimentare, deriveate din acestea. O prezentare amănunțită a tuturor facilităților oferite de comanda SELECT se găsește în secțiunea 8.1.

Limbajul de control a datelor (bazei de date) se referă în principal la comenziile privind controlul tranzacțiilor și al accesului concurrent la date (COMMIT, ROLLBACK, SET TRANSACTION, SAVEPOINT). Acestea sunt prezentate în capitolul 7.

Limbajul SQL poate fi utilizat atât de administratorul bazei de date, cât și de programator sau utilizatorul final. Administratorul folosește în principal DDL, utilizatorul final folosește DML și DQL, iar programatorul folosește atât DML și DQL cât și DCL.

PL/SQL

Dacă administratorul bazei de date poate folosi fără probleme partea de definire a datelor din limbajul SQL pentru a crea schema bazei de date, nu același lucru este valabil și pentru dezvoltatorul de aplicații. În cazul acestuia, facilitățile limitate furnizate de limbajul SQL și în special caracterul sau non-procedural sunt total insuficiente pentru crearea unei aplicații mai complexe. În Oracle, dezvoltatorii de aplicații folosesc PL/SQL (Procedural Language/Structured Query Language), o extensie a limbajului SQL, care îmbină avantajele manipulării datelor din SQL cu instrucțiunile procedurale (IF, LOOP, etc.). PL/SQL permite ca instrucțiunile SQL de interogare (SELECT) și manipulare a datelor (INSERT, UPDATE, DELETE) să fie incluse în blocuri sau unități procedurale de cod.

Unitatea de bază a limbajului PL/SQL este blocul. La rândul lor, blocurile pot fi multurate secvențial sau pot fi conținute unul în altul, putându-se realiza astfel un program cu funcționalitate foarte complexă. În plus, există blocuri speciale care reprezintă subprograme (proceduri și funcții), care pot fi apelate în cadrul unui alt bloc, la fel ca în orice limbaj procedural. În limbajul PL/SQL se scriu și declanșatorii (triggers-ele) bazei de date. Declanșatorul reprezintă o procedură apelată automat atunci când se efectuează modificări asupra datelor dintr-un anumit tabel.

PL/SQL este limbajul folosit de toate utilitarele din Oracle Developer, cum ar fi Oracle Forms și Oracle Reports. Motorul PL/SQL inclus în aceste unele are rol de procesor. O prezentare pe larg a limbajului PL/SQL se găsește în capitolul 9.

Produse Oracle8

În plus de serverul bazei de date, Oracle oferă diferite produse pentru efectuarea funcțiilor administrative ale bazei de date și pentru dezvoltarea de aplicații. Dintre acestea cele mai importante și mai des folosite sunt *Oracle Enterprise Manager (OEM)*, *SQL*Plus* și *Oracle Developer*. Oracle Enterprise Manager este un pachet ce conține mai multe instrumente pentru administrarea bazei de date. SQL*Plus este un utilitar care permite utilizatorilor lansarea de comenzi SQL asupra bazei de date, cât și lansarea unor comenzi PL/SQL, precum crearea procedurilor, funcțiilor și a pachetelor stocate și a declanșatoarelor de date. Oracle Developer cuprinde mai multe instrumente de dezvoltare a aplicațiilor. Dintre acestea, cel mai des folosit este *Oracle Forms* (vezi capitolele 11 și 12), folosit pentru realizarea de formulare pentru interogarea și manipularea datelor bazei de date. Anexa 1 cuprinde o prezentare mai amplă a produselor Oracle.

Capitolul 1

Modelul relațional

Modelul relațional reprezintă datele sub forma unor structuri bidimensionale, asemănătoare tabelelor. Apariția sa a datorat nevoii de a țărua concepția, accesarea și procesarea datelor. Modelele de date existente anterior (ierarhic și rețea) erau extrem de complexe și necesitau personal extren de specializat pentru a concepe și a naviga în baza de date. De asemenea, datorită faptului că performanțele acestor tipuri de baze de date erau dependente de proiectarea și design-ul lor, pentru a asigura o performanță corespunzătoare trebuia investit foarte mult efort în activitatea de proiectare. Aceste motive făceau ca disponibilitatea de personal adecvat să fie foarte limitată. În plus, datorită complexității lor, sistemele care foloseau aceste modele erau foarte greu de instalat. În sfârșit, dar nu în ultimul rând, modelele ierarhice și rețea necesitau ca programatorul aplicației să cunoască implementarea fizică a bazei de date (legăturile existente între înregistrări) și cunoștințe despre limbajul de manipulare a datelor corespunzător.

Modelul relațional este o soluție pentru rezolvarea acestor probleme. Deși sistemul software necesar unui model relațional este foarte complex, învățarea limbajelor de definire a datelor și a celor de manipulare și interogare a datelor necesită mult mai puțin timp și efort decât cele corespunzătoare modelelor ierarhic și rețea. De asemenea, proiectarea unui sistem care folosește modelul relațional este flexibilă, astfel încât modificările sunt destul de ușor de implementat, reducându-se astfel efortul corespunzător fazei de proiectare a bazei de date. În plus, simplitatea modelului relațional și a limbajului său de interogare a datelor permite utilizatorilor să execute propriile lor interogări.

Concepțele teoretice care formează baza modelului relațional au apărut la începutul anilor '70, în principal datorită lui Edgar F. Codd. Acest model diferă de modelele precedente prin aceea că nu se bazează pe dependențele de secvență și cale. Modelele existente anterior permiteau accesul la date prin căi specificate de o schemă. Cum programele asociate depindeau de căile diferitelor înregistrări din baza de date, dacă, datorită unei schimbări de design, calea de acces pentru o înregistrare se schimba, era necesară și modificarea programului de acces la datele respective, pentru a lăua în considerare această schimbare. Pe de altă parte, modelul relațional nu folosește căile de acces în structura sa logică sau în limbajul său de interogare a datelor. De asemenea, în modelele ierarhice și rețea datele puteau fi accesate în mai mult de o secvență, iar programele de aplicație utilizau secvența respectivă a datelor. Evident, dacă secvența datelor era schimbată, programul trebuia schimbat pentru a lăua în considerare noua secvență. Modelul relațional nu utilizează o anumită secvență în structura sa logică sau în limbajul său de interogare. Această independență sporită a datelor permite o tranziție mai ușoară la noi relații între date.

Modelul relațional nu este orientat spre sistemul de calcul, adică modelul nu include regulile, structurile și operațiile referitoare la implementarea fizică a unui sistem de baze de date. De fapt, unul dintre principalele obiective ale sistemului relațional este separarea clară între aspectele fizice și cele logice ale unei baze de date. Spre deosebire de modelele menționate anterior, utilizatorul poate obține date specifice din modelul relațional folosind un limbaj ne-procedural care îi permite să descrie datele cerute în loc de navigația folosită.

În același timp, modelul relațional și operațiile folosite de către acesta au la bază o riguroasă fundamentare matematică; s-au creat *algebra relațională* și *calculul relațional*. Totuși, pe lângă avantajele clare ale rigurozității matematice, matematizarea excesivă a modelului și a limbajului de

manipulare crează o problemă datorită faptului că în afara mediului academic doar un procent mic de utilizatori au pregătirea necesară pentru a le înțelege. Din fericire, folosirea modelului relațional nu necesită neapărat înțelegerea în profunzime a matematicii pe care este bazat. Operațiile de manipulare a datelor pot fi privite ca o serie de operații de proiecție, filtrare, reuniune, intersecție, etc. asupra datelor din tabele.

Deși are la rândul lui imperfecțiuni, modelul relațional a devenit în ultima vreme principalul model de baze de date și a fost adoptat de majoritatea producătorilor de software din acest domeniu. Principalele sale avantaje sunt simplitatea, fundamentarea matematică riguroasă și independența datelor, adică separarea aspectelor logice ale acestora de cele fizice.

1.1. Concepte de bază ale modelului relațional

În general, definirea unui model de date presupune precizarea și identificarea următoarelor elemente:

- Structurile de date folosite;
- Operatorii care acționează asupra structurilor de date;
- Restricțiile care trebuie impuse pentru menținerea corectitudinii datelor, numite și *restricții de integritate*.

Această secțiune prezintă pe larg fiecare dintre aceste componente ale modelului relațional.

În modelul relațional datele sunt reprezentate ca structuri bidimensionale, numite *relații*. O relație este alcătuită dintr-un număr fix de elemente numite *attribute*, fiecare dintre acestea putând lua valori într-o mulțime finită, numită *domeniu*. Numărul de attribute al unei relații se numește *aritatea relației*.

`relație(atribut_1, atribut_2, ..., atribut_n)`

De exemplu datele despre angajații unei firme se pot reprezenta ca o relație de aritate 7, după cum urmează:

`salariat(cod_salariat, nume, prenume, adresa, dată_naștere,
sex, cod_departament)`

Elementele unei relații se numesc *tupluri* sau *înregistrări*. De obicei relațiile sunt reprezentate sub forma unor *tabele*, în care fiecare *rând* reprezintă un tuplu și fiecare *coloană* reprezintă valorile tuplurilor corespunzătoare unui anumit atribut.

salariat

cod_salariat	nume	prenume	adresa	dată_naștere	sex	cod_departament
S1	Ion	Cosmin	Str. Jiului Nr.3	01/01/1970	M	D1
S2	Popescu	Vasile		15/11/1956	M	D1
S3	Ionescu	Gina	Str. Valea Albă Nr.4	28/02/1976	F	D2
S4	Costache	Viorel	Str. Crișana Nr. 11	12/02/1975	M	D1

Relațiile ce constituie o bază de date trebuie să satisfacă mai multe condiții, după cum urmează:

1. Fiecare atribut trebuie să poarte un nume, care este unic în cadrul relației. Modelul relațional nu permite ca două attribute din cadrul aceleiași relații să poarte același nume. Pe de altă parte, este posibil ca două attribute din două relații diferite să poarte același nume.
2. Fiecare atribut poate avea doar valori atomice, deci care nu se pot descompune din punct de vedere logic. Să presupunem, de exemplu, că un producător de automobile atribuie fiecarui automobil un cod de identificare, care se obține prin concatenarea mai multor coduri, reprezentând marca automobilului, culoarea, fabrica unde este produs, seria și numărul de fabricație, etc. Atunci, definirea acestui cod de identificare ca atribut al unei relații ar încălca această regulă, deoarece fiecare valoare a atributului s-ar putea descompune în mai multe valori cu semnificație logică.
3. Fiecare tuplu este unic. Nu sunt permise tupluri identice (duplicate). În plus, unicitatea unui tuplu nu este limitată la datele existente: fiecare tuplu trebuie să fie unic tot timpul.

Aceasta înseamnă că într-o relație există întotdeauna unul sau mai multe attribute care asigură că tuplul va rămâne unic tot timpul. Un atribut sau set de attribute care identifică în mod unic un tuplu se numește *cheie candidată* sau mai simplu *cheie*. Pentru fiecare relație se alege din mulțimea cheilor candidate o cheie care se numește *cheie primară* a relației. Eventualele alte chei candidate, diferite de cheia primară, se numesc *chei alternative*. Când mai mult de un singur atribut este necesar pentru a crea o cheie, se spune că avem o *cheie compusă*. În cazul unei chei compuse, attributele care fac parte din cheie sunt alese astfel încât nici unul să nu fie în plus, adică nici un atribut nu poate fi șters din cheia candidată astfel încât partea care rămâne din cheia candidată să identifice încă în mod unic plurile; de aceea, se spune că o cheie trebuie să fie *minimală*.

Uneori, în unele tupluri dintr-o relație, un atribut poate fi necunoscut sau neaplicabil. Pentru a reprezenta un astfel de atribut se folosește o valoare convențională, notată cu *Null*. Ca regulă generală, nici unul dintre attributele care alcătuiesc cheia primară nu poate avea valoarea *Null* pentru nici unul din plurile relației.

Exemplu:

Pentru relația *salariat* definită mai sus, o cheie candidată este *cod_salariat*. Dacă se presupune că nume, prenume și dată_năștere identifică în mod unic salariatul, atunci combinația acestor trei attribute este o altă cheie candidată, care, datorită faptului că este alcăută din mai multe attribute, este și cheie compusă. Dacă cheia *cod_salariat* este aleasă cheie primară, atunci combinația celor trei attribute (nume, prenume și dată_năștere) devine cheie alternativă.

În alegerea cheilor candidate să țină cont de principiul minimalității în sensul că din cheia compusă nume, prenume și dată_năștere nu poate fi șters nici un atribut și nici nu mai trebuie adăugat altul, aceste trei attribute identificând în mod unic plurile. De exemplu, combinația (nume, prenume, dată_năștere, cod_departament) nu este o cheie a relației *salariat* pentru că și în absența atributului *cod_departament*, un salariat este identificat în mod unic de primele trei attribute. Observăm că tuplul identificat prin valoarea cheii primare egală cu S2 are atributul *adresa* necunoscut, adică acesta are valoarea *Null*. Prin urmare, atributul *adresa* nu poate fi cheie primară și nici nu poate intra în componența unei chei compuse care este și cheie primară.

4 Dezvoltarea aplicațiilor de bază de date în Oracle8 și Forms6

Se numește *cheie străină* un atribut sau o mulțime de atribute ale unei relații R1 care există și într-o altă relație R2, nu neapărat distinctă de R1, și care formează cheia primară a relației R2. În cazul acesta, cheia străină din R1 se spune că *face referință* la cheia primară din R2. Valorile pe care le ia cheia străină, dacă nu au valoarea Null, trebuie să se găsească printre valorile cheii primare la care face referință.

Exemplu:

Dacă mai definim o relație de aritate 3, după cum urmează:

departament (cod_departament, denumire, localitate)

cod_departament	denumire	localitate
D1	Analiză financiară	București
D2	Depozit	Pitești
D3	Contabilitate	Constanța
D4	Magazin	Pitești

pentru evidența departamentelor din firmă, atributul cod_departament din relația salariat devine cheie străină care face referință la cheia primară a relației departament. Valoarea cheii străine cod_departament din relația salariat trebuie ori să fie ori Null, ori să coincidă cu o valoare a cheii primare la care face referință. Prin urmare, dacă atributul cod_departament din relația salariat ar fi avut pentru un anumit tuplu valoarea D5, valoare ce nu se regăsește printre valorile cheii primare cod_departament din relația departament, atunci s-ar fi încălcătat o regulă de integritate.

Există posibilitatea ca o cheie străină să facă referință la cheia primară a propriei relații. Pentru a ilustra acest lucru, să adăugăm la relația salariat un nou atribut cod_manager, reprezentând codul superiorului ierarhic al salariatului (facem presupunerea că un angajat poate avea cel mult un superior ierarhic).

cod_salariat	nume	prenume	cod_manager
S1	Ion	Cosmin	
S2	Popescu	Vasile	S1
S3	Ionescu	Gina	S1
S4	Costache	Viorel	S2

Deci salariatul cu codul S1 este superiorul ierarhic al salariatelor cu codurile S2 și S3, salariatul cu codul S2 este superiorul ierarhic al salariatului cu codul S4, iar S1 nu are un superior ierarhic.

4. Navigația în cadrul modelului relațional se face prin intermediul valorii pe care o ia un atribut. Aceasta înseamnă că limbajul de interogare va prelua ca parametri de intrare doar numele unor atribute și valorile posibile ale acestora, corespunzătoare cerinței utilizatorului, și va returna tuplurile care satisfac această cerință. Spre deosebire de altele modele, nu există legături între tupluri și deci nu există dependență de calea urmată.

5. Tuplurile pot fi prezentate utilizatorului în orice ordine. Deci acesta nu trebuie să facă nici o presupunere în privința ordinii tuplurilor.
6. Atributele pot fi prezentate în orice ordine, deci utilizatorul nu trebuie să facă nici o presupunere în privința ordinii acestora. Cu alte cuvinte, în modelul relațional nu există dependență de secvență. De exemplu, deși relațiile

`Salariat(cod_salariat, nume, prenume, adresa, dată_naștere,
sex, cod_departament)`

și

`Salariat(cod_salariat, nume, prenume, cod_departament,
dată_naștere, sex, adresa)`

par diferite, ele sunt echivalente funcțional pentru că au atributе identice, chiar dacă sunt în altă ordine.

7. Relațiile pot fi manipulate pentru a furniza utilizatorului diferite vederi asupra datelor, rezultatul fiind noi relații. Cu alte cuvinte, rezultatul manipulării relațiilor sunt noi relații. În plus, relațiile produse ca rezultat al comenziilor limbajului de interogare a datelor satisfac toate regulile la care sunt supuse relațiile inițiale.

1.2. Constrângeri de integritate

Pentru asigurarea integrității datelor, o bază de date trebuie să satisfacă un număr de constrângeri, numite *constrângeri de integritate*. Constrângările de integritate ale modelului relațional se pot împări în două clase: *constrângeri structurale*, care trebuie satisfăcute de orice bază de date care folosește modelul relațional și *constrângeri de comportament*, care sunt specifice fiecărei baze de date particulare.

Constrângările de integritate structurale exprimă proprietăți fundamentale, inerente modelului relațional și sunt în general specificate la definiția bazei de date, ca parte a schemei bazei de date. În modelul relațional există două tipuri de constrângeri structurale: *de entitate* și *de referință*. Ele au fost menționate mai înainte, când am vorbit despre proprietățile sistemului relațional, fără însă a le menționa explicit numele:

1. *Integritatea entității*: O cheie primară nu poate conține atrbute ce pot avea valoarea Null. În plus, prin însăși definiția unei chei primare, ea trebuie să fie unică și minimală.
2. *Integritatea referirii*: Valoarea unei chei străine trebuie ori să fie ori Null, ori să coincidă cu o valoare a cheii primare la care face referință.

Aceste două tipuri de constrângeri pot fi impuse în Oracle prin simpla lor adăugare la definiția tabelelor respective, vezi secțiunea 6.1.3.

Constrângările de integritate de comportament sunt specifice unei anumite baze de date și își conțin de semnificația valorilor atributelor din baza respectivă. De exemplu, constrângările de domeniu restricționează valorile unui atribut la o anumită mulțime, iar constrângările sintactice se pot referi la tipul datelor, lungimea atributelor, etc. Constrângările de comportament pot exprima legături între valorile unor atrbute diferite, de exemplu valoarea unui atribut este dependentă de valoarea altui atribut sau set de atrbute sau o expresie formată din valorile mai multor atrbute trebuie să se încadreze în anumite limite etc.

Constrângerile de comportament pot fi impuse în Oracle fie prin adăugarea lor la definiția tabelelor (vezi secțiunea 6.1.3), fie prin definirea unor secvențe de program, numite *declanșatori* (*triggers*) care sunt atașate tabelelor și care intră în acțiune la încălcarea acestor constrângeri, împiedicând operațiile care ar duce la încălcarea integrității (vezi secțiunea 9.16).

1.3. Operatorii sistemului relațional

În afara relațiilor și a proprietăților acestora, modelul relațional este definit și prin setul de operații care se pot efectua asupra acestor relații. Există două moduri de descriere matematică a acestor operatori, și anume *algebra relațională* și *calculul relațional*, vezi [3].

Algebra relațională, introdusă de către Codd, este formată dintr-o mulțime de opt operatori, ce acționează asupra relațiilor și generează tot o relație. Operatorii algebrei relaționale sunt fie operatorii tradiționali pe mulțimi (UNION, INTERSECT, DIFFERENCE, PRODUCT), fie operatori relaționali speciali (PROJECT, SELECT, JOIN, DIVISION). Cum ieșirea generată de fiecare dintre acești operatori este tot o relație, este posibilă combinarea și compunerea lor. Cinci dintre operatori (PROJECT, SELECT, DIFFERENCE, PRODUCT, UNION) sunt operatorii primiți ai limbajului, iar ceilalți trei (JOIN, DIVISION, INTERSECT) sunt operatori derivați, putând fi definiți în funcție de primii. Unii dintre operatori se aplică unei singure relații (operatori unari), iar alții operează asupra a două relații (operatori binari).

Calculul relațional reprezintă o adaptare a calculului predicatorilor la domeniul bazelor de date relaționale. Ideea de bază este de a identifica o relație cu un predicat. Pe baza unor predicate inițiale, prin aplicarea unor operatori ai calculului cu predicate (conjuncția, disjuncția, negația, cuantificatorul existențial și cel universal) se pot defini noi predicate, adică noi relații.

Algebra relațională și calculul relațional sunt echivalente unul cu celălalt, în sensul că orice relație care poate fi definită în algebra relațională poate fi definită și în calculul relațional și reciproc.

1.3.1 Algebra relațională și limbajul SQL

În prezent, limbajul dominant folosit pentru interogarea bazelor de date relaționale este SQL (*Structured Query Language*), care este un limbaj bazat pe operațiile algebrei relaționale. Cu alte cuvinte, orice operator al algebrei relaționale poate fi descris folosind comanda SELECT a limbajului SQL cu diverse clauze. În ceea ce urmează vom defini operatorii algebrei relaționale și vom exemplifică modul de implementare a acestor operatori în SQL. Pentru o explicație a sintaxei comenzi SELECT din limbajul SQL vezi secțiunea 8.1.

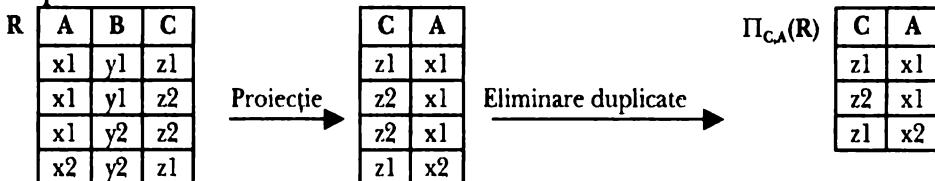
Notă: Comanda SELECT din limbajul SQL nu reprezintă același lucru și nu trebuie confundată cu operatorul SELECT din algebra relațională.

Operatorul PROJECT (proiecția)

Acesta este un operator unar care are ca parametri un atribut sau mai multe atribute ale unei relații și care elimină din relație toate celelalte atribute, producând o submulțime "verticală" a acesteia. Datorită faptului că suprimarea unor atribute poate avea ca efec-

apariția unor tupluri duplicate, acestea vor fi eliminate din relația rezultată deoarece, prin definiție, o relație nu poate conține tupluri cu valori identice. Notațiile folosite de obicei pentru acest operator sunt $\Pi_X(R)$ și $\text{PROJECT}(R, X)$, unde R reprezintă relația, iar X este atributul sau mulțimea de atribute care constituie parametrii proiecției.

Exemplu



În SQL, proiecția fără dubluri se obține folosind comanda SELECT cu specificația DISTINCT:

```
SELECT DISTINCT C, A
FROM R;
```

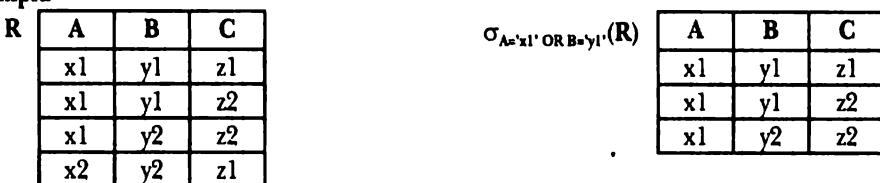
În cazul folosirii comenzi SELECT fără clauza DISTINCT se va obține proiecția cu dubluri:

```
SELECT C, A
FROM R;
```

Operatorul SELECT (selecția)

Acesta este un operator unar care este utilizat pentru extragerea tuturor tuplurilor dintr-o relație care satisfac o condiție specificată, producând astfel o submulțime "pe orizontală" a relației. Condiția este o formulă logică ce poate conține nume de atribute, constante, operatori logici (AND, NOT, OR), operatori de comparație (<, =, >, <=, >=, !=). Notațiile folosite de obicei pentru acest operator sunt $\sigma_C(R)$ sau $\text{SELECT}(R, C)$, unde R reprezintă relația, iar C este condiția care trebuie satisfăcută de tuplurile selectate.

Exemplu



În SQL, selecția se obține folosind comanda SELECT cu clauza WHERE:

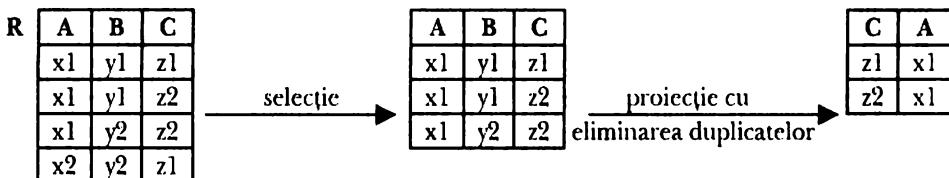
```
SELECT *
FROM R
WHERE A = 'x1' OR B = 'y1';
```

Notă: Spre deosebire de alte limbi de programare, cum ar fi Pascal, limbajul SQL acordă operatorilor de comparație o prioritate mai mare decât operatorilor logici, reducându-se astfel numărul de paranteze necesare pentru expresii complexe.

8 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

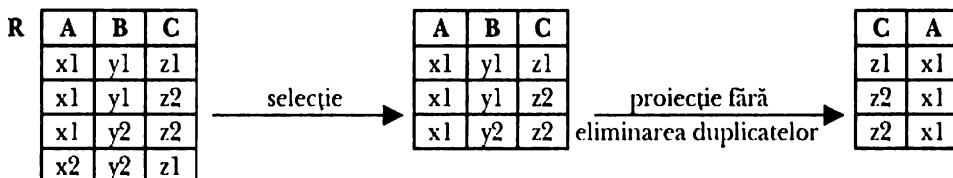
Combinarea selecției cu proiecția fără dubluri se face în modul următor:

```
SELECT DISTINCT C, A
FROM R
WHERE A = 'x1' OR B = 'y1';
```



Combinarea selecției cu proiecția cu dubluri se face în modul următor:

```
SELECT C, A
FROM R
WHERE A = 'x1' OR B = 'y1';
```



PRODUCT (produsul cartezian)

Acesta este un operator binar. Produsul cartezian a două relații R și S este mulțimea tuturor tuplurilor care se obțin prin concatenarea unui tuplu din R cu un tuplu din S. Prin urmare, dacă aritatea relației R este m , iar aritatea relației S este n , atunci produsul cartezian dintre R și S va avea aritatea $m + n$. Notațiile folosite de obicei pentru acest operator sunt $R \times S$, PRODUCT(R, S), TIMES(R, S).

Exemplu

R	A	B	C
x1	y1	z1	
x2	y1	z2	
x3	y2	z1	

S	D	E
z1	u1	
z2	u2	

$R \times S$	A	B	C	D	E
x1	y1	z1	z1	u1	
x1	y1	z1	z2	u2	
x2	y1	z2	z1	u1	
x2	y1	z2	z2	u2	
x3	y2	z1	z1	u1	
x3	y2	z1	z2	u2	

Produsul cartezian poate fi exprimat în SQL printr-o comandă SELECT pe mai multe tabele fără clauza WHERE:

```
SELECT *
FROM R, S;
```

Compatibilitate la reuniune

Două relații R și S se numesc *compatibile la reuniune* dacă ele conțin același număr de atribut (au aceeași aritate) și atributurile cu același număr de ordine din fiecare relație au același domeniu din care pot lua valori. Operatorii UNION, INTERSECT, DIFFERENCE, prezentați în continuare, sunt operatori binari ce nu pot fi aplicati decât asupra relațiilor compatibile la reuniune.

UNION (reuniunea)

Reuniunea a două relații R și S este mulțimea tuplurilor aparținând fie lui R, fie lui S. Reuniunea celor două mulțimi va cuprinde fiecare tuplu o singură dată, chiar dacă el face parte din amândouă mulțimile. Reuniunea se poate aplica doar relațiilor compatibile la reuniune. Notațiile folosite de obicei pentru acest operator sunt $R \cup S$ sau $\text{UNION}(R, S)$. Reuniunea este o operație binară comutativă, adică $R \cup S = S \cup R$.

Exemplu

R	A	B
x1	y1	
x2	y1	
x3	y1	

S	C	D
x1	y1	
x1	y2	

$R \cup S$	A	B
x1	y1	
x2	y1	
x3	y1	
x1	y2	

În SQL reuniunea se poate exprima folosind operatorul UNION:

```
SELECT A, B
FROM R
UNION
SELECT C, D
FROM S;
```

DIFFERENCE (diferența)

Diferența a două relații R și S este mulțimea tuplurilor care aparțin lui R, dar nu aparțin lui S. Diferența este o operație binară ne-comutativă, adică $R - S \neq S - R$, care se poate aplica doar relațiilor compatibile la reuniune. Notațiile folosite de obicei pentru acest operator sunt $R - S$, DIFFERENCE(R, S), MINUS(R, S).

Exemplu

R	A	B
x1	y1	
x2	y1	
x3	y1	

S	C	D
x1	y1	
x1	y2	

$R - S$	A	B
x2	y1	
x3	y1	

În SQL diferența se poate exprima folosind operatorul MINUS:

```
SELECT A, B
FROM R
MINUS
```

10 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

```
SELECT C, D  
FROM S;
```

În plus, diferența poate fi similită și prin operatorul NOT EXISTS. De exemplu, comanda SQL de mai sus este echivalentă cu următoarea:

```
SELECT A, B  
FROM R  
WHERE NOT EXISTS  
(SELECT *  
FROM S  
WHERE R.A = S.C AND R.B = S.D);
```

Notă: Pentru simplitate, în comanda SQL de mai sus am presupus că nici un atribut din relațiile R sau S nu poate avea valoarea Null. Dacă, de exemplu, atributul A din relația S și atributul C din relația S pot avea valoarea Null, atunci expresia $R.A = S.C$ trebuie înlocuită cu $R.A = S.C \text{ OR } (R.A \text{ IS NULL AND } S.C \text{ IS NULL})$. Acest lucru se datorează faptului că în SQL, expresia $\text{NULL} = \text{NULL}$ are valoarea False.

INTERSECT (intersecția)

Intersecția a două relații R și S este mulțimea tuplurilor care aparțin atât lui R cât și lui S. Intersecția este o operație binară comutativă care se poate aplica doar relațiilor compatibile la reuniune. Notațiile folosite de obicei pentru acest operator sunt $R \cap S$, $\text{INTERSECT}(R, S)$, $\text{AND}(R, S)$. Intersecția este un operator derivat, putând fi exprimat cu ajutorul reuniunii și diferenței: $R \cap S = R - (R - S)$ sau $R \cap S = S - (S - R)$

Exemplu

R	A	B
x1	y1	
x2	y1	
x3	y1	

S	C	D
x1	y1	
x1	y2	

$R \cap S$	A	B
x1	y1	

În SQL diferența se poate exprima folosind operatorul INTERSECT:

```
SELECT A, B  
FROM R  
INTERSECT  
SELECT C, D  
FROM S;
```

În plus, intersecția poate fi similită și prin operatorul EXISTS. De exemplu, comanda SQL de mai sus este echivalentă cu următoarea:

```
SELECT A, B  
FROM R  
WHERE EXISTS  
(SELECT *  
FROM S  
WHERE R.A = S.C AND R.B = S.D);
```

În cazul când operatorii UNION, DIFFERENCE sau INTERSECT se aplică unor relații care sunt obținute prin selecție din aceeași relație, atunci aceștia pot fi simulați prin aplicarea operatorilor logici corespunzători (OR, AND NOT, AND) asupra condițiilor de selecție. De exemplu, următoarele comenzi sunt echivalente:

```

SELECT A, B
FROM R
WHERE A = 'x1'
MINUS
SELECT A, B
FROM R
WHERE B = 'y1';

SELECT A, B
FROM R
WHERE A = 'x1' AND NOT B = 'y1';

```

DIVISION (diviziunea)

Diviziunea este o operație binară care se aplică asupra a două relații R și S, astfel încât mulțimea atributelor lui R include mulțimea atributelor lui S. Dacă R este o relație cu aritatea m , iar S o relație cu aritatea n , unde $m > n$, atunci diviziunea lui R la S este mulțimea tuplurilor de dimensiune $m - n$ la care, adăugând orice tuplu din S, se obține un tuplu din R. Notațiile utilizate cel mai frecvent sunt $R \div S$, DIVISION(R, S), DIVIDE(R, S).

Exemplu

R	A	B	C	S	C	R + S	A	B
x1	y1	z1			z1		x1	y1
x1	y2	z1			z2			
x1	y1	z2						
x2	y1	z2						
x2	y2	z2						

Diviziunea este o operație derivată care se exprimă cu ajutorul diferenței, produsului cartezian și proiecției: $R \div S = R_1 - R_2$, unde $R_1 = \Pi_X(R)$, $R_2 = \Pi_X((R_1 \times S) - R)$, iar X este mulțimea atributelor lui R care nu există în S. Pentru exemplul de mai sus:

R ₁	A	B	R ₁ × S	A	B	C	(R ₁ × S) - R	A	B	C
x1	y1		x1	y1	z1		x2	y1	z1	
x1	y2		x1	y2	z1		x2	y2	z1	
x2	y1		x2	y1	z1		x1	y2	z2	
x2	y2		x2	y2	z1		x1	y2	z2	
			x1	y1	z2					
			x1	y2	z2					
			x2	y1	z2					
			x2	y2	z2					

R ₂	A	B
x2	y1	
x2	y2	
x1	y2	

R ₁ ÷ R ₂	A	B
x1	y1	

Operatorul DIVISION este legat de cuantificatorul universal (\forall) care nu există în SQL, dar care poate fi simulațat cu ajutorul cuantificatorului existențial (\exists), care există în SQL, utilizând relația: $\forall x P(x) \equiv \text{NOT } \exists x \text{ NOT } P(x)$. Pentru a ilustra exprimarea operatorului DIVISION în cele două moduri (folosind cuantificatorul universal și cuantificatorul existențial), să considerăm relațiile student_curs și curs_fundamental de mai jos:

curs_student	
cod_student	curs
S1	matematica
S1	fizica
S1	mecanica
S2	matematica
S2	informatica
S3	fizica
S4	matematica
S4	fizica

curs_fundamental	
curs	
matematica	
fizica	

curs_student ÷ curs_fundamental

curs
S1
S4

Atunci relația curs_student ÷ curs_fundamental poate fi definită prin următoarea întrebare: care sunt studenții care urmează *toate* cursurile fundamentale? Alternativ, această relație poate fi definită prin întrebarea: care sunt studenții pentru care *nu există* curs fundamental care să *nu fie* urmat de către aceștia? Utilizând a două formulare, rezultă că operatorul DIVISION poate fi simulațat în SQL prin doi operatori NOT EXISTS:

```
SELECT DISTINCT cod_student
FROM curs_student cs1
WHERE NOT EXISTS
  (SELECT *
   FROM curs_fundamental cf
   WHERE NOT EXISTS
     (SELECT *
      FROM curs_student cs2
      WHERE cf.curs = cs2.curs
      AND cs1.cod_student = cs2.cod_student));
```

JOIN (componerea, joncțiunea)

Operatorul de componere permite regăsirea informației din mai multe relații corelate. Componerea este o operație binară care are ca rezultat o nouă relație în care fiecare tuplu este o combinație a unui tuplu din prima relație cu un tuplu din a doua relație.

Operatorul JOIN este un operator derivat, putând fi simulațat printr-o combinație de produs cartezian, selecție și proiecție. În general, se construiește un produs cartezian, se elimină tupluri prin selecție și se elimină atrbute prin proiecție. După modalitățile în care se face selecția și proiecția, se disting mai multe tipuri de componere: THETA-JOIN, NATURAL-JOIN, SEMI-JOIN, OUTER-JOIN. Fiecare dintre acestea vor fi prezentate în continuare.

THETA-JOIN

Operatorul THETA-JOIN combină perechile de tupluri din două relații, cu condiția ca între valorile atributelor specificate să existe o anumită legătură, adică să satisfacă o anumită condiție specificată explicit în cadrul operației. În cadrul condiției operatorului THETA-JOIN se poate folosi orice operator de comparație ($>$, \geq , $<$, \leq , \neq , $=$). În cazul în care este folosit operatorul de comparație $=$, tipul de componere se numește EQUI-JOIN. Operatorul THETA-JOIN se reprezintă de obicei cu ajutorul simbolului \bowtie sub care se scrie condiția, $R \bowtie_{condiție} S$, sau prin $JOIN(R, S, condiție)$.

Exemplu

R	A	B	C
x1	y1	1	
x2	y2	3	
x3	y3	5	

S	D	E
2	z1	
4	z2	

.

R $\bowtie_{C < D}$ S	A	B	C	D	E
x1	y1	1	2	z1	
x1	y1	1	4	z2	
x2	y2	3	4	z2	

Următorul exemplu ilustrează realizarea operatorului THETA-JOIN în SQL:

```
SELECT *
FROM R, S
WHERE R.C < S.D;
```

NATURAL-JOIN (Componerea naturală)

Componerea naturală este o operație binară comutativă care combină tupluri din două relații, R și S, cu condiția ca atrbutele comune să aibă valori identice. În cazul componerii naturale atrbutele specificate trebuie să aibă același nume. Practic diferența dintre NATURAL-JOIN și EQUI-JOIN constă în faptul că în primul caz numele atrbutelor sunt identice iar în cel de al doilea caz acestea sunt diferite. De obicei, componerea naturală se notează prin $R \bowtie S$ sau $JOIN(R, S)$.

Pentru două relații R și S, componerea naturală pe un set de atrbute comune X constă în efectuarea succesivă a următoarelor operații:

- Se calculează produsul cartezian $R \times S$.

14 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

- Se selectează din $R \times S$ acele tupluri obținute pentru care valorile atributelor X din tuplul R sunt identice cu valorile atributelor X din tuplul S.
- Deoarece în relația astfel obținută attributele X apar de două ori (o dată provenind din R și o dată din S), se elimină una dintre aparițiile acestor attribute.

Exemplu

lucrător

nr_lucrător	cod_sectie	nr_atelier
1	S1	1
2	S1	2
3	S2	1
4	S1	2
5	S1	1

atelier

cod_sectie	nr_atelier	denumire
S1	1	Proiectare
S1	2	Informatica
S2	1	Mecanica
S2	2	Electrotehnica

lucrător \bowtie atelier

nr_lucrător	cod_sectie	nr_atelier	denumire
1	S1	1	Proiectare
2	S1	2	Informatica
3	S2	1	Mecanica
4	S1	2	Informatica
5	S1	1	Proiectare

În exemplul de mai sus, {cod_sectie, nr_atelier} este cheie primară în tabelul atelier și cheie străină în tabelul lucrător.

Următorul exemplu ilustrează realizarea compunerii naturale în SQL:

```
SELECT lucrator.nr_lucrator, lucrator.cod_sectie,
       lucrator.nr_atelier, atelier.denumire
  FROM lucrator, atelier
 WHERE lucrator.cod_sectie = atelier.cod_sectie
   AND lucrator.nr_atelier = atelier.nr_atelier;
```

SEMI-JOIN (semi-compunerea)

Operația de semi-compunere aplicată asupra a două relații R și S generează o relație care conține toate tuplurile din R care sunt corelate măcar cu un tuplu din S. Operația nu este comutativă și se notează de obicei prin SEMI-JOIN(R, S).

Exemplu

R	A	B	C
x1	y1	z1	
x2	y1	z1	
x3	y2	z1	
x4	y2	z2	

S	B	C	D
y1	z1	u1	
y2	z2	u2	
y2	z2	u3	

SEMIJOIN(R, S)

A	B	C
x1	y1	z1
x2	y1	z1
x4	y2	z2

Următorul exemplu ilustrează realizarea compunerii naturale în SQL:

```
SELECT DISTINCT R.A, R.B, R.C
FROM R, S
WHERE R.B = S.B
AND R.C = S.C
```

OUTER-JOIN (componere externă)

Operația de componere externă este o extindere a compunerii naturale. În cazul aplicării operatorului NATURAL-JOIN se pot pierde tupluri atunci când există un tuplu într-o din relații care nu este corelat cu nici un tuplu din cealaltă relație. Operatorul OUTER-JOIN elimină acest inconvenient. Practic, la aplicarea operatorului OUTER-JOIN, se realizează compunerea naturală a celor două relații, la care se adaugă tuplurile din S care nu sunt conținute în compunere, completate cu valori Null pentru atributele rămase din R. Operatorul se notează cu OUTERJOIN(R, S). Există și alte variante ale acestui operator, de exemplu o altă variantă adaugă la tuplurile obținute din compunerea naturală a lui R și S tută tuplurile din R care nu sunt conținute în compunere cât și tuplurile din S care nu sunt conținute în compunere, completând restul cu Null. În mod evident, această variantă a operatorului se poate obține cu ușurință din varianta prezentată de noi. Operatorul OUTER-JOIN, în varianta prezentată de noi, este ne-comutativ.

Exemplu:

student

nr_stud	nume	prenume	cod_facult
1	Popescu	Ion	F1
2	Ionescu	Vasile	F1
3	Ionescu	Viorel	F2
4	Costache	Ion	F2
5	Matache	Mihai	F1

facultate

cod_facult	nume_facult	localitate
F1	Matematica	București
F2	Fizica	București
F3	Informatica	Pitești
F4	Mecanica	Ploiești

OUTERJOIN (student, facultate)

nr_stud	nume	prenume	cod_facult	nume_facult	localitate
1	Popescu	Ion	F1	Matematica	București
2	Ionescu	Vasile	F1	Matematica	București
3	Ionescu	Viorel	F2	Fizica	București
4	Costache	Ion	F2	Fizica	București
5	Matache	Mihai	F1	Matematica	București
Null	Null	Null	F3	Informatica	Ploiești
Null	Null	Null	F4	Mecanica	Ploiești

În versiunea SQL folosită de Oracle, operatorul OUTER JOIN este specificat prin sufixul (+) adăugat la câmpul după care se face compunerea corespunzător tuplului ale căruiatribute pot fi completate cu Null:

```
SELECT student.nr_stud, student.nume, student.prenume,
       student.cod_facult, facultate.nume_facult,
       facultate.localitate
```

```
FROM student, facultate
WHERE student.cod_facult (+) = facultate.cod_facult;
```

1.4. Tabele, rânduri, coloane

Modelul relațional este bazat pe matematica relațională și acest lucru se reflectă în terminologia pe care am prezentat-o până acum. Pe de altă parte, folosirea acestui model în economie, unde concepțele matematice sunt foarte puțin sau deloc utilizate și înțelese, a necesitat o nouă terminologie. Așa că, odată cu preluarea modelului relațional de către economie are loc o transformare a terminologiei relaționale într-ună care poate fi ușor înțeleasă de cei fără o pregătire specială în domeniul. Pentru cei care nu sunt experți în procesarea datelor, relațiile devin *tabele*, tuplurile devin *rânduri* și atributele devin *coloane*. Aceasta este și terminologia pe care o vom folosi în continuare în această carte.

1.5. Sisteme de Gestiune a Bazelor de Date Relaționale (SGBDR).

În principiu, un *sistem de gestiune a bazelor de date relaționale (SGBDR)* este un SGBD care utilizează drept concepție de organizare a datelor modelul relațional. În mod evident însă, această definiție este mult prea generală pentru a putea fi folosită în practică, deoarece modul de implementare a modelului relațional diferă de la un producător la altul. În 1985, Codd a publicat un set de 13 reguli în raport cu care un SGBD poate fi apreciat ca relațional. Trebuie remarcat că nici un SGBD comercializat în prezent nu satisface în totalitate regulile lui Codd, dar aceasta nu împiedică etichetarea acestora drept relaționale. Cu alte cuvinte, regulile lui Codd nu trebuie folosite pentru a aprecia dacă un sistem este sau nu relațional, ci măsura în care acesta este relațional, adică numărul regulilor lui Codd respectate de către acesta.

1.5.1. Regulile lui Codd

Regula 1. Regula reprezentării logice a datelor:

Într-o bază de date relațională, toate datele sunt reprezentate la nivel logic într-un singur mod, și anume sub formă de valori atomice în tabele.

Deci toate datele trebuie memorate sub formă de tabele, iar valoarea corespunzătoare intersecției dintre un rând și o coloană trebuie să fie atomică, adică să nu mai poată fi descompusă din punct de vedere logic. Așa cum am discutat mai înainte, un exemplu de încâlcare a acestei reguli este stocarea ca o singură coloană a unui cod al automobilului obținut prin concatenarea mai multor coduri, reprezentând marca automobilului, culoarea fabrica unde este produs, seria și numărul de fabricație, etc. Uneori această regulă este încâlcată în practică, dar acest lucru este de cele mai multe ori semnul unui design de calitate slabă, creând probleme de integritate a bazei de date.

Această regulă este cea mai importantă dintre cele definite de Codd, iar un SGBD care nu respectă această regulă nu poate fi în nici un caz considerat relațional.

Regula 2. Regula accesului la date:

Toate datele individuale din tabele trebuie să fie accesibile prin furnizarea numelui tabelului, numelui coloanei și valorii cheii primare.

Conform modelului relațional, într-un tabel nu pot exista rânduri identice, iar fiecare rând poate fi identificat prin valoarea cheii primare. În consecință, orice dată individuală poate fi identificată folosind numele tabelului, al coloanei și valoarea cheii primare. Oracle nu respectă această regulă deoarece permite existența mai multor rânduri identice în același tabel. Totuși, acest lucru poate fi evitat, de exemplu prin definirea unei chei primare, care elimină implicit și posibilitatea existenței rândurilor identice. Pe de altă parte, această regulă este încălcată în Oracle și de existența identificatorului de rând, ROWID, care poate fi folosit pentru accesarea rândului respectiv.

Regula 3. Regula reprezentării valorilor necunoscute:

Un sistem relațional trebuie să permită declararea și manipularea sistematică a valorilor Null, cu semnificația unor valori necunoscute sau inaplicabile.

Această regulă, implică, de exemplu, că un SGBDR trebuie să facă diferență între valoarea numerică 0 și Null sau între șirul de caractere "spațiu" și valoarea Null. Valoarea Null trebuie să reprezinte absența informației respective și are un rol important în implementarea restricțiilor de integritate structurală (integritatea entității și integritatea referirii). Oracle respectă această regulă, limbajul SQL permisând declararea și manipularea valorilor Null.

Regula 4. Regula dicționarului de date:

Descrierea bazei de date (dicționarul de date) trebuie să fie reprezentată la nivel logic tot sub formă de tabele, astfel încât asupra acesteia să se poată aplica aceleași operații ca și asupra datelor propriu-zise.

Cu alte cuvinte, dicționarul de date trebuie să fie organizat la nivel logic și accesat la fel ca orice tabel din baza de date. Această regulă este respectată de către Oracle, dicționarul de date constând din tabele și tabele virtuale (vederi) care pot fi interogate la fel ca oricare alte tabele sau vederi, folosind comanda SELECT.

Regula 5. Regula limbajului de acces:

Într-un sistem relațional trebuie să existe cel puțin un limbaj de accesare a datelor, care să asigure următoarele operații: definirea tabelelor de bază și a tabelelor virtuale (vederilor), manipularea și interogarea datelor (atât interactiv cât și prin program), definirea restricțiilor de integritate, autorizarea accesului la date, delimitarea tranzacțiilor.

Limbajul SQL folosit de către Oracle permite definirea tabelelor (comenzile CREATE TABLE, ALTER TABLE, DROP TABLE), a vederilor (comenzile CREATE VIEW, ALTER VIEW, DROP VIEW), manipularea (comenzile INSERT, UPDATE, DELETE) și interogarea baza de date (comanda SELECT), definirea restricțiilor de integritate (clauza CONSTRAINT folosită la definirea tabelelor), autorizarea accesului la date (printr-un set de privilegii de sistem și la nivel de obiect), delimitarea tranzacțiilor (operațiile COMMIT și ROLLBACK).

Regula 6. Regula de actualizare a tabelelor virtuale (vederilor).

Un SGBD trebuie să poată determina dacă o vedere poate să fie actualizată sau nu.

Un *tabel virtual (vedere)* este un tabel logic, în sensul că el organizează datele sub forma unor rânduri și coloane, ca orice alt tabel, dar în schimb el nu stochează datele, fiind construit pe baza unor interogări asupra uneia sau mai multor tabele de bază. De exemplu să considerăm tabelul :

```
salariu(cod_salariat,salariu_brut, zile_totale, zile_lucrare).
```

Pe baza acestui tabel se poate defini vederea

```
salariu_r(cod_salariat,salariu_brut,salariu_realizat)
```

unde *salariu_realizat* este definit ca

```
salariu_realizat=salariu_brut*zile_totale/zile_lucrare
```

Să presupunem acum că se dorește actualizarea coloanei *salariu_brut* din vedere. Acest lucru este posibil, datorită faptului că actualizarea se propagă înapoi la coloana din tabelul de bază, producându-se și actualizarea acesteia. Pe de altă parte, nu este posibilă actualizarea coloanei *salariu_realizat*, datorită faptului că schimbarea valorii acesteia să ar putea produce datorită schimbării valorilor mai multor coloane (*salariu_brut*, *zile_totale* sau *zile_lucrare*), SGBD-ul neștiind care din aceste coloane trebuie actualizată în tabelul de bază. Oracle respectă această regulă, existând un set de reguli care determină dacă o coloană a unei vederi poate sau nu să fie actualizată.

Regula 7. Regula manipulării datelor:

Un sistem relațional trebuie să ofere posibilitatea procesării tabelelor (de bază sau virtuale) nu numai în operațiile de interogare a datelor cât și în cele de inserare, actualizare și stergere.

Aceasta înseamnă că operațiile de manipulare a datelor (inserare, actualizare și stergere) trebuie să se poată efectua asupra oricărei mulțimi de rânduri dintr-un tabel, pornind de la întregul tabel și terminând cu un singur rând sau cu nici unul. Deci, un SGBD relațional nu obligă utilizatorul să caute într-un tabel rând cu rând pentru a regăsi, modifica sau șterge informații dorită, deoarece operațiile prin care se manipulează conținutul bazei de date lucrează la nivel de mulțime de rânduri. Limbajul SQL asigură această facilitate prin instrucțiunile: INSERT și subinterrogare, UPDATE și DELETE (vezi secțiunile 8.2, 8.3 și 8.4).

Regula 8. Regula independenței fizice a datelor:

Programele de aplicație nu trebuie să depindă de modul de stocare și accesare fizică a datelor.

Deci un SGBD relațional trebuie să separe complet aspectele de ordin fizic ale datelor (modul de stocare și modul de acces la date) de cele de ordin logic. Cu alte cuvinte, programele de aplicație nu trebuie să depindă de stocarea fizică a datelor. De exemplu, dacă un fișier care conține un tabel de date este mutat pe o altă unitate de disc sau îl este schimbat numele, acesta nu trebuie să aibă vreun efect asupra aplicațiilor care folosesc acel tabel, utilizatorilor fiindu-și transparentă această schimbare. În general, Oracle respectă această regulă, deși stocarea fizică a datelor trebuie luată în considerație la proiectarea bazei de date.

Regula 9. Regula independenței logice a datelor:

Programele de aplicație nu trebuie să fie afectate de nici o restructurare logică a tabelelor bazei de date care conservă datele.

Deci orice modificare efectuată asupra unui tabel care conservă datele din acesta (de exemplu, dacă un tabel trebuie divizat în două, din rațiuni de creștere a performanțelor) nu trebuie să afecteze funcționarea programelor de aplicație. Această regulă este respectată de către Oracle prin posibilitatea definirii vederilor: dacă un tabel este divizat în două, atunci se poate crea o vedere care alătură cele două tabele, astfel încât această împărțire nu va avea nici un efect asupra aplicației.

Regula 10. Regula independenței datelor din punctul de vedere al integrității:

Regulile de integritate a bazei de date trebuie să fie definite în limbajul utilizat de sistem pentru definirea datelor și nu în cadrul aplicațiilor individuale; în plus, aceste reguli de integritate trebuie stocate în dicționarul de date.

Cu alte cuvinte, restricțiile de integritate trebuie impuse la definirea tabelelor bazei de date și nu în cadrul aplicațiilor care folosesc aceste tabele. În general, Oracle respectă această regulă, la definirea tabelelor (în cadrul comenzi CREATE TABLE) putându-se defini atât restricțiile de integritate structurală (NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY) cât și unele restricții de comportament (CHECK). Informații despre aceste restricții sunt stocate în dicționarul bazei de date.

Regula 11. Regula independenței datelor din punctul de vedere al distribuirii:

Programele de aplicație nu trebuie să fie afectate de distribuirea pe mai multe calculatoare a bazei de date.

Cu alte cuvinte, baza de date trebuie să funcționeze corect indiferent dacă se găsește pe un singur calculator sau este distribuită în mai multe noduri ale unei rețele. Această regulă este o extensie a regulii 8, privind independența programelor de aplicație față de modul de stocare fizică a datelor. Această regulă este în general respectată de Oracle, existând totuși restricții privind accesarea unor obiecte aflate în alt nod al rețelei. În plus, în Oracle există posibilitatea replicării locale a tabelelor aflate în alte noduri ale rețelei, evitându-se astfel transmiterea în mod repetat a informațiilor prin rețea.

Regula 12. Regula privind prelucrarea datelor de către un limbaj de nivel inferior.

Orice limbaj nerelațional folosit pentru accesarea datelor trebuie să respecte aceleași condiții de integritate ca și limbajul relațional de acces.

De exemplu, dacă sistemul posedă un limbaj procedural, prin care se accesează datele la nivel de rând și nu, potrivit sistemului relațional, la nivelul mulțimilor de rânduri, în acest limbaj nu se pot evita restricțiile de integritate (integritatea entității, integritatea referențială, restricții de comportament) pe care trebuie să le respecte limbajul relațional de acces la date. Această regulă este respectată de către Oracle prin faptul că singurul limbaj de accesare a datelor este SQL, care este un limbaj relațional.

Dacă un SGBD îndeplinește principiile sistemului relațional (folosește ca structuri de date tabele conforme cu modelul relațional, asigură cele două reguli de integritate structurală și operațiile relaționale) și respectă aceste 12 reguli, atunci el poate fi numit relațional. Codd rezumă aceste lucruri prin regula zero:

20 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Regula 0. Regula de bază:

Un SGBD Relațional trebuie să fie capabil să gestioneze baza de date exclusiv pe baza caracteristicilor sale relaționale.

Aceasta înseamnă că sistemul trebuie să-și îndeplinească toate funcțiile prin manipulări în care unitatea de procesare să fie tabelul (multimea de rânduri), asupra căruia să se efectueze operațiile specifice modelului relațional. Trebuie spus că regula 0 nu este respectată în totalitate de nici un SGBD existent pe piață, inclusiv Oracle, implementarea acestora folosind atât caracteristici relaționale cât și nerelaționale.

Se obișnuiește ca, în conformitate cu tipul de cerințe pe care le exprimă, regulile să fie grupate în cinci categorii, și anume:

1. reguli de bază: Regula 0 și Regula 12;
2. reguli structurale: Regula 1 și Regula 6;
3. reguli privind integritatea datelor: Regula 3 și Regula 10;
4. reguli privind manipularea datelor: Regula 2, Regula 4, Regula 5 și Regula 7;
5. reguli privind independența datelor: Regula 8, Regula 9 și Regula 11.

Trebuie spus că nici unul dintre SGBD-urile existente în prezent nu satisface în totalitate toate cele 13 reguli ale lui Codd. De aceea, în practică nu sunt utilizate regulile lui Codd, fiind formulate în schimb un set de cerințe minime pe care trebuie să le satisfacă un sistem SGBD pentru a putea fi considerat relațional.

Un SGBD este denumit *minimal relațional*, dacă satisface următoarele condiții:

1. Toate datele din cadrul bazei de date sunt reprezentate prin valori în tabele.
2. Nu există pointeri observabili de către utilizator între tabele.
3. Sistemul asigură operatorii relaționali de proiecție, selecție și compunere naturală, fără limitări impuse de considerente interne.

Un SGBD este denumit *complet relațional* dacă este minimal relațional și satisface în plus următoarele condiții:

4. Sistemul asigură toate operațiile de bază ale algebrei relaționale, fără limitări impuse de considerente interne.
5. Sistemul asigură restricțiile de integritate de bază ale modelului relațional (integritatea entității și integritatea referențială)

SGDB-ul Oracle este complet relațional și chiar se apropie destul de mult de un SGBD relational ideal, definit prin regulile lui Codd.

Capitolul 2

Proiectarea bazelor de date relationale

După mai bine de două decenii de folosire a modelului relațional, proiectarea (designul) bazelor de date rămâne încă mai degrabă artă decât știință. Au fost sugerate un număr de metode, dar până în prezent nici una nu este dominantă. Pe de altă parte proiectarea bazelor de date trebuie să fie bazată pe considerații practice care stau la baza oricărei activități de procesare a datelor. Pentru a crea un design adecvat este necesară o cunoaștere aprofundată a funcționării antreprizei, a modului în care aceasta folosește datele și a sistemului de management al bazelor de date folosit.

Metodele curente de proiectare a bazelor de date sunt în general divizate în trei etape separate: crearea schemei conceptuale, crearea design-ului logic al bazei de date și crearea design-ului fizic al bazei de date.

1. **Crearea schemei conceptuale.** Aceasta este un design de nivel înalt (inclusiv relațiile dintre datele întregului sistem), care descrie datele și relațiile necesare pentru execuția operațiilor necesare, fiind independent de orice model de baze de date. Designul de la acest nivel este foarte general, se realizează într-o perioadă scurtă de timp și prezintă modul în care grupările de date sunt integrate în sistemul de ansamblu.
2. **Crearea design-ului logic al bazei de date.** În această fază schema conceptuală este transformată în structuri specifice unui anumit sistem de gestiune a bazei de date. La acest nivel designul este rafinat, sunt definite elemente de date specifice care sunt grupate în înregistrări. În cazul modelului relational, la sfârșitul acestei etape vom avea un număr de tabele care vor permite stocarea și manipularea *corectă* a tuturor datelor necesare sistemului.
3. **Crearea design-ului fizic al bazei de date.** În această etapă designul logic este transformat într-o structură fizică eficientă.

Primele două etape vor fi prezentate pe larg în acest capitol. Cel de-al treilea va fi abordat în capitolul 6, când va fi prezentată organizarea logică a bazei de date Oracle.

2.1. Crearea schemei conceptuale

Procesul de design al schemei conceptuale începe prin determinarea datelor necesare activităților din antrepriză. Este creată o echipă de design a schemei conceptuale care se ocupă cu determinarea datelor necesare, eventual prin folosirea de interviuri cu managerii antreprizei. După ce echipa proiectează datele, ea le revizuește și le organizează.

2.1.1. Modelul entitate-legătură (entitate-relație)

Una dintre tehniciile folosite pentru organizarea rezultatelor din etapa de colectare a datelor este modelul *entitate-legătură*, care împarte elementele unui sistem real în două categorii și

anume entități și legături (relații) între aceste entități. Principalele concepte folosite în acest model sunt cele de *entitate*, *relație* (legătură) și *atribut*.

Notă: Nu trebuie confundat conceptul de relație în sensul de legătură sau asociere, care intervine în modelul entitate-legătură cu cel definit în capitolul 1.

Entitate

O *entitate* este un obiect de interes pentru care trebuie să existe date înregistrate. O entitate poate fi atât un obiect tangibil – precum persoane, locuri sau lucruri - cât și abstracte – precum comenzi, conturi bancare, etc. De exemplu, să considerăm o universitate formată din mai multe facultăți; în fiecare facultate studiază mai mulți studenți și predau mai mulți profesori. Fiecare student urmează mai multe cursuri, după cum un profesor poate predă mai multe cursuri. În plus, un curs poate fi predat de mai mulți profesori (de exemplu la grupe/serii diferite). Elementele semnificative ale acestui sistem sunt: facultate, student, profesor și curs; acestea sunt entitățile sistemului. Ele sunt reprezentate în figura 2.1 împreună cu relațiile dintre ele. Remarcați că entitățile se reprezintă prin dreptunghiuri, iar relațiile dintre ele prin arce neorientate.

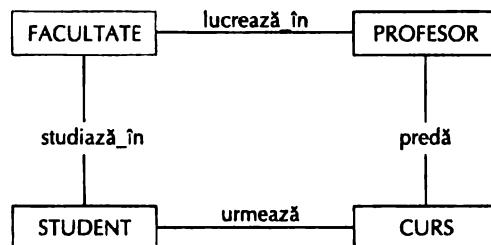


Figura 2.1

Ideile de bază pentru identificarea și reprezentarea entităților sunt următoarele:

- Fiecare entitate este denumită în mod unic; nu pot exista două entități cu același nume. Entitățile sunt reprezentate întotdeauna prin substantive, dar nu orice substantiv folosit în descrierea sistemului este o entitate a acestuia. Entitățile sistemului sunt doar acele substantive care au o semnificație deosebită în descrierea sistemului. De exemplu, chiar dacă suntem interesați de numărul de ore de predare efectuate de un profesor pe săptămână, aceasta nu înseamnă că vom crea o entitate pentru aceasta. De fapt, vom vedea în continuare că numărul de ore predate va fi un atribut al entității PROFESOR.
- De asemenea, pentru fiecare entitate trebuie să se dea o descriere detaliată; de exemplu putem spune că un PROFESOR este un cadru didactic angajat al universității pe o perioadă nedeterminată, din această categorie făcând parte atât profesorii permanenți cât și cei asociați, dar fiind excluși cei care predau la universitate numai o perioadă limitată.

Relație (legătură)

Entitățile pot forma relații între ele. O relație este o asociere nedirecționată între două entități. Ea exprimă un raport care există între entitățile respective. De exemplu "lucrează_în" este o relație între entitățile PROFESOR și FACULTATE, iar "predă" este o relație între entitățile PROFESOR și CURS.

Principalele idei pentru identificarea și reprezentarea relațiilor sunt următoarele:

- Relațiile sunt reprezentate prin verbe, dar nu orice verb este o relație.
- Între două entități poate exista mai mult decât o singură relație. De exemplu, dacă luăm în vedere că fiecare facultate este condusă de un decan și că acesta este ales din rândurile profesorilor, atunci între entitățile PROFESOR și FACULTATE va mai exista o relație numită "conduce".
- Pot exista relații cu același nume, dar relațiile care asociază aceleași entități trebuie să poarte nume diferite.

Cardinalitatea unei relații indică numărul maxim de instanțe din fiecare entitate care poate participa la relație. Cu alte cuvinte, cardinalitatea unei relații reprezintă răspunsul la întrebări de genul: Câți studenți pot studia la o facultate? Mulți. Dar la câte facultăți poate studia un student? La cel mult una. Deci cardinalitatea relației "studiază_la" este de mulți-la-unu. *Cardinalitatea unei relații* poate fi de trei feluri: mulți-la-unu, unu-la-unu sau mulți-la-mulți.

- mulți-la-unu (many-to-one, N:1): Relația dintre entitățile A și B este de tipul mulți-la-unu dacă fiecarei instanțe din A îi poate fi asociată cel mult o singură instanță din B și fiecarei instanțe din B îi poate fi asociate mai multe instanțe din A. De exemplu, relațiile "lucrează_in" dintre PROFESOR și FACULTATE și "studiază_la" dintre STUDENT și FACULTATE sunt de tipul N:1. O relație mulți-la-unu se reprezintă în modul următor:



Figura 2.2

- unu-la-unu (one-to-one, 1:1): Relația dintre entitățile A și B este de tipul unu-la-unu dacă fiecarei instanțe din A îi poate fi asociată cel mult o singură instanță din B și fiecarei instanțe din B îi poate fi asociată cel mult o singură instanță din A. De exemplu, relația "conduce" dintre PROFESOR și FACULTATE este o relație 1:1.

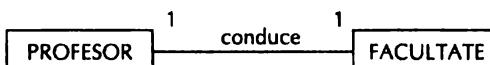


Figura 2.3

- mulți-la-mulți (many-to-many, N:M): Relația dintre entitățile A și B este de tipul mulți-la-mulți dacă fiecarei instanțe din A îi pot fi asociate mai multe instanțe din B și fiecarei instanțe din B îi pot fi asociate mai multe instanțe din A. De exemplu, relațiile "predă" dintre PROFESOR și CURS și "urmează" dintre STUDENT și CURS sunt de tipul N:M. O relație N:M se reprezintă în modul următor:

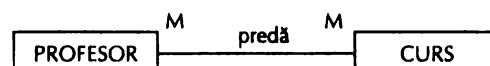


Figura 2.4

Valorile discutate până acum (N:1, 1:1, N:M) reprezintă *cardinalitatea maximă* a unei relații. Pe de altă parte, o relație este caracterizată și de o *cardinalitate minimă*, care indică obligativitatea participării entităților la relație. Cu alte cuvinte, aceasta furnizează răspunsul la întrebări de genul: Câți studenți trebuie să studieze la o facultate? Zero (de exemplu dacă facultatea este nou

înșință). Dar la câte facultăți trebuie să studieze un student? Cel puțin una. Deci cardinalitatea minimă a relației "studiază_la" dintre STUDENT și FACULTATE este de 0:1. În mod similar, relația "predă" dintre PROFESOR și CURS are cardinalitatea minimă 0:0 (un profesor trebuie să predea zero cursuri și un curs trebuie să fie predat de zero profesori – de exemplu dacă cursul este nou și nu s-a stabilit încă titularul de curs). Deci cardinalitatea minimă a unei relații poate avea valorile 0:0, 0:1 și 1:1. Dacă participarea unei entități la o relație este obligatorie (cardinalitatea minimă respectivă este 1) se mai spune și că participarea acestia la relație este *totală*. În caz contrar (cardinalitatea minimă respectivă este 0), participarea entității la relație se numește *parțială*. De exemplu participarea entității STUDENT la relația "studiază_la" este parțială, pe când participarea entității FACULTATE la aceeași relație este totală.

În cadrul reprezentării grafice, cardinalitatea maximă a unei relații se va indica fără paranteze, în timp ce cardinalitatea minimă, dacă este diferită de cea maximă, se va scrie în paranteze, vezi figurile 2.5, 2.6 și 2.7. De multe ori, cardinalitatea minimă nu este indicată în diagrama entității legătură, pe când cardinalitatea maximă trebuie indicată întotdeauna, ca fiind esențială.

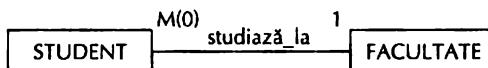


Figura 2.5

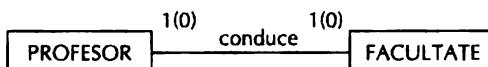


Figura 2.6

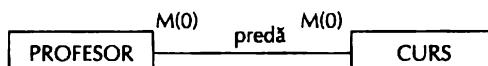


Figura 2.7

Un alt mod de a reprezenta relațiile, indicând doar cardinalitatea lor maximă este următorul:

- relații mulți-la-unu



Figura 2.8

- relații unu-la-unu

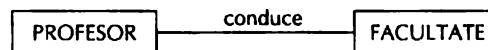


Figura 2.9

- relații mulți-la-mulți

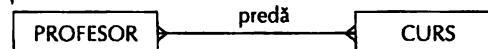


Figura 2.10

Atribut

Un atribut este o caracteristică a unei entități sau a unei relații. Fiecare entitate are un anumit număr de attribute despre care sunt înregistrate date. De exemplu, numele, prenumele, vârsta și numărul de ore predate sunt attribute ale entității PROFESOR. Fiecare

atribut poate lua o valoare care furnizează informații despre entitatea respectivă. Exemplu de valori de atribut sunt "Ionescu" pentru nume, "Mihai" pentru prenume etc. Pe de altă parte și relațiile pot avea atribut. De exemplu, relația "urmează" dintre STUDENT și CURS poate avea ca atribut nota obținută la examen și nota obținută la reședință - pentru cei care își au promovat examenul - iar relația "lucrează_în" dintre PROFESOR și FACULTATE poate avea ca atribut data angajării.

Principalele idei pentru identificarea și reprezentarea atributelor sunt următoarele:

- Numele unui atribut este unic în cadrul unei entități sau al unei relații;
- Atributele sunt întotdeauna substantive, dar nu orice substantiv este un atribut;
- Pentru fiecare atribut, trebuie furnizată o descriere, împreună cu domeniul de valori (intreg, sir de caractere, dată calendaristică, etc.);
- Alegerea atributelor trebuie făcută în aşa fel încât să se evite aşa-numitele *atribute indirekte*. Un atribut indirect al unei entități sau relații este un atribut care nu apare în mod real acelei entități sau relații, fiind o caracteristică a unui alt obiect al sistemului. De exemplu, numele facultății este un atribut indirect al entității STUDENT, el descriind de fapt o proprietate a entității FACULTATE. De accea, el va trebui redistribuit acestei entități.

Modelul entitate-legătură și modelul relațional.

Modelul entitate-legătură poate fi transformat în mod natural într-o bază de date relațională. Fără a intra deocamdată în amănuntele acestei transformări, enumărăm în continuare principalele idei ale acestei transformări:

- O entitate devine un tabel;
- Un atribut al unei entități devine o coloană a tabelului respectiv;
- O relație va fi reprezentată fie printr-un tabel special, fie printr-o cheie străină într-unul dintre cele două tabele entitate, care face referire la cheia primară a celuilalt tabel entitate.

Chei primare. Chei naturale și chei artificiale.

În concordanță cu terminologia folosită în capitolul 1, o *cheie* a unei entități va fi un atribut sau un set de attribute care identifică în mod unic o instanță a acelei entități. Cu alte cuvinte, o cheie face distincție între oricare două rânduri diferite ale tabelului provenit din entitatea respectivă. De exemplu, putem presupune că fiecare student va fi identificat în cadrul universității printr-un cod unic; atunci codul studentului este o cheie a entității STUDENT. Pe de altă parte, numele studentului nu poate fi cheie a acestei entități deoarece pot exista mai mulți studenți cu același nume. Dacă însă presupunem că nu pot exista studenți cu același nume, prenume și dată de naștere atunci combinația acestor attribute este la rândul ei cheie a entității STUDENT.

Există două tipuri de chei: *naturale* și *artificiale*. O cheie naturală este constituită dintr-un atribut sau o combinație de attribute cu semnificație reală pentru entitatea în cauză. De exemplu, combinația nume, prenume, dată de naștere este o cheie naturală a entității STUDENT. O cheie artificială este un atribut al unei entități care nu are semnificație reală pentru entitatea în cauză, fiind folosită doar pentru a face distincție între instanțele entității. De exemplu, codul studentului este o cheie artificială a entității STUDENT.

Una dintre cheile entității va fi declarată *cheie primară*. Deci, în principiu, oricare dintre cele două chei ale entității STUDENT poate fi declarată cheie primară. Pe de altă parte însă,

este preferată folosirea cheilor primare artificiale, excepție făcând cazul când cheia primară respectivă nu va fi stocată în alte tabele ca și cheie străină. Principalele avantaje ale cheilor primare artificiale față de cele naturale sunt următoarele:

- *Stabilitatea.* Valoarea unei chei artificiale rămâne aceeași pe parcursul funcționării sistemului, în timp ce valoarea unei chei naturale poate fi în general modificată, această modificare atrăgând, la rândul ei, schimbarea cheilor străine care fac referire la ea. De exemplu, numele unei studențe se poate schimba prin căsătorie; dacă se consideră combinația nume, prenume și data nașterii ca fiind cheia primară a entității STUDENT, atunci orice schimbare a numelui va impune modificarea valorilor cheilor străine corespunzătoare. Ca o regulă generală, valoarea cheii primare a unui tabel nu trebuie să poată fi modificată, aceasta creând probleme privind păstrarea integrității datelor - cu alte cuvinte schimbarea cheii primare a unui tabel va trebui însoțită de schimbarea cheilor străine care fac referire la aceasta;
- *Simplitatea.* În general, o cheie artificială este mai simplă decât una naturală. Cheile naturale sunt mai complexe, atât din punct de vedere fizic (numărul de octeți) cât și a numărului de coloane. De exemplu, este mult mai comodă stocarea codului studentului ca și cheie străină, decât a combinației dintre numele, prenumele și data nașterii;
- *Nu prezintă ambiguități.* O cheie primară trebuie să nu prezinte ambiguități, astfel încât să poată fi folosită cu ușurință de către dezvoltator sau utilizator în filtrările efectuate pe tabele. Și în această privință, o cheie naturală creează probleme. De exemplu, numele și prenumele unui student pot fi formate dintr-unul sau mai multe cuvinte care pot fi despărțite de un spațiu sau de o linie, etc;
- *Elimină valorile Null.* În cazul cheilor primare naturale, valorile Null reprezintă un problemă. De exemplu, aceasta înseamnă că un student nu poate fi înregistrat dacă nu se știe data de naștere.

În concluzie, o cheie primară trebuie să fie unică, diferită de Null, scurtă, simplă, fără ambiguități, să nu conțină informații descriptive, să fie ușor de manipulat, să fie stabilă și familiară utilizatorului. Cheile artificiale îndeplinește toate aceste condiții în afară de ultima, fiind preferate aproape întotdeauna celor naturale.

Diagrama entitate-legătură

Entitățile sistemului, împreună cu relațiile dintre ele se reprezintă prin aşa numită diagramă *entitate-legătură*, în care entitățile sunt reprezentate prin dreptunghiuri, iar relațiile dintre acestea prin arce neorientate, specificându-se și cardinalitatea acestora. Pentru fiecare entitate se specifică cheile primare și eventual atributele mai semnificative. Atributele care reprezintă chei primare trebuie să fie subliniate. Diagrama entitate-legătură a sistemului descris la începutul acestui capitol este reprezentată în figura 2.11.

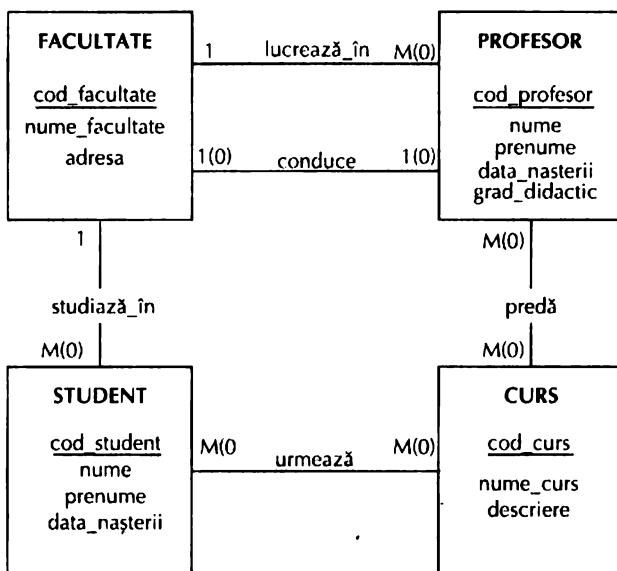


Figura 2.11

Cazuri speciale de entități, relații și attribute

În continuare vom considera câteva cazuri speciale de entități, relații și attribute, încercând în același timp o clasificare a acestora.

- **Subentitate/Superentitate.**

O subentitate este o submulțime a unei alte entități, numită superentitate. De exemplu, să presupunem că în sistemul prezentat mai sus nu vom reține date numai despre profesorii universității, ci și despre tot personalul din universitate. Atunci vom crea o superentitate PERSONAL, pentru care PROFESOR este o subentitate. O altă subentitate a acestei superentități va fi PERSONAL_ADMINISTRATIV. O subentitate se reprezintă printr-un dreptunghi inclus în dreptunghiul care reprezintă superentitatea corespunzătoare, vezi figura 2.12. Cheia primară, attributele și relațiile unei superentități sunt valabile pentru orice subentitate, reciprocă fiind evident falsă. De exemplu, cheia primară a entității PROFESOR va fi acum "cod_personal", care este cheia primară a entității PERSONAL, în timp ce unele dintre attributele subentității PROFESOR (de exemplu "nume", "prenume", "data_nasterii") se regăsesc printre attributele entității PERSONAL. Pe de altă parte însă, subentitatea PROFESOR poate avea și alte attribute decât cele specifice superentității PERSONAL, de exemplu gradul didactic. Cu alte cuvinte, attributele comune vor fi repartizate superentității, în timp ce attributele specifice vor fi repartizate subentităților.

Între o subentitate și superentitatea corespunzătoare există întotdeauna o relație 1:1, având cardinalitatea minimă 1:0.

Uneori este convenabil să se creeze superentități din entități cu mai multe attribute comune. De exemplu, din entitățile PROFESOR și PERSONAL_ADMINISTRATIV s-a creat superentitatea PERSONAL. Superentitatea astfel creată va conține attributele comune, iar attributele specifice vor fi repartizate subentităților componente. În plus, se

va crea o nouă cheie artificială pentru superentitatea nou formată. De exemplu, pentru PERSONAL s-a creat un cod personal, care a devenit cheie primară a acestei entități.

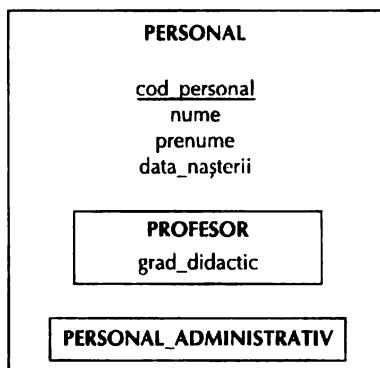


Figura 2.12

- *Entitate dependentă (detaliu)/entitate master.*

O entitate dependentă (detaliu) este o entitate care nu poate exista de sine stătătoare, ci numai atașată unei alte entități, aceasta din urmă fiind numită entitatea master a acestei legături. De exemplu, dacă presupunem că fiecare curs poate fi constituit dintr-unul sau mai multe module, atunci entitatea MODUL va fi o entitate dependentă de CURS, vezi figura 2.13. Între entitățile master și detaliu va exista întotdeauna o relație 1:N, având cardinalitatea minimă 1:0. Cheia primară a unei entități detaliu va fi formată din cheia primară a entității master plus una sau mai multe atribută ale entității detaliu. De exemplu, cheia entității MODUL poate fi aleasă ca fiind combinația dintre cod_curs și nr_modul, acesta din urmă specificând numărul de ordine al unui modul în cadrul unui curs.

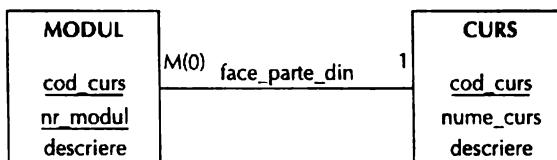


Figura 2.13

- *Relații recursive.*

Pot exista relații nu numai între două entități diferite, ci și între o entitate și ea însăși; acestea se numesc relații recursive. De exemplu, dacă presupunem că activitatea de cercetare în universitate este organizată pe o structură ierarhică, adică un profesor poate avea un șef și poate fi la rândul lui șeful mai multor profesori, atunci entitatea PROFESOR admite o relație recursivă de tipul N:1, vezi figura 2.14.

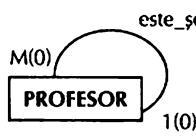


Figura 2.14

- *Relații binare (de tip 2)/ relații între mai mulți de două entități (de tip 3).*

Până acum am discutat doar despre relațiile dintre două entități, numite relații binare sau de tip 2. Pot însă exista și relații între mai multe decât două entități, pe care le vom numi relații de tip 3. De exemplu, să presupunem că fiecare student trebuie să efectueze mai multe proiecte, iar pentru fiecare proiect el poate să-și aleagă unul sau mai mulți profesori coordonatori, un profesor putând coordona același student în mai multe proiecte. Deci relația "efectuează_coordonază" este o relație de tip 3 între entitățile STUDENT, PROIECT și PROFESOR, vezi figura 2.15. O relație de tip 3 nu poate fi spartă în relații binare între entitățile componente, un exemplu este oferit în figura 2.16, unde prin spargerea relației "efectuează_coordonază" în trei relații binare prin proiecție se obțin informații eronate, relația inițială nemaiputând fi reconstituată din relațiile componente.

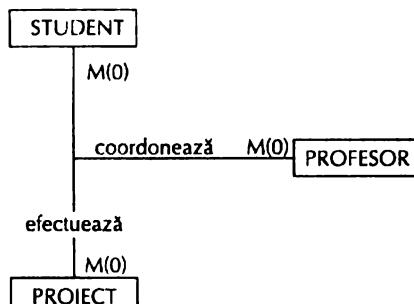


Figura 2.15

STUDENT	PROIECT	PROFESOR
s1	p1	x2
s1	p2	x1
s2	p1	x1

a) Relația de tip 3 inițială

STUDENT	PROIECT
s1	p1
s1	p2
s2	p1

STUDENT	PROFESOR
s1	x2
s1	x1
s2	x1

PROIECT	PROFESOR
p1	x2
p2	x1
p1	x1

b) Descompunerea relației de tip 3 în 3 relații binare prin proiecție

STUDENT	PROIECT	PROFESOR
s1	p1	x1
s1	p1	x2
s1	p2	x1
s2	p1	x1

c) Reconstituirea eronată a relației inițiale

Figura 2.16.

- **Atribute simple/compuse/repetitive(multivaloare)/calculate(deduse)**

Atributele pot fi de patru feluri: simple, compuse, repetitive (multivaloare) și calculate (deduse). Unui atribut simplu îi corespunde o singură valoare, atomică. De exemplu, numele și prenumele unui student sunt atrbute simple. Un atrbut compus este format din mai multe

attribute simple, numite componente sale. Valoarea unui atribut compus este reprezentată de valorile atributelor componente. Dacă presupunem, de exemplu, că o adresă se poate descompune în componentele țară, oraș, stradă, număr și cod, atunci "adresa" este un atribut compus din 5 componente. Un atribut repetitiv (multivaloare) este un atribut care poate avea mai multe valori, numărul acestora variind de la o instanță la alta. De exemplu, un studenț poate avea mai multe numere de telefon, deci acesta este un atribut repetitiv. Un atribut calculat reprezintă un atribut a cărui valoare nu este cunoscută direct, ci calculată pe baza valorilor altor attribute. De exemplu atributul "valoare" este calculat ca produs între attribute "cantitate" și "preț". Attributele calculate se folosesc foarte rar deoarece ele reprezintă de fapt redundanță a datelor.

Probleme în identificarea entităților, relațiilor și atributelor

- *Relație sau entitate?*

Uneori este greu de identificat dacă o componentă a sistemului este relație sau entitate. Dacă o entitate are o cheie provenită din combinația cheilor primare a două sau mai multe entități atunci trebuie definită o relație. Deci entitatea PREDĂ din figura 2.17 a) va avea semnificația unei relații între entitățile PROFESOR și CURS, reprezentată în figura 2.17 b).

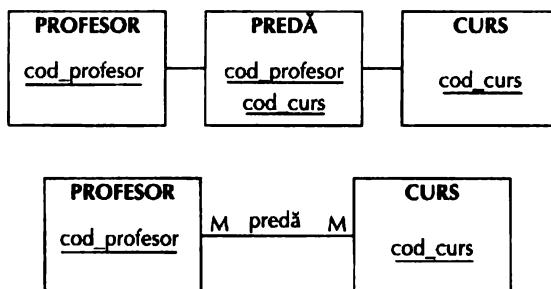


Figura 2.17

- *Relație sau atribut?*

După cum am văzut până acum, o relație poate fi reprezentată ca un atribut al unei entități, după cum attributele unei entități pot fi înlocuite cu relații. Deci, care este diferența între o relație și un atribut? Atunci când un atribut al unei entități reprezintă cheia primară a altrei entități, el exprimă de fapt o relație. Deci figura 2.18 a) reprezintă o relație între entitățile STUDENT și FACULTATE, vezi figura 2.18 b).

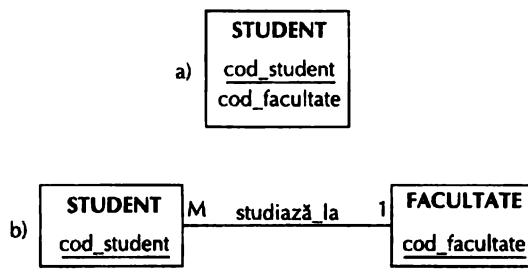


Figura 2.18

•**Etapele obținerii modelului entitate-legătură.**

Pentru realizarea modelului entitate-legătură a sistemului analizat sunt parcuse următoarele **trepte**:

- Identificarea entităților sistemului;
- Identificarea relațiilor sistemului și stabilirea cardinalității acestora;
- Identificarea atributelor entităților și relațiilor sistemului;
- Stabilirea cheilor primare ale entităților;
- Trasarea diagramei entitate-legătură.

Diagrama entitate-legătură a sistemului prezentat ca exemplu în această secțiune, înțelezând entitățile și relațiile menționate mai sus, este prezentată în figura 2.19.

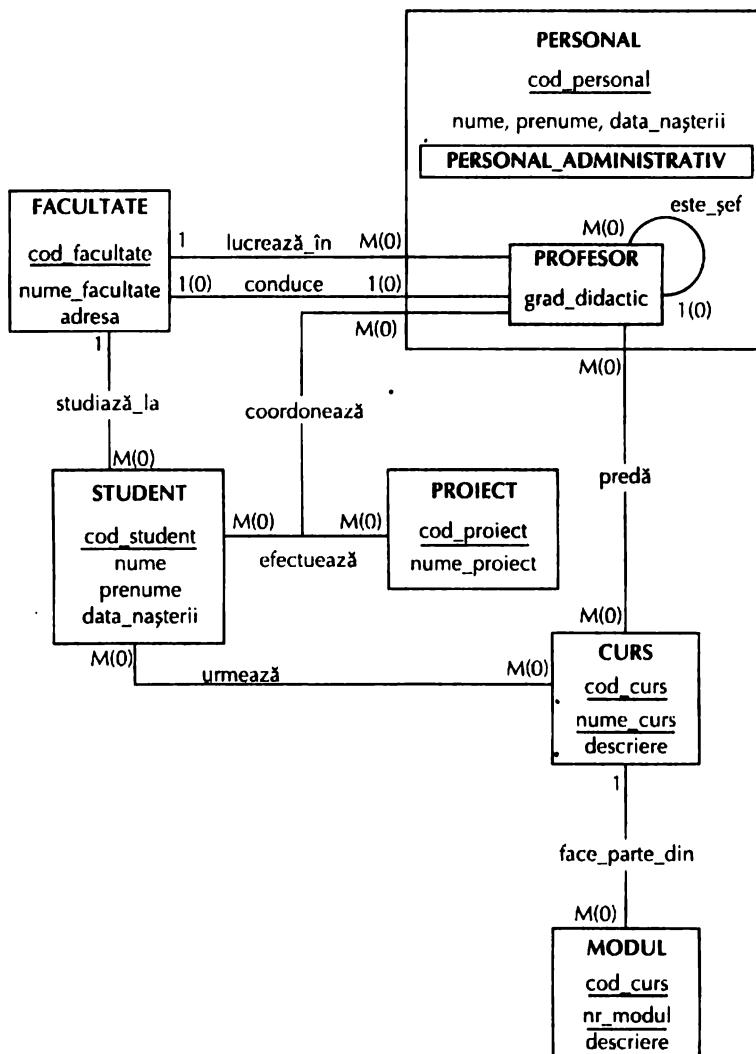


Figura 2.19

Trebuie remarcat că aceeași realitate poate fi percepția diferită de către analiști diferiți, așa că este posibilă obținerea de modele diferite pentru același sistem, după cum și un sistem poate să se modifice în timp, ceea ce va atrage la rândul său modificarea modelului asociat. În sfârșit, există și alte moduri grafice de prezentare a diagramei entitate-legătură, vezi de exemplu [15], în acest capitol prezentându-se doar una dintre notațiile existente.

2.2. Crearea design-ului logic al bazei de date

Pentru realizarea design-ului logic al bazei de date, schema conceptuală este transformată într-un design al bazei de date care va funcționa într-un sistem de gestiune al bazelor de date specific. Design-ul logic al bazei de date este o reafinare a modelului inițial furnizat de schema conceptuală. Aceasta nu înseamnă că modelul conceptual nu este corect, dar trebuie stabilite detalii suplimentare necesare dezvoltării proiectului.

2.2.1. Transformarea modelului entitate legătură în modelul relațional

Deci pentru obținerea design-ului logic al unei baze de date relaționale se pornește de la schema conceptuală, mai precis de la modelul entitate-legătură, și se încearcă reprezentarea entităților și a legăturilor sub formă de tabele relaționale. Regulile de conversie ale entităților, legăturilor și atributelor sunt următoarele:

Transformarea entităților:

Regula generală este că entitățile devin tabele, distingându-se următoarele subcategorii:

- Entitățile independente devin *tabele independente*, adică tabele a căror cheie primară nu conține chei străine. De exemplu, entitatea STUDENT va deveni un tabel a cărui cheie primară este "cod_student".
- Entitățile dependente devin *tabele dependente (tabele detaliu)* adică tabele a căror cheie primară conține cheia străină ce face referință la cheia primară a entității de care depinde entitatea în cauză. De exemplu, cheia primară a entității MODUL va fi formată din "cod_curs", care reprezintă o cheie străină pentru entitatea CURS, plus "nr_modul".
- Subentitățile devin *subtabele* adică tabele a căror cheie primară este cheia străină pentru tabelul superentitate. De exemplu, cheia primară a tabelului PROFESOR este "cod_personal", care este o cheie străină ce face referință la cheia primară "cod_personal" din tabelul PERSONAL.

Uneori, se preferă construirea unor *supertabele*, formate atât din attributele superentității - cele comune tuturor subentităților - cât și din attributele specifice fiecărei subentități. Avantajul unor astfel de supertabele este simplificarea programelor de manipulare a datelor. Pe de altă parte însă, ele creează probleme suplimentare privind integritatea datelor, de exemplu dacă vom avea un singur tabel pentru tot personalul din facultate, atunci attributele specifice profesorului pot avea valori diferite de Null numai atunci când în tabel se inserează un rând corespunzător unui profesor. În plus, subtabelele obținute din descompunerea unui astfel de supertabel sunt mai stabile, mai flexibile, ocupă spațiu fizic mai mic și conțin mai puține valori Null.

Transformarea relațiilor:

- Relațiile 1:1 devin chei străine, cheia străină fiind plasată în tabelul cu mai puține linii. De exemplu, relația "conduce" dintre PROFESOR și FACULTATE se realizează prin inserarea unei chei străine în tabelul FACULTATE care face referință la cheia primară a tabelului PROFESOR, vezi figura 2.20.

Notă: Plasamentul cheii străine va fi indicat printr-o săgeată (\rightarrow), iar când cheia străină va fi conținută în cheia primară, atunci vârful săgeții va fi umplut ($\rightarrow\!\!\!$).

Deci într-o relație 1:1 poziția cheii străine depinde de cardinalitatea minimă a relației. Dacă aceasta este tot 1:1, atunci cheia străină poate fi plasată în oricare din cele două tabele. Dacă însă această cardinalitate minimă este 1:0, atunci cheia străină este plasată în tabelul a cărui cardinalitate minimă în relație este 0. În cazul în care cardinalitatea minimă e 0:0 se alege pur și simplu tabelul cu mai puține linii.

- Relațiile N:1 devin chei străine plasate în tabelul care se află de partea "mulții" a relației. De exemplu relația "lucrează_în" va fi realizată prin inserarea unei chei străine în tabelul PROFESOR care va face referință la cheia primară a tabelului FACULTATE, vezi figura 2.21. Și în cazul relațiilor N:1 se disting două cazuri în funcție de cardinalitatea minimă a relației. Dacă aceasta este 0:1, atunci cheia străină respectivă nu poate avea valoarea Null, iar în cazul entităților dependente ea va face chiar parte din cheia primară a tabelului. Dacă însă cardinalitatea minimă a relației este 0:0 atunci cheia străină poate avea valoarea Null și nu poate face parte din cheia primară.
- O relație mulți-la-mulți se transformă într-un tabel special, numit tabel *asociativ*, care are două chei străine pentru cele două tabele asociate; cheia primară a tabelului asociativ este compusă din aceste două chei străine plus eventual alte coloane adiționale. În acest caz se spune că o relație mulți-la-mulți se sparge în două relații mulți-la-unu, tabelul asociativ fiind în relație de mulți-la-unu cu fiecare dintre cele două tabele entitate. De exemplu, relația "predă" dintre PROFESOR și CURS se realizează printr-un tabel a cărui cheie primară este combinația cheilor străine ale acestor două entități, vezi figura 2.22.
- O relație de tip 3 (relație între mai mult de două entități) devine un tabel asociativ care are căte o cheie străină pentru fiecare dintre tabelele asociate; cheia primară este compusă din aceste chei străine plus eventual alte coloane adiționale. De exemplu, tabelul reprezentat în figura 2.23 exprimă relația "efectuează_coordonatează" dintre STUDENT, PROIECT și PROFESOR. În acest caz, cheia primară este combinația cheilor străine corespunzătoare celor trei entități.

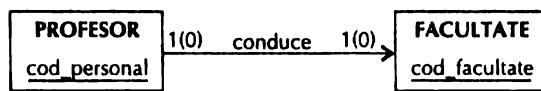


Figura 2.20

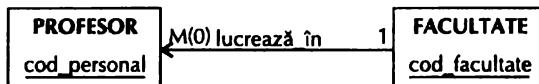


Figura 2.21

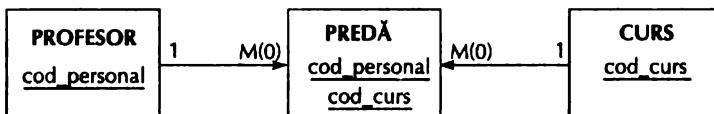


Figura 2.22

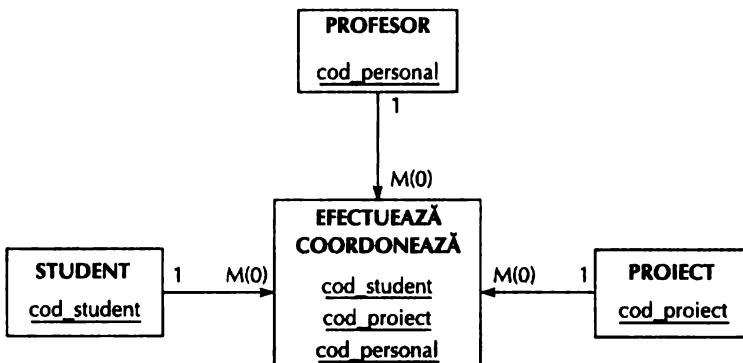


Figura 2.23

Transformarea atributelor:

- Atributele simple ale unei entități devin coloane în tabelul provenit din entitatea corespunzătoare. De asemenea, fiecare componentă a unui atribut compus devine o coloană în tabel. De exemplu, pentru atributul compus "adresă", format din "țară", "oraș", "stradă", "număr" și "cod", vom avea cinci coloane, câte una pentru fiecare componentă a sa.
- Atributele repetitive (multivaloare) ale unei entități devin tabele dependente ce conțin o cheie străină (care face referință la cheia primară a entității) și atributul multivaloare, cheia primară a acestui nou tabel este formată din cheia străină plus una sau mai multe coloane adiționale. De exemplu, dacă presupunem că un student poate avea mai multe numere de telefon, atunci "nr_telefon" este un atribut multivaloare al entității STUDENT, care va da naștere unui tabel TELEFON, a cărui cheie primară va fi combinația dintre "cod_student" și "nr_telefon", vezi figura 2.24.
- Atributele simple ale unei relații 1:1 sau 1:N vor deveni coloane ale tabelului care conțin cheia străină. De exemplu, data înscrierii, care este un atribut al relației "studiază_la" dintre STUDENT și FACULTATE, va fi reprezentată ca o coloană în tabelul STUDENT.

De asemenea, fiecare atribut compus al unei relații 1:1 sau 1:N va deveni o coloană în tabelul care conține cheia străină.

- Atributele simple ale unei relații N:M vor deveni coloane ale tabelului asociativ. De exemplu, nota obținută la examen, care este un atribut al relației "urmează" dintre STUDENT și CURS va fi reprezentată ca o coloană în tabelul asociativ corespunzător acestei relații. De asemenea, fiecare componentă a unui atribut compus al unei relații N:M va deveni o coloană în tabelul asociativ.
- Atributele repetitive (multivaloare) ale unei relații 1:1 sau 1:N vor deveni tabele dependente de tabelul care conține cheia străină, iar atributurile repetitive ale unei relații N:M vor deveni tabele dependente de tabelul asociativ corespunzător relației. Evident, cheia primară a acestor tabele dependente va fi o combinație formată din cheia străină respectivă și una sau mai multe coloane adiționale. De exemplu, dacă presupunem că în cadrul anumitor cursuri studenții trebuie să dea un număr de teste, atunci "test" va fi un atribut multivaloare al relației "urmează" dintre STUDENT și CURS și care va da naștere unui tabel dependent de tabelul asociativ al acestei relații, vezi figura 2.25.



Figura 2.24

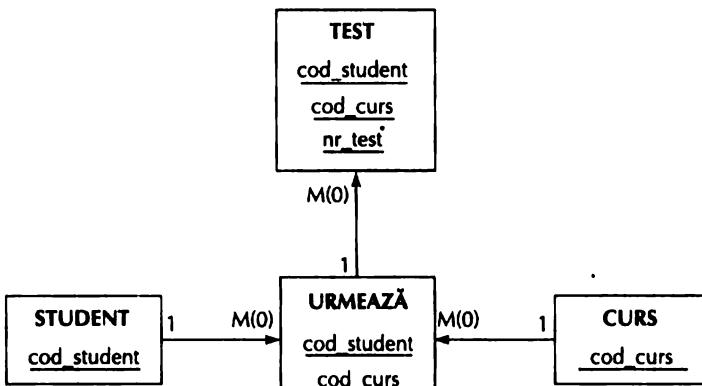


Figura 2.25

În figura 2.26 este prezentată *diagrama logică a bazei de date* pentru sistemul descris ca exemplu. Aceasta a rezultat din diagrama entitate-legătură, în urma transformărilor prezentate mai sus.

Tabelele asociate acestei diagrame sunt următoarele:

PERSONAL (cod_personal, nume, prenume, data_nastere, sex, stare_civilă, data_angajare)

PERSONAL_ADMINISTRATIV (cod_personal, profesie, funcție)

PROFESOR (cod_personal, grad_didactic, titlu, *șef*, ore_preadate, cod_facultate)

CURS (cod_curs, nume_curs, descriere, nr_ore)

PREDA (cod_personal, cod_curs, dată_inceput)

MODUL (cod_curs, nr_modul, descriere)

FACULTATE (cod_facultate, nume_facultate, localitate, strada, nr, cod_poștal, cod_decan)

STUDENT (cod_studenții, nume, prenume, data_nașterii, țara, localitate, strada, nr, cod_poștal, studii_anterioare, dată_înscriere)

TELEFON (*cod_student, nr_telefon, tip_telefon*)

PROJECT (cod_proiect, nume_proiect, domeniu)

EFFECTUEAZA COORDONEAZA (cod student, cod proiect, cod personal)

URMEEAZA (*cod_student, cod_curs, nota_examen, nota_restanță, observații*)

TEST (cod_student, cod_curs, nr_test, nota_test, observatii)

Notă: Atributele subliniate constituie cheia primară a tabelului iar cele italicice constituie chei străine.

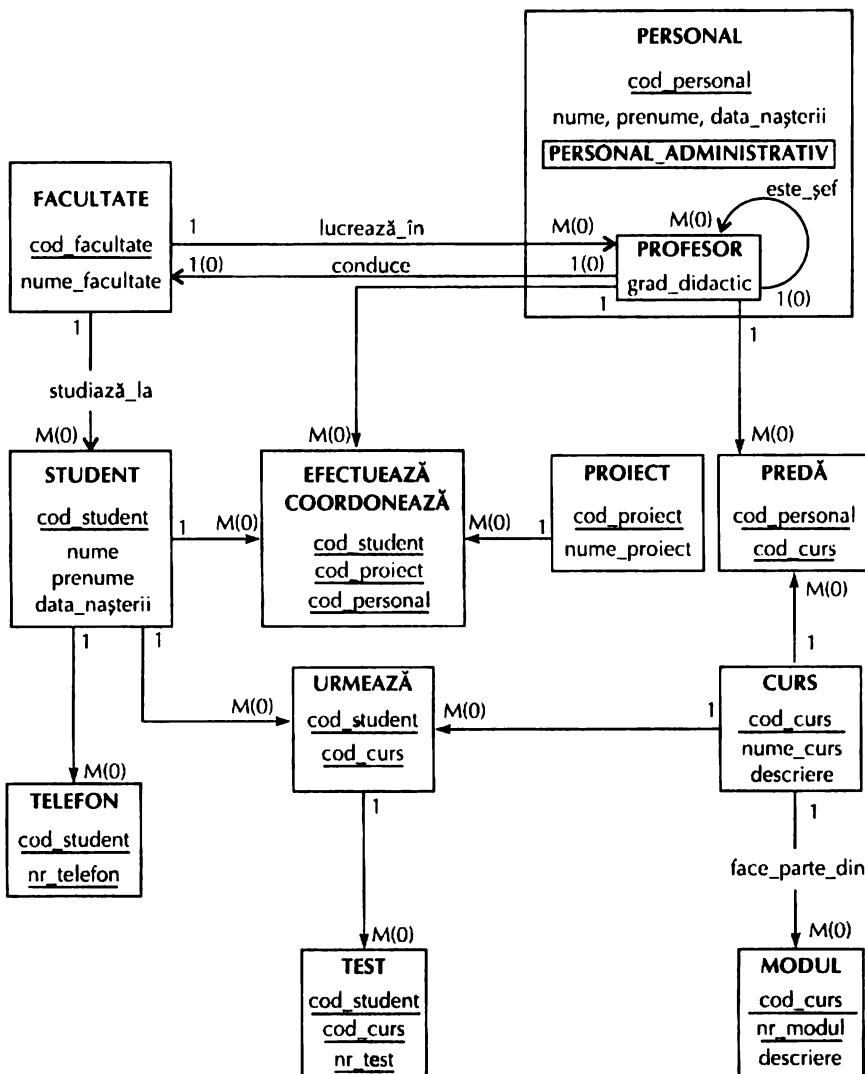


Figura 2.26

Exemplu

Pentru următorul scenariu să se realizeze diagrama logică a bazei de date și să se definească înbelele corespunzătoare evidențiindu-se cheile primare și cheile străine.

Un spital este organizat în saloane, fiecare având mai mulți pacienți. Fiecare doctor din spital aparține unei anumite secții condusă de un șef de secție care este tot doctor. Fiecare doctor este responsabil de mai mulți pacienți, un pacient putând fi tratat de unul sau mai mulți doctori. Un salon poate fi supravegheat de una sau mai multe asistente iar o asistentă nu poate lucra decât într-un singur salon.

Diagrama logică a bazei de date pentru acest sistem este reprezentată în figura 2.27. Tabelele asociate acestei diagrame sunt următoarele:

SALON (nr_salon, capacitate, etaj)

ASISTENTA (cod_asistentă, nume, prenume, nr_salon)

PACIENT (cod_patient, nume, prenume, data_naștere, sex, nr_salon)

DOCTOR (cod_doctor, nume, prenume, grad, statut, cod_sectie)

TRATAMENT (cod_patient, cod_doctor, descriere)

SECTIE (cod_sectie, denumire, cainera_garda, șef)

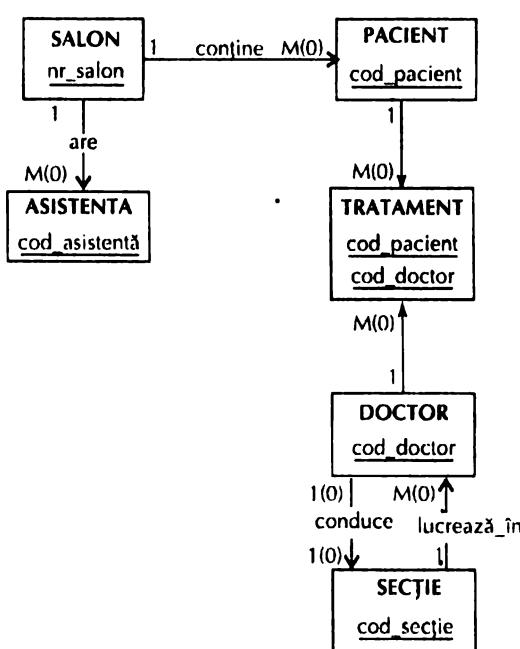


Figura 2.27

2.3. Normalizarea bazei de date

În general, pentru proiectarea unei baze de date relaționale se creează mai întâi schema conceptuală a acesteia (folosind de obicei modelul entitate-legătură), care este transformată apoi într-un design logic, folosind metodologia descrisă în secțiunile precedente. În trecut însă, în locul

acestei tehnici de modelare conceptuală, proiectarea unei baze de date relaționale se face folosindu-se o tehnică numită *normalizare*, ce constă din descompunerea unui tabel relațional în mai multe tabele care satisfac anumite reguli și care stochează aceleși date ca și tabelul inițial. În prezent, normalizarea se folosește pentru a rafina design-ul logic al bazei de date rezultat în urma transformării schemei conceptuale, eliminând unele probleme care pot apărea în urma procesului de proiectare inițial. Evident, dacă schema conceptuală este corectă și dacă transformarea acesteia în tabele relaționale este corectă, procesul de normalizare nu mai este necesar. Totuși, cunoașterea acestei tehnici este necesară deoarece ea stabilește criteriile pe care trebuie să le îndeplinească un design corect al unei baze de date relaționale.

După transformarea modelului entitate-legătură în tabele relaționale, în baza de date rezultată poate exista redundanță în date cât și anomalii de actualizare, adică anumite rezultate nedorite în timpul încărcării, exploatarii și întreținerii bazei de date.

În continuare vom ilustra cu ajutorul unui exemplu, aceste posibile deficiențe ale bazei de date, precum și modul în care tehnica normalizării înlătură aceste rezultate nedorite. Înainte însă, dăm următoarea definiție:

Fie R un tabel relațional și fie X și Y două submulțimi de coloane ale lui R. Vom spune că X *determină funcțional* pe Y sau Y *depinde funcțional* de X dacă nu există două rânduri în tabelul R care să aibă aceleași valori pentru coloanele din X și să aibă valori diferite pentru cel puțin o coloană din Y. Cu alte cuvinte, o valoare a lui X determină în mod unic o valoare a lui Y, adică oricare două rânduri din R care au aceeași valoare pentru X trebuie să ia aceeași valoare pentru Y. Notația utilizată pentru descrierea dependenței funcționale este $X \rightarrow Y$. X se va numi *determinantă* iar Y *determinat*. Spunem că dependența $X \rightarrow Y$ este *trivială* dacă toate elementele lui Y sunt și elemente ale lui X, $Y \subseteq X$.

Să considerăm acum tabelul relațional:

VÂNZĂRI (cod_client, nume_client, nr_telefon,
cod_comandă, data, cod_articol,
 nume_articol, cost_articol, cantitate)

Tabelul de mai sus se folosește pentru înregistrarea tranzacțiilor unui magazin ce vinde articole la comandă, vezi figura 2.28. Coloanele subliniate constituie cheia primară a tabelului.

VÂNZĂRI

cod_client	nume_client	nr_telefon	cod_comandă	data	cod_articol	nume_articol	cost_articol	cantitate
A1	Popescu	3215576	C1	12.05.99	P1	cămașă	100.000	2
A1	Popescu	3215576	C1	12.05.99	P3	tricou	50.000	1
A2	Ionescu	2325587	C2	13.05.99	P1	cămașă	100.000	3
A2	Ionescu	2325587	C2	13.05.99	P3	tricou	50.000	2
A2	Ionescu	2325587	C2	13.05.99	P2	pantaloni	200.000	1
A1	Popescu	3215576	C3	14.05.99	P3	tricou	50.000	3
A3	Georgescu	4555895	C4	14.05.99	P1	cămașă	100.000	1

Figura 2.28: tabelul VÂNZĂRI

În mod evident, numele clientului și numărul său de telefon vor fi dependente de codul

Înțelesului, data la care a fost efectuată comanda va depinde de codul acesteia, iar numele și numărul articolului vor fi dependente de codul acestuia. În plus, vom presupune, aşa cum este în normal, că o comandă poate fi făcută de un singur client, cu alte cuvinte codul clientului, și implicit și numele și numărul de telefon al acestuia, vor depinde de codul comenzii. Cu alte cuvinte, în tabelul de mai sus am identificat următoarele dependențe funcționale pentru care determinantul nu este cheie a tabelului:

```
{cod_articol} → {nume_articol, cost_articol}
{cod_comandă} → {data, cod_client, nume_client, nr_telefon}
{cod_client} → {nume_client, nr_telefon}
```

Notă: dependența $\{cod_comandă\} \rightarrow \{nume_client, nr_telefon\}$ poate fi dedusă din dependențele $\{cod_comandă\} \rightarrow \{cod_client\}$ și $\{cod_client\} \rightarrow \{nume_client, nr_telefon\}$. Astfel de dependențe se numesc *dependențe tranzitive* și vor fi definite mai înainte în secțiunea 2.3.3.

Datorită acestor dependențe, în tabelul prezentat mai sus pot să apară următoarele inconveniente:

- Poate exista *redundanță* în date, adică unele informații apar de mai multe ori. De exemplu, numele și costul articolului cu codul P1 (P1, cămașă, 100.000) sunt specificate de 3 ori și tot de 3 ori sunt specificate numele și numărul de telefon ale clientului cu codul A1 (A1, Popescu, 3215576).
- Pot apărea anomalii la actualizare:
 - * *Anomalie la inserție.* Să presupunem că magazinul a achiziționat un nou articol pe care urmează să-l vândă, de exemplu un articol cu codul P4, numele "pantofi" și costul 250.000. Totuși, tuplul (P4, pantofi, 250.000) nu poate fi inserat în tabelul VĂNZĂRI decât dacă există o comandă pentru acest articol - aceasta datorită faptului că trebuie să dăm o valoare diferită de Null coloanei "cod_comanda", care face parte din cheia primară. La fel se întâmplă și dacă vrem să inserăm datele (numele și numărul de telefon) unui nou client.
 - * *Anomalie la stergere.* Să presupunem că în cadrul comenzii cu codul C2 este anulat articolului P2 (pantaloni). Ștergând rândul corespunzător - cel pentru care valoarea cheii primare este (C2, P2) - se pierde și informația referitoare la numele și costul articolului respectiv. La fel, dacă se șterge rândul pentru care valoarea cheii primare este (C4, P1), se pierde informația referitoare la clientul cu codul A3.
 - * *Anomalie la modificare.* Să presupunem că s-a schimbat numele articolului P1 de la "cămașă" la "bluză". Dacă modificarea respectivă se face doar pentru un rând al tabelului, atunci în tabel vor apărea date incorecte - celelalte două rânduri vor conține numele vechi al articolului. Deci această modificare trebuie făcută în toate rândurile pentru care valoarea coloanei "cod_articol" este P1, costul modificării fiind în acest caz semnificativ. La fel se întâmplă și în cazul în care se modifică numărul de telefon sau numele unui client.

Redundanța și anomaliiile care apar în lucrul cu baza de date sunt datorate dependențelor care există în cadrul tabelelor bazei de date, mai precis acele dependențe pentru care determinantul nu este cheie a tabelului. *Normalizarea* este procesul reversibil de descompunere a unui tabel relațional în tabele cu o structură mai simplă, proces care are scop tocmai evitarea redundanței datelor și evitarea anomalieiilor de actualizare.

Spunem că procesul este *reversibil* în sensul că descompunerea se face *fără pierdere de informație*, adică tabelul inițial poate fi reconstruit prin compunerea naturală pe atribute comune

a tabelelor rezultate. Pentru un tabel R care se descompune prin proiecție în tabelele R_1, R_2, \dots, R_n , condiția de descompunere fără pierdere de informație presupune ca R să fie obținut prin compunerea naturală a tabelelor R_1, R_2, \dots, R_n (adică coloanele din R să fie reuniunea coloanelor din R_1, R_2, \dots, R_n și fiecare rând din R să fie obținut prin compunerea rândurilor din cele n tabele pentru care valoarea atributelor comune este identică).

Un caz particular al acestei descompuneri, cel mai des utilizat în construcția formelor normale, este regula Casey-Delobel: fie un tabel R(X, Y, Z) care se descompune prin proiecție în tabelele R1(X, Y) și R2(X, Z) - unde prin X am notat coloanele comune ale tabelelor R_1 și R_2 , iar prin Y și Z coloanele specifice lui R, și respectiv R_2 ; în acest caz, condiția de descompunere fără pierdere de informație presupune ca tabelul R să fie obținut prin compunerea naturală a tabelelor R_1 și R_2 (adică prin compunerea rândurilor celor două tabele pentru care valorile coloanelor X sunt identice). Într-un limbaj pseudo-SQL, compunerea naturală a celor două tabele se scrie în modul următor:

```
SELECT R1.X, R1.Y, R2.Z
FROM R1, R2
WHERE R1.X = R2.X
```

De exemplu, aplicând succesiv regula Casey-Delobel, tabelul VÂNZĂRI se poate descompune fără pierdere de informații în tabelele

VÂNZĂRI_1 (cod_client, nume_client, nr_telefon, cod_comandă,
data, cod_articol, cantitate)

și

ARTICOL (cod_articol, nume_articol, cost_articol), vezi figura 2.29,
iar tabelul VÂNZĂRI_1 poate fi descompus în
VÂNZĂRI_2 (cod_comandă, cod_articol, cantitate)

și

COMANDA_2 (cod_comandă, data, cod_client, nume_client,
nr_telefon), vezi figura 2.30

La rândul lui, tabelul COMANDA_2 poate fi descompus în tabelele
COMANDA_3 (cod_comandă, data, cod_client)

și

CLIENT (cod_client, nume_client, nr_telefon), vezi figura 2.31.

În consecință, tabelul VÂNZĂRI poate fi descompus fără pierdere de informații în tabelele VÂNZĂRI_2, ARTICOL, COMANDA_3 și CLIENT.

VÂNZĂRI_1

<u>cod_client</u>	<u>nume_client</u>	<u>nr_telefon</u>	<u>cod_comandă</u>	<u>data</u>	<u>cod_articol</u>	<u>cantitate</u>
A1	Popescu	3215576	C1	12.05.99	P1	2
A1	Popescu	3215576	C1	12.05.99	P3	1
A2	Ionescu	2325587	C2	13.05.99	P1	3
A2	Ionescu	2325587	C2	13.05.99	P3	2
A2	Ionescu	2325587	C2	13.05.99	P2	1
A1	Popescu	3215576	C3	14.05.99	P3	3
A3	Georgescu	4555895	C4	14.05.99	P1	1

ARTICOL

cod_articol	nume_articol	cost_articol
P1	cămașă	100.000
P2	pantaloni	200.000
P3	tricou	50.000

Figura 2.29

VÂNZĂRI_2

cod_comandă	cod_articol	cantitate
C1	P1	2
C1	P3	1
C2	P1	3
C2	P3	2
C2	P2	1
C3	P3	3
C4	P1	1

COMANDA_2

cod_comandă	data	cod_client	nume_client	nr_telefon
C1	12.05.99	A1	Popescu	3215576
C2	13.05.99	A2	Ionescu	2325587
C3	14.05.99	A1	Popescu	3215576
C4	14.05.99	A3	Georgescu	4555895

Figura 2.30

COMANDA_3

cod_comandă	data	cod_client
C1	12.05.99	A1
C2	13.05.99	A2
C3	14.05.99	A1
C4	14.05.99	A3

CLIENT

cod_client	nume_client	nr_telefon
A1	Popescu	3215576
A2	Ionescu	2325587
A3	Georgescu	4555895

Figura 2.31

În plus, procesul de normalizare trebuie să realizeze o *descompunere minimală* a tabelului inițial, deci nici o coloană din tabelele rezultate nu poate fi eliminată fără a duce la pierderea de informații și implicit la pierderea caracterului ireversibil al transformării. Ca o consecință, nici unul dintre tabelele rezultate nu trebuie să fie conținut într-altul.

În sfârșit, este de dorit ca procesul de normalizare să *conserve dependențele* dintre date. Aceasta înseamnă că pentru fiecare dependență netranzitivă trebuie ca atât determinantul cât și determinatul să existe într-unul din tabelele rezultate prin descompunere. Se poate vedea că această cerință este îndeplinită de descompunerea tabelului VÂNZĂRI, fiecare dintre dependențele netranzitive existente în tabel, {cod_articol} → {nume_articol, cost_articol}, {cod_comandă} → {data, cod_client} și {cod_client} → {nume_client, nr_telefon}, regăsindu-se într-unul dintre tabelele ARTICOL, COMANDA_3 și CLIENT. Dependențele tranzitive nu trebuie conservate deoarece pot fi deduse din celelalte. Totuși, aşa cum vom vedea în continuare, nu întreg procesul de normalizare conservă dependențele, această cerință putând fi încălcată în procesul de transformare în formele normale superioare (incepând cu BCNF).

Există o teorie matematică a normalizării ai căror autori sunt Codd și Fagin. Un tabel este într-o anumită *formă normală* dacă satisfacă un anumit set de constrângeri. Există șase forme normale: prima formă normală, a doua formă normală, a treia formă normală, forma normală Boyce-Codd, a patra formă normală și a cincea formă normală. Constrângerile pentru o formă normală sunt întotdeauna mai puternice decât cele pentru formele normale inferioare, de aceea tabelele aflate în a doua formă normală constituie un subset al tabelelor aflate în prima formă normală și.a.m.d.

2.3.1. Prima formă normală (1NF – First Normal Form)

Definiție 1NF

Un tabel relațional este în prima formă normală (1NF) dacă fiecărei coloane îi corespunde o valoare indivizibilă (atomică), deci orice valoare nu poate să fie o mulțime sau un tuplu de valori. În plus, nu pot să apară grupuri de atribute repetitive.

Tabelul VÂNZĂRI (figura 2.28) se află în prima formă normală (se presupune că pentru fiecare client se reține doar un număr de telefon).

Prima formă normală elimină atributele compuse și atributele sau grupurile de atribute repetitive. De exemplu, pentru o coloană care conține date calendaristice, sub forma zz-ll-aa (doi digiți pentru zi, doi pentru lună și doi pentru an), se consideră că valoarea respectivă nu se poate descompune în ziua, luna și anul corespondătoare acestei valori. Pe de altă parte însă, dacă vom considera că "adresa" este un atribut compus format din componentele "țară", "oraș", "stradă", "număr" și "cod", fiecare având semnificație proprie și putând fi folosite independent la interogarea bazei de date, atunci adresa va fi reprezentată în tabel prin 5 coloane în loc de una. De asemenea, un tabel aflat în 1NF nu poate conține atribute sau grupuri de atribute repetitive. De exemplu, în tabelul de mai jos, dacă un student poate avea mai multe numere de telefon, atunci coloanele "telefon1", "telefon2", "telefon3" constituie un grup de atribute repetitive. Deci tabelul STUDENT de mai jos nu este în prima formă normală, el conținând atât un atribut compus ("adresa") cât și un grup de atribute repetitive.

STUDENT

cod_student	nume	prenume	adresa	telefon1	telefon2	telefon3	materia	nota
101	Ionescu	Vasile	România, Bucureşti, Str. Polizu 5, 7355	6245981	3215678	092659019	Drept	10
101	Ionescu	Vasile	România, Bucureşti, Str. Polizu 5, 7355	6245981	3215678	092659019	Engleza	8
102	Pop	Costică	România, Cluj, Str. Unirii 7, 3551	3215469			Fizică	8

Figura 2.32

Algoritmul 1NFA permite aducerea unei relații în 1NF prin eliminarea atributelor compuse și a celor repetitive.

Algoritmul 1NFA

- Se înlocuiesc în tabel coloanele corespunzătoare atributelor compuse cu coloane ce conțin componente elementare ale acestora.
- Se plasează grupurile de atrbute repetitive, fiecare în câte un nou tabel.
- Se introduce în fiecare tabel nou creat la pasul 2 cheia primară a tabelului din care a fost extras atributul respectiv, care devine cheie străină în noul tabel.
- Se stabilește cheia primară a fiecărui nou tabel creat la pasul 2. Aceasta va fi compusă din cheia străină introdusă la pasul 3 plus una sau mai multe coloane adiționale.

Prin aplicarea algoritmului de mai sus, tabelul STUDENT se descompune în tabelele STUDENT_1 și TELEFON, aflate amândouă în 1NF.

STUDENT_1

cod_student	nume	prenume	țară	oraș	stradă	nr	cod_postal	materia	nota
101	Ionescu	Vasile	România	Bucureşti	Polizu	5	7355	Drept	10
101	Ionescu	Vasile	România	Bucureşti	Polizu	5	7355	Engleza	8
102	Pop	Costică	România	Cluj	Unirii	7	3551	Fizică	8

TELEFON

cod_student	telefon
101	6245981
101	3215678
101	092659019
102	3215469

Figura 2.33

Prima formă normală este o cerință minimală a tuturor sistemelor relaționale. Sistemele de baze de date care nu respectă nici măcar această formă nu pot fi numite relaționale. Tabelele aflate în prima formă normală permit o referire simplă a datelor prin indicarea numelui tabelului, a coloanei și a cheii rândului din care face parte informația respectivă. Operatorii pentru aceste tabele sunt mai simpli și permit definirea unor tehnici de proiectare și utilizare a bazelor de date.

2.3.2. A doua formă normală (2NF– Second Normal Form)

Să vedem acum modul în care procesul de normalizare înlătură redundanța logică și anomaliiile de actualizare semnalate la începutul acestei secțiuni. Pentru acest lucru dăm întâi următoarea definiție.

Fie R un tabel relațional și fie X și Y două submulțimi de coloane ale lui R. O dependență funcțională $X \rightarrow Y$ se numește *totală* dacă pentru orice subset de coloane Z al lui X, $Z \subseteq X$, dacă $Z \rightarrow Y$ atunci $Z = X$. Cu alte cuvinte, nu există nici un subset Z al lui X, $Z \neq X$, pentru care $Z \rightarrow Y$. O relație funcțională care nu este totală se numește *parțială*.

Se observă că în tabelul VÂNZĂRI (figura 2.28) există redundanță în date - de exemplu (P1, cămașă, 100.000) - datorată faptului că atributele "nume_articol" și "cost_articol" nu depind de "cod_comandă", care este o componentă a cheii primare a tabelului. Cu alte cuvinte, în tabelul VÂNZĂRI există o dependență parțială de cheia primară. Acest tip de dependențe vor fi înlăturate în a doua formă normală.

Definiție 2NF

Un tabel relațional R este în a doua formă normală (2NF) dacă și numai dacă:

- R este în 1NF
- Orice coloană care depinde parțial de o cheie a lui R este inclusă în acea cheie.

Cu alte cuvinte, a doua formă normală nu permite dependențe funcționale parțiale față de cheile tabelului, cu excepția dependențelor triviale, de inclusiune. Deci tabelul VÂNZĂRI nu este în 2NF, coloanele "nume_articol" și "cost_articol" depinzând parțial de cheia primară a tabelului.

Pentru a obține tabele relaționale în 2NF, tabelul inițial se descompune fără pierdere de informație după următoarea regulă: Fie R(K1, K2, X, Y) un tabel relațional unde K1, K2, X și Y sunt submulțimi de coloane ale lui R astfel încât K1 \cup K2 este o cheie a lui R, iar K1 \rightarrow X este o dependență funcțională totală. Dacă X \subset K1 atunci tabelul este deja în 2NF, altfel tabelul R poate fi descompus prin proiecție în R1(K1, K2, Y) - având cheia K1 \cup K2 - și R2(K1, X) - având cheia K1. Remarcă că această descompunere conservă nu numai datele, ci și dependențele funcționale, atât determinantul cât și determinantul dependenței eliminate regăsindu-se în tabelul nou creat.

Folosind această regulă, algoritmul 2NFA permite aducerea în 2NF a unui tabel relațional aflat în 1NF prin eliminarea dependențelor funcționale parțiale.

Algoritm 2NFA

- Pentru fiecare coloană X care depinde funcțional parțial de o cheie K, $K \rightarrow X$, și care nu este inclusă în K, se determină $K1 \subset K$ un subset al lui K, astfel încât dependența $K1 \rightarrow X$ este totală și se creează un nou tabel $R1(K1, X)$, adică un tabel format din determinantul ($K1$) și determinatul (X) acestei dependențe.
- Dacă în tabelul R există mai multe dependențe totale ca mai sus cu același determinant, atunci pentru acestea se creează un singur tabel format din determinant - luat o singură dată - și din determinații dependențelor considerate
- Se elimină din tabelul inițial R toate coloanele, X, care formează determinantul dependenței considerate.
- Se determină cheia primară a fiecărui tabel nou creat, $R1$. Acesta va fi $K1$, determinantul dependenței considerate.
- Dacă noile tabele create conțin alte dependențe parțiale, atunci se merge la pasul 1, altfel algoritmul se termină.

Pentru tabelul VÂNZĂRI cheia primară este {cod_comandă, cod_articol}. Atributele "nume_articol" și "cost_articol" depind funcțional parțial de această cheie primară și depind funcțional total de atributul "cod_articol" conținut în cheie primară. Același lucru se întâmplă și cu attributele "data", "cod_client", "nume_client" și "nr_telefon" care depind funcțional total numai de atributul "cod_comandă" conținut în cheie primară. Prin urmare vom avea următoarele dependențe totale:

$$\begin{aligned} \{\text{cod_comandă}\} &\rightarrow \{\text{data, cod_client, nume_client, nr_telefon}\} \\ \{\text{cod_articol}\} &\rightarrow \{\text{nume_articol, cost_articol}\} \end{aligned}$$

În consecință, tabelul VÂNZĂRI va fi descompus în tabelele VÂNZĂRI_2, COMANDA_2 și ARTICOL, care sunt toate în 2NF, vezi figura 2.34.

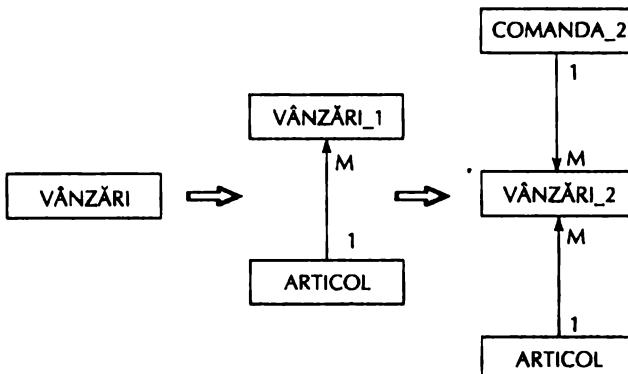


Figura 2.34

Se observă că un tabel care are cheia primară formată dintr-un singur atribut este automat în 2NF. Prin urmare, algoritmul 2NFA nu se poate aplica decât în cazul în care cheia primară a unui tabel este o cheie compusă.

2.3.3. A treia formă normală (3NF –Third Normal Form)

Deși tabelul COMANDA_2 este în 2NF, se observă că încă mai există redundanță în date - tuplul (A1, Popescu, 3215576) apare de două ori. Intuitiv, aceasta se explică prin faptul că atributele "nume_client" și "nr_telefon" depind indirect de cheia primară a tabelului, dependența făcându-se prin intermediul atributului "cod_client". Aceste dependențe indirekte vor fi îndepărtate în 3NF.

Definiție 3NF

Un tabel relațional R este în a treia formă normală (3NF) dacă și numai dacă:

- R este în 2NF;
- Pentru orice coloană A neconținută în nici o cheie a lui R, dacă există un set de coloane X astfel încât $X \rightarrow A$, atunci fie X conține o cheie a lui R, fie A este inclusă în X.

A doua condiție din definiție interzice dependențele funcționale totale față de alte coloane în afara celor care constituie chei ale tabelului. Prin urmare, un tabel este în 3NF dacă orice coloană care nu este conținută într-o cheie depinde de cheie, de întreaga cheie și numai de cheie. În mod evident tabelul COMANDA_2 nu este în 3NF existând dependențele $\{cod_client\} \rightarrow \{nume_client\}$ și $\{cod_client\} \rightarrow \{nr_telefon\}$. Pe de altă parte, tabelele VÂNZĂRI_2 și ARTICOL sunt în 3NF.

Adesea, cea de-a doua condiție din definiția de mai sus se formulează folosind noțiunea de *dependență tranzitivă*. Fie R un tabel relațional, X o submulțime de coloane a lui R și A o coloană a lui R. Spunem că A este *dependentă tranzitivă* de X dacă există o submulțime de coloane Y care nu include coloana A și care nu determină funcțional pe X astfel încât $X \rightarrow Y$ și $Y \rightarrow A$. Dacă în această definiție se dorește să se evidențieze și Y atunci se spune că A depinde funcțional de X prin intermediul lui Y și se scrie $X \rightarrow Y \rightarrow A$. De exemplu, în tabelul COMANDA_2 coloanele "nume_client" și "nr_telefon" depind tranzitiv de cheia primară "cod_comandă" prin intermediul coloanei "cod_client". Folosind această definiție, condiția ca un tabel să fie în 3NF se poate reformula astfel:

Definiție 3NF

Un tabel relațional R este în a treia formă normală (3NF) dacă și numai dacă:

- R este în 2NF;
- Orice coloană neconținută în nici o cheie a lui R nu este dependentă tranzitivă de nici o cheie a lui R.

Pentru a obține tabele relaționale în 3NF, tabelul inițial se descompune fără pierdere de informație după următoarele reguli. Fie R(K, X, Y, Z) un tabel relațional unde K este o cheie a lui R, iar X, Y și Z sunt submulțimi de coloane ale lui R.

- Dacă există dependență tranzitivă $K \rightarrow X \rightarrow Y$, atunci R se poate descompune în R1(K, X, Z) - având cheia K - și R2(X, Y) - având cheia X.
- Dependența tranzitivă poate fi mai complexă. Fie $K_1 \subset K$ o parte a cheii K astfel încât există dependență tranzitivă $K \rightarrow X_1 \rightarrow Y$, unde $X_1 = K_1 \cup X$. În acest caz, R poate fi descompus în R1(K, X, Z) - având cheia K - și R2(K1, X, Y) - având cheia $K_1 \cup X$.

Remarcați că descompunerile corespunzătoare regulilor de mai sus conservă nu numai dusele, ci și dependențele funcționale, determinantul și determinatul dependențelor eliminate regăsindu-se în tabelele nou create.

Un exemplu de aplicare a primei reguli este descompunerea tabelului COMANDA_2 în COMANDA_3 și CLIENT, amândouă fiind în 3NF.

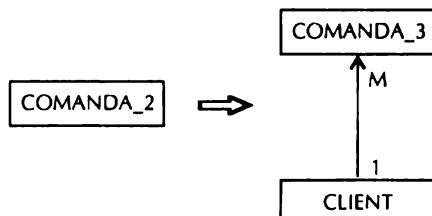


Figura 2.35

Pentru a exemplifica cea de-a două regulă, să luăm tabelul PROIECTE (cod_angajat, cod_proiect, rol_in_proiect, suma_objinută), vezi figura 2.36, care stocăază date privind repartizarea pe proiecte a angajaților unei firme. Să presupunem că suma obținută de un angajat împinde de proiectul respectiv și de rolul angajatului în acel proiect, deci avem dependența (cod_proiect, rol_in_proiect) → suma_objinută. Aplicând regula a doua de mai sus, tabelul PROIECTE se descompune în tabelele PROIECTE_3 (cod_angajat, cod_proiect, rol_in_proiect) și SUMA (cod_proiect, rol_in_proiect, suma_objinută), vezi figurile 2.37 și 2.38.

PROIECTE			
<u>cod_angajat</u>	<u>cod_proiect</u>	<u>rol_in_proiect</u>	<u>suma_objinută</u>
A1	P1	Programator	100.000
A2	P1	Coordonator	150.000
A3	P1	Programator	100.000
A4	P1	Programator	100.000
A1	P2	Programator	90.000
A4	P2	Analist	140.000

Figura 2.36

PROIECTE_3		
<u>cod_angajat</u>	<u>cod_proiect</u>	<u>rol_in_proiect</u>
A1	P1	Programator
A2	P1	Coordonator
A3	P1	Programator
A4	P1	Programator
A1	P2	Programator
A4	P2	Analist

48 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

SUMA		
cod_proiect	rol_in_proiect	suma_objinută
P1	programator	100.000
P1	coordonator	150.000
P2	programator	90.000
P2	analist	140.000

Figura 2.37

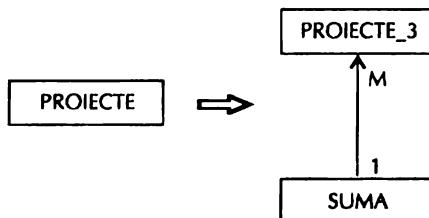


Figura 2.38

Folosind această regulă, algoritmul 3NFA permite aducerea în 3NF a unui tabel relațional aflat în 2NF prin eliminarea dependențelor funcționale tranzitive.

Algoritmul 3NFA

1. Pentru fiecare dependență funcțională tranzitivă $K \rightarrow X \rightarrow Y$, unde K și X nu sunt neapărat disjuncte, se transferă coloanele din X și Y într-un nou tabel.
2. Se determină cheia primară a fiecărui nou tabel creat la pasul 1, aceasta fiind formată din coloanele din X.
3. Se elimină din tabelul principal coloanele din Y.
4. Dacă tabelele rezultate conțin alte dependențe tranzitive, atunci se merge la pasul 1, altfel algoritmul se termină.

Aplicând algoritmii 2NFA și 3NFA, tabelul VÂNZĂRI a fost descompus în tabelele VÂNZĂRI_2, ARTICOL, COMANDA_3 și CLIENT, care sunt toate în 3NF. Se poate constata cu ușurință că în aceste tabele nu mai există nici redundanță în date și nici anomalii de actualizare.

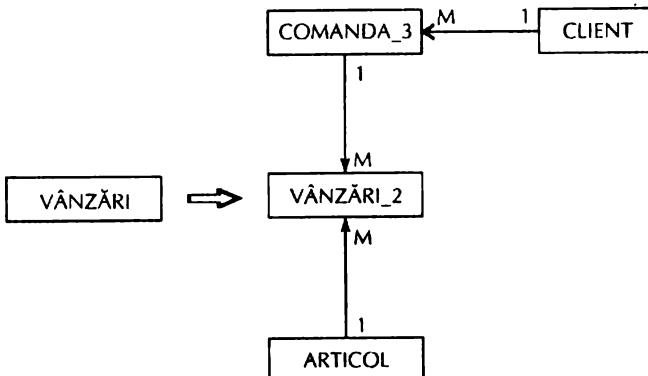


Figura 2.39

2.3.4. Forma normală Boyce-Codd (BCNF – Boyce-Codd Normal Form)

Totuși, nu toate tabelele aflate în a treia formă normală sunt lipsite de redundanță în date și nu pot fi normalizate. Pentru a ilustra această situație să considerăm următorul exemplu. O companie de transporturi efectuează curse, în care poate folosi unul sau mai mulți șoferi - de exemplu prima jumătate a curselor conduce un șofer, cea de-a doua alt șofer - și mai multe între autobuzele aflate în dotare, cu condiția ca într-o cursă un șofer să conducă un singur autobuz. Pe de altă parte însă, un autobuz este repartizat unui șofer și deci nu poate fi con dus de către decât de acesta. Această situație poate fi modelată printr-un tabel TRANSPORTURI (cod_cursă, cod_șofer, cod_autobuz, loc_plecare, loc_sosire), unde ultimele două coloane indică locul de plecare și locul de pornire a șoferului în cursă, vezi figura 2.40. În acest tabel există dependențe:

$$\begin{aligned} \{ \text{cod_cursă}, \text{cod_șofer} \} &\rightarrow \{ \text{cod_autobuz}, \text{loc_plecare}, \text{loc_sosire} \} \\ \{ \text{cod_autobuz} \} &\rightarrow \{ \text{cod_șofer} \} \end{aligned}$$

În cheile tabelului sunt {cod_cursă, cod_șofer} și {cod_cursă, cod_autobuz}. Se poate vedea cu ușurință că tabelul TRANSPORTURI este în 3NF și totuși în acest tabel există redundanță în date, însoțită de nereguli (S1, A1) și (S2, A2) apărând de 2 ori, datorită dependenței {cod_autobuz} → {cod_șofer}.

TRANSPORTURI

cod_cursă	cod_șofer	cod_autobuz	loc_plecare	loc_sosire
C1	S1	A1	Constanța	București
C1	S2	A2	București	Brașov
C2	S2	A2	Cluj	Sibiu
C2	S1	A3	Sibiu	București
C3	S1	A1	București	Constanța

Figura 2.40

Forma Boyce-Codd elimină acest tip de redundanță. Intuitiv, un tabel R este în BCNF dacă fiecare determinant al unei dependențe funcționale este cheie candidată a lui R. Tabelul TRANSPORTURI de mai sus nu este în BCNF, „cod_autobuz” nefiind o cheie a lui R. O definiție mai riguroasă pentru BCNF este prezentată în continuare.

Definiție BCNF

Un tabel relațional este în forma normală Boyce-Codd (BCNF) dacă și numai dacă pentru orice dependență funcțională totală $X \rightarrow A$, unde X este un subset de coloane iar A o coloană neconținută în X , X este o cheie a lui R.

Orice tabel în BCNF este și în 3NF, reciprocă fiind falsă, după cum demonstrează contraexemplul de mai sus. În plus, orice tabel care are cel mult două coloane este în BCNF. Orice tabel relațional se poate descompune fără pierdere de informație în tabele aflate în BCNF, dar nu același lucru se poate spune despre descompunerea cu păstrarea dependențelor funcționale, după cum vom vedea în continuare.

Un algoritm care permite aducerea în BCNF a unui tabel relațional aflat în 3NF prin eliminarea dependențelor non-cheie este prezentat în continuare.

Algoritmul BCNFA

- Pentru fiecare dependență non-cheie $X \rightarrow Y$, unde X și Y sunt subseturi de coloane ale lui R, se creează două tabele. Una dintre ele va fi formată din coloanele $\{X, Y\}$, iar cealaltă va fi formată din toate coloanele inițiale, mai puțin coloanele Y .
- Dacă tabelele rezultate conțin alte dependențe non-cheie, atunci se merge la pasul 1, altfel algoritmul se termină.

Aplicând acest algoritm, tabelul TRANSPORTURI de mai sus se va descompune în tabelele TRANSPORTURI_BC (cod_cursă, cod_autobuz, loc_plecare, loc_sosire) și AUTOBUZ (cod_autobuz, cod_sofer), vezi figura 2.41. Se observă că descompunerea să facă fără pierdere de informație, dar a fost pierdută dependența funcțională $(\text{cod_cursă}, \text{cod_sofer}) \rightarrow (\text{cod_autobuz})$.

TRANSPORTURI_BC			
cod_cursă	cod_autobuz	loc_plecare	loc_sosire
C1	A1	Constanța	București
C1	A2	București	Brașov
C2	A2	Cluj	Sibiu
C2	A3	Sibiu	București
C3	A1	București	Constanța

AUTOBUZ	
cod_autobuz	cod_sofer
A1	S1
A2	S2
A3	S1

Figura 2.41

2.3.5. A patra formă normală (4NF – Fourth Normal Form)

Dacă BCNF elimină redundanțele datorate dependențelor funcționale, 4NF determină redundanțele datorate dependențelor multivaloare. Pentru a ilustra acest tip de redundanțe, s-

considerăm tabelul ANGAJATI (cod_angajat, limba_straină, mașina) aflat în BCNF, vezi figura 2.42. Un angajat poate cunoaște mai multe limbi străine și poate avea mai multe mașini, dar nu există nici o legătură între limba străină și mașină. Cu alte cuvinte, redundanța datelor din tabelul ANGAJATI este cauzată de existența a două relații N:M independente, vezi figura 2.43. A patra formă normală va înălțura aceste relații N:M independente.

ANGAJATI		
cod_angajat	limba_straină	mașina
A1	Engleză	Ford Mondeo
A1	Franceză	Ford Mondeo
A1	Italiană	Ford Mondeo
A1	Engleză	Skoda Octavia
A1	Franceză	Skoda Octavia
A1	Italiană	Skoda Octavia
A2	Engleză	Ford Mondeo
A2	Franceză	Ford Mondeo

Figura 2.42.

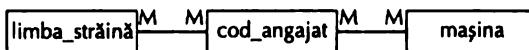


Figura 2.43

Pentru a defini mai riguroș constrângările impuse de 4NF dăm întâi următoarea definiție. Fie R un tabel relațional, X și Y două submulțimi de coloane ale lui R și Z = R - X - Y mulțimea coloanelor din R care nu sunt nici în X nici în Y. Spunem că există o *dependență multivaloare* Y de X sau că X *determină multivaloare* pe Y, și notăm $X \rightarrow\rightarrow Y$, dacă, pentru orice valoare a coloanelor lui X, sunt asociate valori pentru coloanele din Y care nu sunt corelate în nici un fel cu valourile coloanelor lui Z. Cu alte cuvinte $X \rightarrow\rightarrow Y$ dacă și numai dacă oricare ar fi u și v două rânduri ale lui R cu $u(X) = v(X)$, există s și t două rânduri ale lui R astfel încât $s(X) = u(X)$, $s(Y) = v(Y)$, $s(Z) = v(Z)$ și $t(X) = u(X)$, $t(Y) = v(Y)$, $t(Z) = u(Z)$ (vezi figura 2.44), unde prin $u(X)$ suntem notat valoarea coloanelor X corespunzătoare rândului u , etc. În mod evident, dacă $X \rightarrow\rightarrow Y$ atunci și $X \rightarrow\rightarrow Z$. Dependența multivaloare se mai numește și *multidependență*.

	X	Y	Z
u	x	y1	z1
v	x	y2	z2
s	x	y1	z2
t	x	y2	z1

Figura 2.44.

De exemplu, în tabelul ANGAJATI avem dependențele multivaloare "cod_angajat" $\rightarrow\rightarrow$ "limba_straină" și "cod_angajat" $\rightarrow\rightarrow$ "mașina". Pe de altă parte însă, nici "limba_straină" și nici "mașina" nu depinde funcțional de "cod_angajat".

Orice dependență funcțională este și o dependență multivaloare, dar afirmația reciprocă nu este în general adeverată, după cum demonstrează contraexemplul de mai sus. Un tabel care se află în BCNF și pentru care această afirmație reciprocă este adeverată se află în 4NF.

Definiție 4NF

Un tabel relațional R este în a patra formă normală (4NF) dacă și numai dacă:

- R este în BCNF;
- Orice dependență multivaloare $X \rightarrow\rightarrow Y$ este de fapt o dependență funcțională $X \rightarrow Y$.

Condiția a două arată că dacă există o dependență multivaloare $X \rightarrow\rightarrow Y$, atunci orice coloană A a lui R va depinde funcțional de coloanele din X, $X \rightarrow A$ - aceasta deoarece existența unei dependențe multivaloare $X \rightarrow\rightarrow Y$ implică și existența unei dependențe multivaloare $X \rightarrow\rightarrow (R - X - Y)$. Înțând cont de faptul că R este în BCNF, înseamnă că există o cheie candidată a lui R inclusă în X. Deci definiția de mai sus se poate reformula astfel:

Definiție 4NF

Un tabel relațional R este în a patra formă normală (4NF) dacă și numai dacă pentru orice dependență multivaloare $X \rightarrow\rightarrow Y$ există o cheie a lui R inclusă în X.

Un tabel poate fi adus în 4NF prin descompunere fără pierdere de informație după următoarea regulă. Fie R(X, Y, Z) un tabel relațional în care există o dependență multivaloare $X \rightarrow\rightarrow Y$ astfel încât X nu conține nici o cheie a lui R. Atunci tabelul R poate fi descompus prin proiecție în două tabele R1(X, Y) și R2(X, Z).

Aplicând această regulă tabelul ANGAJATI se descompune în tabelele ANGAJATI_4A (cod_angajat, limba_straină), ANGAJATI_4B (cod_angajat, mașina), vezi figura 2.45, aflate amândouă în 4NF.

ANGAJATI_4A	
cod_angajat	limba_straină
A1	Engleză
A1	Franceză
A1	Italiană
A2	Engleză
A2	Franceză

ANGAJATI_4B	
cod_angajat	mașina
A1	Ford Mondeo
A1	Skoda Octavia
A2	Ford Mondeo

Figura 2.45

Aplicând recursiv această regulă, algoritmul 4NFA permite aducerea unui tabel relațional din BCNF în 4NF prin eliminarea dependențelor multivaloare.

Algoritmul 4NFA

- Se identifică dependențele multivaloare $X \rightarrow\!\!\!\rightarrow Y$ pentru care X și Y nu conțin toate coloanele lui R și X nu conține nici o cheie a lui R . Se poate presupune că X și Y sunt disjuncte datorită faptului că din $X \rightarrow\!\!\!\rightarrow Y$ rezultă $X \rightarrow\!\!\!\rightarrow (Y - X)$.
- Se înlocuiește tabelul inițial R cu două tabele, primul format din coloanele $[X, Y]$, iar celălalt din toate coloanele inițiale, mai puțin coloanele Y .
- Dacă tabelele rezultate conțin alte dependențe multivaloare, atunci se face transfer la pasul 1, altfel algoritmul se termină.

2.3.6. A cincea formă normală (5NF – Fifth Normal Form)

A cincea formă se întâlnește destul de rar în practică, ea având mai mult valoare teoretică. Dacă în prima formă normală sunt eliminate relațiile $N:M$ independente, a cincea formă normală are ca scop eliminarea relațiilor $N:M$ dependente. Redundanțele datorate unor astfel de relații pot fi înălțurate prin descompunerea tabelului în 3 sau mai multe tabele. Pentru a ilustra acest tip de redundanță, să considerăm tabelul LUCRĂTOR_ATELIER_PRODUS (cod_lucrător, cod_atelier, cod_produs), aflat în 4NF. Aparent acest tabel este ilustrarea unei relații de tip 3 care există între lucrător, atelier și produs. Dacă însă presupunem că între lucrător și atelier, lucrător și produs, atelier și produs există relații $N:M$, vezi figura 2.47, atunci în tabel pot exista redundanțe în date - exemplul (L2, A1), (L2, P1) și (A1, P1) apar de 2 ori, vezi figura 2.46, care pot fi înălțurate prin descompunerea tabelului LUCRĂTORI în 3 tabele, LUCRĂTOR_ATELIER (cod_lucrător, cod_atelier), LUCRĂTOR_PRODUS (cod_lucrător, cod_produs) și ATELIER_PRODUS (cod_atelier, cod_produs), vezi figura 2.48. Se observă că tabelul LUCRĂTOR_ATELIER elimină redundanța (L2, A1), tabelul LUCRĂTOR_PRODUS elimină redundanța (L2, P1), în timp ce tabelul ATELIER_PRODUS elimină redundanța (A1, P1). De asemenea, se poate vedea că tabelul inițial LUCRĂTOR_ATELIER_PRODUS nu poate fi reconstituit din compunerea a doar două din tabelele componente, vezi figura 2.49, unde tabelul R_12 rezultă din compunerea tabelelor LUCRĂTOR_ATELIER și LUCRĂTOR_PRODUS, R_13 rezultă din compunerea tabelelor LUCRĂTOR_ATELIER și ATELIER_PRODUS, iar R_23 rezultă din compunerea tabelelor LUCRĂTOR_PRODUS și ATELIER_PRODUS. Pe altă parte, tabelul inițial LUCRĂTOR_ATELIER_PRODUS poate fi obținut prin compunerea tuturor celor trei tabele componente, de exemplu el rezultă prin compunerea lui R_12 cu ATELIER_PRODUS.

LUCRĂTOR_ATELIER_PRODUS		
<u>cod_lucrător</u>	<u>cod_atelier</u>	<u>cod_produs</u>
L1	A1	P1
L2	A1	P2
L2	A1	P1
L2	A2	P1

Figura 2.46

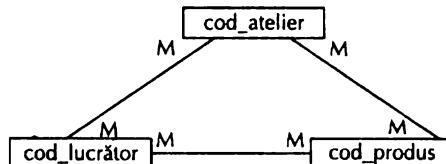


Figura 2.47

LUCRĂTOR_ATELIER	
cod_lucrător	cod_atelier
L1	A1
L2	A1
L2	A2

LUCRĂTOR_PRODUS	
cod_lucrător	cod_produs
L1	P1
L2	P2
L2	P1

ATELIER_PRODUS	
cod_atelier	cod_atelier
A1	P1
A1	P2
A2	P1

Figura 2.48

R_12

cod_lucrător	cod_atelier	cod_produs
L1	A1	P1
L2	A1	P2
L2	A1	P1
L2	A2	P2
L2	A2	P1

R_13

cod_lucrător	cod_atelier	cod_produs
L1	A1	P1
L1	A1	P2
L2	A1	P2
L2	A1	P1
L2	A2	P1

R_23

cod_lucrător	cod_atelier	cod_produs
L1	A1	P1
L1	A2	P1
L2	A1	P2
L2	A1	P1
L2	A2	P1

Figura 2.49

Dependența funcțională și dependența multivaloare, și implicit regulile de descompunere pentru formele normale 1NF-4NF, permit descompunerea prin proiecție a unui tabel relațional în *două* tabele relaționale. Totuși, regulile de descompunere asociate acestor forme normale nu dau toate descompunerile posibile prin proiecție a unui tabel relațional. Există tabele care nu pot fi descompuse în două tabele, dar pot fi descompuse în trei sau mai multe tabele fără pierdere de informație. Astfel de descompuneri, în trei sau mai multe tabele, sunt tratate de 5NF. Pentru a arăta că un tabel se poate descompune fără pierderi de informație a fost introdus concepțul de *join-dependență* sau *dependență la compunere*, definit în cele ce urmează.

Fie R un tabel relațional și R_1, R_2, \dots, R_n o mulțime de tabele relaționale care nu sunt disjuncte (au coloane comune) astfel încât reunirea coloanelor din R_1, R_2, \dots, R_n este mulțimea coloanelor din R. Se spune că R satisface join-dependență $*\{R_1, R_2, \dots, R_n\}$ dacă R se descompune prin proiecție pe R_1, R_2, \dots, R_n , fără pierdere de informație, adică tabelul inițial poate fi reconstruit prin compunerea naturală pe atrbute comune ale tabelelor rezultante. În exemplul de mai sus, tabelul LUCRATOR_ATELIER_PRODUS satisface dependență de uniune $*\{LUCRATOR_ATELIER, LUCRATOR_PRODUS, ATELIER_PRODUS\}$.

Join-dependență este o generalizare a dependenței multivaloare. Mai precis, dependența multi-valoare corespunde join-dependenței cu două elemente. Într-adevăr, dacă în relația $R(X, Y, Z)$ avem multidependență $X \rightarrow\rightarrow Y$, atunci avem și join-dependență $*\{(X \cup Y), (X \cup Z)\}$. Invers, dacă avem join-dependență $*\{R_1, R_2\}$, atunci avem și dependență multivaloare $(R_1 \cap R_2) \rightarrow\rightarrow (R_1 - (R_1 \cap R_2))$.

Definiție 5NF.

Un tabel relațional R este în a cincea formă normală (5NF) dacă și numai dacă orice join-dependență $*\{R_1, R_2, \dots, R_n\}$ este consecință cheilor candidate ale lui R, adică fiecare dintre R_1, R_2, \dots, R_n include o cheie candidată a lui R.

Orice tabel relațional care este în 5NF este în 4NF deoarece, aşa cum am arătat mai sus, orice dependență multivaloare poate fi privită ca un caz particular de dependență la uniune. Trecerea de la 4NF la 5NF constă în identificarea join-dependențelor cu mai mult de trei elemente și descompunerea tabelului inițial prin proiecție pe aceste componente. După cum am mai spus, 5NF are o importanță practică redusă, cazurile când apare în practică fiind extrem de rare.

Orice tabel relațional poate fi descompus fără pierderi de informație într-o mulțime de tabele relaționale care sunt în 5NF. Se garantează faptul că un tabel în 5NF nu conține anomaliile ce pot fi eliminate luând proiecțiile pe diferite submulțimi de coloane ale acestuia.

Concluzii

Normalizarea este procesul de transformare a structurilor de date și are ca scop eliminarea redundanțelor și promovarea integrității datelor. Normalizarea este un pilon de bază al bazelor de date relaționale. În general, un set de structuri de date nu sunt considerate relaționale decât dacă sunt complet normalizate.

Normalizarea datelor este împărțită în șase etape, numite forme normale. Fiecare formă normală are asociat atât un criteriu cât și un proces. Criteriul unei anumite forme normale este mai restrictiv decât al formei normale inferioare, astfel încât orice tabel relațional care este într-o anumită formă normală este și în formă normală inferioară. Procesul asociat unei forme normale se referă la trecerea unor structuri de date din forma normală inferioară în forma normală curentă. Etapele normalizării sunt rezumate pe scurt în continuare.

- 1NF \rightarrow 2NF elimină dependențele funcționale parțiale față de chei;
- 2NF \rightarrow 3NF elimină dependențele funcționale tranzitive față de chei;
- 3NF \rightarrow BCNF elimină dependențele funcționale pentru care determinantul nu este cheie;

- BCNF → 4NF elimină toate dependențele multivaloare care nu sunt și dependențe funcționale;
- 4NF → 5NF elimină toate join-dependențele care nu sunt implicate de o cheie.

În practică, cel mai adesea apar primele forme normale (1NF - 3NF), astfel încât un tabel relațional aflat în 3NF este de obicei complet normalizat. În special 5NF apare extrem de rar, având valoare practică foarte scăzută.

2.3.7. Denormalizare

În principiu, denormalizarea este exact procesul invers normalizării. Denormalizarea este procesul de creștere a redundanței datelor, care are ca scop creșterea performanței sau simplificarea programelor de interogare a datelor. Totuși, trebuie reținut că o denormalizare corectă nu înseamnă nicidcum a nu normaliza structurile de date inițiale. Din contră, denormalizarea are loc după ce structurile bazice de date au fost complet normalizate, și se face prin selectarea strategică a celor structuri unde denormalizarea aduce avantaje semnificative. Pe de altă parte, orice denormalizare trebuie însoțită de introducerea de măsuri suplimentare, care să asigure integritatea datelor chiar și în cazul unei baze de date care nu este complet nenormalizată.

Creșterea performanței

Principalul obiectiv al denormalizării este creșterea performanței programelor de integrare a datelor. Una dintre cele mai întâlnite situații de acest gen este denormalizarea folosită pentru operații sau calcule efectuate frecvent. Ilustrăm această situație cu exemplul următor. Să presupunem că pentru înregistrarea tranzacțiilor unui magazin care vinde articole la comandă se folosesc tabelele:

VÂNZĂRI_2 (cod_comandă, cod_articol, cantitate),
ARTICOL (cod_articol, nume_articol, cost_articol),
COMANDA_3 (cod_comandă, data, cod_client),
CLIENT (cod_client, nume_client, nr_telefon)

descrise mai sus, vezi figurile 2.29, 2.30 și 2.31. Aceste tabele sunt în 5NF și conțin toate datele necesare pentru obținerea oricărui raport privind vânzările din magazin - de exemplu, în funcție de articol, dată, cantitate, client etc. Totuși, să presupunem că majoritatea rapoartelor cerute de conducerea magazinului utilizează cantitatea totală vândută într-o lună pentru fiecare articol. Tabelele de mai sus conțin toate datele necesare pentru obținerea acestei informații, de exemplu ea se poate obține printr-o simplă interogare SQL. Pe de altă parte însă, aceasta ar însemna că totalul pentru fiecare articol să fie recalculat de fiecare dată când este cerut. Deoarece este puțin probabil ca acest total să se schimbe după încheierea lunii respective, repetarea acestora ar putea fi înlocuită cu un tabel suplimentar

ARTICOL_LUNA (cod_articol, luna, cantitate_totală)

Datele conținute în acest nou tabel sunt redundante, ele putând fi obținute în orice moment din datele conținute în tabelele VÂNZĂRI_2 și COMANDA_3, dar prezintă avantajul că folosirea lor este mai facilă și mai rapidă decât repetarea calculelor. Pe de altă parte însă, folosirea acestui tabel suplimentar are și dezavantaje, putând duce la pierderea integrității datelor. Modificarea datelor din tabelele VÂNZĂRI_2 și COMANDA_3 nu se

reflectă automat în datele din noul tabel. De exemplu, dacă după efectuarea calculelor și inserarea totalului în noul tabel se efectuează o vânzare suplimentară, această vânzare nu este reflectată în datele din noul tabel. De aceea, în situațiile în care baza de date nu este total normalizată, trebuie luate măsuri suplimentare pentru păstrarea integrității datelor. De exemplu, pentru ca orice modificare a tabelelor inițiale să fie reflectată automat în noul tabel, în Oracle se pot crea așa numitele *trigger-e* (declanșatoare) ale bazei de date bazate pe tablele VÂNZĂRI_2 și COMANDA_3 care se declanșează după fiecare modificare (INSERT, UPDATE, DELETE) a acestor tabele și care actualizează datele din tabelul ARTICOL_LUNA. Aceste trigger-e pe bază, la rândul lor, vor începe operațiile de actualizare din tabelele pe care sunt bazate.

Datorită problemelor legate de redundanță și integritate, orice denormalizare trebuie să fie gândită cu atenție pentru a fi siguri că avantajele acestia în privința performanțelor și a simplității programelor depășesc efortul necesar pentru impunerea integrității. În exemplul de mai sus, se pot crea tabele suplimentare pentru totaluri pe lună sau pe zi. Dacă aceste totaluri sunt utilizate rar, este mai convenabil ca ele să fie calculate direct din tabelele normalizate, fără a mai introduce alte date redundante. În concluzie, fiecare caz trebuie studiat cu atenție pentru a vedea dacă problemele inerente asociate denormalizării (redundanță, menținerea integrității) sunt compensate de creșterea în performanță pentru anumite situații.

Simplificarea codului

Un alt motiv invocat pentru folosirea denormalizării este acela al simplificării codului pentru manipularea datelor. Cu alte cuvinte, pentru un dezvoltator, un singur tabel poate părea uneori mai ușor de utilizat decât două sau mai multe. De exemplu, să considerăm tabelul

STOCURI(cod_depozit, cod_material, nume_material, cantitate)

utilizat de o firmă pentru a înregistra cantitățile de materiale existente în fiecare din depozitele sale. Evident, acest tabel nu este în 2NF și poate fi normalizat prin descompunerea în două tabele:

STOCURI_2(cod_depozit, cod_material, cantitate)

MATERIAL(cod_material, nume_material)

Totuși, dacă se dorește aflarea tuturor depozitelor în care există ciment, de exemplu, pentru dezvoltator este mai convenabil să folosească varianta nenormalizată, în acest caz interogarea SQL corespunzătoare fiind:

```
SELECT cod_depozit, cantitate
FROM stocuri
WHERE nume_material = 'CIMENT';
```

Evident, interogarea de mai sus este mai simplă decât varianta în care se folosesc cele două tabele normalize:

```
SELECT cod_depozit, cantitate
FROM stocuri_2, material
WHERE stocuri_2.cod_material = material.cod_material
AND nume_material = 'CIMENT';
```

Pe de altă parte însă, diferența de performanță (timp de execuție) dintre cele două variante de interogări este neglijabilă, de multe ori chiar interogarea pe structuri normalize va fi chiar mai rapidă decât cealaltă, astfel încât singurul avantaj al folosirii structurii nenormalizate este simplitatea.

58 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms8

Dacă pentru un dezvoltator simplificarea invocată mai sus este nesemnificativă - orice dezvoltator trebuie să fie capabil să scrie o interogare pe mai multe tabele de tipul celei de mai sus - problema se pune cu mai mare stringență în cazul în care utilizatorii pot să-și scrie propriile interogări ale bazei de date. În acest caz, pentru a păstra atât avantajele normalizării cât și pentru a permite în același timp simplificarea interogărilor, se poate crea o vedere bazată pe cele două tabele normalize:

```
CREATE VIEW stocuri AS
SELECT cod_depozit,stocuri_2.cod_material,nume_material,cantitate
FROM stocuri_2, material
WHERE stocuri_2.cod_material = material.cod_material;
```

O astfel de vedere "ascunde" tabelele normalize și permite utilizatorilor scrierea unor interogări mai simple. Pe de altă parte însă, performanța programelor poate scădea mult în cazul vederilor complexe, în acest caz fiind uneori de preferat a se utiliza tabelele de bază pentru creșterea performanței.

Concluzii

Design-ul logic al bazei de date are un impact profund asupra performanțelor și structurii sistemului. Deși este posibil să compenseze un design prost prin scrierea unui volum mare de cod, acest lucru nu este în nici un caz de dorit, mai ales că în majoritatea cazurilor el va duce inevitabil la scăderea performanțelor sistemului. În mod ideal, dezvoltarea oricărei aplicații trebuie să înceapă de la o structură complet normalizată, care să constituie o temelie solidă a acesteia. Denormalizarea, dacă ea este necesară, trebuie făcută numai după ce se obține o structură complet normalizată. Ca regulă generală, denormalizarea este folosită rar și numai după o analiză atentă.

Capitolul 3

Arhitectura SGBDR Oracle

Arhitectura Oracle descrisă în această secțiune este o arhitectură generică, independentă de platformele pe care aceasta poate rula. Între diferite platforme pot apărea însă unele diferențe privind detalii de configurație și folosire a sistemului Oracle. Atunci când vom discuta despre astfel de detalii, ne vom referi la cele specifice sistemului de operare Windows NT, datorită popularității acestuia.

SGBDR Oracle constă din două elemente fundamentale: *baza de date* și *instanța*. Baza de date este o colecție de date și are la rândul ei un *nivel fizic* și un *nivel logic*.

Nivelul fizic constă dintr-un set de fișiere localizate pe hard disk. Aceste fișiere sunt niște fișiere binare ce pot fi accesate numai prin utilizarea software-ului nucleului Oracle.

Din punct de vedere *logic*, baza de date este împărțită în conturi ale utilizatorilor. Fiecare dintre aceste conturi este identificat printr-un nume și o parolă unică pentru baza de date respectivă. Un cont de utilizator este proprietarul unor tabele sau al altor obiecte (tabele, vederi, indecsi, secvențe, sinonime) ale bazei de date. Obiectele schemei și relațiile dintre ele formează design-ul relațional al unei baze de date.)

În momentul când un utilizator dorește să acceseze datele conținute în baza de date, nu poate face acest lucru decât introducând un nume de utilizator și o parolă validă, adică practic este necesară conectarea sa la baza de date cu ajutorul unui cont. Conectarea la baza de date este necesară în momentul când se dorește utilizarea diferitelor instrumente Oracle. În general, la lansarea în execuție a unui instrument Oracle, va apărea o fereastră ce va solicita numele și parola utilizatorului. Numele utilizatorului și parola de conectare la baza de date este diferită de numele și parola utilizatorului sistemului de operare sub care rulează Oracle.

În două conturi diferite de utilizator poate exista același nume de tabel (sau orice alt obiect al bazei de date). Deși tabelele au același nume, datorită faptului că se găsesc în conturi diferite, ele diferă prin structură sau conținut. Un tabel nu poate avea decât un singur proprietar. Prin urmare, un tabel (sau orice alt obiect al bazei de date) este identificat prin nume și prin contul de utilizator care este proprietarul tabelului. Dacă se dorește ca și alt cont de utilizator să folosească acel tabel, proprietarul tabelului trebuie să acorde anumite drepturi.

De multe ori, aceeași bază de date (același set de fișiere fizice) este folosită pentru a păstra în conturi separate diferite versiuni ale tabelelor. În acest sens pot exista versiuni ale tabelelor pentru dezvoltatorii de aplicații, pentru cei care testează cum funcționează sistemul sau anumite aplicații, etc. Se poate utiliza două conturi de utilizator pentru a păstra datele necesare a două aplicații complet separate. Cu alte cuvinte, pot exista două baze de date logice implementate în aceeași bază de date fizică, utilizând două conturi de utilizator.

Notă: Nu trebuie confundată noțiunea de cont al unui utilizator, cu utilizatorii efectivi ai bazei de date. Practic pot exista mai mulți utilizatori care să se conecteze la un anumit moment la baza de date folosind același cont de utilizator. Deoarece în terminologia folosită de către Oracle un "cont de utilizator" este denumit "utilizator" vom folosi și noi în continuare această denumire, semnificația acestui termen fiind determinată de contextul în care va fi folosit.

Corespondența dintre structurile nivelului logic și modul lor de stocare la nivel fizic se realizează prin intermediul *structurilor logice de stocare*. Scopul principal al unei structuri logice de stocare este de a defini modul în care va fi utilizat spațiul fizic de stocare a bazei de date. Practic, Oracle dictează modul în care este utilizat spațiul fizic al bazei de date.

Instanța este metoda folosită de către sistem pentru a accesa baza de date și constă dintr-un set de procese de fundal necesare pentru susținerea bazei de date și un segment de memorie a sistemului alocat bazei de date.

În continuare ne vom referi pe scurt la modul în care se creează o bază de date Oracle la instalarea SGBD-ului Oracle Server precum și la componentele nivelului fizic al bazei de date, adică fișierele propriu-zise ale bazei de date și fișierele adiționale. Apoi vom trece succint în revistă conceptul de instanță a bazei de date, legătura dintre o instanță și o bază de date, pornirea și oprirea unei baze de date. Deoarece crearea unei baze de date presupune cunoașterea mai multor concepții Oracle, ultimul paragraf din acest subcapitol va conține modul efectiv de creare în mod manual a unei baze de date. Pentru cei interesați, informații suplimentare pot fi găsite în [2], [23]. Structurile logice de stocare, utilizatorii și nivelul logic al bazei de date vor fi prezentate mai târziu în capituloare separate. O listă a principalelor comenzi care se pot efectua asupra bazei de date și a utilităților ce pot fi folosite pentru executarea acestora se găsește în Anexa 2.

3.1. Crearea unei baze de date

La instalarea SGBD-ului Oracle8 Server pe o platformă Windows NT se poate crea și o bază de date. În acest sens există următoarele opțiuni:

- *Typical* – se creează o bază de date exemplu, care poate fi folosită ca model la crearea unor noi baze de date. Acesta este opțiunea cea mai convenabilă și mai des folosită.
- *Custom* – se lansează în execuție instrumentul *Oracle Database Assistant* ce permite personalizarea parametrilor de configurare pentru baza de date ce urmează a fi creată.
- *None* – nu se creează nici o bază de date.

Baza de date poate fi creată nu numai în momentul instalării ci și după aceea (acest lucru este neapărat necesar dacă se instalează Oracle Server fără a se crea o bază de date, folosind opțiunea "None"). Acest lucru se poate realiza cu ajutorul comenzi CREATE DATABASE sau cu ajutorul instrumentului *Oracle Database Assistant*, instrument specific numai platformei Windows NT. În cazul folosirii comenzi CREATE DATABASE se înai spune că baza de date a fost creată *manual*.

Baza de date creată prin opțiunea "Typical" are o configurație predefinită referitoare la numele utilizatorilor, identificatorii de sistem, spațiile tabel, fișierele de date, parametrii de inițializare, fișierele de control, fișierele jurnal pentru recuperare, segmentele de revenire, etc. Despre toate aceste opțiuni vom vorbi în paragrafele următoare, menționând caracteristicile pe

care le are baza de date instalată prin opțiunea "Typical", pe care o vom numi în continuare *bază de date exemplu*. Dacă baza de date exemplu este instalată pentru prima oară pe un calculator, identificatorul de sistem SID (System IDentifier) precum și numele bazei de date este ORCL. Pentru fiecare instalare adițională creată pe același calculator, identificatorul SID și numele bazei de date devine ORC0, ORC1, etc.

Utilitarul *Oracle Database Assistant* permite utilizatorului crearea, modificarea sau stergerea unei baze de date prin intermediul unei interfețe grafice ușor de utilizat. La rândul său, și acest utilitar oferă cele două opțiuni (Typical și Custom) pentru crearea unei baze de date. Față de comanda CREATE DATABASE, *Oracle Database Assistant* are atât avantajul simplității interfeței grafice, cât și faptul că odată cu baza de date, prin intermediul acestui utilitar se pot crea și componente ale acesteia precum spații tabel și segmente de rollback.. Evident însă că *Oracle Database Assistant* reprezintă doar o interfață grafică folosită pentru generarea unor comenzi SQL (CREATE DATABASE, CREATE TABLESPACE, CREATE ROLLBACK SEGMENT, etc.). De fapt, acest utilitar oferă atât posibilitatea creării imediate a bazei de date cât și posibilitatea creării doar a unui fișier batch și a fișierelor SQL necesare și care pot fi folosite mai târziu pentru a crea baza de date.

Deoarece crearea unei baze de date presupune cunoașterea componentelor nivelului fizic al bazei de date, a conceptului de instanță a bazei de date, a legăturii dintre o instanță și o bază de date, a modului de pornire și oprire a unei baze de date, în continuare ne vom referi la toate acestea, urmând ca ultima secțiune din acest capitol să conțină modul efectiv de creare manuală a unei baze de date (vezi secțiunea 3.7).

3.2. Nivelul fizic al bazei de date

Așa cum s-a menționat la începutul acestui capitol, la nivel fizic baza de date Oracle este alcătuită din mai multe fișiere. Aceste fișiere se pot împărți în două categorii:

- *fișierele propriu-zise ale bazei de date* - conțin datele utilizatorilor;
- *fișierele adiționale* - conțin informații necesare pentru ca baza de date să funcționeze corespunzător.

Fișierele propriu-zise ale bazei de date Oracle:

1) Fișierele de date

Toate datele din baza de date Oracle sunt stocate în fișierele de date. Deci acestea conțin toate obiectele bazei de date (tabele, indecsi, sevențe, segmente de rollback, segmente temporare, dicționarul de date, etc.). Aceste fișiere sunt de obicei de dimensiune mare, în funcție de informațiile conținute în baza de date. Fișierele de date Oracle există ca fișiere ale sistemului de operare și pot fi manipulate la fel ca orice alte fișiere ale acestuia. Datele stocate în aceste fișiere sunt într-un format binar astfel încât ele nu pot fi citite decât de SGBDR-ul Oracle. O bază de date Oracle are cel puțin un fișier de date (adecvat pentru o bază de date mică), dar în mod uzual există mai multe fișiere. Mai multe informații despre modul de stocare a datelor sunt prezentate în capitolul următor.

Baza de date exemplu, instalată cu opțiunea "Typical", conține patru fișiere de date localizate în directorul \$ORACLE_HOME\DATABASE. Aceste fișiere sunt: sys1SID.ora, usr1SID.ora, tmp1SID.ora, rbs1SID.ora, unde SID este un sir de caractere ce reprezintă identificatorul de sistem.

2) Fișierele de control

Orice bază de date trebuie să aibă cel puțin un fișier de control. Fișierul de control reprezintă punctul central al bazei de date. El conține numele bazei de date, data și ora la care a fost creată, informații vitale pentru pornirea bazei de date, precum localizarea și starea fișierelor de date și a fișierelor redo log. Majoritatea parametrilor conținuți în fișierul de control iau valori în timpul creării bazei de date și sunt relativ stabili, în sensul că nu se schimbă de la o zi la alta. Fișierul de control este binar și nu poate fi citit sau editat manual. Având în vedere că fără un fișier de control corect baza de date nu poate fi pornită, acest fișier trebuie protejat. Oracle asigură un mecanism de stocare a mai multor fișiere de control, astfel încât majoritatea bazelor de date Oracle operează cu fișiere de control multiple.

Baza de date exemplu, instalată cu opțiunea "Typical", conține un singur fișier de control localizat în directorul \$ORACLE_HOME\DATABASE. Acest fișier are denumirea ct11SID.ora.

3) Fișierele jurnal pentru recuperare (redo log files)

Orice bază de date conține cel puțin două fișiere jurnal pentru recuperare. Acestea sunt fișiere care păstrează informații folosite în cazul unei defecțiuni de funcționare a sistemului. Ele conțin un jurnal al tuturor schimbărilor făcute asupra bazei de date. Aceste informații sunt utilizate în cazul unei erori în funcționarea sistemului în scopul de a efectua din nou schimbările care au fost făcute asupra bazei de date și care au fost apoi permanentizate (committed) dar care, datorită defecțiunii de funcționare, nu au fost făcute și asupra fișierelor de date corespunzătoare. Fișierele redo log trebuie protejate împotriva defecțiunilor hardware; dacă informațiile conținute în acestea sunt pierdute, nu mai este posibilă recuperarea sistemului.

Fișierele jurnal pentru recuperare se mai numesc și *fișiere jurnal pentru recuperare online (online redo log files)*. În momentul în care o instanță Oracle este pornită se utilizează întotdeauna două sau mai multe fișiere redo log. Modificările făcute sunt înregistrate pe rând în aceste fișiere. Astfel, în momentul în care unul din fișiere este plin, modificările se înregistrează în fișierul următor, s.a.m.d. Când toate fișierele sunt pline, primul fișier este rescris, ciclul continuând.

Baza de date exemplu, instalată cu opțiunea "Typical", conține patru fișiere jurnal pentru recuperare online ce au dimensiunea de 1 MB fiecare și sunt localizate în directorul \$ORACLE_HOME\DATABASE. Aceste fișiere se numesc log1SID.ora, log2SID.ora, log3SID.ora, log4SID.ora.

Fișiere adiționale:

1) Fișierul de inițializare a parametrilor

Fișierul de inițializare a parametrilor reprezintă principalul mijloc de configurare a SCBDR-ului. El este pur și simplu o colecție de valori ale unor parametri, care fiecare controlează sau modifică un anumit aspect din modul de funcționare al unei bazei de date și

instanțe. Fișierul de inițializare este un fișier ASCII care poate fi editat de către administratorul bazei de date pentru optimizarea performanțelor acesteia. Numele atribuit în mod implicit fișierului de inițializare a parametrilor este `initSID.ora`, unde SID este identificatorul de sistem pentru baza de date pe care o controlează (de exemplu, pentru SID test, numele fișierului de inițializare va fi `inittest.ora`). Fișierul de inițializare este citit înainte de pornirea bazei de date; schimbările făcute parametrilor în acest fișier nu sunt luate în considerare până când instanța nu este închisă și re-pornită. Totuși, unele dintre parametrii de inițializare au caracter *dinamic*, adică pot fi modificați în timp ce o instanță este pornită. Efectul schimbării acestor parametrii va fi imediat, nefiind necesară închiderea și re-pornirea instanței. Schimbarea parametrilor în mod dinamic se poate face cu ajutorul comenziilor `ALTER SESSION`, `ALTER SYSTEM` sau `ALTER SYSTEM DEFERRED`.

Tabelul următor conține o descriere succintă a parametrilor cei mai importanți. Despre unele dintre parametri vom mai discuta și în secțiunile următoare.

Nume Parametru	Descriere
<code>BACKGROUND_DUMP_DEST</code>	Locația unde vor fi plasate fișierele istoric create de procesele de fundal (inclusiv fișierul <code>alert.log</code>)
<code>COMPATIBLE</code>	Versiunea cu care este compațibilă această instanță. Acest lucru previne utilizarea unor facilități ale bazei de date specifice unor versiuni mai recente decât valoarea acestui parametru.
<code>CONTROL_FILES</code>	Numele fișierelor de control. Dacă acest parametru nu se schimbă, fișierele de control din alte baze de date pot fi rescrise de nouă instanță, făcând celelalte instanțe inutilizabile.
<code>DB_BLOCK_BUFFERS</code>	Numărul de blocuri din memoria SGA (Shared Global Area). Valoarea inițială și minimă în același timp este 50.
<code>DB_BLOCK_SIZE</code>	Mărimea în bytes a blocurilor bazei de date. Acest parametru nu poate fi modificat după ce baza de date a fost creată.
<code>DB_FILES</code>	Numărul maxim de fișiere ale bazei de date ce pot fi deschise simultan.
<code>DB_NAME</code>	Identificatorul bazei de date. Acest identificator este opțional, dar dacă se folosește trebuie să fie identic cu numele bazei de date folosit în comanda <code>CREATE DATABASE</code> .
<code>LOG_ARCHIVE_DEST</code>	Locația fișierelor redo log arhivate.
<code>LOG_ARCHIVE_START</code>	Activează sau dezactivează arhivarea automată. Dacă este setat pe true, procesul ARCH este pornit automat la pornirea instanței.
<code>LOG_BUFFER</code>	Numărul de bytes alocați pentru fișierele redo log în memoria SGA (Shared Global Area).
<code>MAX_DUMP_FILE_SIZE</code>	Dimensiunea maximă a fișierelor istoric, specificată ca număr de blocuri de memorie care, în funcție de sistemul de operare, pot avea diverse dimensiuni implicate.
<code>PROCESSES</code>	Numărul maxim de procese ale sistemului de operare ce se pot conecta la această instanță.
<code>ROLLBACK_SEGMENTS</code>	Lista segmentelor de revenire care vor fi activate (online) automat la deschiderea bazei de date.

Unitarea aplicațiilor de baze de date în Oracle8 și Forms6

Nume Parametru	Descriere
SHARED_POOL_SIZE	Mărimea în bytes a unei anumite zone de memorie din SGA. Valoarea implicită este 3.500.000.
SQL_TRACE	Activează sau dezactivează crearea unor fișiere istoric suplimentare.
USER_DUMP_DEST	Locația fișierelor istoric suplimentare create la cererea utilizatorului.

Pentru baza de date exemplu, fișierul de inițializare se găsește în directorul \$ORACLE_HOME\DATABASE.

Parametrii care nu sunt specificați în fișierul de inițializare iau în mod automat valori implicate. Pentru a putea afla valoarea unui parametru se poate folosi utilitarul Server Manager versiunea 3.0. Acest utilitar se lansează în execuție din prompterul MS-DOS:

```
C:\> SVRMGR30
```

Comanda care afișează valoarea unui parametru este următoarea:

```
SVRMGR> SHOW PARAMETER nume_parametru
```

Unde nume_parametru este numele parametrului a cărui valoare se dorește a se afișa. De exemplu:

```
SVRMGR> SHOW PARAMETER DB_FILES
```

2) Fișiere jurnal arhivate pentru recuperare (archived redo log files)

Fișierele jurnal pentru recuperare arhivate (denumite și *offline redo log files*) sunt copii ale fișierelor redo log care au fost umplute cu informații referitoare la modificările înregistrate. Această facilitate este optională. Pentru a putea activa această facilitate se pot folosi două metode: prin setarea parametrului LOG_ARCHIVE_START din fișierul de inițializare cu valoarea TRUE sau în mod dinamic prin folosirea comenzi ALTER SYSTEM cu clauza ARCHIVE LOG START după ce instanța a fost pornită. Când arhivarea automată este activată, procesul ARCH (cel care arhivează) copiază fișierele redo log completate în directorul specificat de parametrul de inițializare LOG_ARCHIVE_DEST. Numele fișierelor redo log arhivate este dat de parametrul de inițializare LOG_ARCHIVE_FORMAT.

Dacă există un set complet de fișiere redo log arhivate aferente modificărilor făcute unei baze de date, precum și baza de date inițială, în cazul în care fișierele de date sunt corupte se pot reaplica schimbările făcute bazei de date astfel încât să se recupereze datele pierdute.

3) Fișiere istoric (trace files)

Bazele de date Oracle au cel puțin un fișier în care sunt scrise toate mesajele, erorile și evenimentele importante. Acesta poartă numele SIDalert.log, unde SID este identificatorul de sistem pentru baza de date. Acest fișier (numit și "alert file") este primul care trebuie cercetat în cazul când se investighează o defecțiune a bazei de date. El conține un istoric al mesajelor privind defecțiunile importante ale bazei de date, pornirea și oprirea acesteia cât și alte evenimente importante. Dacă fișierul "alert" nu există, acesta se creează automat în momentul pornirii bazei de date.

Pe lângă acest fișier istoric mai există și alte fișiere create automat de către procesele de fundal. Acestea poartă numele sidPROC.trc, unde sid este identificatorul de sistem

pentru baza de date iar PROC este numele procesului de fundal (DBWR, LGWR, SMON, PMON, etc.). Toate aceste fișiere istoric create în mod automat se află localizate în directorul specificat de parametrul de inițializare BACKGROUND_DUMP_DEST. Pentru baza de date exemplu fișierelor istoric se găsesc în directorul \$ORACLE_HOME\RDBMS80\TRACE.

Pe lângă fișierelor istoric create în mod obligatoriu, la cererea utilizatorului se pot crea fișiere istoric suplimentare. Această facilitate este activată prin setarea parametrului de inițializare SQL_TRACE. Acest parametru poate fi modificat și în mod dinamic prin folosirea comenzii ALTER SESSION. Numele acestor fișiere este oraSECV.trc, unde SECV este o secvență de numere unică generată automat de către sistem. Locația unde vor fi create aceste fișierelor istoric este dată de alt parametru de inițializare: USER_DUMP_DEST. Se recomandă ca locația conținută de acest parametru să fie diferită de locația conținută de parametrul BACKGROUND_DUMP_DEST.

3.3 Instanța unei baze de date

O bază de date Oracle depozitează datele în fișiere fizice și permite accesul controlat al utilizatorului la aceste fișiere printr-un set de procese ale sistemului de operare. Aceste procese sunt activate în timpul instanței de pornire. Din cauza faptului că acestea operează fără interacțiunea directă a utilizatorului, ele sunt cunoscute sub denumirea de *procese de fundal*. Pentru a manipula eficient datele și pentru a activa comunicarea între diferite procese, Oracle folosește memoria partajată, cunoscută și sub numele de *Shared Global Area (SGA)*. Aceste procese de fundal, împreună cu segmentul memoriei partajate (SGA), constituie o instanță Oracle.

Fiecare instanță Oracle utilizează mai multe procese de fundal, dar în mod implicit orice instanță conține 5 procese de fundal. Pentru a înțelege, la modul general, ce reprezintă un proces de fundal, vom enunța în continuare funcțiile celor 5 procese de fundal implicate:

- 1) DBWR (Database Writer) – scrie în baza de date modificările efectuate asupra datelor;
- 2) LGWR (Log Writer) – înregistrează schimbările făcute în memoria partajată;
- 3) SMON (System Monitor) – verifică în primul rând consistența bazei de date și în funcție de rezultatele acestui proces inițiază procesul de recuperare al datelor în momentul deschiderii bazei de date;
- 4) PMON (Process Monitor) – derulează tranzacțiile nefăcute și eliberează resursele în cazul în care un proces eșuează;
- 5) CKPT (Checkpoint) – este responsabil de modificările privind starea bazei de date atunci când schimbările făcute în memoria partajată sunt înregistrate permanent în baza de date.

În afară de aceste procese de fundal mai există și altele cum ar fi: ARCH (Archiver Process), RECO (Recoverer Process), SNPn (Snapshot Process), LCK (Lock), etc.

3.4. Legătura dintre o bază de date și o instanță

Legătura dintre o bază de date Oracle și o instanță constă în faptul că o bază de date nu este accesibilă utilizatorilor până când nu este deschisă prin intermediul unei instanțe. Cu alte cuvinte, o instanță reprezintă calea de a accesa o bază de date Oracle. Fiecare instanță are

propriul său set de procese de fundal și propria memorie partajată (SGA). Operația de pornire a unei instanțe implică pornirea proceselor de fundal și alocarea zonei de memorie partajată. O instanță poate deschide și utiliza la un anumit moment numai o singură bază de date. Baza de date Oracle este identificată prin numele ei. Deși este posibil să se folosească un nume aferent bazei de date și alt nume aferent instanței, în cazul în care corespondența dintre instanțe și baze de date este de unu-la-unu (cum se întâmplă în general) este recomandabil ca pentru o mai bună administrare să se folosească același nume.

3.5. Pornirea unei baze de date Oracle

Utilizatorii nu au acces la datele conținute în baza de date până când aceasta nu este pornită. Pornirea unei baze de date presupune parcurgerea a trei etape succesive. În funcție de anumite operații care se execută asupra bazei de date, procesul de pornire se poate opri după oricare din aceste etape. Baza de date nu se poate porni decât dacă există drepturile necesare. Prin urmare, aceste operații trebuie făcute de către administratorul bazei de date. Pentru baza de date exemplu, numai contul de utilizator INTERNAL, cu parola ORACLE, poate porni baza de date. În continuare vom enumera etapele necesare pentru a porni o bază de date.

1) Pornirea unei instanțe

Pornirea unei instanțe este sinonimă cu pornirea unei baze de date. Pe parcursul acestei etape Oracle citește fișierul de inițializare a parametrilor, alocă memoria partajată (SGA), pornește procesele de fundal și deschide fișierele istoric.

În momentul creării unei baze de date sau în momentul când se recreează fișierele de control se parcurge numai etapa de pornire a unei instanțe, fără a se trece mai departe la etapa de montare a bazei de date.

2) Montarea bazei de date

Montarea unei baze de date presupune asocierea unei baze de date cu o instanță pornită anterior, localizarea și deschiderea fișierelor de control specificate în fișierul de inițializare a parametrilor precum și obținerea informațiilor referitoare la numele și starea fișierelor de date și a fișierelor jurnal pentru recuperare.

Dacă se dorește redenumirea fișierelor de date, activarea sau dezactivarea unor opțiuni de arhivare sau dacă se dorește recuperarea în totalitate a bazei de date, atunci procesul se va opri la etapa de montare a bazei de date, fără a se trece mai departe la etapa de deschidere efectivă a bazei de date.

3) Deschiderea bazei de date

Deschiderea bazei de date este procesul prin care baza de date este pusă la dispoziția utilizatorilor pentru operații obișnuite (baza de date este deschisă în mod normal). În această etapă se realizează deschiderea efectivă a fișierelor de date și a fișierelor jurnal pentru recuperare și se verifică consistența bazei de date.

3.6. Oprirea unei baze de date Oracle

Oprirea unei baze de date Oracle presupune parcurgerea următoarelor trei etape:

1. Închiderea bazei de date

Închiderea bazei de date determină scrierea tuturor datelor bazei de date în fișierele de date. Fișierele de control rămân deschise deoarece baza de date este încă montată.

2. Demontarea bazei de date

Demontarea bazei de date presupune disocierea bazei de date de instanță. În această etapă Oracle închide fișierele de control.

3. Închiderea instanței

Închiderea instanței presupune închiderea tuturor proceselor de fundal, eliberarea memoriei partajate, precum și închiderea fișierelor istoric.

Oprirea unei baze de date se poate realiza în patru moduri:

- Normal* – este modul implicit de închidere a unei baze de date. În timpul închiderii normale, Oracle nu mai acceptă conectarea unor noi utilizatori și așteaptă ca toți utilizatorii curenti să se deconecteze, apoi închide și demontează baza de date, după care oprește instanța. Următoarea pornire a bazei de date nu va necesita o operație de recuperare a instanței.
- Tranzacțional* – este similar cu modul de închidere *Normal*, cu diferența că deconectarea unui client se face automat după ce tranzacția curentă a acestuia cu baza de date a luat sfârșit.
- Imediat* – comenziile SQL curente nu se procesează și nu așteaptă ca utilizatorii curenti să se deconecteze, apoi închide și demontează baza de date, după care oprește instanța. Următoarea pornire a bazei de date nu va necesita o operație de recuperare a instanței.
- Abort* - comenziile SQL curente nu se procesează, nu așteaptă ca utilizatorii curenti să se deconecteze, iar tranzacțiile ce nu sunt permanentizate (committed) se derulează înapoi (rolled back). Instanța este opriță fără a se închide fișierele, fapt care va determina o operație de recuperare a instanței în momentul următoarei porniri a bazei de date.

3.7 Crearea manuală a unei baze de date

Comanda CREATE DATABASE este o comandă ce oferă o mai mare flexibilitate în comparație cu celelalte metode de creare a bazei de date deoarece oferă posibilitatea de a putea specifica anumiți parametri ce nu pot fi specificați altfel. Punctul slab este însă faptul că există o mai mare posibilitate de a apărea erori de sintaxă. Crearea manuală a unei baze de date este în general mai complicată decât prin celelalte metode, prin urmare trebuie să fie făcută după ce se acumula o anumită experiență.

Crearea manuală a unei baze de date nu presupune numai executarea comenziilor CREATE DATABASE, ci și efectuarea anumitor pași premergători. Prin urmare, pentru a crea o bază de date în mod manual trebuie efectuate următoarele operații:

1. Setarea anumitor variabile ale sistemului de operare.

Setarea acestor variabile depinde bineînțeles de sistemul de operare pe care va fi instalată baza de date.

Pentru sistemul de operare Windows NT, vor trebui efectuați următorii pași:

- 1) Setarea variabilei ORACLE_SID, variabilă ce va fi utilizată de către SVRMGR30 (Server Manager versiunea 3.0).

Parametrii precum ORACLE_HOME sau ORACLE_SID sunt variabile definite în regisztrii sistemului de operare (se mai numesc și variabile de mediu) și au următoarea semnificație:

- ORACLE_HOME – directorul unde va fi instalat SGBD-ul Oracle
- ORACLE_SID – specifică numele instanței, nume ce trebuie să fie unic între instanțele ce rulează pe aceeași mașină.

Aceste variabile se pot edita cu ajutorul utilitarului `regedit.exe`, și se găsesc în directorul HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE. În consecință, pentru a se putea crea o nouă bază de date, parametrul ORACLE_SID va trebui setat cu ajutorul comenzi:

```
C:\> set ORACLE_SID=test
```

- 2) Crearea unui serviciu și a unui fișier de parole, utilizând de exemplu următoarea comandă din utilitarul ORADIM80:

```
C:\>ORADIM80 -NEW -SID test -INTPWD baza -STARTMODE auto  
-PFILE c:\orant\database\inittest.ora
```

Comanda de mai sus creează un nou serviciu numit `test`, care este pornit automat când pornește Windows NT. Parola pentru contul intern este `baza`, iar parametrul PFILE oferă calea completă a fișierului `initSID.ora`, în cazul nostru `inittest.ora`. În urma executării acestei comenzi nu va apărea nici un mesaj referitor la modul de încheiere al comenzi, adică dacă aceasta a fost executată cu succes sau nu. Pentru a verifica dacă acest serviciu a fost pornit, se poate verifica dacă în sferastră Services din Windows NT Control Panel există serviciul OracleServiceTEST.

2. Pregătirea fișierului de inițializare a parametrilor

Pentru a pregăti fișierul de inițializare a parametrilor este recomandat să se copieze mai întâi fișierul `initSID.ora` (în cazul bazei de date exemplu acesta va fi `initORCL.ora`) cu un alt nume. În cazul exemplului nostru, acesta se va numi `inittest.ora`. Parametrii din acest fișier pot fi apoi modificați sau pot rămâne neschimbați însă trebuie neapărat schimbați următorii parametri:

- DB_NAME – identificatorul bazei de date. Acest identificator este necesar în momentul creării unei noi baze de date. Nu este necesar ca el să fie identic cu identificatorul ORACLE_SID, dar trebuie să fie identic cu numele bazei de date utilizat în comanda CREATE DATABASE. În exemplul nostru parametrul DB_NAME este denumit `test` și este identic cu identificatorul ORACLE_SID. În general cei doi parametrii sunt identici.
- CONTROL_FILES – specifică fișierle de control. Este recomandabil să se specifice cel puțin două fișiere de control situate pe discuri diferite.
- DB_BLOCK_SIZE – determină mărimea blocului bazei de date. Această dimensiune nu poate fi modificată după ce baza de date a fost creată.

3. Pornirea instanței

Pentru a porni instanța unei baze de date este necesară mai întâi conectarea la baza de date cu privilegiul SYSDBA. Conectarea la baza de date se face din utilitarul SVRMGR30 prin comanda:

SVRMGR> CONNECT INTERNAL/parola

INTERNAL nu este un utilizator special ci este de fapt un sinonim al utilizatorului SYS cu privilegiul SYSDBA. (Mai multe detalii despre utilizatorii și privilegiile acestora se găsesc în capitolul 5). Parola utilizatorului INTERNAL este cea folosită la crearea serviciului de la pasul 1 cu ajutorul utilitarului ORDIM80 (în exemplul nostru parola este baza)

După ce s-a realizat conectarea la baza de date trebuie pornită instanța, dar fără a se face și montarea bazei de date, prin următoarea comandă:

SVRMGR> STARTUP NOMOUNT PFILE=inittest.ora

Dacă fișierul de inițializare al parametrilor nu se află în directorul curent, atunci trebuie specificată calea acestuia în parametrul PFILE. De exemplu:

SVRMGR>STARTUP NOMOUNT PFILE=C:\ORANT\DATABASE\inittest.ora

4. Crearea bazei de date

Pentru a crea o bază de date se folosește comanda CREATE DATABASE ce are sintaxa următoare:

```
CREATE DATABASE [nume_baza_de_date]
[CONTROLFILE REUSE]
[LOGFILE [GROUP intreg]specificație_grup_fișiere_redo_log
[, [GROUP intreg] specificație_grup_fișiere_redo_log]
...]
[MAXLOGFILES intreg]
[MAXLOGMEMBERS intreg]
[MAXLOGHISTORY intreg]
[DATAFILE specificație_fișier_de_date
[,specificație_fișier_de_date]...]
[MAXDATAFILES intreg]
[MAXINSTANCES intreg]
[ARCHIVELOG|NOARCHIVELOG]
[CHARACTER SET set_de_caractere]
[NATIONAL CHARACTER SET set_de_caractere]
```

unde:

- nume_baza_de_date specifică numele bazei de date ce va fi creată. Dacă acest nume este omis, va fi utilizat parametrul DB_NAME din fișierul de inițializare;
- CONTROLFILE REUSE specifică faptul că fișierele de control existente identificate de parametrul CONTROL_FILE din fișierul de inițializare pot fi reutilizate;
- LOGFILE GROUP specifică numele fișierelor redo log ce vor fi folosite precum și grupul de care acestea aparțin. Dacă se omite acest parametru, Oracle va crea două grupuri de fișiere redo log, al căror nume și localizare vor depinde de sistemul de operare pe care este instalată baza de date. Un grup de fișiere redo log constă din copii identice ce se recomandă a fi stocate pe discuri diferite;
- MAXLOGFILES specifică numărul maxim de grupuri de fișiere redo log care pot fi create;
- MAXLOGMEMBERS specifică numărul maxim de fișiere redo log care pot fi membre ale unui grup;

70 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

- MAXLOGHISTORY specifică numărul maxim de fișiere redo log arhivate pentru recuperări automate. Acest parametru este folositor numai dacă se dorește utilizarea fișierelor redo log arhivate;
- DATAFILE specifică numele fișierelor de date ce vor fi folosite inițial. Aceste fișiere de date vor face parte din spațiul tabel SYSTEM creat odată cu baza de date (vezi secțiunea 4.1). Dacă această opțiune este omisă, Oracle creează în mod implicit un fișier de date;
- MAXDATAFILES specifică numărul maxim de fișiere de date ce pot fi create pentru această bază de date;
- MAXINSTANCES specifică numărul maxim de instanțe simultane ce pot monta și deschide baza de date;
- ARCHIVELOG | NOARCHIVELOG stabilește dacă fișierele redo log trebuie arhivate înainte de a fi reutilizate, respectiv dacă fișierele redo log pot fi reutilizate fără a mai arhiva conținutul lor. Dacă se omite acest parametru modul implicit (default) este NOARCHIVELOG;
- CHARACTER SET specifică setul de caractere pe care îl folosește baza de date pentru a depozita date. Acest parametru nu poate fi schimbat după ce baza de date este creată. Setul de caractere și valoarea inițială a acestui parametru sunt dependente de sistemul de operare;
- NATIONAL CHARACTER SET specifică setul național de caractere pe care îl folosește baza de date pentru a depozita date de tipul NCHAR, NCLOB, NVARCHAR2. Dacă nu este specificat, setul național de caractere va avea aceeași valoare ca setul de caractere.

Specificația unui grup de fișiere redo log, notată mai sus prin **specificație_grup_fișiere_redo_log** are următoarea sintaxă:

```
{nume_fișier | (nume_fișier [, nume_fișier] ...)}  
[SIZE întreg [K|M]]  
[REUSE]
```

Specificația unui fișier de date, notată mai sus prin **specificație_fișier_de_date** are următoarea sintaxă:

```
nume_fișier  
[SIZE întreg [K|M]]  
[REUSE]  
[AUTOEXTEND {OFF|ON[NEXT întreg [K|M]]  
[MAXSIZE [UNLIMITED] întreg [K|M]]}]]
```

unde SIZE specifică mărimea fișierului, REUSE determină refolosirea unui fișier de date deja existent, iar opțiunea AUTOEXTEND arată dacă fișierul poate crește automat (ON) sau nu (OFF) și, dacă este cazul, mărimea creșterii fișierului (NEXT) și mărimea maximă a fișierului (MAXSIZE). O explicație mai detaliată a acestor parametrii se găsește în secțiunea 4.1.1.

Pentru exemplificare vom crea o bază de date numita **test**:

```
CREATE DATABASE test  
CONTROLFILE REUSE  
LOGFILE GROUP 1
```

```

('c:\ORANT\DATABASE\log1test.ora',
 'd:\DB\logd1test.ora') SIZE 1024K REUSE
GROUP 2
('c:\ORANT\DATABASE\logc2test.ora',
 'd:\DB\logd2test.ora') SIZE 1024K REUSE
MAXLOGFILES 5
MAXLOGMEMBERS 5
MAXLOGHISTORY 1
DATAFILE
  'c:\ORANT\DATABASE\sys1test.ora'
  SIZE 50M REUSE AUTOEXTEND ON NEXT 10M MAXSIZE 200M
MAXDATAFILES 100
MAXINSTANCES 1
CHARACTER SET WE8ISO8859P1;

```

Această comandă creează o bază de date numită test. Baza de date conține două grupuri de fișiere redo log. Fișierul log1test.ora este identic cu fișierul logd1test.ora iar fișierul logc2test.ora este identic cu fișierul logd2test.ora cu mențiunea că acestea se află pe discuri diferite. Aceasta reprezintă o protecție în cazul în care unul din fișierele redo log este pierdut din cauza unor erori. Pentru o mai bună organizare, este de preferat ca numele fișierului redo log să conțină și discul pe care este stocat (în exemplul de mai sus avem log1test.ora respectiv logd1test.ora). Baza de date are un singur fișier de date (sys1test.ora) care are o mărime de 50MB. Fișierul de date poate să se autoextindă automat cu câte 10MB până la dimensiunea de 200 MB. Setul de caractere WE8ISO8859P1 reprezintă de fapt setul de caractere ISO 8859-1 West European. Oracle suportă 180 de seturi de caractere.

Pentru a putea evita pe cât posibil erorile de sintaxă care pot apărea în momentul utilizării comenzii CREATE DATABASE se poate modifica script-ul de creare a bazei de date oferit de Oracle. Numele și locația acestui script variază în funcție de sistemul de operare. În Windows 95/98/NT script-ul se numește BUILD_DB.SQL, iar pentru Unix script-ul se numește CRDB.SQL. Folosind această metodă, script-ul menționat se copiază, apoi se efectuează schimbările necesare pentru a crea o bază de date cu o configurație proprie, iar în final se rulează script-ul respectiv.

În momentul executării comenzii CREATE DATABASE, Oracle efectuează următoarele operații:

- Creează fișierele de date specificate. Dacă au fost specificate fișiere de date existente, datele acestora sunt șterse;
- Creează și inițializează fișierele de control specificate;
- Creează și inițializează fișierele redo log specificate;
- Creează spațiul tabel SYSTEM și segmentul de revenire SYSTEM (vezi secțiunile 4.1 și 4.2.3);
- Creează dicționarul de date (vezi secțiunea 6.11);
- Creează utilizatorii SYS și SYSTEM (vezi secțiunea 5.4);
- Specifică setul de caractere pentru baza de date. Acest set de caractere nu mai poate fi modificat după ce baza de date a fost creată;
- Monteză și deschide baza de date în mod exclusiv, adică numai instanța respectivă poate accesa baza de date în acel moment. Dacă se dorește ca baza de date să fie accesată de mai multe instanțe, atunci aceasta trebuie oprită și apoi repornită;

72 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

5. Rularea script-ului de generare a vederilor dicționarului de date și respectiv de instalare a obiectelor folosite de PL/SQL

Pentru ca baza de date să poată funcționa corect trebuie generat dicționarul de date (vezi 6.11.). Acest lucru se realizează prin rularea script-ului CATALOG . SQL:

```
SVRMGR>@%RDBMS80%\ADMIN\CATALOG . SQL
```

Un alt script care trebuie rulat este script-ul CATPROC . SQL care instalează obiectele folosite de PL/SQL:

```
SVRMGR>@%RDBMS80%\ADMIN\CATPROC . SQL
```

Capitolul 4

Structuri logice de stocare a bazei de date

În Oracle, spațiul folosit pentru stocarea datelor este controlat prin folosirea *structurilor logice de stocare*.

O *structură logică de stocare* este o unitate alocată a spațiului de stocare. Structurile logice dictează modul în care va fi utilizat spațiul fizic al bazei de date, configurația logică a bazei de date având un impact foarte mare asupra performanței acesteia.

Structurile logice de stocare folosite de Oracle sunt următoarele:

- *Spații tabel (tablespaces)*. Spațiul tabel este cea mai mare structură logică de stocare folosită de Oracle. Fiecare obiect al bazei de date este conținut într-un spațiu tabel.
- *Segmente (segments)*. Un segment este o mulțime de extinderi folosite pentru a stoca datele dintr-un obiect al bazei de date, fiind deci omologul fizic al obiectului respectiv.
- *Extinderi (extents)*. O extindere este o structură logică de stocare formată dintr-unul sau mai multe blocuri contigüe.
- *Blocuri de date (data blocks)*. Un bloc este cea mai mică unitate de stocare într-o bază de date Oracle.

Capitolul de față prezintă aceste structuri logice de stocare. În plus este prezentat și *identificatorul de rând (ROWID)*, ce reprezintă adresa fizică a unui rând dintr-un tabel în cadrul bazei de date Oracle.

4.1. Spații tabel

Datele gestionate de SGBDR Oracle sunt împărțite în spații tabel (*tablespaces*). Spațiile tabel sunt unități logice de spațiu pre-alocat ce există în *fișierele de date* asociate acestuia. Cu alte cuvinte, datele din baza de date sunt stocate *logic* în spații tabel și *fizic* în fișiere de date.

Un spațiu tabel este o structură folosită pentru a grupa datele accesate în mod similar. Un spațiu tabel poate fi constituit din mai multe fișiere de date care pot fi distribuite pe mai multe discuri, deci mărimea unui spațiu tabel nu este limitată de mărimea unui fișier sau spațiu fizic de stocare. Pe de altă parte, un fișier de date este asociat doar unui spațiu tabel și unei singure baze de date. Toate obiectele bazei de date (tabele, clustere, indecsi etc.) trebuie să aibă specificat un spațiu tabel unde trebuie să fie create. Datele care alcătuiesc aceste obiecte sunt apoi stocate în fișierele de date alocate spațiului tabel respectiv.

Orice bază de date conține un spațiu tabel numit **SYSTEM**. Ca parte a procesului de creare a bazei de date, Oracle creează automat acest spațiu tabel. Deși o bază de date relativ mică poate începea în spațiu tabel **SYSTEM**, se recomandă crearea unor spații tabel separate pentru datele

aplicației. În spațiul tabel SYSTEM se află obiectele predefinite și dicționarul de date, care conține informații despre obiectele bazei de date (tabele, vederi, indecsi, clustere etc.).

Spațiile tabel sunt create și administrează de către DBA (administratorul bazei de date). Fișierele de date vor fi create odată cu crearea sau modificarea (extinderea) unui spațiu tabel, cu excepția situației când se specifică explicit rezolvarea unui fișier de date existent (folosind opțiunea REUSE explicitată mai jos).

În general, nu trebuie create spații tabel suplimentare fără un motiv întemeiat, fiind recomandată existența pe cât posibil a cât mai puține spații tabel. De asemenea, în cadrul fiecărui spațiu tabel este recomandată crearea a cât mai puține fișiere de date, de dimensiuni cât mai mari. De exemplu, la crearea sau mărirea dimensiunii unui spațiu tabel este recomandată crearea unui singur fișier de date de dimensiune mai mare decât a mai multor fișiere de dimensiuni mici.

Mărimea unui fișier de date este cea specificată la crearea acestuia și nu reprezintă cantitatea datelor stocate în el. De exemplu, un fișier de date care va fi creat cu mărimea de 10 MB, va folosi toți cei 10 MB indiferent dacă el conține un milion de rânduri sau unul singur. Datorită acestui fapt, dimensiunea unui fișier de date trebuie planificată cu atenție pentru ca, odată creat, un fișier de date nu mai poate fi micșorat. Începând cu versiunea 7.3 Oracle oferă posibilitatea ca fișierele de date să se extindă în mod automat (folosind opțiunea AUTOEXTEND explicitată mai jos) dacă spațiul tabel corespunzător devine neîncăpător pentru datele ce trebuie stocate.

Așa cum am văzut până acum, fișierele bazei de date, spațiile tabel și baza de date sunt noțiuni strâns legate între ele. Pentru o mai bună înțelegere a acestor noțiuni, vom schematiza diferențele cele mai importante dintre acestea:

<i>Baza de date și spațiul tabel</i>	O bază de date Oracle constă din una sau mai multe spații tabel. Datele bazei de date sunt stocate colectiv în spații tabel.
<i>Spațiul tabel și fișierele de date</i>	Fiecare spațiu tabel constă din unul sau mai multe fișiere de date. Aceste fișiere de date sunt fișiere fizice ale sistemului de operare pe care rulează Oracle.
<i>Baza de date și fișierele de date</i>	Datele dintr-o bază de date sunt stocate în fișiere de date ce alcătuiesc spațiile tabel. Cea mai simplă bază de date constă dintr-un singur spațiu tabel (SYSTEM, vezi secțiunea 4.1) și dintr-un singur fișier de date. De exemplu, o bază de date mai complicată poate avea trei spații tabel fiecare constând din două fișiere de date, deci în total șase fișiere de date.

4.1.1. Crearea spațiilor tabel

Spațiile-tablet pot fi create folosind comanda SQL CREATE TABLESPACE, având sintaxa următoare:

```
CREATE TABLESPACE nume_spațiu_tabel
DATAFILE specificație_fișier_de_date
[, specificație_fișier_de_date...]
[MINIMUM EXTENT întreg [K|M]]
```

```
[LOGGING | NOLOGGING]
[DEFAULT STORAGE parametrii_de_stocare ]
[ONLINE | OFFLINE]
[PERMANENT | TEMPORARY]
```

unde:

- DATAFILE specifică fișierul sau fișierele de date pe care le va cuprinde spațiul tabel. Aceste fișiere vor fi create odată cu crearea spațiului tabel. Sintaxa specificației unui fișier de date este detaliată mai jos;
- MINIMUM EXTENT specifică mărimea minimă a extinderilor cuprinse în acest spațiu tabel. Aceasta este în bytes (în mod implicit), Kbytes (dacă se folosește opțiunea K) sau Mbytes (dacă se folosește opțiunea M);
- LOGGING și respectiv NOLOGGING¹ determină dacă la executarea anumitor comenzi SQL (încărcarea de date într-un tabel, crearea indecșilor, etc.) asupra obiectelor din spațiul tabel se vor păstra sau nu informații despre aceste operații în fișierul jurnal pentru recuperare (redo log). Dacă nu se specifică nici o opțiune, valoarea implicită este LOGGING;
- DEFAULT STORAGE specifică valorile implicite (default) pentru parametrii de stocare ale tuturor obiectelor create în spațiul tabel. Parametrii de stocare vor fi detaliați mai jos;
- ONLINE activează spațiul tabel imediat după creare, făcându-l accesibil utilizatorilor care au permisiunea de a accesa spațiul tabel. OFFLINE dezactivează spațiul tabel imediat după creare, făcându-l inaccesibil. ONLINE este valoarea implicită;
- PERMANENT arată că spațiul tabel va fi folosit pentru a stoca obiecte permanente. TEMPORARY arată că spațiul tabel va fi folosit doar pentru a stoca obiecte temporare. PERMANENT este valoarea implicită.

Specificația unui fișier de date, notată mai sus prin **specificație_fișier_de_date** are următoarea sintaxă:

```
nume_fișier
[SIZE întreg [K|M]]
[REUSE]
[AUTOEXTEND {OFF|ON[NEXT întreg [K|M]]}
            [MAXSIZE [UNLIMITED] întreg [K|M]]}])
```

unde:

- SIZE specifică mărimea fișierului în bytes (în mod implicit), Kbytes (dacă se folosește opțiunea K) sau Mbytes (dacă se folosește opțiunea M);
- REUSE determină resolosirea unui fișier de date deja existent. În absența opțiunii REUSE, Oracle va crea un nou fișier de date dacă nu există deja un fișier cu numele menționat sau va produce un mesaj de eroare în caz contrar. În cazul folosirii opțiunii REUSE, Oracle va folosi fișierul dacă el există sau va crea unul nou în caz contrar;
- opțiunea AUTOEXTEND arată dacă fișierul poate crește automat (ON) sau nu (OFF) în cazul în care spațiul tabel are nevoie de mai mult spațiu. Valoarea implicită este OFF. În

¹ Opțiunea NOLOGGING este o nouitate adusă de versiunea Oracle8 și poate fi folosită pentru a mări performanța instrucțiunilor SQL în cazul bazelor de date mari. De exemplu, încărcarea într-un tabel a milioane de înregistrări în modul implicit LOGGING ar duce la înscriserea unei mari cantități de informații în fișierele jurnal, consumând astfel spațiu fizic pe disc și încreștinind vizibil procesul.

cazul când opțiunea AUTOEXTEND este ON, se pot specifica atât mărimea creșterii fișierului, în bytes Kbytes (K) sau Mbytes (M), prin opțiunea NEXT, cât și valoarea maximă a fișierului, prin opțiunea MAXSIZE, care poate fi un număr întreg de bytes Kbytes (K) sau Mbytes (M) sau pate fi nelimitată (UNLIMITED).

De exemplu, comanda SQL:

```
CREATE TABLESPACE ts_alfa
DATAFILE '/date/ts_alfa01.dbf'
      SIZE 20M;
```

creează spațiul tabel ts_alfa cu un fișier de date ts_alfa01.dbf de mărime 20M. În cazul în care se dorește ca spațiul tabel să se autoextindă automat cu câte 5M până la dimensiunea de 50 M se folosește comanda:

```
CREATE TABLESPACE ts_alfa
DATAFILE '/date/ts_alfa01.dbf'
      SIZE 20M
      AUTOEXTEND ON NEXT 5M MAXSIZE 50M;
```

Setarea valorilor implicite ale parametrilor de stocare

În mod ideal, un spațiu tabel ar trebui să conțină numai segmente cu spații de mărimi egale. O modalitate de a realiza acest lucru este de a atribui fiecărui spațiu tabel opțiunile dorite de stocare ca opțiuni implicite. Clauza DEFAULT STORAGE a comenzi CREATE TABLESPACE este mijlocul de a face acest lucru. Sintaxa acesteia este următoarea:

```
DEFAULT STORAGE (
  [INITIAL întreg [K|M]]
  [NEXT întreg [K|M]]
  [PCTINCREASE întreg]
  [MINEXTENTS întreg]
  [MAXEXTENTS întreg])
```

unde:

- INITIAL specifică mărimea primei extinderi a segmentului. Acesta este în bytes (în mod implicit), Kbytes (dacă se folosește opțiunea K) sau Mbytes (dacă se folosește opțiunea M). Valoarea implicită este egală cu dimensiunea a 5 blocuri de date;
- NEXT specifică mărimea celei de-a doua extinderi a segmentului în bytes (în mod implicit), Kbytes sau Mbytes. Valoarea implicită este egală cu dimensiunea a 5 blocuri de date;
- PCTINCREASE definește creșterea, măsurată în procente, a fiecărei extinderi ulterioare celei de-a doua. De exemplu, în cazul în care această valoare este 25 (vezi exemplul de mai jos), fiecare extindere începând cu cea de-a treia va fi cu 25% mai mare decât cea precedentă; deci a treia extindere va avea mărimea $40*(1+0.25)K = 50K$, a patra extindere va avea mărimea $50*(1+0.25) = 62.5K$, etc. Valoarea implicită a acestui parametru este 50.
- MINEXTENTS indică numărul de extinderi care vor fi create la crearea obiectului care este stocat în segment.

- MAXEXTENTS indică numărul maxim de extinderi pe care Oracle poate să le aloce pentru obiectul care este depozitat în segment.

Folosirea clauzei implicite de stocare este ilustrată în exemplul următor:

```
CREATE TABLESPACE ts_alfa
DATAFILE '/date/ts_aIfa01.dbf'
  SIZE 20M
DEFAULT STORAGE(
  INITIAL 50K
  NEXT 40K
  PCTINCREASE 25
  MINEXTENTS 5
  MAXEXTENTS 30);
```

Folosirea clauzei DEFAULT STORAGE în comandă de mai sus face ca valorile parametrilor de stocare din această clauză să fie folosiți ori de câte ori un obiect al bazei de date (tabel, cluster, instantaneu sau segment de revenire) este creat în spațiul tabel `ts_alfa` fără a fi setați explicit parametrii de stocare pentru acesta.

4.1.2. Modificarea spațiilor tabel

Comanda SQL ALTER TABLESPACE poate fi folosită pentru modificarea unui spațiu tabel într-unul din modurile următoare:

- Adăugarea de noi fișiere de date:

```
ALTER TABLESPACE ts_alfa
ADD DATAFILE '/date7ts_alfa02.dbf'
  SIZE 20M;
```

- Schimbarea numelui uneia sau mai multor fișiere de date atașate tabelului:

```
ALTER TABLESPACE ts_alfa
RENAME DATAFILE '/date7ts_alfa02.dbf' TO '/date/ts_a02.dbf';
```

- Pentru a schimba parametrii implicați de stocare:

```
ALTER TABLESPACE ts_alfa
DEFAULT STORAGE(
  INITIAL 50K
  NEXT 50K
  PCTINCREASE 25
  MINEXTENTS 5
  MAXEXTENTS 30);
```

- Pentru a activa (ONLINE) sau dezactiva (OFFLINE) tabelul:

```
ALTER TABLESPACE ts_alfa
OFFLINE;
```

- Pentru a arăta că un back-up va fi efectuat asupra fișierelor de date cuprinse în spațiul tabel (BEGIN BACKUP) sau că un back-up al tabelului a luat sfârșit (END BACKUP):

```
ALTER TABLESPACE ts_alfa
START BACKUP;
```

4.1.3. Distrugerea spațiilor tabel

Un tabel poate fi distrus folosind comanda SQL `DROP TABLESPACE`. Pentru a distruge un spațiu tabel care conține obiecte ale bazei de date, este necesară folosirea opțiunii `INCLUDING CONTENTS`:

```
DROP TABLESPACE ts_alfa INCLUDING CONTENTS;
```

Spațiul tabel `SYSTEM` nu poate fi șters datorită faptului că acesta conține dicționarul de date. O altă restricție care se impune de către Oracle este aceea că spațiul tabel `SYSTEM` nu poate fi `OFFLINE`.

În concluzie, spațiile tabel sunt folosite pentru a separa accesul (citire/scriere) la date diferite din baza de date. De exemplu, prin folosirea spațiilor tabel multiple se pot obține următoarele avantaje:

- Separarea datelor de sistem față de cele ale aplicației prin stocarea acestora din urmă în alt spațiu tabel decât `SYSTEM`;
- Separarea aplicațiilor între ele prin stocarea datelor pe două spații tabel diferite;
- Restrângerea accesului la anumite date prin dezactivarea spațiului tabel în care sunt stocate aceste date;
- Posibilitatea de a stoca spații tabel diferite pe unități de disc distincte în vederea îmbunătățirii performanțelor și reducerii conflictelor de acces la date. De exemplu, un spațiu tabel poate fi creat pentru a stoca date, iar altul pentru a stoca indecesi. Dacă acestor două spații tabel le sunt alocate fișiere de date care sunt plasate pe două discuri distincte, atunci se asigură faptul că accesul la indecesi nu interferează cu accesul la datele pentru care sunt creați indecesii.

4.2. Segmente

Un spațiu tabel posedă unul sau mai multe segmente. Un segment este o mulțime de extinderi folosite pentru a stoca datele dintr-un obiect al bazei de date. Deci se poate spune că segmentele sunt omoloagile fizice ale obiectelor bazei de date care stochează date, adică fiecărui obiect stocat fizic îi corespunde un segment. Așa cum vom vedea în capitolul 6, nu toate obiectele unei baze de date sunt stocate fizic, deci există obiecte cărora nu le corespunde un segment. Un segment utilizează un număr de blocuri care se găsesc în același spațiu tabel (deși blocurile se pot găsi în fișiere de date diferite).

Baza de date Oracle folosește patru tipuri de segmente:

4.2.1. Segmente de date (data segments)

Acestea se mai numesc și *segmente tabel (table segment)* și stochează datele din tabelele, clusterele și instantaneele bazei de date. Un astfel de tip de segment este creat în mod

indirect prin intermediul comenziilor SQL CREATE TABLE, CREATE CLUSTER sau CREATE SNAPSHOT.

Valorile parametrilor de stocare (INITIAL, NEXT, PCTINCREASE, MINEXTENTS, MAXEXTENTS, vezi secțiunea 4.1.1) pot fi setate folosind clauza STORAGE a acestor comenzi (vezi primul exemplu de mai jos); în caz contrar (vezi al doilea exemplu de mai jos), sunt folosite valorile implicate pentru spațiul tabel pe care este creat tabelul, indexul sau instantaneul respectiv.

Exemplul 1:

```
CREATE TABLE persoana(
cod_persoana          NUMBER(10),
nume                   VARCHAR2(10),
prenume                VARCHAR2(10),
data_nastere           DATE)
TABLESPACE ts_alfa
STORAGE(
    INITIAL 50K
    NEXT 50K
    PCTINCREASE 0
    MINEXTENTS 10
    MAXEXTENTS 40);
```

Exemplul 2:

```
CREATE TABLE persoana(
cod_persoana          NUMBER(10),
nume                   VARCHAR2(10),
prenume                VARCHAR2(10),
data_nastere           DATE)
TABLESPACE ts_alfa;
```

4.2.2. Segmente de indecsi (index segments)

Acest tip de segment păstrează datele unui index al bazei de date. Un segment de indecsi este creat indirect prin intermediul comenzi SQL CREATE INDEX. La fel ca și în cazul segmentelor de date, valorile parametrilor de stocare pot fi setate folosind clauza STORAGE sau pot fi folosite valorile implicate ale spațiului tabel pe care este creat indexul.

4.2.3. Segmente de revenire (rollback segments)

Un segment de revenire este o porțiune a bazei de date care înregistrează acțiunile unei tranzacții SQL în vederea derulării înapoi, dacă se dorește acest lucru. Un segment de revenire restabilește baza de date în starea în care se găsea la ultimul punct de salvare sau la ultima actualizare permanentă (COMMIT).

Fiecare tranzacție are un segment de revenire asociat. De obicei, acest segment este atribuit automat de către baza de date. Există totuși posibilitatea de a asocia o tranzacție cu un anumit segment de revenire, folosind comanda SQL SET TRANSACTION cu opțiunea

USE ROLLBACK SEGMENT. Un segment de revenire poate fi folosit concurent de una sau mai multe tranzacții.

Segmentele de revenire pot fi create folosind comanda SQL CREATE ROLLBACK SEGMENT, în care se poate specifica spațiul tabel pe care se creează segmentul. Dacă această opțiune este omisă, segmentul de revenire este creat implicit în segmentul tabel SYSTEM. De asemenea, se poate specifica și parametrii de stocare folosind clauza STORAGE sau pot fi folosiți cei implicați ai spațiului tabel. Alături de parametrii de stocare menționați mai sus (INITIAL, NEXT, PCTINCREASE, MINEXTENTS, MAXEXTENTS), în cazul segmentelor de revenire poate fi folosit și parametrul OPTIMAL, a cărui valoare (în bytes, Kbytes sau Mbytes) specifică mărimea optimă a segmentului de revenire (vezi exemplul de mai jos). Oracle încearcă să mențină această mărime pentru segmentul de revenire prin dealocarea dinamică a extinderilor atunci când datele pe care le conțin nu mai sunt necesare. Oracle va dealoca câte extinderi este posibil fără a reduce mărimea totală a segmentului de revenire sub valoarea optimă.

```
CREATE ROLLBACK SEGMENT rb1
TABLESPACE ts_rbs
DEFAULT STORAGE (
    INITIAL 250K
    NEXT 250K
    PCTINCREASE 0
    MINEXTENTS 16
    OPTIMAL 4M);
```

4.2.4. Segmente temporare (temporary segments)

Un segment temporar este creat atunci când o comandă SQL necesită un spațiu de lucru temporar. De exemplu, pentru o operație de sortare (clauza ORDER BY dintr-o comandă SELECT) va fi creat un segment temporar reprezentând zona de lucru pentru această operație. Segmentele temporare sunt distruse atunci când comanda SQL respectivă este terminată.

4.3. Extinderi

Fiecare segment este compus din una sau mai multe extinderi; la rândul lor, o extindere este formată dintr-unul sau mai multe blocuri contigüe. Cu alte cuvinte, orice segment este compus din mai multe grupuri de blocuri contigüe, adică dintr-una sau mai multe zone de memorie contigüe. Atunci când un segment epuizează spațiul de stocare care i-a fost alocat, Oracle îi acordă o nouă extindere. La momentul creării, fiecare segment posedă cel puțin o extindere. Așa cum am văzut, numărul de extinderi precum și dimensiunea acestora pot fi specificate folosind clauza STORAGE a comenzi de creare a obiectului respectiv sau pot fi specificate ca valori implicate pentru spațiul tabel folosind clauza DEFAULT STORAGE la crearea spațiului tabel.

Pentru a putea înțelege mai bine modul de funcționare al extinderilor vom ilustra în continuare modul de gestiune a acestora în momentul creării și inserării înregistrărilor într-un tabel. Dacă un tabel folosește valorile implicate ale parametrilor de stocare (la crearea tabelului și a spațiului tabel aferent lipsesc clauzele STORAGE și respectiv DEFAULT STORAGE), atunci Oracle va aloca 5 blocuri pentru prima extindere (cunoscută și ca extinderea inițială). Cele 5 blocuri se umplu cu date pe măsură ce înregistrările sunt inserate în tabel. Când ultimul bloc a fost umplut și se dorește inserarea unei noi înregistrări, baza de date alocă automat un alt set de 5 blocuri. Această alocare continuă până când nu mai există loc în spațiul tabel. Singura restricție care se impune este aceea că blocurile unei extinderi trebuie să se găsească în același fișier de date. (Această restricție este o consecință logică a faptului că blocurile trebuie să se găsească într-o zonă de memorie contiguă).

La alocarea unei noi extinderi pentru un segment, Oracle parcurge spațiul tabel corespunzător în căutarea primului set contiguu de blocuri care au împreună dimensiunea incrementului stabilit pentru o extindere. În momentul în care o extindere a fost alocată unui segment, blocurile conținute în aceasta nu mai pot fi utilizate de către alt obiect al bazei de date chiar dacă, de exemplu, rândurile dintr-un tabel au fost șterse. Totuși, extinderile segmentelor pot fi dealocate atunci când nu mai sunt necesare. În general, extinderile nu sunt returnate spațiului tabel ca spațiu liber până când nu este returnat întreg segmentul. Acest lucru survine, de exemplu, atunci când un tabel este distrus, iar segmentul și extinderile sale sunt eliberate. Excepție fac segmentele de revenire care elibereză spațiu alocat în mod dinamic.

4.4. Blocuri de date

Blocul de date reprezintă cea mai mică unitate de stocare utilizată de Oracle pentru stocarea datelor. Un bloc de date corespunde unui bloc fizic de octeți de pe disc. În interiorul fișierelor de date spațiul de stocare este compus din blocuri de date. Mărimea unui bloc de date corespunde cu numărul de octeți citiți sau scriși de SGBDR Oracle într-o singură operație I/O (intrare/ieșire, citire/scriere). Mărimea unui bloc de date poate fi setată folosind parametrul DB_BLOCK_SIZE din fișierul de inițializare a parametrilor; pentru Oracle sub Windows NT, valoarea implicită (default) a mărimii unui bloc este de 2 KB. De exemplu, presupunând că mărimea unui bloc este de 2 KB, pentru o bază de date de 50 MB (51200 KB) vor exista 25600 de blocuri.

Dimensiunea blocului este un factor important al eficienței bazei de date deoarece ea determină numărul de octeți citiți din fișierele de date într-o singură operație I/O. Dacă blocul este prea mare, sunt aduse înregistrări inutile pentru a satisface cererea curentă, ceea ce duce la creșterea timpului necesar regăsirii datelor. Dacă dimensiunea blocului este prea mică, sunt necesare operații suplimentare de I/O pentru a satisface cererea. Dacă în mod obișnuit aplicația citește și scrie cantități mici de date la fiecare tranzacție, atunci blocurile de dimensiune mai mici sunt mai eficiente din punct de vedere al performanței operației I/O. Dacă, din contră, aplicația procesează o cantitate mare de date într-o singură tranzacție, atunci blocurile mai mari sunt mai eficiente.

Fiecare bloc este numerotat secvențial pentru fiecare fișier de date începând cu 1. Două blocuri pot avea aceeași adresă numai dacă se găsesc în fișiere de date diferite. Orice bloc de

date conține spațiu pentru informații antet (header information), pentru modificări viitoare de date în blocul respectiv și pentru rândurile stocate efectiv în bloc. Antetul blocului conține informații precum segmentele bazei de date care au rânduri în blocul respectiv, câte tranzacții pot accesa simultan blocul, etc. Fiecare bloc alocă un anumit spațiu pentru modificări viitoare ale rândurilor stocate în bloc. Dacă o astfel de modificare duce la creșterea rândului original, atunci este folosit acest spațiu.

La crearea unui spațiu tabel, se utilizează câteva blocuri din primul fișier de date, restul rămânând libere. Oracle gestionează o listă cu blocurile rămase libere din fiecare fișier de date aferent unui spațiu tabel cu ajutorul dicționarului de date.

Pentru controlul utilizării spațiului liber al blocurilor de date din extinderile unui segment, proiectantul aplicației are la dispoziție doi parametri, PCTFREE și PCTUSED. Ambii parametri pot fi specificați numai în momentul creației sau modificării tabelelor și clusterelor, în cadrul comenzi SQL CREATE TABLE și CREATE CLUSTER. În plus, parametrul PCTFREE poate fi setat și la crearea indecsilor, în cadrul comenzi CREATE INDEX. De exemplu:

```
CREATE TABLE persoana (
cod_persoana          NUMBER(10),
nume                   VARCHAR2(10),
prenume               VARCHAR2(10),
data_nastere           DATE)
PCTFREE 20
PCTUSED 70;
```

unde:

- PCTFREE reprezintă spațiul în procente din fiecare bloc de date rezervat pentru o posibilă expansiune cauzată de actualizările viitoare. Valoarea trebuie să fie un întreg între 0 și 99, valoarea implicită fiind 10. În exemplul de mai sus 20% din fiecare bloc este rezervat pentru modificarea rândurilor existente și permite inserarea noilor înregistrări în 80% din spațiu fiecarui bloc.
- PCTUSED reprezintă procentul de umplere a blocurilor unei extinderi până la care Oracle continuă să insereze noile înregistrări în extinderea curentă. În exemplul de mai sus, în momentul în care continuarea inserării de noi rânduri ar duce la depășirea procentajului de 70%, inserarea este opriță, se alocă o nouă extindere și se continuă inserarea noilor rânduri în noua extindere.

Suma dintre PCTFREE și PCTUSED nu trebuie să depășească 100. O valoare mare a parametrului PCTUSED reduce spațiul de stocare neutilizat, însă reduce și viteza de execuție a instrucțiunilor INSERT (inserare) și UPDATE (actualizare). O valoare mică a acestui parametru duce la creșterea spațiului de stocare neutilizat, dar îmbunătățește eficiența procesării comenzi INSERT și UPDATE.

4.5. Identifierul de rând (ROWID)

Identifierul de rând (ROWID) este o adresă fizică unică pentru fiecare rând al unui tabel. Datorită acestui fapt, identifierul de rând poate fi privit ca pe o pseudo-colonă a

tabelului. Această pseudo-coloană are denumirea ROWID și poate fi folosită în interogări. Identificatorul de rând este creat în mod automat de către Oracle în momentul în care un rând este inserat permanent într-un tabel. Valoarea identificatorului va rămâne neschimbată până când rândul aferent sau tabelul ce conține acest rând va fi șters, prin urmare valorile pseudo-coloanei ROWID nu pot fi modificate.

În versiunea Oracle7 un identificator de rând era alcătuit din trei componente necesare pentru localizarea unei înregistrări:

- Numărul *blocului* ce conține înregistrarea. Numerele blocurilor sunt relative la fișierul de date aferent și nu la spațiul tabel. Prin urmare, putem avea doi identificatori de rând ce au același număr de bloc dar au fișiere de date diferite aflate în același spațiu tabel.
- Numărul *rândului* din bloc ce conține înregistrarea. Numerele rândurilor dintr-un bloc încep întotdeauna cu 0.
- Numărul *fișierului de date* ce conține înregistrarea. Numărul primului fișier dintr-o bază de date este 1. În versiunea Oracle7, numerele fișierelor sunt unice în cadrul unei baze de date

În acest caz, identificatorul unui rând are formatul:

bloc.rând.fișier

Pentru a exemplifica formatul identificatorului de rând putem folosi următoarea interogare:

```
SELECT ROWID, SUBSTR(ROWID,15,4) "FISIER",
       SUBSTR(ROWID,1,8) "BLOC",
       SUBSTR(ROWID,10,4) "RÂND"
  FROM produse;
```

Interogarea de mai sus va avea un rezultat de forma:

ROWID	FISIER	BLOC	RÂND
00000DD5.0000.0001	0001	00000DD5	0000
00000DD5.0001.0001	0001	00000DD5	0001
00000DD5.0002.0001	0001	00000DD5.	0002

În versiunea Oracle7 se putea folosi formatul de mai sus pentru identificatorul de rând deoarece numerele fișierelor de date erau unice în cadrul unei baze de date. Versiunea Oracle8 înălță această restricție, numărul unui fișier trebuind să fie unic doar în cadrul unui spațiu tabel (acesta va fi numit numărul fișierului relativ la spațiul tabel). În consecință, pentru identificarea unică a unui rând se folosește o a patra componentă, numărul de identificare al obiectului, care determină în mod unic segmentul bazei de date în care se află obiectul – obiectele din același segment (de exemplu tabelele dintr-un cluster) au același număr de obiect. Deci în Oracle8 se folosește o versiune extinsă a ROWID, alcătuită din următoarele patru componente:

- număr obiect;
- număr relativ al fișierului de date;

84 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

- număr bloc;
- număr rând din bloc.

În acest caz, identificatorul unui rând necesă tot 18 caractere pentru afișare. Prin urmare, identificatorul de rând are formatul:

000000	FFF	BBBBBB	RRR
<i>Număr obiect</i>	<i>Număr relativ al fișierului de date</i>	<i>Număr bloc</i>	<i>Număr rând</i>

Pentru compatibilitate cu versiunile anterioare, Oracle8 suportă și formatul ROWID din Oracle7, numit ROWID restricționat. Pentru a putea converti o valoare ROWID din format extins în format restricționat se poate folosi pachetul DBMS_ROWID.

Identifierul de rând are mai multe utilizări importante. El reprezintă în primul rând cel mai rapid mod de a localiza o anumită înregistrare. Prin urmare el este utilizat intern de către indecsi (vezi secțiunea 6.3) sau poate fi utilizat de către dezvoltatorii de aplicații atunci când se cunoaște valoarea sa. Identifierul de rând poate fi utilizat și pentru a culege informații referitoare la stocarea fizică a datelor din tabele. De exemplu, dacă ne interesează localizarea fizică a rândurilor unui anumit tabel, adică în câte fișiere de date sunt conținute înregistrările tabelului, vom folosi pentru ROWID extins o interogare de forma:

```
SELECT COUNT(DISTINCT(SUBSTR(ROWID, 7, 3))) "FISIERE"  
FROM produse;
```

Capitolul 5

Securitatea bazei de date

Baza de date Oracle conține propriul ei sistem de securitate care previne accesul neautorizat la baza de date. Sistemul de securitate al bazei de date Oracle este realizat prin intermediul **utilizatorilor** bazei de date. Serverul bazei de date solicită numele utilizatorului și parola pentru fiecare accesare a bazei de date; indiferent de utilizator folosit pentru interfață, serverul bazei de date nu permite accesul la baza de date dacă nu este utilizat un nume și o parolă corectă.

Notă: Așa cum s-a mai menționat, în contextul bazei de date Oracle, un utilizator înseamnă de fapt un cont de utilizator și nu o persoană care accesează baza de date. Evident, o persoană poate accesa baza de date folosind unul sau mai mulți utilizatori Oracle, iar mai multe persoane pot accesa baza de date folosind același utilizator Oracle (acesta este cazul cel mai întâlnit).

O **schemă** este o colecție de obiecte disponibile unui utilizator. Obiectele schemei sunt structuri logice ce se referă efectiv la datele unei baze de date precum tabele, vederi, secvențe, indecsi, sinonime, etc.

Fiecarui utilizator al bazei de date îi sunt acordate anumite drepturi, cunoscute sub numele de privilegii. Un **privilegiu** este permisiunea de a executa o acțiune sau de a accesa un obiect aparținând unui alt utilizator. În Oracle, un utilizator nu poate executa nici un fel de acțiune fără a avea privilegiul să o facă. În acest sens unui utilizator îi pot fi acordate sau revocate privilegii.

Accesul unui utilizator la baza de date este administrat printr-un număr de drepturi numite **privilegii de sistem**, sau privilegii la nivelul bazei de date, care permit utilizatorului să efectueze operații precum conectarea la baza de date și crearea de obiecte. Odată ce utilizatorul a creat obiecte ale bazei de date, el este apoi responsabil de a acorda drepturi altor utilizatori pentru obiectele care sunt proprietatea lui. Aceste drepturi sunt numite **privilegii la nivel de obiect**.

Rolurile sunt utilizate pentru a simplifica administrarea privilegiilor. Astfel, în loc de a acorda un anumit privilegiu direct unui utilizator, privilegiile sunt acordate unui rol, iar un rol este acordat la rândul lui unui utilizator. Cu alte cuvinte, rolurile reprezintă *un grup de privilegii*.

Acest capitol prezintă modul în care se realizează securitatea unei baze de date Oracle, referindu-se la fiecare dintre concepții menționate mai sus.

5.1. Privilegii de sistem

ACTIONILE PE CARE UN UTILIZATOR LE poate efectua asupra bazei de date sunt administrate prin privilegiile de sistem acordate acestuia. În Oracle există peste 80 de privilegii de sistem, denumirea lor fiind inspirată de acțiunile pe care le permit. Ele variază de la permisiunea de a se conecta la o bază de date (CREATE SESSION) la dreptul de a crea un tabel (CREATE ANY TABLE) sau index (CREATE ANY INDEX) sau de a distruge un tabel (DROP ANY TABLE) sau index (DROP ANY INDEX) din schema oricărui utilizator. O listă a tuturor privilegiilor de sistem este prezentată în Anexa 4A.

5.2. Privilegii la nivel de obiect

Securitatea obiectelor unei baze de date este administrată ca un număr de *privilegii la nivel de obiect*, care determină ce acces au utilizatorii la obiectele bazei de date. Privilegii la nivel de obiect existente în Oracle sunt rezumate în tabelul din figura 5.1. Tabelul din figura 5.2 indică obiectele asupra cărora poate fi acordat fiecare privilegiu în parte.

Fiecare privilegiu la nivel de obiect este independent de celelalte, adică existența unui privilegiu nu atrage după sine și existența altor privilegiu.

Privilegiul de obiect	Descrierea permisiunii
SELECT	Selectarea rândurilor dintr-un tabel, vedere sau instantaneu și extragerea numerelor dintr-un generator de secvențe.
INSERT	Inserarea înregistrărilor într-un tabel sau vedere.
UPDATE	Actualizarea înregistrărilor dintr-un tabel sau vedere.
DELETE	Ștergerea înregistrărilor dintr-un tabel sau vedere.
ALTER	Modificarea structurii și parametrilor unui tabel sau a unei secvențe.
REFERENCES	Referirea unui tabel utilizând chei străine.
EXECUTE	Executarea unei proceduri, funcții, pachet sau proceduri externe și accesarea obiectelor declarate în specificația pachetului. În plus, pentru opțiunea obiect din Oracle8 (vezi capitolul 10), acest privilegiu se poate acorda și asupra unui tip de date creat de utilizator, în acest caz acest tip de date putând fi folosit la crearea unor tabele, la definirea unor coloane din tabele și la declararea unor variabile sau parametri.
INDEX	Crearea indecșilor tabelului.
READ	Citirea unui BFILE din directorul specificat (vezi Anexa 3).

Figura 5.1.

Privilegiul de obiect	Tabel	Vedere	Secvență	Procedură - funcție pachet	Instantaneu	Director	Bibliotecă
SELECT	X	X	X		X		
INSERT	X	X					
UPDATE	X	X					
DELETE	X	X					
ALTER	X		X				

Privilegiul de obiect	Tabel	Vedere	Secvență	Procedură funcție pachet	Instantaneu	Director	Bibliotecă
REFERENCES	X						
EXECUTE				X			X
INDEX	X						
READ						X	

Figura 5.2.

Fiecare obiect al bazei de date este proprietatea unui utilizator al bazei de date - se spune că obiectul face parte din schema respectivului utilizator. Proprietarul unui obiect are control deplin asupra acestuia. El poate efectua orice acțiune asupra unui obiect deținut fără să aibă privilegii la nivel de obiect deoarece aceste privilegii sunt implicate. În plus, proprietarul unui obiect poate acorda privilegii asupra obiectului respectiv și altor utilizatori.

Prin urmare, pentru a efectua o acțiune asupra unui obiect (interrogare, actualizare, distrugere, etc.), un utilizator trebuie să se găsească în unul din următoarele cazuri:

- să fie proprietarul acelui obiect;
- să aibă acordat privilegiul la nivelul obiectului respectiv de a efectua acea acțiune ;
- să aibă acordat privilegiul de sistem care să îi permită acest lucru.

De exemplu, pentru a putea interroga (SELECT) un tabel, un utilizator trebuie să fie proprietarul aceluia tabel, să posede privilegiul SELECT pe acel tabel sau să posede privilegiul de sistem SELECT ANY TABLE. Privilegiile necesare pentru efectuarea diverselor acțiuni asupra obiectelor bazei de date sunt sintetizate pe categorii de obiecte în Anexa 4B.

5.3. Roluri

Pentru a simplifica modul de administrare, privilegiile pot fi grupate în *roluri (roles)*. Un grup de privilegii pot fi acordate unui rol, iar un rol poate fi acordat la rândul lui unui utilizator, acordându-se astfel utilizatorului în mod implicit privilegiile asociate cu acel rol. De asemenea, este posibilă acordarea unui rol altui rol, putându-se crea astfel ierarhii de roluri. În plus, unui rol îi poate fi revocat un privilegiu după cum și unui utilizator îi poate fi revocat un rol. În general, rolurile se folosesc atunci când există utilizatori care au nevoie de același seturi de privilegii.

Rolurile permit simplificarea gestionării privilegiilor: în loc de a acorda mai multe privilegii unui utilizator, acestuia îi poate fi acordat rolul care conține aceste privilegii. Pe lângă simplificarea modului de administrare a privilegiilor, rolurile au avantajul de a putea fi administrate dinamic: atunci când privilegiile unui rol se modifică, aceste modificări se reflectă automat în privilegiile acordate utilizatorilor care dețin rolul respectiv. În plus, un rol poate fi activat sau dezactivat în cadrul aceleiași sesiuni, aceasta având ca efect acordarea sau retragerea privilegiilor respective utilizatorului. De asemenea un rol poate fi protejat cu o parolă care va trebui introdusă în momentul în care se dorește activarea acestuia.

Oracle oferă un set de roluri predefinite cu privilegii incluse care sunt create odată cu baza de date (vezi Anexa 4C). Acestea sunt următoarele:

- CONNECT oferă privilegiile de bază pentru utilizatorul unei aplicații;
- RESOURCE oferă privilegiile de bază pentru programatorul unei aplicații;
- DBA oferă toate privilegiile de sistem cu posibilitatea de a acorda aceste privilegii altor utilizatori sau roluri (cu alte cuvinte are opțiunea WITH ADMIN OPTION, vezi secțiunea 5.8);
- EXP_FULL_DATABASE și IMP_FULL_DATABASE sunt pentru utilizatorii utilitarelor Export și Import (vezi Anexa 1), adică oferă privilegiile necesare operațiilor de export și respectiv import ale bazei de date;
- DELETE_CATALOG_ROLE, EXECUTE_CATALOG_ROLE și SELECT_CATALOG_ROLE oferă privilegiile de accesare a vederilor dicționarului de date și a pachetelor;
- Alte roluri: AQ_ADMINISTRATION_ROLE, AQ_USER_ROLE, RECOVERY_CATALOG_OWNER și SNMPAGENT.

5.4. Utilizatorii bazei de date

Odată cu crearea unei baze de date sunt creați și doi utilizatori ai bazei de date: SYS și SYSTEM. Utilizatorul SYS este proprietarul tuturor tabelelor interne ale bazei de date, pachetelor și procedurilor predefinite etc. El este de asemenea proprietarul vederilor și tabelelor dicționarului de date și are atribuite toate rolurile predefinite, printre care și cel de DBA. SYS constituie temelia bazei de date Oracle. Din acest motiv, trebuie evitată pe cât posibil folosirea acestui utilizator; o simplă comandă greșită făcută de cineva conectat la baza de date ca SYS poate avea efect devastator.

Utilizatorul SYS este singurul care poate avea acces la anumite tabele ale dicționarului de date. Datorită faptului că acesta este proprietarul tuturor structurilor dicționarului de date, pentru a acorda privilegii asupra obiectelor dicționarului de date este necesară conectarea ca SYS. Parola inițială pentru utilizatorul SYS este CHANGE_ON_INSTALL, dar, datorită importanței acestui utilizator, este recomandată schimbarea acesteia imediat după instalare.

Utilizatorul SYSTEM este de asemenea creat odată cu crearea bazei de date și, în mod normal, este utilizat inițial pentru crearea altor utilizatori și pentru administrarea bazei de date. Ca și SYS, și SYSTEM are drepturi depline asupra tuturor obiectelor bazei de date, având acordat rolul de DBA și, de aceea, de multe ori contul SYSTEM se utilizează pentru a administra baza de date. Totuși, este preferabil ca pentru administrarea bazei de date să se creeze un utilizator separat cu privilegiile de DBA. Parola inițială pentru utilizatorul SYSTEM este MANAGER și se recomandă schimbarea ei imediat după crearea bazei de date.

Utilizatorii SYS și SYSTEM nu trebuie în nici un caz folosiți pentru a crea sau interoga obiecte specifice aplicațiilor sau pentru a interacționa direct cu aplicațiile.

Pentru baza de date exemplu, se creează automat mai mulți utilizatori. Aceștia sunt prezentați în tabelul de mai jos:

Nume cont utilizator	Parola	Descriere
INTERNAL	ORACLE (dacă baza de date a fost instalată folosind opțiunea "Custom", parola se specifică în timpul instalării)	Numele administratorului bazei de date cu care se realizează anumite operații precum pornirea și oprirea bazei de date. INTERNAL nu este un utilizator, el este de fapt un alias al utilizatorului SYS cu privilegiul suplimentar SYSDBA.
SYS	CHANGE_ON_INSTALL	Numele administratorului bazei de date ce are următoarele roluri: <ul style="list-style-type: none"> - CONNECT; - RESOURCE; - DBA; - EXP_FULL_DATABASE; - IMP_FULL_DATABASE; - DELETE_CATALOG_ROLE; - EXECUTE_CATALOG_ROLE; - SELECT_CATALOG_ROLE; - AQ_ADMINISTRATION_ROLE; - AQ_USER_ROLE; - RECOVERY_CATALOG_OWNER; - SNMPAGENT .
SYSTEM	MANAGER	Numele utilizatorului bazei de date ce deține numai rolul DBA.
SCOTT	TIGER	Numele utilizatorului bazei de date ce deține rolurile CONNECT și RESOURCE.
DEMO	DEMO	Numele utilizatorului bazei de date ce deține rolurile CONNECT și RESOURCE. Este recomandată stergerea acestui cont în cazul în care nu este folosit.
DBSNMP	DBSNMP	Numele utilizatorului bazei de date ce deține rolurile CONNECT, RESOURCE și SNMPPAGENT.

Figura 5.3

5.5. Schema

Schema reprezintă o colecție de obiecte care sunt proprietatea unui utilizator. Un utilizator poate avea o singură schema, care va avea același nume ca acesta. O schema este creată pentru a administra o aplicație, având drepturi depline asupra tuturor obiectelor aplicației. Acest lucru are anumite avantaje dintr-o serie de motive: separă în general administrarea obiectelor bazei de date de administrarea obiectelor aplicației și permite ca toate datele aplicației să poată fi exportate sau mutate de către proprietarul acestora.

Între spațiile tabel și schema unei baze de date nu există nici o legătură; obiectele unei scheme se pot găsi în spații tabel diferite, după cum și într-un spațiu tabel se pot găsi obiecte din mai multe scheme.

90 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Pentru a accesa un obiect din propria schema, un utilizator poate folosi doar numele acestuia. Pentru a accesa un obiect din schema altui utilizator, trebuie specificat atât numele obiectului cât și schema din care face parte, folosind sintaxa:

```
schema . obiect
```

De exemplu, un utilizator, să-l numim **costel**, va putea crea în schema proprie un tabel numit **persoana**, folosind comanda:

```
CREATE TABLE persoana...;
```

Dacă însă un alt utilizator dorește crearea tabelului **persoana** în schema utilizatorului **costel**, va trebui să folosească comanda:

```
CREATE TABLE costel.persoana ...;
```

Pentru interogarea tabelului, utilizatorul **costel** va folosi comanda:

```
SELECT * FROM persoana;
```

În schimb, pentru a interoga tabelul, orice alt utilizator va folosi comanda:

```
SELECT * FROM costel.persoana;
```

În plus, pentru a accesa un obiect din schema altui utilizator, un utilizator va trebui să posede privilegiile necesare respectivului tip de acces. De exemplu, pentru interogarea tabelului **persoana**, orice alt utilizator va trebui să posede privilegiul **SELECT** asupra tabelului **persoana** sau privilegiul de sistem **SELECT ANY TABLE**.

În general, trebuie să existe măcar trei tipuri de acces la o aplicație: administratorul bazei de date, dezvoltatorul și utilizatorul:

- *Administratorul bazei de date* va trebui să administreze structurile și obiectele bazei de date (cum ar fi fișierele, spațiile tabel și tabelele). Administratorul bazei de date este de fapt proprietarul aplicației și al bazei de date. El trebuie să dețină o varietate de privilegii de sistem.
- *Dezvoltatorul* va trebui să poată efectua atât operații de interogare (DQL), cât și de manipulare (DML) și definire (DDL) a datelor. Pentru aplicațiile de dimensiuni reduse, proprietarul aplicației este de obicei utilizat și pentru dezvoltare. În general însă, pentru dezvoltare se folosesc o altă schemă decât cea care deține obiectele aplicației, deci un dezvoltator trebuie să dețină în principal privilegii la nivel de obiect.
- *Utilizatorul*, pe de altă parte, va putea efectua doar operații de interogare (DQL) și manipulare a datelor (DML), fără a avea însă permișionarea de a efectua operații de definire a datelor (DDL).

Schela care deține obiectele aplicației va acorda privilegiile necesare fiecărui tip de acces la aplicație.

5.6. Crearea, modificarea și distrugerea utilizatorilor

Utilizatorii bazei de date pot fi creați folosind comanda SQL CREATE USER, având sintaxa următoare:

```
CREATE USER nume_utilizator
IDENTIFIED {BY parola | EXTERNALLY}
[DEFAULT TABLESPACE nume_spațiu_tabel]
[TEMPORARY TABLESPACE nume_spațiu_tabel]
[QUOTA {întreg [K|M] |UNLIMITED} nume_spațiu_tabel ]...
[PROFILE nume_profil ]
[PASSWORD EXPIRE]
[ACCOUNT {LOCK|UNLOCK}]
```

unde

- IDENTIFIED indică modul în care Oracle permite accesul utilizatorului: prin parolă, când Oracle menține în mod intern o parolă pentru utilizator, pe care acesta o va utiliza pentru accesarea bazei de date; EXTERNALLY, când verificarea accesului utilizatorului este făcută de către sistemul de operare - în acest caz utilizatorul trebuie să fie identic cu cel definit în sistemul de operare.
- DEFAULT TABLESPACE indică spațiul tabel folosit în mod implicit de obiectele create de către utilizator. Dacă această clauză este omisă, spațiul tabel implicit este SYSTEM.
- TEMPORARY TABLESPACE indică spațiul tabel pentru segmentele temporare ale utilizatorului. Dacă această clauză este omisă, spațiul tabel implicit pentru segmentele temporare este SYSTEM.
- QUOTA permite utilizatorului să aloce spațiu într-un spațiu tabel; la folosirea acestei clauze poate fi specificat spațiul maxim alocat (in bytes, Kbytes (K) sau Mbytes (M)) sau poate fi folosită opțiunea UNLIMITED pentru a permite utilizatorului să aloce oricât de mult spațiu în spațiul tabel.
- PROFILE atribuie utilizatorului un profil. Un profil este un set de limitări pentru resursele bazei de date (de exemplu, numărul maxim de sesiuni concurente, timpul maxim al unei sesiuni, etc.). Dacă profilul este atribuit unui utilizator, atunci utilizatorul nu poate depăși limitele specificate de profil. Un profil poate fi creat folosind comanda SQL CREATE PROFILE. Dacă această clauză este omisă, Oracle atribuie în mod implicit utilizatorului profilul DEFAULT.
- PASSWORD EXPIRE forcează ca parola inițială să fie schimbată de către utilizator în momentul în care acesta se va conecta pentru prima dată la baza de date.
- ACCOUNT LOCK|UNLOCK blochează sau deblochează contul permitând sau nu accesul utilizatorului la baza de date. Dacă se folosește opțiunea de blocare (LOCK), contul utilizatorului este creat însă utilizatorul nu se va putea conecta până când nu se deblochează în mod explicit contul. Această opțiune se folosește, de exemplu, în cazul în care se preferă crearea unui cont pentru un angajat nou dar nu i se permite accesul până când nu este pregătit să-l folosească. Opțiunea implicită este UNLOCK.

Pentru a putea crea un alt utilizator, trebuie ca utilizatorul ce realizează acest lucru să dețină privilegiul de sistem CREATE USER.

Exemplul următor creează utilizatorul **costel** cu parola **costica**, având ca spațiu tabel implicit **ts_alfa**, spațiu tabel temporar **ts_temp** și cotă de spațiu nelimitată pe spațiile tabel **ts_alfa**, **ts_beta** și **ts_temp**.

```
CREATE USER costel
IDENTIFIED BY costica
DEFAULT TABLESPACE ts_alfa
TEMPORARY TABLESPACE ts_temp
QUOTA UNLIMITED ON ts_alfa
QUOTA UNLIMITED ON ts_beta
QUOTA UNLIMITED ON ts_temp;
```

Caracteristicile unui utilizator pot fi schimbate folosind comanda **SQL ALTER USER**. Folosind această comandă se poate schimba, de exemplu, parola unui utilizator:

```
ALTER USER costel
IDENTIFIED BY costi;
```

Distrugerea unui utilizator se poate face folosind comanda **SQL DROP USER**. Dacă există obiecte în schema utilizatorului, atunci acestea trebuie distruse înainte de distrugerea utilizatorului. Aceasta se poate face automat folosind opțiunea **CASCADE** a acestei comenzi:

```
DROP USER costel CASCADE;
```

5.7. Crearea, modificarea și ștergerea rolurilor

Un rol poate fi creat folosind comanda **SQL CREATE ROLE**, având sintaxa următoare:

```
CREATE ROLE rol
[ NOT IDENTIFIED | IDENTIFIED {BY parola | EXTERNALLY} ]
```

unde:

- NOT IDENTIFIED arată că orice utilizator căruia îi va fi acordat rolul creat nu va trebui identificat în momentul când activează rolul (comanda **SET ROLE**, vezi secțiunea 5.10);
- IDENTIFIED arată că orice utilizator căruia îi va fi acordat rolul creat va trebui identificat. Această identificare poate fi făcută intern de către Oracle prin parolă sau extern prin verificarea utilizatorului sistemului de operare (**EXTERNALLY**). În acest ultim caz, în funcție de sistemul de operare, utilizatorul va trebui să specifice o parolă sistemului de operare la activarea rolului;
- Dacă ambele opțiuni NOT IDENTIFIED și IDENTIFIED sunt omise, opțiunea implicită este NOT IDENTIFIED.

```
CREATE ROLE rol_costel;
CREATE ROLE tanta IDENTIFIED BY tantica;
```

Un utilizator poate crea un rol numai dacă deține privilegiul de sistem **CREATE ROLE**.

Schimbarea modului de identificare a unui rol poate fi făcută folosind comanda SQL ALTER ROLE:

```
ALTER ROLE tanta NOT IDENTIFIED;
```

Distrugerea unui rol se poate face folosind comanda SQL DROP ROLE:

```
DROP ROLE tanta;
```

5.8. Acordarea rolurilor și privilegiilor

Privilegiile de sistem, privilegiile la nivel de obiect sau rolurile pot fi acordate unui utilizator sau unui rol folosind comanda SQL GRANT. Totuși, sintaxa comenzii GRANT pentru acordarea de privilegii de sistem sau roluri diferă de sintaxa pentru acordarea de privilegii la nivel de obiect.

Bineînțeles, pentru a putea acorda un privilegiu sau un rol, utilizatorul respectiv trebuie să aibă acest drept. De exemplu, pentru a acorda un privilegiu de sistem sau un rol, utilizatorul trebuie să aibă privilegiul sau rolul respectiv acordat cu opțiunea ADMIN OPTION sau să aibă privilegiul GRANT ANY PRIVILEGE, respectiv GRANT ANY ROLE.

Acordarea unui privilegiu de sistem sau a unui rol

Pentru acordarea de privilegii de sistem sau roluri sintaxa comenzii GRANT este următoarea:

```
GRANT {privilegiu_de_sistem|rol}
[,{privilegiu_de_sistem|rol}]{...}
TO {utilizator|rol|PUBLIC} [, {utilizator|rol|PUBLIC}]{...}
[WITH ADMIN OPTION]
```

unde:

- PUBLIC este folosit pentru a acorda privilegiile de sistem sau rolurile specificate tuturor rolurilor și utilizatorilor existenți;
- WITH ADMIN OPTION permite utilizatorului (sau rolului) căruia îi este acordat rolul (sau privilegiul) să acorde la rândul lui rolul (sau privilegiul) altui utilizator (sau rol). De asemenea, dacă unui utilizator sau rol îi este acordat un rol folosind WITH ADMIN OPTION, atunci utilizatorul sau rolul respectiv poate modifica (ALTER ROLE) sau distruge (DROP ROLE) rolul acordat.

Un rol nu se poate acorda lui însuși în mod direct sau printr-un set circular de atribuiri. Dacă se încearcă acest lucru, Oracle va genera un mesaj de eroare.

În continuare vom prezenta câteva exemple de acordare a privilegiilor de sistem sau a rolurilor.

94 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Acordarea privilegiilor de sistem CREATE CLUSTER și CREATE TABLE pentru rolul rol_costel:

```
GRANT CREATE CLUSTER, CREATE TABLE  
TO rol_costel;
```

Acordarea privilegiului de sistem CREATE SESSION pentru utilizatorul costel, permisând acestuia conectarea la baza de date:

```
GRANT CREATE SESSION  
TO costel;
```

Acordarea rolului rol_costel pentru utilizatorul costel:

```
GRANT rol_costel  
TO costel;
```

Acordarea unui privilegiu la nivel de obiect

Pentru acordarea de privilegii la nivel de obiect sintaxa comenzi GRANT este ușor diferită deoarece este nevoie să se identifice un obiect specific:

```
GRANT {privilegiu_de_object|ALL} [(coloana [,coloana]...)]  
      [, {privilegiu_de_object|ALL} [(coloana [,coloana]...)]...]...  
ON obiect  
TO {utilizator | rol | PUBLIC}{[,{utilizator} rol | PUBLIC]}...  
[WITH GRANT OPTION]
```

unde:

- ALL este folosit pentru a acorda toate privilegiile pentru obiectul respectiv;
- coloana reprezintă coloana pentru care este acordat privilegiul. Coloanele pot fi specificate numai când sunt acordate privilegiile INSERT, REFERENCES, UPDATE. Dacă nu este listată nici o coloană, atunci privilegiul se acordă pe toate coloanele tabeliei sau vederii în cauză;
- WITH GRANT OPTION permite celui care îi sunt acordate privilegiile de a le acorda la rândul lui altui utilizator sau rol.

Pentru o mai bună înțelegere a acestei comenzi, vom prezenta în continuare câteva exemple:

Acordarea privilegiului SELECT (dreptul de interogare) asupra tabelului studenti pentru utilizatorul costel și pentru rolul rol_costel, cu opțiunea ca aceștia să îl poată acorda mai departe:

```
GRANT SELECT ON studenti  
TO costel, rol_costel  
WITH GRANT OPTION;
```

Acordarea privilegiului UPDATE (dreptul de actualizare) asupra coloanelor nume și prenume ale tabelului studenti pentru utilizatorul costel:

```
GRANT UPDATE(nume, prenume) ON studenti
TO costel;
```

Acordarea privilegiului INSERT (dreptul de inserare a înregistrărilor) și a privilegiului DELETE (dreptul de stergere a înregistrărilor) asupra tabelului studenti pentru utilizatorul costel:

```
GRANT INSERT, DELETE ON studenti
TO costel;
```

Acordarea tuturor privilegiilor (SELECT, INSERT, UPDATE, DELETE, ALTER, REFERENCES, INDEX, vezi tabelul din figura 5.2) asupra tabelului studenti pentru utilizatorul costel:

```
GRANT ALL ON studenti
TO costel;
```

Acordarea privilegiului SELECT (dreptul de interogare) asupra tabelului studenti pentru toți utilizatorii și pentru toate rolurile existente:

```
GRANT SELECT ON studenti
TO PUBLIC;
```

Trebuie menționat că utilizatorul care este proprietarul schemei din care face parte obiectul are automat toate privilegiile asupra obiectului cu opțiunea WITH GRANT OPTION.

5.9 Revocarea rolurilor și privilegiilor

Pentru a revoca un privilegiu sau un rol unui alt rol sau unui utilizator se poate folosi comanda SQL REVOKE. Comanda REVOKE trebuie folosită cu foarte mare atenție deoarece poate anula un privilegiu propriu. Ca și în cazul comenzii GRANT, și comanda REVOKE are o sintaxă pentru revocarea privilegiilor de sistem sau a rolurilor și altă sintaxă pentru revocarea privilegiilor la nivel de obiect.

Revocarea unui privilegiu de sistem sau a unui rol

Pentru revocarea de privilegii de sistem sau roluri, sintaxa comenzii REVOKE este următoarea:

```
REVOKE{privilegiu_de_sistem|rol}[,{privilegiu_de_sistem|rol}]...
FROM{utilizator|rol|PUBLIC} [,,{utilizator|rol|PUBLIC}]...
```

Dacă se revocă un rol care conține alte roluri, întregul set de privilegii asociat cu fiecare rol va fi revocat. Dacă oricare dintre rolurile și privilegiile conținute de rolul anulat a fost

acordat și în mod direct utilizatorului, atunci utilizatorul va continua să aibă privilegiile corespunzătoare.

Vom prezenta în continuare câteva exemple:

Revocarea privilegiului CREATE CLUSTER pentru rolul rol_costel:

```
REVOKE CREATE CLUSTER
FROM rol_costel;
```

Revocarea rolului rol_costel pentru utilizatorul costel:

```
REVOKE rol_costel
FROM costel;
```

Revocarea unui privilegiu la nivel de obiect

Pentru a revoca un privilegiu la nivel de obiect, sintaxa comenzi REVOKE este următoarea:

```
REVOKE{privilegiu_de_object|ALL}{(coloana[, coloana]...)}
      [{privilegiu_de_object|ALL}{(coloana[, coloana]...)...}]{...}
ON obiect
FROM{utilizator|rol|PUBLIC}, [{utilizator|rol|PUBLIC}]{...}
[CASCADE CONSTRAINTS]
[FORCE]
```

unde:

- CASCADE CONSTRAINTS indică faptul că se va șterge orice restricție de integritate referențială definită asupra tabelului de către utilizatorul respectiv. Această opțiune trebuie să fie folosită dacă privilegiul revocat este REFERENCES sau ALL PRIVILEGES și dacă cel căruia i se revocă privilegiul a definit o restricție de integritate referențială asupra tabelului.
- FORCE forțează revocarea privilegiului EXECUTE asupra unui tip de date definit de către utilizator care are tabele dependente, aceste tabele devenind în acest caz invalide. În absența acestei opțiuni, dacă există tabele dependente, revocarea privilegiului EXECUTE asupra tipului de date va eşua. Această opțiune este folosită numai dacă se utilizează opțiunea obiect din Oracle8 (vezi capitolul 10).

Pentru a revoca toate privilegiile utilizatorului costel definite pe tabelul persoana, se va folosi următoarea comandă:

```
REVOKE ALL ON persoana FROM costel;
```

Pentru a revoca numai privilegiul SELECT definit de toți utilizatorii și rolurile existente pentru tabelul persoana, se va folosi următoarea comandă:

```
REVOKE SELECT ON persoana FROM PUBLIC;
```

5.10. Activarea și dezactivarea rolurilor unui utilizator

La conectarea unui utilizator, Oracle va activa toate rolurile implicate (default) ale utilizatorului. Cu excepția situației în care rolurile implicate se definesc folosind clauza DEFAULT ROLE a comenzi SQL ALTER USER, toate rolurile acordate utilizatorului sunt rouri implicate. Folosind comanda SQL ALTER USER cu clauza DEFAULT ROLE se pot identifica exact care din rolurile acordate utilizatorului sunt active și care sunt inactive atunci când utilizatorul se conectează la baza de date. Orice modificare a listei de roluri implicate (făcută prin intermediul comenzi ALTER TABLE) va avea efect începând cu următoarea conectare a utilizatorului la baza de date.

În cadrul unei sesiuni, rolurile atribuite unui utilizator pot fi activate sau dezactivate folosind comanda SQL SET ROLE, având sintaxa următoare:

```
SET ROLE{rol[IDENTIFIED BY parola]
           [,rol[IDENTIFIED BY parola]]...
           | ALL[EXCEPT rol [,rol]...]
           | NONE}
```

Așa cum se observă din sintaxa prezentată mai sus, există trei opțiuni alternative care au următoarea semnificație:

- Lista rolurilor specifică rolurile care sunt active pentru secțiunea curentă. Toate rolurile care nu sunt listate vor fi dezactivate pentru sesiunea curentă (vezi primele două exemple de mai jos). În cazul când rolul are o parolă, pentru activarea rolului este necesară specificarea acesteia.
- ALL activează pentru sesiunea curentă toate rolurile acordate utilizatorului, mai puțin cele enumerate în clauza EXCEPT, dacă ea există. Această opțiune nu poate fi folosită pentru activarea rolurilor cu parola.
- NONE dezactivează toate rolurile pentru sesiunea curentă

În continuare vom prezenta câteva exemple de folosire a acestei comenzi:

Activarea rolului `rol_costel`:

```
SET ROLE rol_costel;
```

Activarea rolului `tanta` identificat prin parola `tantica`:

```
SET ROLE tanta IDENTIFIED BY tantica;
```

Activarea tuturor rolurilor pentru sesiunea curentă:

```
SET ROLE ALL;
```

Activarea tuturor rolurilor în afară de rolul `tanta`:

```
SET ROLE ALL EXCEPT tanta;
```

Dezactivarea tuturor rolurilor pentru sesiunea curentă:

```
SET ROLE NONE;
```

Pentru a determina rolurile active din sesiunea curentă se poate examina vedereana SESSION_ROLES din dicționarul de date (vezi Anexa 5). Numărul maxim de roluri care pot fi active la un anumit moment este specificat de către parametrul de initializare MAX_ENABLED_ROLES.

Capitolul 6

Organizarea logică a bazei de date

Dezvoltatorul de aplicații trebuie să fie profund conștient de organizarea logică a bazei de date. La nivel logic, baza de date este alcătuiră din *scheme*. O schemă este o colecție de structuri logice de date, numite și obiecte ale schemei. După cum am văzut în capitolul anterior, o schemă este proprietatea unui utilizator al bazei de date și are același nume cu acesta. De aceea se mai spune că obiectele schemei sunt proprietatea utilizatorului respectiv.

Definițiile tuturor obiectelor schemei sunt păstrate în dicționarul bazei de date. După cum vom vedea în continuare, unele dintre obiectele schemei (tabele, clustere, indecsi) conțin date, pentru care este necesar un spațiu de stocare. Datele din fiecare astfel de obiect sunt stocate din punct de vedere logic într-un spațiu tabel. Din punct de vedere fizic, aceste date sunt stocate într-unul sau mai multe din fișierele de date asociate aceluiași spațiu tabel. În general, într-un spațiu tabel sunt stocate mai multe obiecte. După cum am văzut în capitolul 4, la crearea tabelelor, clusterelor sau indecsilor se poate specifica spațiul tabel corespunzător și spațiul alocat obiectului creat (prin intermediul parametrilor de stocare).

Obiectele schemei pot fi create și manipulate folosind comenzi SQL. Principalele obiecte ale schemei sunt următoarele:

- tabele (tables);
- vederi (views);
- indecsi (indexes);
- clustere (clusters) și clustere hash (hash cluster);
- secvențe (sequences);
- sinonime (synonyms);
- proceduri și funcții stocate/rezidente (stored procedures and functions);
- pachete stocate (stored packages);
- declanșatoare ale bazei de date (database triggers);
- instantane (snapshots);
- legături ale bazei de date (database link).

Acest capitol prezintă pe larg fiecare dintre aceste obiecte. În plus, ultima secțiune din capitol este dedicată dicționarului bazei de date.

6.1. Tabele

Tabelul este principala structură logică de stocare a datelor. După cum am văzut în capitolul 1, un tabel este o structură bidimensională formată din *coloane* și *rânduri*. Coloanele mai sunt numite și *câmpuri*, iar rândurile *înregistrări*. În general, fiecare tabel este stocat într-un

spațiu tabel¹. După cum am văzut în secțiunea 4.2.1, porțiunea dintr-un spațiu tabel folosită pentru stocarea datelor unui tabel se numește segment tabel. Cu alte cuvinte, segmentul tabel este omologul fizic al unui tabel.

În anumite situații, pentru a mări eficiența operațiilor de scriere/citire a datelor, mai multe tabele pot fi stocate împreună, formând *clustere* (*grupuri de tabele*). Despre acestea vom vorbi mai târziu, în secțiunea 6.4. O nouitate adusă de versiunea Oracle8 este posibilitatea de crea un tabel pe baza unui index, reducând astfel timpul de acces la date prin interogări care folosesc ca termen de comparație coloanele indexate. Despre acestea vom vorbi în secțiunea 6.3.3, deocamdată referindu-ne doar la tabele obișnuite.

6.1.1. Crearea tabelelor

Un tabel poate fi creat prin comanda SQL CREATE TABLE, în care trebuie specificat numele și tipul de date pentru fiecare coloană a tabelului. De exemplu:

```
CREATE TABLE salariat(
cod_salariat      NUMBER(10),
nume               VARCHAR2(10),
prenume            VARCHAR2(10),
data_nastere       DATE,
salariu            NUMBER(10),
manager             NUMBER(10),
cod_dept           NUMBER(10),
cod_tara            NUMBER(10));
```

O sintaxă simplificată a comenzi CREATE TABLE este prezentată în continuare:

```
CREATE TABLE nume_tabel
(nume_coloana tip_data [DEFAULT expresie]
 [, nume_coloana tip data [DEFAULT expresie] ... ]
 [PCTFREE întreg] [PCTUSED întreg]
 [TABLESPACE spațiu_tabel]
 [STORAGE parametrii_de_stocare]
```

unde:

- DEFAULT desemnează o valoare implicită pentru coloană, folosită în cazul în care la inserarea unui rând în tabel nu este specificată o valoare explicită pentru coloana în cauză;
- TABLESPACE specifică spațiul tabel în care va fi stocat tabelul. Dacă acesta nu este menționat explicit, se va folosi spațiul tabel implicit (default) al utilizatorului care este proprietarul schemei din care face parte tabelul (vezi exemplul de mai sus);
- Valorile parametrilor PCTFREE și PCTUSED determină gradul de utilizare a blocurilor din extinderile segmentului tabel, vezi secțiunea 4.4;
- Clauza STORAGE este folosită pentru setarea parametrilor de stocare (INITIAL, NEXT, PCTINCREASE, MINEXTENTS, MAXEXTENTS) prin intermediul căror se specifică mărimea și modul de alocare a extinderilor segmentului tabel, vezi secțiunea 4.1.1.

```
CREATE TABLE salariat(
cod_salariat      NUMBER(10),
nume               VARCHAR2(10),
```

¹ O nouitate adusă de Oracle8, despre care vom vorbi ceva mai târziu este posibilitatea de a partiona un tabel, fiecare parte putând fi stocată într-un spațiu tabel diferit

```

prenume          VARCHAR2(10),
data_nastere    DATE,
salariu         NUMBER(10),
manager          NUMBER(10),
cod_dept        NUMBER(10),
cod_tara         NUMBER(10) DEFAULT 40)

PCTFREE 20 PCTUSED 70
TABLESPACE ts_alfa
STORAGE (INITIAL 100K NEXT 100K);

```

La crearea unui tabel este necesară specificarea tipului de dată pentru fiecare coloană a tabelului. Următorul tabel arată tipurile de date scalare cel mai des folosite.

Tip de dată	Descriere
VARCHAR2 (n)	Șiruri de caractere de lungime variabilă având lungimea maximă <i>n</i> bytes. Lungimea maximă <i>n</i> trebuie neapărat specificată. În versiunea Oracle 8, valoarea maximă a lungimii <i>n</i> de 4000.
CHAR (n)	Șiruri de caractere de lungime fixă <i>n</i> bytes. Valoarea implicită pentru <i>n</i> este 1. Dacă într-o coloană având acest tip se inserează șiruri de caractere mai scurte decât lungimea specificată, atunci Oracle inserează la dreapta numărul necesar de spații libere (blank-uri) pentru atingerea lungimii specificate. În versiunea Oracle 8, valoarea maximă a lungimii <i>n</i> este de 2000.
NUMBER (n, m)	Numere cu precizia <i>n</i> și scală <i>m</i> . Precizia reprezintă numărul maxim de digiți permis, care nu poate depăși 38. Scala reprezintă numărul de zecimale pe care le va avea numărul și poate avea valori între -84 și 127.
NUMBER (n)	Numere întregi având precizia maximă <i>n</i> . Valoarea maximă pentru <i>n</i> este de 38.
NUMBER	Numere în virgulă inobilă având o precizie (număr maxim de digiți) de 38 de digiți.
DATE	Date calendaristice având valori între 1 Ianuarie 4712 î. Cr. și 31 Decembrie 4712 d. Cr. Pentru fiecare dată calendaristică sunt înregistrate următoarele informații: secolul, anul, luna, ziua, ora, minutul și secunda. Pentru a specifica o valoare de tip dată calendaristică, este necesară convertirea unui șir de caractere sau a unui număr folosind funcția TO_DATE. Atunci când sunt folosite în expresii de tip dată calendaristică, Oracle convertește automat șiruri de caractere care au formatul implicit de dată calendaristică. Formatul implicit de dată calendaristică este specificat de parametrul de inițializare NLS_DATE_FORMAT și este un șir de caractere. De exemplu, acesta poate fi 'DD-MON-YY', care cuprinde un număr de doi digiți pentru ziua din lună, o abreviere a numelui lunii și ultimii doi digiți ai anului – de exemplu '01-JAN-99' reprezintă 1 Ianuarie 1999.
LONG	Șiruri de caractere de dimensiune variabilă până la 2 Gbytes sau $2^{31}-1$ bytes. O singură coloană de tip LONG este admisă în cadrul unui tabel.
RAW (n)	Se folosește pentru a stoca date binare (șiruri de biți) de lungime variabilă, având lungimea maximă <i>n</i> bytes. Valoarea lui <i>n</i> trebuie specificată și trebuie să nu depășească 2000. Se poate folosi pentru stocarea imaginilor grafice sau a sunetului digital. Este similar cu VARCHAR2, cu excepția dimensiunii maxime și a faptului că pentru tipul de dată RAW nu se pot interpreta datele.
LONG RAW	Se folosește pentru a stoca date binare (șiruri de biți) de lungime variabilă de până la 2Gbytes. Tipul de dată LONG RAW este similar tipului de dată LONG, cu excepție făcând faptul că pentru tipul de dată LONG RAW nu se pot interpreta datele.

Tipul de dată poate fi urmat, între paranteze, de unul sau mai multe numere care furnizează informații despre dimensiunea coloanei. Dimensiunea coloanei determină dimensiunea maximă a oricărei valori pe care o poate avea coloana. Coloanele de tip VARCHAR2 trebuie să aibă specificată o mărime. Coloanele NUMBER și CHAR pot avea o mărime specificată, dar în lipsa acesteia se folosește o valoare implicită.

Alte date tipuri de date scalare furnizate de Oracle SQL sunt NCHAR și NVARCHAR2, folosite pentru reprezentarea caracterelor limbilor naționale. Pentru o descriere mai detaliată a acestor tipuri de date se poate consulta Anexa 3, care cuprinde toate tipurile de date din PL/SQL (printre care se regăsesc și tipurile de date Oracle SQL). În Oracle8, alături de aceste tipuri de date scalare, există și tipuri de date LOB (Large Objects), care specifică locația unor obiecte de dimensiuni mari. O descriere mai detaliată a tuturor acestor tipuri de date se găsește în Capitolul 10 și Anexa 3. În plus, opțiunea obiect din Oracle8 permite definirea de către utilizator a unor tipuri de date. Această opțiune va fi discutată în detaliu în Capitolul 10.

La crearea unui tabel nu este nevoie să se specifice dimensiunea maximă a acestuia, ea fiind determinată până la urmă de cât de mult spațiu a fost alocat spațiului tabel în care este creat tabelul. Unui tabel îl poate fi repartizat mai mult spațiu în mod automat, în cazul în care spațiul alocat inițial a fost umplut.

6.1.2. Tabele partitionate

O nouitate introdusă în Oracle8 este posibilitatea de a partiona tabele, adică de a împărti tabelul în mai multe părți independente, fiecare cu parametri de stocare potențial diferiți și cu posibilitatea ca părți diferite ale tabelului să se găsească pe spații tabel diferite. Fiecare partitură a tabelului va conține înregistrări ce au valoarea cheii într-un anumit interval specificat. În acest sens, partitoningarea este foarte folositoare în cazul tabelelor de dimensiuni foarte mari.

Partitoningarea este transparentă pentru utilizatori și aplicații. Utilizarea tabelelor partitionate oferă câteva avantaje. Dacă o parte a tabelului este inaccesibilă, celelalte părți sunt disponibile pentru inserare, selecție, modificare și ștergere; numai acele înregistrări care sunt în acea partitură nu vor fi accesibile. De asemenea, se poate bloca accesul la o parte a tabelului în timp ce restul înregistrărilor sunt disponibile.

Fiecare partitură poate avea proprii săi parametri de stocare PCTFREE și PCTUSED, INITIAL, NEXT, PCTINCREASE, MINEXTENTS, MAXEXTENTS. Acest lucru este important deoarece o parte a tabelului poate să conțină un număr mult mai mare de înregistrări decât alta, necesitând, pentru o funcționare eficientă, parametri diferiți de stocare. Posibilitatea de a atribui în mod individual parametri de stocare fiecărei părți oferă o mai mare flexibilitate în stocarea datelor. De asemenea, fiecare parte poate fi stocată în spații tabel diferite. Acest lucru este avantajos în cazul în care unul dintre spațiile tabel este inaccesibil.

Crearea tabelelor partitionate

Sintaxa comenzi CREATE TABLE în cazul partitoningării tabelului este:

```
CREATE TABLE nume_tabel
  (nume_colonă tip_dată [DEFAULT expresie]
   [, nume_colonă tip_dată [DEFAULT expresie]]... )
```

```
PARTITION BY RANGE (listă_coloane)
(PARTITION nume_partiție
VALUES[LESS|GREATER] THAN (listă_valori)
[PCTFREE întreg] [PCTUSED întreg]
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]
[, PARTITION nume_partiție
VALUES[LESS|GREATER] THAN (listă_valori)
[PCTFREE întreg] [PCTUSED întreg]
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]]...)
```

unde:

- `listă_coloane` este o listă ordonată de coloane care determină partitia;
- `listă_valori` este o listă ordonată de valori pentru coloanele din `listă_coloane`

```
CREATE TABLE salariat_part(
cod_salariat          NUMBER(10),
nume                   VARCHAR2(10),
prenume                VARCHAR2(10),
data_nastere           DATE,
salariu                NUMBER(10),
manager                NUMBER(10),
cod_dept               NUMBER(10),
cod Tara                NUMBER(10))
PARTITIONED BY RANGE(salariu)
(PARTITION salariu_mic VALUES LESS THAN (1000)
  TABLESPACE ts_alfa
  STORAGE (initial 50K next 50K),
PARTITION salariu_mediu VALUES LESS THAN (10000)
  TABLESPACE ts_beta
  STORAGE (initial 100K next 100K),
PARTITION salariu_mare VALUES LESS THAN (MAXVALUE)
  TABLESPACE ts_alfa
  STORAGE (initial 50K next 50K));
```

Notă: `MAXVALUE` are practic semnificația de "infinit", ultima parte a tabelului cuprindând valori de peste 10000.



6.1.3. Constrângerile

Alături de numele și tipurile de date ale coloanelor, la definirea unui tabel se pot specifica și *constrângerile de integritate (constraints)*. În Oracle, constrângerile sunt folosite pentru a impune anumite restricții asupra datelor tabelului sau pentru a păstra integritatea referențială a bazei de date. Constrângerile se pot defini la nivel de coloană sau la nivel de tabel, după cum ele se referă la datele unei singure coloane sau la datele mai multor coloane. În Oracle există următoarele tipuri de constrângerile:

Constrângere	Nivel de definire	Funcționalitate
NOT NULL	Coloană	Impune ca valorile coloanei să fie diferite de Null.
UNIQUE	Coloană, tabel	Impune unicitatea valorilor unei coloane sau a unei combinații de coloane.
PRIMARY KEY	Coloană, tabel	Impune unicitatea valorilor unei coloane sau a unei combinații de coloane. În plus, valorile Null nu sunt permise în coloanele care fac parte din PRIMARY KEY. Într-un tabel poate exista o singură cheie primară.
[FOREIGN KEY] REFERENCES	Coloană, tabel	Impune regula de integritate referențială în cadrul aceluiași tabel sau între tabele diferite. O cheie străină este folosită în relație cu o coloană sau combinație de coloane definite ca UNIQUE sau PRIMARY KEY
CHECK	Coloană	Defineste explicit o condiție pe care trebuie să o satisfacă datele din fiecare rând al tabelului

În cazul folosirii constrângerilor la definirea unui tabel, sintaxa comenzi SQL CREATE TABLE se completează în modul următor:

```
CREATE TABLE nume_tabel
  (nume_coloana tip_data[DEFAULT expresie]
    [constr_coloana [constr_coloana]...]
    [nume_coloana_tip_data[DEFAULT expresie]
      [constr_coloana [constr_coloana]...]] ...
  constr_tabel [,constr_tabel] ....)
  ...
```

Sintaxa unei constrângerii la nivel de coloană este:

```
[CONSTRAINT nume constrângere]
  {NOT NULL | UNIQUE | PRIMARY KEY
  | REFERENCES tabel(coloana) [ON DELETE CASCADE]
  | CHECK (condiție)}
```

iar sintaxa unei constrângerii la nivel de tabel este:

```
[CONSTRAINT nume constrângere]
  {UNIQUE|PRIMARY KEY|
  FOREIGN KEY (coloana [,coloana] ...)
  REFERENCES tabel(coloana)
  [ON DELETE CASCADE]}
```

unde:

- CONSTRAINT permite specificarea unui nume pentru integritatea definită. Dacă această opțiune este omisă, Oracle va genera în mod automat un nume, de forma SYS_C_n, unde _n reprezintă un număr care face ca numele constrângerii să fie unic.
- NOT NULL, UNIQUE, PRIMARY KEY, [FOREIGN KEY] REFERENCES, CHECK sunt tipurile de constrângerii descrise în tabelul de mai sus.
- ON DELETE CASCADE este o clauză care se poate folosi la definirea unei restricții de integritate referențială; în acest caz, în cazul ștergerii unei înregistrări care conține cheia primară sau unică la care face referire cheia străină, integritatea referențială este menținută prin ștergerea tuturor înregistrărilor ce conțin cheii străine dependente.

În exemplul de mai jos sunt create două tabele departament și salariat, fiind impuse următoarele constrângeri:

- combinația (cod_dept, cod_tara) este cheia primară a tabelului departament. În acest caz, constrângerea este definită la nivel de tabel.
- cod_salarat este cheia primară a tabelului salariat. În acest caz, constrângerea este definită la nivel de coloană.
- nume este o coloană a tabelului salariat care nu admite valori Null.
- manager este o cheie străină a tabelului salariat care face referință la cheia primară cod_salarat a același tabel. În acest caz, constrângerea este definită la nivel de coloană.
- valorile pentru coloana salariu din tabelul salariat trebuie să fie mai mari ca 0
- combinația de coloane (nume, prenume, data_nastere) din tabelul salariat trebuie să aibă valori unice. În acest caz, constrângerea este definită la nivel de tabel.
- combinația (cod_dept, cod_tara) este o cheie străină a tabelului salariat care face referință la cheia primară a tabelului departament. În acest caz, constrângerea este definită la nivel de tabel. Remarcați că în cazul constrângerii de cheie străină, sintaxa diferă în cazul definirii la nivel de coloană față de cel al definirii la nivel de tabel, în prima situație lipsind cuvintele "FOREIGN KEY".

```

CREATE TABLE departament(
cod_dept      NUMBER(10),
cod_tara      NUMBER(10),
nume_dept     VARCHAR2(10),
CONSTRAINT dept_pk PRIMARY KEY(cod_dept, cod_tara));

CREATE TABLE salariat(
cod_salarat    NUMBER(10)
    CONSTRAINT sal_pk PRIMARY KEY,
nume           VARCHAR2(10) NOT NULL,
prenume        VARCHAR2(10),
data_nastere   DATE,
manager         NUMBER(10)
    CONSTRAINT sal_sal_fk
        REFERENCES salariat(cod_salarat),
salariu        NUMBER(10)
    CONSTRAINT sal_ck CHECK(salariu > 0),
cod_dept       NUMBER(10),
cod_tara       NUMBER(10),
UNIQUE(nume, prenume, data_nastere),
CONSTRAINT sal_dept_fk FOREIGN KEY(cod_dept, cod_tara)
    REFERENCES departament(cod_dept, cod_tara));

```

Constrângerea FOREIGN KEY impune integritatea referențială între tabelul master (departament) și tabelul detaliu (salarat). De exemplu, aceasta înseamnă că un salariat nu poate fi adăugat decât dacă departamentul corespunzător este fie NULL, fie există în tabelul departament. La fel, nu poate fi șters un departament dacă există angajați în acel departament. Există însă și posibilitatea de a permite ștergerea unui departament în care există salariați; în acest caz, pentru menținerea integrității, este necesară și ștergerea tuturor angajaților dependenti. Acest lucru se poate face prin adăugarea clauzei ON DELETE CASCADE pentru constrângerea FOREIGN KEY:

```

CREATE TABLE salariat
cod_salariat      NUMBER(10)
    CONSTRAINT sal_pk PRIMARY KEY,
nume               VARCHAR2(10) NOT NULL,
prenume            VARCHAR2(10),
data_nastere       DATE,
manager             NUMBER(10)
    CONSTRAINT sal_sal_fk
        REFERENCES salariat(cod_salariat),
salariu            NUMBER(10)
    CONSTRAINT sal_ck CHECK(salariu > 1000),
cod_dept            NUMBER(10),
cod_tara            NUMBER(10),
UNIQUE(nume, prenume, data_nastere),
CONSTRAINT sal_dept_fk FOREIGN KEY(cod_dept, cod_tara)
        REFERENCES departament(cod_dept, cod_tara)
        ON DELETE CASCADE);

```

Toate detaliile despre constrângeri sunt stocate în dicționarul de date Oracle. De exemplu, pentru a vizualiza toate constrângările definite pentru tabelele de mai sus putem executa următoarea interogare asupra vederii ALL_CONSTRAINTS:

```

SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME
FROM ALL_CONSTRAINTS
WHERE TABLE_NAME IN ('SALARIAAT', 'DEPARTAMENT');

```

care va produce rezultatul:

CONSTRAINT_NAME	C TABLE_NAME
SYS_C002725	C SALARIAT
SAL_PK	P SALARIAT
SAL_CK	C SALARIAT
SYS_C002728	U SALARIAT
SAL_SAL_FK	R SALARIAT
SAL_DEPT_FK	R SALARIAT
DEPT_PK	P DEPARTAMENT

7 rows selected.

Fiecare constrângere are asociat un nume. În general este convenabil ca acesta să fie dat în mod explicit de cel care creează tabelul (cum este cazul constrângерilor CHECK, PRIMARY KEY și FOREIGN KEY din exemplul de mai sus) pentru că în acest mod constrângerea poate fi referată mai ușor după aceea. În caz contrar (de exemplu constrângările UNIQUE și NOT NULL din exemplul de mai sus) numele este generat automat și are forma "SYS_C...".

Constrângeri amânate

În versiunea Oracle7 fiecare constrângere este verificată de fiecare dată când este executată o instrucțiune DML (inserare, actualizare sau ștergere). În versiunea Oracle8 există posibilitatea ca o constrângere să fie *amânată* (DEFERRED). În cazul acesta, mai multe comenzi SQL pot fi executate fără a se verifica restricția, acesta fiind verificată numai la sfârșitul tranzacției, atunci când este executată instrucțiunea COMMIT. Dacă vreuna dintre

comenzile DML ale tranzacției încalcă restricția, atunci întreaga tranzacție este derulată înapoi și este returnată o eroare.

În Oracle8, orice constrângere pe tabelă sau pe coloană poate fi definită ca amânabilă folosind cuvântul cheie DEFERRABLE; opțiunea contrară este NOT DEFERRABLE care este și opțiunea implicită:

```
{constr_tabel|constr_coloana)
[NOT DEFERRABLE | DEFERRABLE
[INITIALLY IMMEDIATE | INITIALLY DEFERRED]]
```

Când o constrângere este specificată ca fiind DEFERRABLE, se poate specifica în plus starea inițială a constrângerii, care poate fi INITIALLY DEFERRED sau INITIALLY IMMEDIATE, setarea implicită fiind INITIALLY IMMEDIATE. Dacă o constrângere are starea inițială INITIALLY IMMEDIATE, ea este pornită în modul fără amânare, fiind verificată imediat după fiecare instrucție executată. Dacă starea inițială este INITIALLY DEFERRABLE, atunci constrângerea este verificată la executarea unei comenzi COMMIT sau la schimbarea stării constrângerii în IMMEDIATE.

Schimbarea stării unei constrângerii se poate face folosind comanda SQL SET CONSTRAINT:

```
SET CONSTRAINT [DEFERRED | IMMEDIATE]
```

Possibilitatea de a amâna verificarea unei constrângerii este folosită special în cazul unor restricții de integritate referențială, în acest mod fiind posibilă inserarea unor rânduri în tabela copil (detaliu), care conține cheia străină, înaintea rândului corespunzător din tabela părinte (master), care conține cheia primară.

În exemplul de mai jos, cele două restricții de integritate referențială au fost definite ca amânabile.

```
CREATE TABLE departament(
cod_dept          NUMBER(10),
cod_tara          NUMBER(10),
nume_dept         VARCHAR2(10),
CONSTRAINT dept_pk PRIMARY KEY(cod_dept, cod_tara));

CREATE TABLE salariat(
cod_salariat      NUMBER(10)  CONSTRAINT sal_pk PRIMARY KEY,
nume              VARCHAR2(10) NOT NULL,
prenume           VARCHAR2(10),
data_nastere      DATE,
manager            NUMBER(10)
CONSTRAINT sal_sal_fk
    REFERENCES salariat(cod_salariat)   DEFERRABLE,
salariu           NUMBER(10)
CONSTRAINT sal_ck check(salariu > 0),
cod_dept          NUMBER(10),
cod_tara          NUMBER(10),
UNIQUE(nume, prenume, data_nastere),
```

```
CONSTRAINT sal_dept_fk FOREIGN KEY(cod_dept, cod_tara)
    REFERENCES departament(cod_dept, cod_tara) DEFERRABLE;
```

Deoarece starea inițială a restricțiilor nu a fost precizată, ea va fi implicit INITIALLY IMMEDIATE. Pentru a trece restricțiile în starea DEFERRED se folosește instrucțiunea SET CONSTRAINT:

```
SET CONSTRAINT sal_sal_fk DEFERRED;
SET CONSTRAINT sal_dept_fk DEFERRED;
```

De exemplu, următoarea secvență de instrucțiuni SQL se execută cu succes dacă restricția referențială `sal_dept_fk` este DEFERRED, dar eșuează în caz contrar.

```
INSERT INTO salariat(cod_salariat, nume, cod_dept, cod_tara)
VALUES(100, 'Popescu', 1, 40);
```

```
INSERT INTO departament(cod_dept, cod_tara, nume_dept)
VALUES(1, 40, 'IT');
```

```
COMMIT;
```

6.1.4. Crearea și popularea simultană a tabelelor

Atunci când se creează un tabel există posibilitatea ca în același timp tabelul să fie și populat. Pentru aceasta, în cadrul comenzi SQL CREATE TABLE se va utiliza clauza AS urmată de o interogare pe unul sau mai multe tabele. În mod evident, numărul coloanelor din definiția tabelului trebuie să coincidă cu cel din interogare.

Exemplul următor creează un tabel care conține toate înregistrările din tabelul `salariat` având țara cu codul 100:

```
CREATE TABLE salariat_100(
cod_salariat, nume, prenume, data_nastere, manager,
salariu, cod_dept)
AS
SELECT cod_salariat, nume, prenume, data_nastere,
       manager, salariu, cod_dept
FROM salariat
WHERE cod_tara = 100;
```

Atunci când la crearea unui tabel se folosește clauza AS nu este permisă specificarea tipurilor de date ale coloanelor, acestea fiind preluate automat de la tabelul de bază. Pe de altă parte însă, restricțiile definite pentru tabelul de bază, cu excepția celor NOT NULL nu sunt preluate automat de noul tabel. De exemplu, tabelul `salariat_100` va avea o singură restricție de integritate, NOT NULL pentru coloana `nume`. Folosind clauza AS se pot crea tabele și din mai multe tabele de bază, de exemplu un tabel care conține codul, numele și prenumele salariaților precum și numele departamentului în care lucrează se poate crea în modul următor:

```

CREATE TABLE sal_dept_temp(
cod_salariat, nume, prenume, nume_dept)
AS
SELECT s.cod_salariat, s.nume, s.prenume, d.nume_dept
FROM salariat s, departament d
WHERE s.cod_dept = d.cod_dept
AND s.cod_tara = d.cod_tara;

```

6.1.5. Modificarea tabelelor

Un tabel existent poate fi modificat folosind comanda SQL ALTER TABLE. Se pot efectua următoarele tipuri de modificări:

- Adăugarea de noi coloane (împreună cu eventualele constrângeri pentru aceste coloane):

```

ALTER TABLE departament
ADD (localitate VARCHAR2(10) NOT NULL);

```

- Modificarea tipului de date sau a mărimiții unor coloane existente:

```

ALTER TABLE departament
MODIFY (nume_dept VARCHAR2(20));

```

Notă: schimbarea tipului de date al unei coloane sau scăderea dimensiunii acesteia nu este posibilă decât dacă acea coloană este goală; în caz contrar, o astfel de operație ar putea duce la modificarea datelor din tabel.

- Ștergerea unor constrângeri existente:

```

ALTER TABLE salariat
DROP CONSTRAINT sal_ck;

```

Trebuie remarcat că o constrângere PRIMARY KEY la care face referință o constrângere FOREIGN KEY nu poate fiștearsă decât dacă împreună cu constrângerea PRIMARY KEY sunt șterse și toate constrângările referențiale asociate. Pentru acesta se folosește clauza CASCADE.

```

ALTER TABLE departament
DROP CONSTRAINT dept_pk CASCADE;

```

Comanda SQL de mai sus șterge atât constrângerea PRIMARY KEY dept_pk de pe tabelul departament, cât și constrângerea FOREIGN KEY de pe tabelul salariat.

- Adăugarea de noi constrângeri:

```

ALTER TABLE salariat
ADD (CONSTRAINT data_ck CHECK(data_nastere>
'1-Jan-1900'));

```

- Activarea (ENABLE) sau dezactivarea (DISABLE) unor constrângeri existente;

```
ALTER TABLE salariat DISABLE CONSTRAINT sal_dept_fk;
```

La crearea unui tabel, toate constrângările definite sunt implicit active dacă nu a fost folosită opțiunea DISABLE. Dacă o constrângere este dezactivată, atunci asupra datelor pot fi executate operații care încalcă acea constrângere. O constrângere care a fost dezactivată poate fi ulterior activată numai dacă datele care au fost introduse, actualizate sau șterse cât timp ea a fost dezactivată, nu încalcă aceasta constrângere. De exemplu, constrângerea sal_dept_fk poate fi reactivată numai dacă după execuția comenzi de mai sus nu au fost introduse date în tabelul salariat care încalcă integritatea referențială:

```
ALTER TABLE salariat ENABLE CONSTRAINT sal_dept_fk;
```

În versiunea Oracle8, alături de starea activă (ENABLED) și inactivă (DISABLED), o constrângere poate avea o a treia stare: impusă (ENFORCED). Atât restricțiile activate cât și cele dezactivate pot fi trecute în starea ENFORCED. O restricție poate fi trecută în starea ENFORCED folosind comanda ALTER TABLE cu clauza ENFORCE CONSTRAINT:

```
ALTER TABLE salariat ENFORCE CONSTRAINT sal_dept_fk;
```

În cazul executării acestei comenzi, restricția este impusă după executarea comenzi. Deci comanda ALTER TABLE ... ENFORCE CONSTRAINT nu va eșua dacă în tabel există înregistrări care încalcă restricția respectivă (cum se întâmplă în cazul executării unei comenzi ALTER TABLE ... ENABLE CONSTRAINT). Dar, după ce restricția a fost impusă, ea nu va mai permite inserarea sau actualizarea înregistrărilor care nu o respectă, cum s-ar fi întâmplat dacă restricția era dezactivată.

Notă: Comanda SQL ALTER TABLE nu permite ștergerea dintr-un tabel a unei coloane existente. Dacă totuși se dorește acest lucru, se poate folosi comanda CREATE TABLE cu clauza AS, în care se selectează coloanele dorite. De asemenea, comanda SQL ALTER TABLE nu permite modificarea definiției unei constrângeri existente.

- Din punct de vedere fizic, comanda ALTER TABLE permite schimbarea parametrilor PCTFREE și PCTUSED și a parametrilor din clauza STORAGE folosind sintaxa:

```
ALTER TABLE nume_tabel
[PCTFREE întreg] [PCTUSED întreg]
[STORAGE parametrii_de_stocare]
```

- De asemenea, comanda ALTER TABLE permite alocarea și dealocarea manuală a spațiului utilizat de către un tabel. Alocarea manuală a spațiului pentru un tabel se face prin adăugarea de noi extinderi. Alocarea manuală se poate face în general:
 - înainte de o încărcare masivă a datelor;
 - pentru a controla distribuția extinderilor unui tabel în cadrul fișierelor.

Deallocarea spațiului asociat unui tabel reprezintă eliberarea spațiului nefolosit de acesta (care nu a fost niciodată folosit sau care a devenit între timp gol datorită ștergerii de rânduri).

Pentru a aloca sau dealoca spațiu utilizat de un tabel se folosește comanda ALTER TABLE cu următoarele sintaxe:

```
ALTER TABLE nume_tabel
  ALLOCATE EXTENT [([SIZE intreg [K|M]]
    [DATAFILE nume_fișier_de_date] )]
```

respectiv

```
ALTER TABLE nume_tabel
  DEALLOCATE UNUSED [KEEP intreg [K|M]]
```

unde:

- DATAFILE specifică fișierul de date (din spațiu tabel asociat tabelului) care va cuprinde noua extindere. Dacă această opțiune este omisă, fișierul este ales de către Oracle.
- SIZE specifică dimensiunea noii extinderi. Dacă opțiunea SIZE este omisă atunci Oracle va stabili dimensiunea extinderii pe baza parametrilor de stocare a tabelului.
- Cu ajutorul opțiunii KEEP se poate specifica un număr de bytes (Kbytes, Mbytes) din spațiu liber al tabelului ce nu vor fi dealocați.

6.1.6. Distrugerea tabelelor

Pentru a distruge un tabel în Oracle se poate folosi comanda SQL DROP TABLE:

```
DROP TABLE salariat;
```

Pe de altă parte însă, dacă vom folosi o comandă similară pentru a distruge un tabel a cărui cheie primară face referință la o cheie străină a altui tabel, adică ultimul tabel are definită o constrângere FOREIGN KEY corespunzătoare, de exemplu:

```
DROP TABLE departament;
```

atunci existența constrângerii de integritate referențială va împiedica distrugerea tabelului, astfel încât la încercarea de a executa comanda de mai sus se va genera un mesaj de eroare. În astfel de situații, tabelul trebuie distrus împreună cu toate constrângerile FOREIGN KEY care fac referire la cheia primară a acestuia. Acest lucru se poate face prin folosirea comenzii SQL DROP TABLE cu clauza CASCADE CONSTRAINTS:

```
DROP TABLE departament CASCADE CONSTRAINTS;
```

Execuția comenzii de mai sus va duce la distrugerea tabelului departament și a constrângerii referențiale sal_dept_fk.

În momentul în care un tabel este distrus, vor fi șterse automat și toate datele din tabel cât și indecsii asociați lui. Vederile și sinonimele asociate unui tabel care a fost distrus vor rămâne dar vor deveni invalide.

6.2. Vederi

O *vedere* este un “*tabel logic*”, fiind de asemenea organizată în rânduri și coloane. Ea preia rezultatul unei interogări și îl tratează ca pe un tabel, de unde și numele de *tabel logic*. De exemplu, dacă din tabelul *salariat* se dorește vizualizarea doar a cinci coloane (*cod_salariat*, *nume*, *prenume*, *data_nastere*, *cod_dept*) și numai a rândurilor pentru care *cod_tara* = 40, se poate crea o vedere care conține numai aceste linii și coloane, vezi figura 6.1.

tabel de bază *salariat*

<i>cod_salariat</i>	<i>nume</i>	<i>prenume</i>	<i>data_nastere</i>	<i>salariu</i>	<i>manager</i>	<i>cod_dept</i>	<i>cod_tara</i>
101	Popescu	Ion	11-DEC-77	5000		1	44
102	Vasilescu	Vasile	12-JAN-77	3000	101	1	40
103	Georgescu	Ilie	01-MAY-78	3000	101	1	44
104	Enescu	Gica	11-JUN-66	2000	102	1	40
105	Georgescu	Viorel	02-APR-77	2000	104	2	40

vedere *salariat* 40

<i>cod_salariat</i>	<i>nume</i>	<i>prenume</i>	<i>data_nastere</i>	<i>cod_dept</i>
102	Vasilescu	Vasile	12-JAN-77	1
104	Enescu	Gica	11-JUN-66	1
105	Georgescu	Viorel	02-APR-77	2

Figura 6.1: Un exemplu de vedere.

O vedere poate fi construită din una sau mai multe tabele sau chiar alte vederi și permite ca datele din mai multe tabele să fie reînărandate, reunite logic sau ca noi date să fie calculate pe baza acestora. Din punct de vedere al aplicației, vederile au același comportament ca și tabelele: vederile pot și fi interogate și, cu anumite excepții care vor fi menționate mai târziu, asupra vederilor se pot efectua operații DML (INSERT, DELETE, UPDATE). Vederea este un instrument foarte puternic pentru dezvoltatorul de aplicații. Ea poate fi bazată pe mai multe tabele sau vederi care se pot găsi pe mașini diferențiate sau pot aparține unor utilizatori diferenți, acestea fiind prezentate ca și cum ar fi un singur tabel logic.

Spre deosebire de tabel, vederea nu stochează date și nici nu are alocat vreun spațiu de stocare; vederea doar extrage sau deriva datele din tabelele la care aceasta se referă. Aceste tabele poartă numele de *tabele de bază ale vederii*. Acestea pot fi tabele sau pot fi ele însele vederi. Oracle stocă definiția vederii în dicționarul de date sub forma textului interogării care definește vederea, de aceea o vedere poate fi gândită ca o “interrogare stocată”. Pentru vizualizarea definițiilor vederilor se poate folosi coloana TEXT a vederilor ALL_VIEWS, DBA_VIEW și USER_VIEW din dicționarul de date.

Vederile pot fi interogate exact la fel ca tabelele, folosind comanda SELECT. Când o interogare SQL se referă la o vedere, Oracle combină această interogare cu interogarea care definește vedere.

În general, vederile sunt create pentru următoarele scopuri:

- Asigurarea unui nivel mai mare de securitate a bazei de date prin limitarea accesului la un număr mai restrâns de linii și coloane ale unui tabel.
- Simplificarea interogărilor SQL, permitând vizualizarea unor date care în mod normal necesită interogări SQL destul de complicate. De exemplu, o vedere poate permite utilizatorilor vizualizarea datelor din mai multe tabele fără ca aceștia să fie obligați să folosească un SELECT pe mai multe tabele.
- Prezentarea diferită a datelor față de cea din tabelele de bază. De exemplu, coloana unei vederi poate avea alt nume decât coloana corespunzătoare din tabelul de bază, acest lucru neafectând în nici un fel tabelul de bază.
- Efectuarea unor interogări care nu ar putea fi efectuate fără existența unei vederi. De exemplu, este posibilă definirea unei vederi care realizează un join între o vedere care include clauza GROUP BY și un alt tabel; acest lucru nu poate fi făcut într-o singură interogare.
- Pentru a menține calcule mai complicate. Interogarea care definește vederea poate efectua calcule complicate asupra datelor dintr-un tabel; prin menținerea acestei interogări ca o vedere, calculele pot fi efectuate de fiecare dată când se face referire la vedere.
- Asigurarea transparentă a datelor pentru anumiți utilizatori și aplicații. O vedere poate conține date din mai multe tabele, care pot fi proprietatea mai multor utilizatori.

6.2.1. Crearea vederilor

O vedere este creată folosind comanda SQL CREATE VIEW. De exemplu, pentru vedere de mai sus vom avea:

```
CREATE VIEW salariat_40
AS SELECT cod_salariat, nume, prenume, salariu, cod_dept
FROM salariat
WHERE cod_tara = 40;
```

O sintaxă simplificată a comenzii CREATE VIEW este următoarea:

```
CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW nume_vedere
[(alias [,alias]...)]
AS subinterrogare
[WITH READ ONLY]
[WITH CHECK OPTION [CONSTRAINT nume_cronstrangere]]
```

unde:

- OR REPLACE recreează vedere dacă ea există deja. Această opțiune poate fi folosită pentru a schimba definiția unei vederi existente fără a o distruge în prealabil. Avantajul recreării vederii prin opțiunea REPLACE este că în acest caz se păstrează toate privilegiile acordate asupra acestei vederi. De exemplu, să presupunem că după crearea unei vederi, au fost acordate privilegii asupra vederii pentru anumite roluri sau pentru anumiți utilizatori. Dacă după aceea vedere este distrusă și recreată, atunci toate privilegiile asupra vederii au fost pierdute și trebuie acordate din nou. Dacă vedere este însă recreată folosind opțiunea OR REPLACE, atunci privilegiile acordate sunt păstrate și nu mai este necesară acordarea lor încă o dată.
- FORCE este o opțiune care permite crearea vederii indiferent dacă tabelele de bază și coloanele la care se face referire există sau nu, sau dacă utilizatorul posedă sau nu privilegiile

corespunzătoare în legătură cu tabelele respective. Opțiunea opusă, NOFORCE, creează vederea numai dacă tabelele de bază există și dacă utilizatorul posedă privilegiile corespunzătoare în legătură cu tabelele respective; NOFORCE este opțiunea implicită. Dacă se folosește opțiunea FORCE și un tabel de bază nu există sau una dintre coloane nu este validă, atunci Oracle va crea vederea cu erori de compilare. Dacă mai târziu tabelul în cauză este creat sau coloana este corectată, atunci vederea poate fi folosită, Oracle recompilând-o dinamic înainte de folosire.

- alias specifică numele expresiilor selectate de interogarea vederii. Numărul alias-urilor trebuie să fie același cu numărul de expresii selectate de către interogarea vederii. Un alias trebuie să fie unic în cadrul unei interogări. Dacă sunt omise alias-urile, Oracle va folosi denumirile coloanelor din interogare. Atunci când interogarea vederii conține și expresii, nu doar simple coloane, trebuie folosite alias-uri.
- AS indică interogarea vederii. Aceasta poate fi orice instrucțiune SELECT care nu conține clauzele ORDER BY și FOR UPDATE.
- opțiunea WITH READ ONLY asigură că nici o operație DML (inserare, ștergere, modificare) nu poate fi efectuată asupra vizualizării.
- WITH CHECK OPTION este o constrângere care arată că toate actualizările efectuate prin intermediul vederii vor afecta tabelele de bază numai dacă actualizările respective vor avea ca rezultat numai rânduri care pot fi vizualizate prin intermediul vederii. CONSTRAINT furnizează un nume pentru constrângerea CHECK OPTION. Asupra acestor opțiuni vom reveni puțin mai târziu.

Exemplul următor ilustrează crearea unei vederi care conține codul, numele, prenumele, salariul și sporul salarial pentru toți salariații din departamentul 1 și țara cu codul 40.

```
CREATE OR REPLACE VIEW salariat_1
(cod, nume, prenume, salariu, spor_salariu)
AS SELECT cod_salariat, nume, prenume, salariu,
salariu*0.1
FROM salariat
WHERE cod_dept = 1
AND cod_țara = 40;
```

Dacă datele din tabelul salariat sunt cele din figura 6.1, atunci vederea salariat_1 va conține următoarele date:

salariat_1

cod	nume	prenume	salariu	spor_salariu
102	Vasilescu	Vasile	3000 .	300
104	Enescu	Gica	3000	300

Figura 6.2

6.2.2. Operații DML asupra vederilor

În momentul în care în tabelele de bază sunt adăugate noi date sau sunt actualizate sau șterse cele existente, aceste modificări se reflectă corespunzător în vederile bazate pe aceste tabele. Acest lucru este adevărat și viceversa, cu singura mențiune că există anumite restricții la inserarea, actualizarea sau ștergerea datelor dintr-o vedere. Aceste restricții sunt redate pe scurt în continuare:

- Nu pot fi inserate, șterse sau actualizate datele din vederi care conțin una dintre următoarele:
 - operatorul DISTINCT (pentru eliminarea duplielor);
 - clauzele GROUP BY, HAVING, START WITH, CONNECT BY;
 - pseudo-coloana ROWNUM (această pseudo-coloană conține un număr ce indică ordinea în care Oracle selectează înregistrările dintr-un tabel);
 - funcțiile de grup (COUNT, SUM, MAX, MIN, AVG, STDDEV, VARIANCE, GLB);
 - operatorii de mulțimi (UNION, UNION ALL, INTERSECT, MINUS).
- Nu pot fi inserate sau actualizate valorile coloanelor care rezultă prin calcul, de exemplu coloana spor_salariu de mai sus. De asemenea nu se pot efectua operații DML asupra valorilor coloanelor care au fost calculate folosind funcția DECODE.
- Nu pot fi inserate sau actualizate date care ar încălca constrângerile din tabele de bază. De exemplu, dacă în tabela salariat coloana nume este definită ca NOT NULL, atunci în orice vedere bazată pe acest tabel care nu conține coloana nume (de exemplu salariat_exemplu definită mai jos) nu va fi posibilă inserarea de date deoarece aceasta ar duce la încălcarea constrângerii NOT NULL.

```
CREATE VIEW salariat_exemplu
AS SELECT cod_salarat, prenume, data_nastere
FROM salariat;
```

- În versiunea Oracle 7 nu pot fi inserate, șterse sau actualizate datele din vederi bazate pe mai multe tabele; în versiunea Oracle 8 acest lucru este posibil, însă cu anumite excepții, care vor fi discutate într-o altă secțiune.

În plus față de regulile de mai sus, la crearea unei vederi se poate utiliza clauza WITH CHECK OPTION care impune că singurele date care pot fi inserate sau actualizate prin intermediul vederii să fie numai acelea care pot fi vizualizate de aceasta.

Pentru a clarifica acest aspect, să considerăm vedere salariat_2000 definită mai jos și posibilitățile existente de a insera rânduri în această vedere.

```
CREATE VIEW salariat_2000
AS SELECT cod_salarat, nume, prenume, data_nastere, salariu
FROM salariat
WHERE salariu > 2000;
```

Prima comandă SQL de mai jos va duce la inserarea unui rând în tabela de bază salariat, care poate fi vizualizat și prin intermediul vederii salariat_2000, deoarece valoarea corespunzătoare coloanei salariu este mai mare decât 2000. Pe de altă parte, a doua comandă SQL va duce la inserarea unui rând în tabelul de bază care nu poate fi însă vizualizat prin intermediul vederii.

```
INSERT INTO salariat_2000
(cod_salarat, nume, prenume, data_nastere, salariu)
VALUES (106, 'Ionescu', 'Vasile', '11-JUL-60', 3000);
```

```
INSERT INTO salariat_2000
(cod_salarat, nume, prenume, data_nastere, salariu)
VALUES (107, 'Popescu', 'Viorel', '22-JAN-69', 1000);
```

De exemplu, dacă înaintea execuției acestor comenzi în tabelul **salariat** există datele din figura 6.1, atunci după executarea acestor comenzi, datele din tabelul **salariat** și vederea **salariat_2000** vor fi următoarele:

tabel de bază salariat

cod_salarat	nume	prenume	data_nastere	salariu	manager	cod_dept	cod_tara
101	Popescu	Ion	11-DEC-77	5000		1	44
102	Vasilescu	Vasile	12-JAN-77	3000	101	1	40
103	Georgescu	Ilie	01-MAY-78	3000	101	1	44
104	Enescu	Gică	11-JUN-66	2000	102	1	40
105	Georgescu	Viorel	02-APR-77	2000	104	2	40
106	Ionescu	Vasile	11-JUL-60	3000			
107	Popescu	Viorel	22-JAN-69	1000			

vedere salariat_2000

cod_salarat	nume	prenume	data_nastere	salariu
101	Popescu	Ion	11-DEC-77	5000
102	Vasilescu	Vasile	12-JAN-77	3000
103	Georgescu	Ilie	01-MAY-78	3000
106	Ionescu	Vasile	11-JUL-60	3000

Figura 6.3

Să presupunem acum că aceeași vedere, **salariat_2000**, este creată folosind clauza WITH CHECK OPTION:

```
CREATE VIEW salariat_2000
AS SELECT cod_salarat, nume, prenume, data_nastere, salariu
FROM salariat
WHERE salariu > 2000
WITH CHECK OPTION;
```

Și în acest caz, inserarea în vedere a unui rând pentru care valoarea coloanei salariu este mai mare decât 2000 se face fără probleme. Pe de altă parte însă, orice încercare de a insera în vedere rânduri pentru care salariul este mai mic sau egal cu 2000 va produce o eroare indicând încălcarea constrângerii WITH CHECK OPTION, comanda nemodificând tabelul de bază. De exemplu, executarea celor două comenzi de INSERT de mai sus va avea în acest caz ca rezultat următoarele date din **salariat** și **salariat_2000**.

tabel de bază salariat

cod_salarat	nume	prenume	data_nastere	salariu	manager	cod_dept	cod_tara
101	Popescu	Ion	11-DEC-77	5000		1	44
102	Vasilescu	Vasile	12-JAN-77	3000	101	1	40
103	Georgescu	Ilie	01-MAY-78	3000	101	1	44
104	Enescu	Gică	11-JUN-66	2000	102	1	40
105	Georgescu	Viorel	02-APR-77	2000	104	2	40
106	Ionescu	Vasile	11-JUL-60	3000			

vedere salariat 2000

cod_salarat	nume	prenume	data_nastere	salariu
101	Popescu	Ion	11-DEC-77	5000
102	Vasilescu	Vasile	12-JAN-77	3000
103	Georgescu	Ilie	01-MAY-78	3000
106	Ionescu	Vasile	11-JUL-60	3000

Figura 6.4.

În cazul folosirii constrângerii CHECK OPTION, acesteia î se poate atribui un nume folosind opțiunea CONSTRAINT din cadrul comenzi CREATE VIEW. De exemplu:

```
CREATE VIEW salariat_2000
AS SELECT cod_salarat, nume, prenume, data_nastere, salariu
FROM salariat
WHERE salariu > 2000
WITH CHECK OPTION CONSTRAINT salariu_2000;
```

Dacă opțiunea CONSTRAINT este omisă, Oracle atribuie în mod automat constrângerii CHECK OPTION un nume de forma "SYS_Cn" unde n este un întreg care face ca numele constrângerii să fie unic în baza de date.

6.2.3. Operații DML asupra vederilor bazate pe mai multe tabele (Join-Views)

Așa cum am menționat înainte, în versiunea Oracle8 este posibilă inserarea, actualizarea sau stergerea datelor dintr-o vedere bazată pe mai multe tabele, cu anumite restricții însă. Alături de restricțile generale, aplicabile tuturor vederilor, prezентate în secțiunea precedentă, există și restricții specifice numai vederilor bazate pe mai multe tabele. Acestea sunt redate de următoarele reguli:

- *Regula generală:* Orice operație de INSERT, UPDATE sau DELETE pe o vedere bazată pe mai multe vederi poate modifica datele din doar unul dintre tabelele de bază.

Înainte de a enumera regulile specifice pentru fiecare dintre operațiile INSERT, UPDATE sau DELETE este necesară definirea conceptului de *label protejat prin cheie (key-preserved table)*. Dată fiind o vedere bazată pe mai multe tabele, un tabel de bază al vederii este protejat prin cheie dacă orice cheie selectată a tabelului este de asemenea și cheie a vederii. Deci, un tabel protejat prin cheie este un tabel ale căruia chei se păstrează și la nivel de vedere. Trebuie reținut că, pentru a fi protejat prin cheie, nu este necesar ca un tabel să aibă toate cheile selectate în vedere. Este suficient ca, atunci când cheia tabelului este selectată, aceasta să fie și cheie a vederii. Proprietatea unui tabel de a fi protejat prin cheie nu este o proprietate a datelor din tabel, ci o proprietate a schemei. În exemplul de mai jos, dacă pentru fiecare combinație (cod_tara, cod_dept) ar exista un singur salariat, atunci combinația (cod_tara, cod_dept) din tabelul departament ar fi unică pentru datele din vedere rezultată, dar tabelul departament tot nu ar fi protejat prin cheie.

Pentru a ilustra această noțiune cât și regulile următoare, să considerăm o definiție simplificată a tabelelor departament și salariat în care păstrăm doar constrângerile de cheie primară și integritate referențială.

```
CREATE TABLE departament(
cod_dept      NUMBER(10),
cod_tara      NUMBER(10),
nume_dept     VARCHAR2(10),
PRIMARY KEY(cod_dept, cod_tara));

CREATE TABLE salariat(
cod_salariat   NUMBER(10) PRIMARY KEY,
nume           VARCHAR2(10),
prenume        VARCHAR2(10),
data_nastere   DATE,
manager        NUMBER(10)
    REFERENCES salariat(cod_salariat),
salariu        NUMBER(10),
cod_dept       NUMBER(10),
cod_tara       NUMBER(10),
FOREIGN KEY(cod_dept, cod_tara)
    REFERENCES departament(cod_dept, cod_tara));
```

Să mai considerăm și următoarea vedere bazată pe aceste două tabele:

```
CREATE VIEW sal_dept
(cod_salariat, nume, prenume, salariu, cod_dept,
cod_tara, nume_dept)
AS SELECT s.cod_salariat, s.nume, s.prenume, s.salariu,
s.cod_dept, s.cod_tara, d.nume_dept
FROM salariat s, departament d
WHERE s.cod_dept = d.cod_dept
AND s.cod_tara = d.cod_tara;
```

În exemplul de mai sus, tabelul salariat este protejat prin cheie. Regulile specifice pentru fiecare dintre operațiile DML (INSERT, UPDATE sau DELETE) sunt redate în continuare:

Reguli de actualizare (UPDATE):

- Toate coloanele care pot fi actualizate printr-o vedere trebuie să corespundă coloanelor dintr-un tabel protejat prin cheie. Dacă o coloană provine dintr-un tabel neprotejat prin cheie, atunci Oracle nu va putea identifica în mod unic înregistrarea care va trebui actualizată. De exemplu, comanda SQL de mai jos se va executa cu succes.

```
UPDATE sal_dept
SET salariu = salariu + 100
WHERE cod_tara = 40;
```

Pe de altă parte, comanda următoare va eșua:

```
UPDATE sal_dept
SET nume_dept = 'IT'
WHERE cod_dept = 1
AND cod_tara = 40;
```

Comanda de mai sus va eșua pentru că ea încearcă să actualizeze o coloană din tabelul departament, tabel care nu este protejat prin cheie.

- Dacă vederea este definită folosind clauza WITH CHECK OPTION, atunci toate coloanele de joncțiune și toate coloanele tabelelor repetitive nu pot fi modificate. De exemplu, dacă vederea sal_dept ar fi fost definită folosind clauza WITH CHECK OPTION, atunci comanda următoare va eșua deoarece încearcă modificarea unei coloane de joncțiune.

```
UPDATE sal_dept
SET cod_dept = 2
WHERE cod_salariat = 101
AND cod_tara = 40;
```

Reguli de inserare (INSERT):

- O comandă INSERT nu poate să se refere în mod explicit sau implicit la coloane dintr-un tabel care nu este protejat prin cheie. De exemplu, dacă în tabelul departament există o linie cu cod_dept = 3 și cod_tara = 40, atunci următoarea comandă SQL va fi executată cu succes:

```
INSERT INTO sal_dept
(cod_salariat, nume, cod_dept, cod_tara)
VALUES(110, 'Marinescu', 3, 40);
```

În caz contrar, comanda va eșua, fiind încălcată constrângerea de integritate referențială. Pe de altă parte, comanda următoare va eșua deoarece ea încearcă inserarea de date în mai multe tabele.

```
INSERT INTO sal_dept
(cod_salariat, nume, nume_dept)
VALUES(111, 'Georgescu', 'IT');
```

- Dacă o vedere este definită folosind clauza WITH CHECK OPTION, atunci nu se poate executa comenzi INSERT în acea vedere.

Reguli de ștergere (DELETE):

- Rândurile dintr-o vedere pot fi șterse numai dacă în joncțiune există un tabel protejat prin cheie și numai unul. Dacă ar exista mai multe tabele, Oracle nu ar ști din care tabel

să șteargă rândul. De exemplu, comanda SQL de mai jos se va executa cu succes deoarece ea poate fi tradusă într-o operație de ștergere pe tabelul `salariat`:

```
DELETE FROM sal_dept
WHERE nume = 'Popescu';
```

Pe de altă parte, dacă se încearcă executarea unei comenzi `DELETE` pe vederea de mai jos, ea va eşua deoarece ambele tabele de bază, `s1` și `s2`, sunt protejate prin cheie:

```
CREATE VIEW emp_emp AS
SELECT s1.nume, s2.prenume
FROM salariat s1, salariat s2
WHERE s1.cod_salariat = s2.cod_salariat;
```

- Dacă vederea este definită folosind clauza `WITH CHECK OPTION`, atunci nu pot fi șterse rânduri din vedere. De exemplu, nu se poate executa o instrucțiune `DELETE` pe vederea de mai jos deoarece ea este definită ca auto-joncțiune a unui tabel protejat prin cheie.

```
CREATE VIEW emp_manag AS
SELECT s1.nume, s2.nume nume_manager
FROM salariat s1, salariat s2
WHERE s1.manager = s2.cod_salariat
WITH CHECK OPTION;
```

Vederile `ALL_UPDATABLE_COLUMNS`, `DBA_UPDATABLE_COLUMNS` și `USER_UPDATABLE_COLUMNS` ale dicționarului de date conțin informații care arată care dintre coloanele vederilor existente pot fi actualizate. Vederile care nu pot fi actualizate direct pot fi actualizate folosind trigger-e `INSTEAD OF`, vezi secțiunea 9.16.

6.2.4 Recompilarea vederilor

Recompilarea unei vederi permite detectarea eventualelor erori referitoare la vederea respectivă înaintea executării vederii. După fiecare modificare a tabelelor de bază este recomandabil ca vederea să se recompileze. Acest lucru se poate face folosind comanda SQL `ALTER VIEW ... COMPILE`:

```
ALTER VIEW salariat_2000 COMPILE;
```

6.2.5. Distrugerea vederilor

Pentru distrugerea unei vederi se folosește comanda `DROP VIEW`:

```
DROP VIEW salariat_2000;
```

Utilizarea unei vederi este cădeodată o sabie cu două tăișuri. Deși vederile constituie un instrument extrem de convenabil pentru dezvoltatorul de aplicații, ele pot avea un impact

negativ asupra performanței acestora. Dacă pentru vederi simple (de exemplu simple copii ale unui tabel) impactul asupra performanței nu este sesizabil, pentru vederi complexe, care cuprind interogări pe mai multe tabele, acest impact poate fi foarte sever. De aceea folosirea vederilor trebuie planificată cu grijă, pentru a se vedea dacă avantajul creat de simplitatea în manipulare a acestora compensează impactul negativ asupra performanței.

6.3. Indecși

Un *index* este o structură opțională a bazei de date care permite accesarea directă a unui rând dintr-un tabel. Indecșii pot fi creați pentru una sau mai multe coloane ale unui tabel, în acest ultim caz folosindu-se denumirea de *indecsi compuși* sau *indecsi concatenati*. Un index este utilizat de către baza de date pentru a găsi rapid valori pentru coloana sau coloanele pentru care a fost creat indexul, în acest mod furnizând o cale de acces directă la linile asociate acestora fără a mai fi necesară investigarea fiecărui rând din tabel. Practic, în momentul în care se dorește căutarea anumitor înregistrări ale căror valori îndeplinesc un anumit criteriu, în loc să se parcurgă tabelul, se parurge indexul, acesta din urmă furnizând localizarea exactă a înregistrărilor ce îndeplinesc criteriul de căutare, prin indicarea ROWID. Așa cum s-a arătat în secțiunea 4.5, identificatorul de rând ROWID reprezintă în primul rând cel mai rapid mod de a localiza o anumită înregistrare. Prin urmare pentru a găsi o anumită înregistrare în modul cel mai rapid, se determină ROWID-ul acesteia în loc să se parcurgă secvențial tabelul respectiv. Fără indecsi, operații precum determinarea unicății sau ordonarea valorilor unei coloane ar putea consuma multe resurse și timp.

Prezența indecsilor este transparentă pentru utilizator și aplicație, ei neputând fi referiți în mod direct prin interogări. În plus, sintaxa unei comenzi SQL nu este în nici un fel influențată de existența indecsilor, iar rezultatele oricărei interogări vor fi aceleași indiferent dacă există indecsi sau nu. Pe de altă parte însă, utilizarea corectă a acestora poate influența în cel mai mare grad eficiența unei interogări: diferența va consta în rapiditatea cu care se execută comanda și nu în rezultatele acesteia. Foarte multe sisteme ce raportează probleme de performanță suferă din cauza lipsei unui index sau din cauza absenței unui index optim. Informația conținută în indecsi este redundantă, ea fiind derivată din informația care există în tabele.

Indecșii sunt independenți din punct de vedere fizic și logic de datele din tabelul de bază. Un index poate fi creat și distrus fără ca datele din tabelul de bază sau ceilalți indecsi să aibă de suferit. Singurul lucru pe care indexul îl va modifica va fi durata accesului la datele tabelului, acesta devenind mai lent în absența indexului. Evident, odată cu ștergerea unui tabel sunt ștersi și indecsii asociați acestuia.

Indecșii pot să fie *unici* sau *ne-unici*. Indecșii unici garantează faptul că nu va exista nici o pereche de linii care să aibă valori identice pentru coloana sau grupul de coloane pentru care a fost definit indexul. Valorile Null nu sunt considerate pentru unicitate. Un rând cu valoarea Null în coloana indexată nu va fi înregistrat în index, deci un index unic nu va împiedica stocarea mai multor rânduri cu o valoare Null în coloana indexată. Un index ne-unic nu impune nici o restricție în legătură cu valorile din coloanele care îl definesc.

O dată definit, un index este actualizat de către baza de date ori de căte ori au loc modificări ale datelor tabelului. Aceasta înseamnă, că ori de căte ori au loc inserări, ștergeri sau modificări ale

datelor unui tabel, toți indecsii aceluia tabel trebuie actualizați în mod automat, acest lucru având ca efect încetinirea acestor operații asupra datelor tabelului. Cu alte cuvinte, existența indecsilor sporește viteza accesului la datele tabelului pe de o parte, dar în același timp încetinește operațiile de modificare ale acestora. De aceea, nu trebuie creat un index pentru fiecare coloană a unui tabel (deși acest lucru este posibil), ci numai pentru anumite coloane cheie ale acestuia. De exemplu, se recomandă indexarea coloanelor care conțin în majoritate valori unice sau un domeniu larg de valori sau coloane după care se fac dese căutări sau ordonări. Deoarece prezența indecsilor poate avea un impact semnificativ asupra eficienței aplicației în continuare prezentăm câteva sugestii privind folosirea acestora:

Ce tabele trebuie indexate:

- Indexați tabelele pentru care majoritatea interogărilor selectează doar un număr redus de rânduri (sub 5%). Interogările care selectează un număr mare de rânduri nu folosesc în mod eficient indecsii.
- Nu indexați tabele ce conțin puține înregistrări deoarece în acest caz accesul secvențial va fi mai rapid.
- Indexați tabelele care sunt interogate folosind clauze SQL simple. Clauzele SQL mai complexe nu folosesc cu aceeași eficiență indecsii.
- Nu indexați tabelele care sunt actualizate frecvent. Inserările, modificările și ștergerile sunt îngreunate de existența indecsilor. Decizia de a indexa un tabel trebuie luată pe baza raportului dintre numărul de interogări și cel de actualizări efectuat asupra acestuia.

Ce coloane trebuie indexate:

- Folosiți coloanele cele mai frecvent folosite în clauza WHERE a interogărilor.
- Nu indexați coloane care nu au multe valori unice¹.
- Coloanele care au valori unice sunt candidate foarte bune pentru indexare. De altfel, Oracle creează în mod automat indecsi unici pentru coloanele definite ca PRIMARY KEY sau UNIQUE. În plus, din punct de vedere logic, este preferabil ca indecsii unici să nu fie definiți în mod explicit, ci prin intermediul constrângerilor PRIMARY KEY și UNIQUE - deoarece unicitatea este un concept logic și ar trebui să fie asociat cu definiția tabelului.
- Coloanele care sunt folosite pentru a face legătura dintre tabele sunt în general candidate pentru indexare. În general, se recomandă indexarea cheilor străine.
- În anumite situații, folosirea indecsilor compuși poate fi mai eficientă decât a celor individuali. De exemplu, crearea indecsilor compuși este recomandabilă când două coloane nu sunt unice fiecare în parte dar combinația lor este unică sau are în majoritate valori unice. De asemenea, se recomandă crearea indecsilor compuși atunci când interogările uzuale ale tabelului conțin în clauza WHERE mai multe coloane individuale legate prin AND.

¹ Totuși, începând cu versiunea Oracle8 se pot indexa coloane care nu au multe valori distincte cu ajutorul indexului de tip bitmap asupra căruia vom reveni în cuprinsul acestui capitol.

Indecșii consumă spațiu în baza de date la fel ca și tabelele și există într-un spațiu tabel la fel ca și acestea. Spațul necesar indecșilor pentru tabele mari este de obicei semnificativ, așa că trebuie planificat din momentul în care se proiectează baza de date.

6.3.1. Crearea, modificarea și distrugerea indecșilor

În SQL un index se creează folosind comanda CREATE INDEX. O sintaxă simplificată a acestei comenzi este prezentată în continuare:

```
CREATE [UNIQUE] INDEX nume_index
ON tabel (coloana [, coloana] ...)
[PCTFREE intreg] [PCTUSED intreg]
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]
```

unde:

- Dacă este specificată opțiunea UNIQUE, indexul creat este unic, altfel nu.
- Valorile parametrilor PCTFREE și PCTUSED determină gradul de utilizare a blocurilor din extinderile segmentului de index, vezi secțiunea 4.4.
- TABLESPACE specifică spațiul tabel în care va fi stocat indexul. Dacă acesta nu este menționat explicit, se va folosi spațiul tabel implicit (default) al utilizatorului care este proprietarul schemei din care face parte indexul. În general, se recomandă stocarea indexului într-un spațiu tabel diferit de cel în care este stocat tabelul său; în acest caz Oracle poate accesa tabelul și indexul în paralel, obținându-se astfel o creștere a performanțelor interogărilor.
- Clauza STORAGE este folosită pentru setarea parametrilor de stocare (INITIAL, NEXT, PCTINCREASE, MINEXTENTS, MAXEXTENTS) prin intermediul căror se specifică mărimea și modul de alocare a extinderilor segmentului de index, vezi secțiunea 4.1.1.

Urnațoarele comenzi SQL creează un index compus sal_dept_ind pentru coloanele cod_dept și cod_tara din tabelul salariat și respectiv un index unic pentru coloana nume_dept a tabelului departament.

```
CREATE INDEX sal_dept_ind ON salariat(cod_dept, cod_tara);
CREATE UNIQUE INDEX nume_dept_ind ON
departament(nume_dept);
```

Crearea indecșilor unui tabel se recomandă a se face după ce tabelul a fost populat cu date deoarece existența indecșilor incetează în mod evident inserarea datelor.

La definiția constrângerilor PRIMARY KEY sau UNIQUE sau la activarea acestora, Oracle creează în mod automat indeci unici pentru coloanele sau grupurile de coloane respective. În acest caz numele indexului coincide cu numele constrângerii.

La fel ca și în cazul tabelelor, parametrii de stocare a indexelor pot fi modificați ulterior. Pentru aceasta se folosește comanda ALTER INDEX cu următoarea sintaxă:

```
ALTER INDEX nume_index
  STORAGE parametrii_de_stocare
```

De exemplu:

```
ALTER INDEX sal_dept_ind
  STORAGE (NEXT 400K MAXEXTENTS 100);
```

De asemenea, Oracle8 permite alocarea și dealocarea manuală a spațiului utilizat de un index.

Alocarea manuală a spațiului pentru un index înseamnă adăugarea manuală a unei noi extinderi. De exemplu, alocarea manuală se poate face înaintea unei perioade în care se va înregistra o activitate intensă de inserări în tabela pe care este bazat indexul. În acest caz, alocarea manuală previne extinderea dinamică a indexului (alocarea dinamică a unei noi extinderi), împiedicând astfel creșterea timpului de execuție. Dealocarea spațiului asociat unui index reprezintă eliberarea spațiului nefolosit de acesta.

Pentru a aloca sau dealoca spațiu utilizat de un index se folosește comanda ALTER INDEX cu următoarele sintaxe:

```
ALTER INDEX nume_index
  ALLOCATE EXTENT([SIZE intreg [K|M]]
                  [DATAFILE specificație_fisier_de_date])
```

respectiv

```
ALTER INDEX nume_index
  DEALLOCATE UNUSED [KEEP intreg [K|M]]
```

Semnificația parametrilor din aceste comenzi este aceeași ca și în cazul tabelelor, vezi secțiunea 6.1.5.

Un index poate fi distrus folosind comanda SQL DROP INDEX cu următoarea sintaxă:

```
DROP INDEX nume_index;
```

De exemplu:

```
DROP INDEX sal_dept_ind;
```

Un index nu poate fi șters dacă el a fost creat în mod automat, ca parte a definirii sau a activării unei restricții PRIMARY KEY sau UNIQUE. În acest caz, indexul este șters automat la ștergerea sau dezactivarea constrângerii.

Ștergerea unui index se face de obicei dacă indexul nu mai este necesar, dacă se dorește reconstruirea sa (despre care vom discuta în continuare) sau înainte de încărcarea masivă a datelor într-un tabel; ștergerea unui index înainte de încărcarea masivă a datelor și recrearea lui după terminarea încărcării va duce la îmbunătățirea performanței încărcării precum și la utilizarea spațiului alocat indexului în mod mai eficient.

Pentru a reconstrui un index se pot folosi două metode. Prima este de a șterge indexul folosind comanda `DROP INDEX` și de a îl recrea folosind comanda `CREATE INDEX`. A doua este folosirea `ALTER INDEX` cu opțiunea `REBUILD` folosind sintaxa:

```
ALTER INDEX nume_index REBUILD
[ PCTFREE intreg ]-[ PCTUSED intreg ]
[ TABLESPACE spațiu_tabel ]
[ STORAGE parametri_de_stocare ]
[ REVERSE | NOREVERSE ]
```

Reconstruirea unui index se face în următoarele situații:

- indexul existent trebuie mutat într-un spațiu tabel diferit (de exemplu, dacă indexul a fost creat în mod automat de o constrângere de integritate, caz în care el se găsește în același spațiu tabel cu tabelul).
- un index normal trebuie convertit într-un index cu cheie inversă (despre tipurile de indecsă vom vorbi în continuare). Pentru a converti un index obișnuit într-un index cu cheie inversă se folosește comanda `ALTER INDEX ... REBUILD` cu opțiunea `REVERSE` (opțiunea contrară este `NOREVERSE`, care este și opțiunea implicită)
- pentru a recupera spațiul de stocare sau pentru a schimba atributele fizice de stocare.

În cazul reconstruirii indexului folosind comanda `ALTER INDEX ... REBUILD`, în timpul creării noului index este păstrat și indexul original, astfel că indexul poate fi folosit de interogări și în timpul acestei operații. Principalele avantaje ale fiecărei dintre cele două metode sunt rezumate în tabelul de mai jos:

Șterge și recrează	Folosește opțiunea REBUILD
Poate redenumi indexul.	Nu poate redenumi indexul.
Poate schimba între UNIQUE și non-UNIQUE.	Nu poate schimba între UNIQUE și non-UNIQUE.
Poate schimba între tipul de index bazat de arbore B* și indexul de tip bitmap (vezi paragrafele următoare).	Nu poate schimba între tipul de index bazat de arbore B* și indexul de tip bitmap.
Are nevoie de spațiu doar pentru o copie a indexului.	Are nevoie de spațiu suplimentar pentru a duplica indexul în mod temporar.
Necesită sortare.	Nu necesită sortare, folosindu-se indexul inițial.
Indexul este indisponibil în perioada dintre ștergere și recreare.	Indexul este disponibil pentru interogări.
Nu se poate folosi această metodă dacă indexul a fost creat de o constrângere PRIMARY KEY sau UNIQUE. În acest caz ștergerea indexului se poate face prin ștergerea sau dezactivarea constrângerii respective.	Se poate folosi această metodă dacă indexul a fost creat de o constrângere PRIMARY KEY sau UNIQUE.

6.3.2. Tipuri de indecsă

Index de tip arbore B*

Cel mai întâlnit tip de index folosit de Oracle este *indexul de tip arbore B** (*B*-tree index* sau *balanced tree index*). Aceasta este tipul de index creat la executarea unei comenzi standard `CREATE INDEX` și tipul de index la care ne-am referit în secțiunea de mai sus.

Un arbore B*, este un arbore în care pentru găsirea oricărei valori din arbore sunt necesari același număr de pași, indiferent de valoarea căutată. Figura 6.5 ilustrează structura unui index de tip arbore B*.

Algoritmul de căutare într-un astfel de arbore compară valoarea cerută cu valorile din nivelul superior de blocuri; în funcție de această comparație, valoarea căutată este apoi comparată cu unul din blocurile inferioare și comparația continuă până când se ajunge la ultimul nivel de blocuri, numite *blocuri frunză*. Blocurile frunză ale indexului conțin toate valorile datelor indexate și valoarea ROWID a rândului asociat pentru fiecare dintre aceste valori. Un arbore B* este întotdeauna *balansat*, adică distanța de la vârf la oricare nod frunză este aceeași pentru toate nodurile. Cu cât este mai mare această distanță, cu atât este mai mare numărul de blocuri ce trebuie examinate pentru a ajunge la un bloc frunză și, prin urmare cu atât mai lent este indexul. Pentru un index unic, există un singur ROWID pentru fiecare valoare. Când un bloc frunză se umple, atunci este creat un bloc nou. Unele informații din blocul plin sunt mutate în noul bloc, acesta din urmă devenind bloc frunză. Deoarece un arbore B* este un arbore balansat, activitățile într-un index nu se termină aici, ducând uneori până la schimbarea informațiilor din nodul rădăcină al arborelui sau chiar până la înlocuirea nodului rădăcină cu un altul. Teoria indecșilor de tip arbore B* este dincolo de scopul acestei cărți, pentru mai multe informații se poate consulta [3] sau literatura de specialitate care se ocupă cu structuri de date.

Pentru indecși neunici, dacă există mai multe ROWID pentru aceeași valoare a datelor indexate, atunci și acestea sunt sortate, deci indecși neunici sunt sortați întâi după valoarea datelor și apoi după ROWID. Valorile nule nu sunt indexate, cu excepția indecșilor de cluster, care vor fi prezentate în secțiunea 6.4.

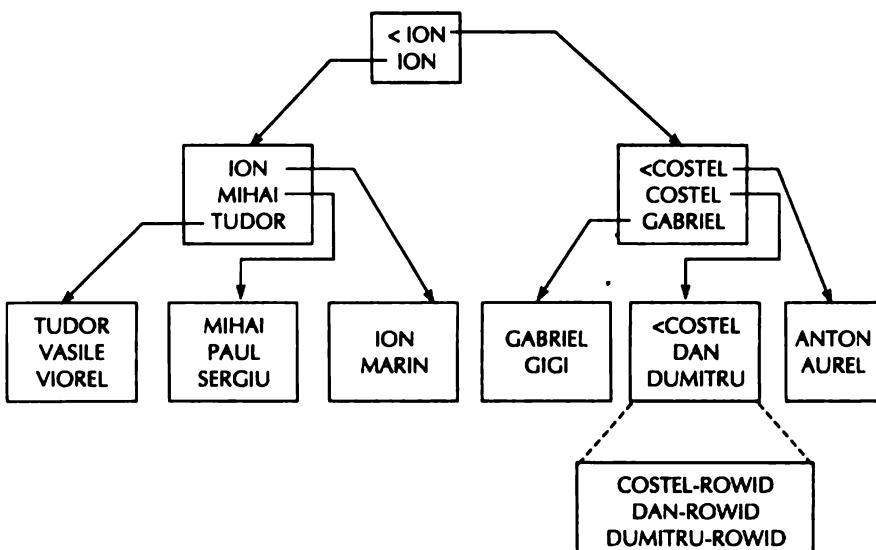


Figura 6.5 : structura internă a unui index de tip arbore B*.

Index partitonal

O nouitate adusă de versiunea Oracle8 este posibilitatea de partitonalare a unui index. Așa cum un tabel poate fi partitonalat, la rândul lui un index de tip arbore B* poate fi și el partitonalat. În deciții partitonați sunt folosiți în cazul tabelelor mari pentru a stoca valorile coloanei indexate în mai multe segmente. Practic, pentru fiecare parte a tabelului poate fi creată o parte a unui index. Prin urmare, partitonalarea permite stocarea unui index în mai multe spații tabel. Unul din avantajele partitonalării este descreșterea domeniului de valori în care indexul cauță o anumită valoare.

Pentru a crea un index partitonal se folosește comanda CREATE INDEX cu clauza PARTITION. Un index partitonal, la fel ca și un tabel partitonalat (vezi secțiunea 6.1.2) permite definirea unor spații tabel cu parametrii de stocare diferiți pentru fiecare parte. Există două moduri de a defini un index partitonal: local și global.

- *local*: partiiile indexului sunt similare cu partiiile tabelului; în acest caz indexul este partitonalat în funcție de aceeași coloană, pe același interval de valori ca și tabelul la care se referă; partiiile trebuie să enunțe în aceeași ordine ca și partiiile tabelului la care se referă. Aceasta este metoda cel mai des folosită deoarece partitonalarea este dirijată de tabelul de bază. Sintaxa pentru crearea unui index partitonal local este următoarea:

```
CREATE INDEX nume_index
ON tabel (coloana[, coloana] ...)
LOCAL
(PARTITION nume_partiție
  [PCTFREE intreg] [PCTUSED intreg]
  [TABLESPACE spațiu_tabel]
  [STORAGE parametrii_de_stocare]
  [, PARTITION nume_partiție
  [PCTFREE intreg] [PCTUSED intreg]
  [TABLESPACE spațiu_tabel]
  [STORAGE parametrii_de_stocare]]...)
```

Urmatorul exemplu creează un index partitonal local pe baza tabelului partitonalat salariat_part din secțiunea 6.1.2.

```
CREATE INDEX nume_sal_part_ind
ON salariat_part(nume)
LOCAL
(PARTITION salariu_mic
  TABLESPACE ts_ind_alfa
  STORAGE (INITIAL 10K NEXT 10K),
  PARTITION salariu_mediul
  TABLESPACE ts_ind_beta
  STORAGE (INITIAL 20K NEXT 20K),
  PARTITION salariu_mare
  TABLESPACE ts_ind_alfa
  STORAGE (INITIAL 10K NEXT 10K));
```

- *global*: partiiile indexului sunt definite de utilizator și nu sunt similare cu partiiile tabelului la care se referă indexul. Sintaxa pentru crearea unui index partitonal global este următoarea:

```
CREATE INDEX nume_index
ON tabel (coloana[, coloana] ...)
```

```

GLOBAL
PARTITION BY RANGE (listă_coloane)
(PARTITION nume_partiție
VALUES [LESS|GREATER] THAN (listă_valori)
[PCTFREE întreg] [PCTUSED întreg]
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]
[,PARTITION nume_partiție
VALUES [LESS|GREATER] THAN (listă_valori)
[PCTFREE întreg] [PCTUSED întreg]
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]]...)

```

Următoarea comandă creează indexul global nume_sal_ind pentru tabela salariat, index ce va avea două partiții:

```

CREATE INDEX nume_sal_ind
ON salariat(nume)
GLOBAL
PARTITION      BY RANGE (nume)
(PARTITION VALUES LESS THAN ('N')
    TABLESPACE ts_alfa_ind,
PARTITION VALUES LESS THAN (MAXVALUE)
    TABLESPACE ts_beta_ind);

```

Index de cluster

Un *index de cluster* (*index de grup*) este un index bazat pe coloanele comune ale unui cluster. Nu se pot efectua nici un fel de comenzi DML asupra unui cluster până când nu a fost creat un index de cluster. Clusterele și indecșii de cluster sunt prezenți pe larg în secțiunea 6.4. Gruparea tabelelor în cluster nu afectează crearea de indecși suplimentari pentru tabele individuale; aceștia pot să creați sau distrui ca de obicei.

Index cu cheie inversă

Un index cu cheie inversă (*reverse-key index*) reprezintă o nouă metodă (adusă de versiunea Oracle8) de a îmbunătăți anumite tipuri de căutări. Așa cum s-a discutat până acum, pentru sporirea eficienței căutărilor unor valori în baza de date se folosesc de obicei arborii B*. Există totuși unele cazuri în care aceștia îngreunează accesul la date. De exemplu, presupunem că avem o coloană indexată ce conține prenumele a mii de persoane. Să presupunem de asemenea că în această coloană există mii de prenume ce încep cu litera 'S'. În momentul în care se dorește inserarea mai multor înregistrări ce încep cu litera 'S' pot apărea ștrangulări ale operațiilor de citire scriere deoarece modificările în index-ul asociat coloanei vor apărea în același nod al arborelui. Indecșii cu cheie inversă sunt folosiți tocmai în această situație deoarece ei stochează datele în mod invers. Prin urmare, prenumele 'SANDU' va fi stocat tot într-un arbore B* ca 'UDNAS'. Acest tip de index este folosit numai în cazul căutării în arbore a unor valori exacte.

Un index cu cheie inversă se creează folosind comanda CREATE INDEX cu opțiunea REVERSE, de exemplu:

```
CREATE INDEX sal_prenume_ind ON salariat(prenume) REVERSE;
```

Un index obișnuit se poate transforma în index cu cheie inversă folosind comanda ALTER INDEX ... REBUILD cu opțiunea REVERSE. De exemplu:

```
ALTER INDEX sal_prenume_ind REBUILD REVERSE;
```

Index de tip bitmap

Un alt tip de index, introdus în Oracle8, este indexul de tip *bitmap*. Într-un astfel de index, în loc de a se stoca valorile propriu-zise ale coloanei indexate, indexul stochează un bitmap format pe baza acestor valori. Cu alte cuvinte, indexul ține un bitmap pentru fiecare rând, bitmap care conține un bit pentru fiecare rând din tabel. Bitul este 1 dacă valoarea respectivă este conținută în acel rând și 0 dacă nu este. De exemplu, să presupunem că avem un index de tip bitmap creat pe baza coloanei culoare dintr-un tabel masina, vezi figura 6.6.

Tabel masina

nr_masina	marca	culoare
1	Ford Mondeo	alb
2	Dacia Nova	negră
3	Daewoo Cielo	alb
4	Daewoo Tico	verde
5	Ford Mondeo	roșu
6	Ford Mondeo	albastru
7	Dacia Nova	alb
8	Dacia Nova	negră
9	Daewoo Cielo	verde
10	Ford Mondeo	verde
11	Dacia Nova	verde
12	Daewoo Tico	verde
13	Ford Mondeo	albastru
14	Dacia Nova	verde
15	Daewoo Tico	albastru
16	Dacia Nova	verde
17	Ford Mondeo	alb
18	Daewoo Tico	negră
19	Dacia Nova	verde
20	Ford Mondeo	verde

Index de tip bitmap pentru coloana culoare

Culoare = alb	Culoare = negru	Culoare = verde	Culoare = albastru	Culoare = roșu
1	0	0	0	0
0	1	0	0	0
1	0	0	0	0
0	0	1	0	0
0	0	0	0	1
0	0	0	1	0
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	0	1	0
0	0	1	0	0
0	0	0	1	0
0	0	1	0	0
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	1	0	0

Figura 6.6

În acest exemplu, coloana **culoare** poate avea numai cinci valori: alb, negru, roșu, verde, albastru, iar în tabel sunt 20 de rânduri. Bitmap-ul corespunzător fiecărei culori va avea 20 de biți, fiecare dintre aceștia având valoarea 1 când mașina va avea culoarea respectivă și 0 în caz contrar. Deci bitmap-ul corespunzător culorii alb va avea 1 pe pozițiile 1, 3, 7, 17, cel corespunzător culorii negru va avea 1 pe pozițiile 2, 8, 18, etc. De exemplu, pentru aflarea numărului de mașini albe se va executa interogarea:

```
SELECT COUNT(*) FROM masina
WHERE culoare = 'alb';
```

Un index de tip bitmap poate procesa foarte eficient o astfel de interogare prin simpla numărare a valorilor 1 din bitmap-ul corespunzător valorii alb.

Un index de tip bitmap se creează folosind comanda CREATE INDEX cu opțiunea BITMAP, de exemplu:

```
CREATE BITMAP INDEX culoare_ind ON masina(culoare);
```

Spre deosebire de indecșii tradiționali de tip arbore B*, folosirea indecșilor de tip bitmap se recomandă atunci când:

- numărul de valori distințe ale coloanei indexate este relativ mic (coloana are cardinalitate mică); de exemplu, în cazul unei coloane ce conține starea civilă sau sexul unei persoane.
- majoritatea interogărilor conțin combinații multiple de condiții WHERE ce implică operatorul OR.

În acest caz, indecșii de tip bitmap pot avea o dimensiune mult mai mică decât indecșii de tip arbore B*. Pe de altă parte însă, indecșii de tip bitmap sunt ineficienți în cazul unor coloane cu număr mare de valori.

6.3.3. Tabele organizate pe bază de index

Tabelele organizate pe bază de index (*index-organised tables*) sunt o nouitate adusă de versiunea Oracle 8. Un tabel organizat pe bază de index diferă față de un tabel obișnuit prin faptul că datele tabelului sunt stocate în indexul asociat. Ori de câte ori se vor face modificări asupra datelor din tabel, precum adăugarea de noi rânduri, modificarea sau stergerea rândurilor existente, se va modifica doar indexul.

Mai exact, un tabel organizat pe bază de index este ca un tabel obișnuit având un index de tip arbore B* pe una sau mai multe coloane, dar în loc de a folosi spații separate de stocare pentru tabel și index, Oracle folosește doar un singur index de tip B*, care conține atât valorile coloanelor indexate, cât și valorile celorlalte coloane pentru rândul corespunzător. Deci, în loc ca fiecare intrare a indexului să conțină valoarea coloanei sau coloanelor indexate și valoarea ROWID pentru rândul corespunzător, ea conține întreg rândul. Coloana sau coloanele după care se face indexarea sunt cele care constituie cheia primară a tabelului. Din această cauză, tabelele organizate pe index sunt eficiente pentru accesarea datelor prin intermediul cheii primare sau un prefix valid al acesteia.

Un tabel organizat pe bază de index poate fi manipulat de către aplicații la fel ca un tabel obișnuit, folosind comenzi SQL. Diferența constă în faptul că în cazul tabelului organizat pe bază de index, toate operațiile sunt efectuate numai asupra indexului.

Următorul tabel rezumă diferențele cele mai importante dintre un tabel obișnuit și un tabel organizat pe bază de index.

Tabel obișnuit	Tabel organizat pe baza de index
ROWID identifică în mod unic un rând; specificarea cheii primare este optională.	Cheia primară identifică în mod unic un rând; specificarea cheii primare este obligatorie.
Are coloana implicită ROWID.	Nu are coloana implicită ROWID.
Accesul la date se face prin intermediul ROWID.	Accesul la date se face prin intermediul cheii primare.
Permite definirea de indecși pentru coloane care nu fac parte din cheia primară.	Nu permite definirea de indecși pentru coloane care nu fac parte din cheia primară.
Permite definirea constrângerii UNIQUE și a trigger-elor.	Nu permite definirea constrângerii UNIQUE, dar permite definirea trigger-elor.
Poate fi stocat într-un cluster conținând alte tabele.	Nu poate fi stocat într-un cluster conținând alte tabele.
Poate fi particionat.	Nu poate fi particionat.

Principalele avantaje ale tabelelor organizate pe bază de index sunt următoarele:

- Se reduce timpul de acces la date prin interogări care folosesc ca termen de comparație coloanele indexate;
- Se reduce spațiul de stocare datorită faptului că nu mai este creat un index suplimentar pentru cheia primară a tabelului.

6.3.4. Crearea tabelelor organizate pe bază de index

Pentru a crea un tabel organizat pe bază de index, se folosește comanda SQL CREATE TABLE cu specificația ORGANIZATION INDEX. La crearea unui tabel organizat pe bază de index, trebuie neapărat specificată cheia primară a tabelului. În plus, se pot folosi clauzele OVERFLOW, THRESHOLD și INCLUDING prezentate mai jos, care sunt opționale.

O intrare dintr-un index de tip arbore B* este de obicei destul de mică, constând dintr-o pereche (valoare indexată, ROWID). Pe de altă parte însă, într-un tabel organizat pe bază de index, intrările din indexul arborelui B* corespunzător pot fi foarte mari deoarece ele constau dintr-o pereche (*cheie_primară, coloane_secundare*), adică un rând întreg. Dacă acestea sunt foarte mari, se poate ajunge la situația în care fiecare nod frunză să conțină doar un singur rând sau o porțiune de rând, distrugându-se astfel densitatea indexului de tip arbore B*.

Pentru a evita această problemă, la crearea unui tabel organizat pe index se poate folosi clauza OVERFLOW, care specifică un spațiu tabel de depășire (OVERFLOW TABLESPACE). Alături de această clauză se mai poate specifica și o valoare de prag (PCTTHRESHOLD). Valoarea de prag este specificată ca procent din mărimea unui bloc. Dacă dimensiunea rândului depășește valoarea de prag specificată, atunci coloanele care nu fac parte din cheia primară pentru rândurile care depășesc valoarea de prag vor fi stocate în spațiul tabel de depășire specificat. În acest caz, intrările din index vor conține perechea (*cheie_primară, capăt_rând*), unde *capăt_rând* conține partea de început a restului de coloane; acesta este exact ca o porțiune normală de rând, cu excepția faptului că face referință la porțiunea rămasă de rând, care este stocată în spațiul tabel de depășire. Dacă nu este specificată clauza OVERFLOW, atunci toate liniile care depășesc valoarea de prag sunt eliminate complet și nu mai sunt inserate în tabela organizată pe index.

Clauza INCLUDING, dacă este specificată, determină coloana care va împărți tabelul în porțiunea de index și porțiunea de depășire. Fiecare coloană după coloana INCLUDING este stocată în porțiunea de depășire a tabelului organizat pe index.

Deci, sintaxa comenzi CREATE TABLE pentru un tabel organizat pe index este următoarea:

```
CREATE TABLE nume_tabel
  (nume_coloana tip_data ... PRIMARY KEY ...)
  ORGANIZATION INDEX
  TABLESPACE spațiu_tabel
  [PCTTHRESHOLD intreg]
  [INCLUDING nume_coloana]
  [OVERFLOW TABLESPACE spațiu_tabel_depășire]
```

```

CREATE TABLE carte(
    serie          VARCHAR2(5),
    numar         NUMBER(7),
    titlu          VARCHAR2(20),
    cod_autor     VARCHAR2(10),
    editura        VARCHAR2(10),
    descriere      VARCHAR2(200),
    CONSTRAINT pk_carte PRIMARY KEY (serie, numar)
    ORGANIZATION INDEXED
    TABLESPACE ts_alfa
    PCTTHRESHOLD 40
    INCLUDING editura
    OVERFLOW TABLESPACE ts_beta;

```

6.4. Clustere

6.4.1. Clusterul de index

Clusterul (grupul de tabele), uneori numit și *clusterul de index*, reprezintă o metodă optională de stocare a datelor tabelelor. Tabelele care sunt accesate frecvent împreună pot fi stocate fizic împreună în aceleași blocuri de date. Tabelele dintr-un grup sunt stocate împreună pentru a reduce numărul de operații de I/O cu discul. Cu alte cuvinte, clusterul de index este o regrupare fizică a două sau mai multe tabele, relativ la una sau mai multe coloane, cu scopul creșterii performanțelor.

Cheia unui cluster este definită ca fiind coloana sau coloanele pe care tabelele grupate le au în comun. Cheia unui cluster nu trebuie confundată cu cheia primară a tabelelor grupate. Cheia clusterului poate fi și cheia unuia sau mai multora dintre tabelele grupate, dar acest lucru nu este obligatoriu. Valorile cheii clusterului sunt stocate o singură dată în schemă, reducându-se astfel spațiul de stocare necesar și procesările suplimentare. Rândurile care au aceeași valoare pentru cheia clusterului sunt stocate de către Oracle în același bloc de date.

Așa cum s-a discutat mai devreme, operațiile de I/O cu discul sunt efectuate de către Oracle la nivel de bloc, nu la nivel de rând; dacă datele sunt stocate împreună, ele vor fi copiate de pe disc în memorie în cadrul aceluiași bloc. Când blocul de date este citit, sunt citite toate datele din tabelele clusterului din blocul de date. Acest lucru poate reprezenta un avantaj real dacă datele din tabelele clusterului se folosesc împreună în mod frecvent. De exemplu, folosirea clusterelor este beneficiă în cazul joncțiunilor pentru că toate datele sunt citite în cadrul aceleiași operației de I/O. Dacă însă clusterul este utilizat pentru tabele actualizate sau șterse în mod frecvent, se va înregistra o scădere a performanțelor.

Clusterul este un mecanism transparent aplicațiilor ce utilizează tabelele acestuia. Datele din tabele sunt manipulate ca și când acestea ar fi stocate într-un tabel normal numai că în momentul interogării se va observa o creștere semnificativă a performanțelor.

6.4.2. Crearea unui tabel într-un cluster de index

Pentru a crea un tabel într-un cluster de index se parcurg în mod obligatoriu următorii pași:

1. Crearea clusterului de index;
2. Adăugarea tabelelor la cluster;
3. Crearea indexului de cluster.

Adăugarea tabelelor la cluster se poate face și după ce indexul de cluster a fost creat.

Crearea clusterului de index

Pentru crearea clusterelor de index se folosește comanda SQL CREATE CLUSTER în care trebuie specificată coloana sau grupul de coloane după care sunt grupate tabelele. O sintaxă simplificată a comenzi CREATE CLUSTER este prezentată în continuare:

```
CREATE CLUSTER nume_cluster
  (nume_coloană tip_dată [, nume_coloană tip_dată] ... )
  [PCTFREE intreg] [PCTUSED intreg]
  [SIZE intreg]
  [TABLESPACE spațiu_tabel]
  [STORAGE parametrii_de_stocare]
```

unde clauza optională SIZE specifică spațiul (în bytes, Kbytes sau Mbytes) folosit pentru a stoca toate rândurile având aceeași valoare pentru cheia clusterului. Această valoare nu trebuie să depășească mărimea unui bloc de date. Dacă se omite această opțiune, Oracle rezervă un bloc de date pentru fiecare valoare a cheii.

Exemplul următor creează un cluster sal_dept, în care tabelele sunt grupate după coloanele cod_dept și cod_tara.

```
CREATE CLUSTER sal_dept
  (cod_dept NUMBER(10),
   cod_tara NUMBER(10));
```

Adăugarea tabelelor la cluster

După crearea clusterului, adăugarea tabelelor la cluster se face folosind comanda CREATE TABLE cu clauza CLUSTER:

```
CREATE TABLE nume_cluster
  (nume_coloana tip_data ... [, nume_coloana tip_data ...] ... )
CLUSTER nume_cluster
```

De exemplu:

```
CREATE TABLE departament(
  cod_dept      NUMBER(10),
  cod_tara      NUMBER(10),
  nume_dept     VARCHAR2(10),
  PRIMARY KEY(cod_dept, cod_tara))
CLUSTER sal_dept(cod_dept, cod_tara);
```

```

CREATE TABLE salariat(
    cod_salariat      NUMBER(10)          PRIMARY KEY,
    nume               VARCHAR2(10)        NOT NULL,
    prenume             VARCHAR2(10),
    data_nastere       DATE,
    salariu            NUMBER(10),
    cod_dept           NUMBER(10),
    cod_tara            NUMBER(10),
FOREIGN KEY(cod_dept, cod_tara)
    REFERENCES departament(cod_dept, cod_tara))
CLUSTER sal_dept(cod_dept, cod_tara);

```

Datorită faptului că tabelele clusterului folosesc alocarea de spațiu a acestuia, la folosirea comenzii CREATE TABLE cu opțiunea CLUSTER nu trebuie specificați parametrii PCTFREE, PCTUSED și nici opțiunea TABLESPACE sau parametrii de stocare.

Nu se poate asocia un cluster pentru un tabel care există deja. Dacă se dorește totuși acest lucru, se creează un nou tabel ce va fi grupat în cluster și care va conține înregistrările tabelului existent (folosind opțiunea AS SELECT). Tabelul vechi se șterge iar cel nou va fi redenumit cu numele celui vechi.

Index de cluster

Pentru coloanele cheie ale clusterului *trebuie* creat în mod explicit un *index de cluster* (*index de grup*). Indexul este utilizat pentru localizarea univocă a liniiilor. Indexul de cluster trebuie creat înaintea efectuării oricărui operație de interogare sau DML asupra clusterului. Deci, indexul de cluster trebuie creat înainte de încărcarea datelor în tabelele clusterului; în absența indexului, tabelele nu pot fi utilizate.

Pentru a localiza un rând dintr-un cluster, întâi este folosit indexul de cluster pentru a găsi valoarea cheii clusterului, care face referire la blocul de date asociat.

Spre deosebire de un index de tip arbore B*, un index de cluster include și valorile nule ale cheii. În plus, fiecare element din indexul de cluster conține o intrare pentru fiecare valoare a cheii, și nu o intrare pentru fiecare rând, ca un index de tip B* obișnuit.

Pentru crearea unui index de cluster se folosește comanda SQL CREATE INDEX cu opțiunea ON CLUSTER:

```

CREATE INDEX nume_index
ON CLUSTER nume_cluster
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]
[PCTFREE întreg]

```

Următoarea comandă SQL creează indexul pentru clusterul sal_dept:

```

CREATE INDEX sal_dept_ind
ON CLUSTER sal_dept;

```

6.4.3. Clusterul hash

Un alt tip de cluster pe care îl oferă Oracle este *clusterul hash*. Spre deosebire de clusterul obișnuit, pentru a accesa o înregistrare, clusterul hash nu folosește un index, ci o funcție numerică, numită *funcția hash*. Această funcție are ca parametru *cheia clusterului* și returnează o anumită valoare numită *valoare hash*. Valoarea hash corespunde blocului de date din cluster pe care Oracle îl va căuta sau îl va scrie pe baza comenzi executate. De exemplu în momentul în care un rând este inserat în tabel, acesta va fi stocat în funcție de valoarea hash asociată. Numărul de valori hash este specificat atunci când se creează clusterul. Deci, într-un cluster hash, rândurile care au aceeași valoare hash sunt depozitate împreună pentru a activa accesul rapid.

De exemplu, considerăm un sistem de plată a telefonului în care un anumit tabel depozitează informații despre numărul de telefon al fiecărui client, durata apelurilor efectuate, precum și data fiecărei con vorbiri. Dacă regulile de plată indică faptul că toate apelurile efectuate de un client într-o anumită lună vor fi facturate împreună, puteți depozita aceste apeluri într-un cluster hash a cărui cheie este formată din coloanele ce conțin numărul telefonului, anul și luna în care a avut loc con vorbirea. În momentul în care acest tabel va fi interogat în vederea întocmirii facturii lunare, se va observa o creștere semnificativă a performanțelor.

Un cluster hash poate fi folosit pentru a stoca unul sau mai multe tabele. Ca și în cazul unui cluster de index, pentru clusterul hash cheia poate fi alcătuită din una sau mai multe coloane.

Folosirea clusterelor hash se recomandă în special pentru tabele statice (ale căror date nu se modifică prea mult în timp) și care sunt interogate frecvent cu condiții de filtrare ce sunt egalități care conțin cheia clusterului.

Pentru a găsi sau stoca un rând într-un tabel indexat sau într-un cluster obișnuit sunt necesare cel puțin două operații de I/O (de obicei chiar mai multe): una sau mai multe pentru a afla sau stoca valoarea cheii și alta pentru a căuta sau scrie rândul în tabel sau cluster. În contrast, Oracle folosește funcția hash pentru a localiza rândul într-un cluster hash (pentru aceasta nu este necesară nici o operație de I/O). Ca rezultat, pentru a căuta sau a scrie un rând într-un cluster hash este necesară doar o operație de I/O.

6.4.4. Crearea unui tabel într-un cluster hash

Pentru a crea un tabel într-un cluster hash se parcurg următoarele pași:

1. Crearea clusterului hash;
2. Adăugarea tabelului la cluster;

Crearea clusterului hash

Pentru crearea clusterelor hash se folosește comanda SQL CREATE CLUSTER cu clauza HASHKEYS. O sintaxă simplificată a acestei comenzi este prezentată în continuare:

```
CREATE CLUSTER nume_cluster
  (nume_colonă tip_dată [, nume_colonă tip_dată] ... )
  HASHKEYS întreg
```

```
[HASH IS expresie]
[SIZE întreg]
[PCTFREE întreg] [PCTUSED întreg]
[TABLESPACE spațiu_tabel]
[STORAGE parametrii_de_stocare]
```

unde:

- Clauza HASHKEYS specifică faptul că clusterul ce urmează a fi creat este de tip hash și furnizează numărul de valori hash distințe ce pot fi returnate de funcția hash. Valoarea minimă a numărului de valori distințe este 2. De fapt Oracle rotunjescă numărul specificat în această clauză la primul număr prim mai mare decât valoarea specificată. De exemplu, dacă acest număr furnizat de clauza HASHKEYS este 100, înseamnă că pentru orice valoare pe care o are cheia clusterului funcția hash va returna o valoare între 0 și 100 (adică vor fi 101 valori hash). Dacă clauza HASHKEYS lipsește Oracle va crea implicit un cluster de index.
- HASH IS specifică o expresie ce va fi utilizată de funcția hash a clusterului. Funcția hash va reprezenta restul împărțirii valorii acestei expresii la valoarea specificată de HASHKEYS (expresie MOD valori_hash). Valorile expresiei specificate trebuie să fie valori numerice fără zecimale (întregi) și uniform distribuite pe domeniul acesteia. Dacă acest lucru nu este posibil, atunci clauza HASH IS nu trebuie folosită, caz în care Oracle va folosi implicit o funcție hash internă. De obicei această clauză se folosește atunci când cheia clusterului este formată dintr-o singură coloană care ia valori întregi uniform distribuite (vezi exemplul de mai jos).
- SIZE specifică spațiul (în bytes, Kbytes sau Mbytes) folosit pentru a stoca toate rândurile având aceeași valoare pentru funcția hash. Această valoare nu trebuie să depășească mărimea unui bloc de date. De exemplu, dacă spațiul disponibil dintr-un bloc este de 1700 bytes iar valoarea specificată de SIZE este de 500 bytes, atunci unui bloc îi vor reveni 3 valori hash. Dacă se omite această opțiune, Oracle rezervă un bloc de date pentru fiecare valoarea a funcției hash.

```
CREATE CLUSTER test_cls(cod NUMBER(5))
HASHKEYS 1000
HASH IS cod
SIZE 500;
```

Adăugarea tabelului la cluster

După crearea clusterului hash, adăugarea tabelului sau tabelelor la cluster se face folosind comanda CREATE TABLE cu clauza CLUSTER. Această operație este identică cu operația de adăugare a unui tabel la un cluster de index, de aceea nu necesită lămuriri suplimentare.

```
CREATE TABLE test
(cod      NUMBER(5)    PRIMARY KEY,
...)
CLUSTER test_cls(cod);
```

6.4.5. Modificarea clusterelor

Comanda ALTER CLUSTER poate fi folosită pentru a modifica parametrii de stocare ai clusterului (PCTFREE, PCTUSED și parametrii din clauza STORAGE). Tot cu această comandă se poate realiza alocarea și dealocarea spațiului aferent unui cluster. În cazul unui cluster hash nu se poate schimba valoarea parametrului SIZE, precum și HASH IS sau HASHKEYS. Comanda ALTER CLUSTER urmează aceeași sintaxă ca și în cazul modificării tabelelor.

6.4.6. Distrugerea clusterelor

Pentru distrugerea unui cluster se folosește comanda SQL DROP CLUSTER:

```
DROP CLUSTER sal_dept;
```

Totuși, folosind comanda de mai sus clusterul nu poate fi distrus dacă există tabele în el. Pentru a permite acest lucru se folosește clauza optională INCLUDING TABLES:

```
DROP CLUSTER sal_dept
INCLUDING TABLES;
```

În plus, dacă există tabele din afara clusterului care conțin restricții de integritate care se referă la chei din tabelele clusterului, acestea trebuie de asemenea eliminate. Pentru acest lucru se folosește clauza optională CASCADE CONSTRAINTS:

```
DROP CLUSTER sal_dept
INCLUDING TABLES CASCADE CONSTRAINTS;
```

6.5. Secvențe

De multe ori este necesară crearea unei secvențe de numere pentru a fi folosite ca valori ale cheii unui tabel, de exemplu coloana cod_salariat din tabelul salariat. În loc de a genera aceste numere manual, Oracle oferă o facilitate pentru generarea unei astfel de secvențe în mod automat. Pentru a crea manual o secvență de numere, ar fi necesară blocarea rândului care conține ultima valoare a secvenței (pentru a evita preluarea acestei valori de mai multe ori) generarea noii valori și apoi deblocarea rândului. Blocarea acestor rânduri poate fi evitată prin folosirea generatorului de secvențe furnizat de Oracle. Acesta poate genera secvențe de numere de până la 38 de digiti, fără a fi necesară blocarea manuală a rândurilor. Secvențele sunt memorate și generate indiferent de tabele. Prin urmare, aceeași secvență poate fi utilizată pentru mai multe tabele. O secvență este practic un obiect al bazei de date care servește la generarea unor numere întregi unice, putând fi folosită simultan de mai mulți utilizatori, evitând apariția conflictelor și a blocării. Ca orice alt obiect al bazei de date, o secvență poate fi creată, inadăpostită sau distrusă. Oracle stochează definițiile secvențelor în dicționarul de date.

6.5.1. Crearea, modificarea și distrugerea secvențelor

Pentru a crea o secvență se folosește comanda SQL CREATE SEQUENCE, având următoarea sintaxă:

```
CREATE SEQUENCE nume_secvență
  [INCREMENT BY întreg]
  [START WITH întreg]
  [MAXVALUE întreg | NOMAXVALUE]
  [MINVALUE întreg | NOMINVALUE]
  [CYCLE | NOCYCLE]
  [CACHE întreg | NOCACHE]
  [ORDER | NOORDER]
```

unde:

- INCREMENT BY specifică intervalul dintre două numere ale secvenței. Aceasta poate fi orice valoare pozitivă sau negativă, dar nu poate fi 0. Dacă valoarea este negativă atunci secvența este în ordine descrescătoare, dacă este pozitivă, atunci secvența este în ordine crescătoare. Valoarea implicită (default) este 1.
- START WITH specifică prima valoare generată de secvență. Această opțiune poate fi folosită pentru a începe o secvență crescătoare cu o valoare mai mare decât valoarea sa minimă sau pentru a începe o secvență descrescătoare cu o valoare mai mică decât valoarea sa maximă. Pentru secvențe crescătoare valoarea implicită este valoarea minimă a secvenței, iar pentru secvențe descrescătoare valoarea implicită este valoarea maximă a secvenței.
- MINVALUE specifică valoarea minimă a secvenței. NOMINVALUE specifică o valoare minimă de 1 pentru o secvență crescătoare sau -10^{20} pentru o secvență descrescătoare. NOMINVALUE este valoarea implicită.
- MAXVALUE specifică valoarea maximă a secvenței. NOMAXVALUE specifică o valoare maximă de -1 pentru o secvență descrescătoare și 10^{27} pentru o secvență crescătoare. NOMAXVALUE este valoarea implicită.
- CYCLE arată că secvența continuă să genereze valori și după ce a atins valoarea maximă (în cazul secvențelor crescătoare) sau valoarea minimă (în cazul secvențelor descrescătoare). În acest caz, după ce o secvență crescătoare a atins valoarea maximă, ea va genera valoarea sa minimă; după ce o secvență descrescătoare a atins valoarea minimă, ea va genera valoarea sa maximă. NOCYCLE arată că secvența nu mai poate genera valori după ce a atins valoarea maximă (în cazul secvențelor crescătoare) sau valoarea minimă (în cazul secvențelor descrescătoare). Valoarea implicită este NOCYCLE.
- CACHE arată câte valori ale secvenței sunt preallocate de către Oracle și ținute în memorie pentru acces mai rapid. Valoarea minimă a acestui parametru este 2. NOCACHE arată că nu există valori preallocated ale secvenței. Dacă se omit amândouă opțiunile, CACHE și NOCACHE, Oracle prealocă implicit 20 de valori.

Următorul exemplu creează secvența sal_seq:

```
CREATE SEQUENCE sal_seq
  INCREMENT BY 1
  START WITH 1
  NOMAXVALUE
```

NOCYCLE
CACHE 10:

Fiecare dintre parametrii specificați la crearea unei sevențe pot fi modificați prin executarea comenzi SQL **ALTER SEQUENCE**:

```
ALTER SEQUENCE sal_seq  
MAXVALUE 100000  
CYCLE  
CACHE 20;
```

O secvență poate fi distrusă folosind comanda SQL `DROP SEQUENCE`:

```
DROP SEQUENCE sal_seq;
```

6.5.2. Utilizarea sevențelor

După definirea unei secvențe, aceasta poate fi utilizată și incrementată de mai mulți utilizatori. Oracle nu așteaptă închiderea unei tranzacții care accesează secvența pentru a permite utilizarea ei de către un alt utilizator.

O secvență poate fi referită într-o comandă SQL cu ajutorul pseudo-coloanelor NEXTVAL și CURRVAL, sub forma nume_secventa.NEXT_VAL și respectiv nume_secventa.CURRVAL. Valoarea următoare este generată de pseudo-coloana NEXTVAL, în timp ce valoarea curentă este repetată la folosirea pseudo-coloanei CURRVAL.

În exemplul următor, secvența `sal_seq` este folosită pentru a insera o nouă linie în tabelul `salariat`:

```
INSERT INTO salariat(cod_salariat, nume, prenume,  
                     data_nastere)  
VALUES (sal_seq.NEXTVAL, 'Georgescu', 'Vasile', '11-MAY-69');
```

La generarea unui nou număr din secvență, acesta este disponibil numai sesiunii care l-a creat. Orice altă sesiune care folosește aceeași secvență va obține o nouă valoare din secvență - evident, tot prin intermediul pseudo-coloanei NEXTVAL.

Pentru a utiliza valoarea curentă a secvenței în sesiunea curentă de lucru se folosește pseudo-colonă CURRVAL. Valoarea CURRVAL este disponibilă numai dacă NEXTVAL a fost folosită în sesiunea curentă, în caz contrar încercarea de a folosi pseudo-colonă CURRVAL va produce o eroare. De exemplu, să presupunem că și generarea de valori pentru coloana cod_dept din tabelul departament se face prin intermediul unei secvențe dept_seq. Atunci, următoarele comenzi SQL vor insera două departamente și câte doi salariaj pe unuī fiecare departament:

```
INSERT INTO departament (cod_dept, cod_tara, nume_dept)
VALUES (dept seq.NEXTVAL, 40, 'Proiectare');
```

```
INSERT INTO salariat(cod_salariat, nume, prenume,  
                     data_nastere, cod_dept, cod_tara)
```

```

VALUES(sal_seq.NEXTVAL, 'Vasilescu', 'Costel',
      '17-MAY-70', dept_seq.CURRVAL, 40);

INSERT INTO salariat(cod_salariat, nume, prenume,
                      data_nastere, cod_dept, cod_tara)
VALUES(sal_seq.NEXTVAL, 'Popescu', 'Vasile',
      '17-JUN-72', dept_seq.CURRVAL, 40);

INSERT INTO departament (cod_dept, cod_tara, nume_dept)
VALUES (dept_seq.NEXTVAL, 40, 'Vanzari');

INSERT INTO salariat(cod_salariat, nume, prenume,
                      data_nastere, cod_dept, cod_tara)
VALUES(sal_seq.NEXTVAL, 'Ionescu', 'Gheorghe',
      '12-MAY-71', dept_seq.CURRVAL, 40);

INSERT INTO salariat(cod_salariat, nume, prenume,
                      data_nastere, cod_dept, cod_tara)
VALUES(sal_seq.NEXTVAL, 'Diaconescu', 'Marian',
      '15-MAY-57', dept_seq.CURRVAL, 40);

```

CURRVAL și NEXTVAL pot fi folosite în următoarele locuri:

- clauza VALUES a unei comenzi INSERT, ca în exemplul de mai sus.
- clauza SET a unei comenzi UPDATE, ca în exemplul de mai jos:

```

        UPDATE salariat
        SET cod_salariat = sal_seq.NEXTVAL
        WHERE cod_salariat = 7;

```

- lista unei comenzi SELECT, ca în exemplul de mai jos (pentru descrierea tabelului DUAL vezi secțiunea 6.11):

```

SELECT sal_seq.NEXTVAL
FROM DUAL;

```

Notă: În PL/SQL, pentru a atribui unei variabile valorile NEXTVAL sau CURRVAL se folosește o comandă SELECT de forma:

```

SELECT sal_seq.NEXTVAL
INTO sal_var
FROM DUAL;

```

CURRVAL și NEXTVAL nu pot fi folosite în următoarele locuri:

- o subinterrogare;
- interogarea unei vederi sau a unui instantaneu;
- o comandă SELECT cu operatorul DISTINCT;
- o comandă SELECT cu clauza GROUP BY sau ORDER BY;
- o comandă SELECT care este combinată cu altă comandă SELECT printr-un operator de mulțime (UNION, INTERSECT, MINUS);

- clauza WHERE a unei comenzi SELECT;
- valoarea DEFAULT a unei coloane într-o comandă CREATE TABLE sau ALTER TABLE;
- condiția unei constrângeri CHECK.

Atunci când un număr este generat de către o secvență, aceasta este incrementată indiferent dacă tranzacția a fost actualizată permanent (committed) sau derulată înapoi (rolled back). Dacă doi sau mai mulți utilizatori incrementează aceeași secvență în mod concurrent, numerele generate la cererea unuia dintre ei pot să nu fie consecutive deoarece între timp au putut fi generate alte valori la cererea celorlalți utilizatori. Un utilizator nu poate avea niciodată acces la un număr generat de o secvență la cererea altui utilizator. În plus, un utilizator nu poate ști dacă un alt utilizator generează numere folosind aceeași secvență. Dacă numerele generate de către o secvență sunt folosite la popularea unui câmp, este posibil ca valorile din acel câmp să nu fie numere consecutive – în cazul când o tranzacție a fost derulată înapoi. De exemplu, dacă valoarea CURRVAL este 10 în cazul secvenței sal_seq de mai sus, acest lucru nu implică neapărat faptul că tabelul salariat conține 10 înregistrări.

6.6. Sinonime

Un *sinonim* este un obiect care face referire la un alt obiect din baza de date, cu alte cuvinte este un alias al acestuia. Cu alte cuvinte, Oracle oferă posibilitatea de a atribui mai multe nume aceluiași obiect. În Oracle se pot crea sinonime pentru tabele, vederi, instantanee, secvențe, funcții, proceduri sau pachete din baza de date sau chiar pentru alte sinonime din baza de date. Deoarece un sinonim este doar un alias, el nu necesită nici un spațiu de stocare, în afară de definiția sa din dicționarul bazei de date.

Sinonimele sunt folosite de obicei pentru a permite unui utilizator să folosească obiecte din schema altui utilizator, fără a specifica proprietarul acestora. Când un utilizator folosește un sinonim, acesta trebuie să cunoască doar numele sinonimului, nu și proprietarul sau numele obiectului referit de sinonim. Sinonimele sunt mecanisme puternice prin intermediul cărora se realizează independența unui obiect al bazei de date față de utilizatorul care este proprietarul său.

Un sinonim este de două feluri: *public* sau *privat*. Un sinonim public este proprietatea grupului special de utilizatori numit PUBLIC și poate fi folosit de către toți utilizatorii bazei de date. Un sinonim privat este proprietatea numai a unui utilizator, putând fi accesat la fel ca orice obiect din schema acestuia. Pe de altă parte, trebuie reținut că definirea unui sinonim, fie el public sau privat, nu implică accesul la obiectul referit de acesta de către alți utilizatori. Pentru a accesa respectivul obiect, utilizatorul trebuie să posedă privilegiile adecvate pentru accesarea obiectului.

6.6.1. Crearea, modificarea și distrugerea sinonimelor

Pentru a crea un sinonim se folosește comanda SQL CREATE SYNONYM, având sintaxa:

```
CREATE [PUBLIC] SYNONYM [schema.]nume_sinonim
FOR [schema.]obiect
```

Dacă se specifică opțiunea PUBLIC, atunci sinonimul creat este public, altfel sinonimul este privat.

Pentru a înțelege cum funcționează sinonimele, să considerăm, de exemplu că tabelul salariat aparține schemei utilizatorului costica și că un alt utilizator, mitica, creează un sinonim public, sal1, și un sinonim privat, sal2, pentru acesta:

```
CREATE PUBLIC SYNONYM sal1 FOR costica.salariat;
```

```
CREATE SYNONYM sal2 FOR costica.salariat;
```

Atunci, pentru a vizualiza toate datele din tabela salariat, orice utilizator poate folosi următoarea comandă SQL:

```
SELECT * FROM sal1;
```

În schimb, doar utilizatorul mitica poate folosi comanda SQL de mai jos pentru a vizualiza datele din tabelul salariat - aceasta deoarece sal2 face parte din schema acestui utilizator.

```
SELECT * FROM sal2;
```

În plus, aşa cum s-a menționat mai devreme, pentru a putea executa comenzi de mai sus, utilizatorii respectivi trebuie să posede privilegiul SELECT pe tabelul de bază salariat.

Numele atribuite sinonimelor pot fi aceleași cu ale obiectelor de bază (bineînțeles, cu excepția situației când sinonimul este privat și aparține aceleiași scheme ca și obiectul de bază).

Pentru a distruge un sinonim din baza de date se folosește comanda:

```
DROP [PUBLIC] SYNONYM [schema.]nume_sinonim
```

De exemplu:

```
DROP PUBLIC SYNONYM sal1;
```

```
DROP SYNONYM sal2;
```

6.7. Proceduri, funcții și pachete stocate

După cum vom vedea în capitolul 9, în PL/SQL se pot defini subprograme, acestea fiind blocuri PL/SQL cărora li se pot transmite parametrii și care pot fi apoi apelate. Există două feluri de subprograme PL/SQL: proceduri și funcții. Funcțiile returnează în mod explicit un parametru, pe când procedurile nu returnează în mod explicit nici un parametru.

Subprogramele pot fi definite de orice utilitar Oracle care folosește limbajul PL/SQL. Utilitare precum Oracle Forms și Oracle Reports pot stoca aceste subprograme local, în acest caz ele fiind accesibile numai aplicației și utilitarului respectiv. Pentru ca aceste subprograme să poată fi folosite de orice utilitar Oracle, ele trebuie stocate în baza de date. Astfel de proceduri sau funcții se numesc *proceduri sau funcții stocate*. Odată ce sunt compilate și stocate în dicționarul de date, procedurile și funcțiile devin obiecte ale bazei de date care pot fi accesate de orice număr de aplicații conectate la acea bază de date.

Un pachet este un obiect care grupează mai multe tipuri, obiecte și subprograme PL/SQL. În mod similar cu procedurile și funcțiile de sine stătătoare, pentru ca un pachet să poată fi utilizat de orice utilitar sau aplicație Oracle, este necesar ca acesta să fie stocat în baza de date după ce a fost în prealabil compilat. Astfel de pachete se numesc *pachete stocate*.

Pachetele, procedurile și funcțiile stocate sunt prezentate pe larg în capitolul 9, secțiunile 9.14 și 9.15.

6.8. Declanșatoare ale bazei de date

Un *trigger (declanșator)* al bazei de date este un bloc PL/SQL stocat în baza de date, care este asociat unui tabel. Diferența dintre un trigger și o procedură stocată constă în faptul că trigger-ul este executat în mod *implicit* ori de câte ori asupra tabelului este lansată o anumită comandă DML (INSERT, UPDATE sau DELETE). O nouitate adusă de versiunea Oracle8 sunt trigger-ele INSTEAD OF. Acestea se execută în locul instrucțiunilor DML efectuate asupra unei vederi, furnizând astfel o modalitate de a actualiza vederi ce nu pot fi actualizate direct prin comenzi SQL.

Trigger-ele bazei de date sunt prezentate pe larg în secțiunea 9.16.

6.9. Instantanee

Conceptul de *instantanee (snapshot)* este strâns legat de conceptele de bază de date distribuită și replicarea unei baze de date. O *bază de date distribuită* este o colecție de baze de date stocate fiecare pe calculatoare diferite dar care la nivelul unei aplicații sunt văzute ca o bază de date singulară. Practic pe fiecare calculator există un server Oracle care gestionează baza de date locală dar cooperează și la menținerea consistenței datelor din celelalte baze de date situate la distanță, asigurând astfel funcționarea unei baze de date distribuite. Prin urmare o aplicație poate accesa și modifica prin intermediul unei rețele datele conținute în mai multe baze de date, acestea fiind văzute ca o bază de date singulară ce are proprietatea de a avea datele distribuite pe mai multe calculatoare. *Replicarea unei baze de date* reprezintă facilitatea de a putea copia datele dintr-o bază de date în locații multiple și sisteme situate la distanță. Aceste date pot fi modificate și accesate numai în anumite împrejurări. Replicarea unei baze de date poate fi făcută la nivelul întregii baze de date sau la nivelul unor anumite tabele. Replicarea constituie o modalitate foarte folosită de a spori performanțele unei baze de date distribuite, minimizând traficul prin rețea. Există mai multe opțiuni de replicare a unei baze de date dar opțiunea de bază o reprezintă folosirea instantaneelor.

Instantaneul este un tabel care conține rezultatele unei interogări asupra unuia sau mai multor tabele care în general se regăsesc într-o bază de date situată la distanță. Tabelele conținute în interogare se mai numesc și *tabele master* iar baza de date ce conține aceste tabele se numește *bază de date master*. Interogarea pe care se bazează instantaneul nu poate conține tabele sau vederi care au ca proprietar utilizatorul SYS.

Instantaneul reprezintă în esență o imagine statică a datelor, de unde și numele de instantaneu. Practic, instantaneul este o copie locală a unei baze de date situate la distanță și este utilizat în contextul bazelor de date distribuite. Existând aceste copii locale a datelor, baza de date nu trebuie să transmită date prin intermediul rețelei în mod repetat, reducând în acest fel traficul în rețea și îmbunătățind performanțele aplicației. În plus, instantaneul protejează împotriva pierderii accesului la baza de date situată la distanță și dă posibilitatea de a continua lucruul indiferent de starea rețelei.

Instantanele oferă avantaje în cazul în care tabelele master sunt:

- actualizate rar dar interogate masiv;
- interogate de mai mulți utilizatori situați la distanță.

Există două tipuri de instantane: simple și complexe. Un instantaneu simplu este un instantaneu a cărui interogare de definire este realizată astfel încât fiecare rând din instantaneu corespunde cu un rând sau o parte de rând din tabelul master. În mod specific, interogarea unui instantaneu simplu nu poate conține următoarele elemente:

- clauza GROUP BY sau CONNECT BY;
- funcții de grup sau operatorul DISTINCT;
- operatori de mulțimi (UNION, UNION ALL, INTERSECT, MINUS);
- joncțiuni, cu excepția subinterrogărilor permise (care nu conțin elementele de mai sus).

Instantanele simple pot conține subinterrogări în interogarea prin care sunt definite (cu excepția celor nepermise) deoarece, practic, subinterrogarea filtrează anumite înregistrări selectate de interogarea bazată pe un singur tabel master. Acest lucru nu implică obligativitatea subinterrogării de a se baza pe același tabel master pe care se bazează interogarea principală.

Dacă un instantaneu conține unul din elementele enumerate mai sus atunci acesta este un instantaneu complex. Reîmprospătarea unui instantaneu simplu este mai eficientă decât în cazul unui instantaneu complex însă în momentul interogării instantanele complexe sunt mai eficiente decât cele simple.

Din punctul de vedere al unui utilizator un instantaneu este considerat drept un singur obiect însă în reprezentarea internă a serverului Oracle acesta este văzut ca o colecție de obiecte. Obiectele create se află în baza de date locală sau în baza de date situată la distanță localizate în cadrul schemei unde a fost creat instantaneul. Prin urmare în momentul creării unui instantaneu se vor crea următoarele obiecte:

- în baza de date situată la distanță, un tabel de bază denumit SNAPS_nume_instantaneu, care va conține rezultatele interogării;
- pentru cheia primară a acestui tabelului de mai sus va fi creat un index;
- pentru fiecare instantaneu, Oracle creează o vedere ce va avea același nume ca și instantaneul; această vedere va furniza accesul read-only la instantaneu;
- pentru anumite tipuri de instantanee vor fi creați îndecsi adiționali.

6.9.1. Reîmprospătarea instantaneelor

Datele dintr-un instantaneu nu sunt neapărat identice cu datele curente din tabelele master. Un instantaneu reflectă datele din tabelele master la un anumit moment. Datele dintr-un instantaneu nu sunt actualizate automat odată cu actualizarea tabelului master. Deci dacă datele din tabelele master se modifică, instantaneul nu va reflecta aceste modificări până când asupra acestuia nu este efectuată o *reîmprospătare* (*refresh*). În acest sens existența instantaneelor prezintă și unele dezavantaje. Pentru fiecare instantaneu serverul Oracle trebuie să efectueze procesări suplimentare care să reîmprospăteze periodic datele. Pe lângă toate acestea se adaugă o creștere a necesarului de spațiu de stocare datorită prezenței instantaneului. Prin urmare, trebuie ținut cont de faptul că utilizarea instantaneelor poate duce atât la îmbunătățirea căi și la scăderea performanțelor sistemului.

În cazul unui instantaneu, utilizatorul trebuie să decidă când și cum să aibă loc fiecare operațiune de reîmprospătare a acestuia. De exemplu, dacă tabelul master pe care se bazează un instantaneu este actualizat frecvent atunci este necesar ca procesul de reîmprospătare să aibă loc în mod frecvent. Cu alte cuvinte trebuie analizate cu atenție caracteristicile și cerințele unei aplicații pentru a determina intervalele de timp la care trebuie să aibă loc reîmprospătarea unui instantaneu. Pentru a se păstra integritatea referențială și consistența tranzacțiilor în cazul instantaneelor ce se bazează de tabele master cu legături între ele, Oracle organizează și reîmprospătarea instantaneelor ca parte a unui grup. Crearea unui grup de reîmprospătare pentru instantane se face folosind procedura DBMS_REFRESH.MAKE () .

Reîmprospătarea unui instantaneu se poate face în două moduri:

- manual – prin folosirea procedurii DBMS_SNAPSHOT.REFRESH ()
- automat – prin specificarea modului și a intervalului la care are loc împrospătarea în momentul creării instantaneului. Acest lucru se realizează folosind clauza REFRESH a comenzi CREATE SNAPSHOT.

În cazul în care reîmprospătarea unui instantaneu este făcută în mod automat trebuie specificat momentul de timp la care se va face prima reîmprospătare precum și după cât timp se va repeta această operațiune. La rândul ei, reîmprospătarea automată a unui instantaneu se poate face în două moduri: *complet* sau *rapid* (*fast*).

Reîmprospătarea completă

Pentru a realiza o reîmprospătare completă serverul va gestiona executarea interogării instantaneului. Rezultatul acestei execuții va înlocui datele existente și astfel se va realiza practic reîmprospătarea instantaneului. Reîmprospătarea completă poate fi făcută pentru orice tip de instantaneu.

Reîmprospătarea rapidă

Pentru a realiza o reîmprospătare rapidă, serverul ce gestionează instantaneul identifică mai întâi schimbările care au avut loc în tabela master din momentul în care a avut loc

ultima reîmprospătare și până în prezent iar apoi efectuează aceste modificări și în instantaneu. Reîmprospătările rapide sunt mult mai eficiente decât cele complete în cazul în care modificările făcute în tabelele master sunt puține. Reîmprospătarea rapidă nu poate fi disponibilă pentru un instantaneu decât dacă tabelul master are asociat un jurnal al instantaneului.

Jurnalul unui instantaneu este practic un tabel asociat tabelului master pe care se bazează instantaneul și conține evidența modificărilor datelor din tabelul master. Acest jurnal al instantaneului stocă cheia primară, data calendaristică și ora modificării iar în mod optional poate conține și identificatorul de rând ROWID al acelor înregistrări care au suferit anumite schimbări ale datelor. Jurnalul unui instantaneu nu poate fi folosit decât în cazul unui instantaneu simplu și se poate crea cu ajutorul comenzi CREATE SNAPSHOT LOG. Numele unui jurnal este de forma MLOG\$_nume_tabel_master și este localizat în baza de date de la distanță, în aceeași schemă ca și tabelul master pe care se bazează instantaneul. Dacă există mai multe instantanee simple create prin interogarea aceluiași tabel, toate aceste instantanee vor utiliza în comun același jurnal de instantaneu. Datorită faptului că jurnalul unui instantaneu nu poate fi folosit decât în cazul unui instantaneu simplu, reîmprospătarea rapidă nu se poate aplica decât acestui tip de instantaneu. Totodată reîmprospătarea rapidă nu poate avea loc decât dacă jurnalul instantaneului a fost creat înainte de crearea sau de ultima reîmprospătare a instantaneului.

Pentru un instantaneu se pot specifica segmentele de rollback ce vor fi folosite în nodul local sau în nodul situat la distanță în timpul unei reîmprospătări. Începând cu versiunea Oracle8, instantaneele pot fi partizionate în același mod ca tabelele pentru că în esență ele sunt de fapt tabele.

Un instantaneu creat în versiunea Oracle8 este în mod implicit un instantaneu de tip cheie primară, deoarece un instantaneu identifică rândurile din tabelul master în funcție de cheia primară a acestuia. Pentru compatibilitatea cu versiunile anterioare, Oracle8 suportă și instantanee de tip ROWID. Pentru mai multe informații vezi documentația Oracle [26]/*Oracle8 Database/ Oracle8 Replication*.

6.9.2. Crearea instantaneelor

Datorită faptului că în reprezentarea internă a serverului Oracle un instantaneu este alcătuit dintr-o colecție de obiecte, un utilizator nu poate crea un instantaneu în propria schemă decât dacă deține privilegiile de sistem CREATE SNAPSHOT, CREATE TABLE, CREATE VIEW precum și privilegiul SELECT pe tabelele master. Pentru a crea un instantaneu se folosește comanda CREATE SNAPSHOT cu următoarea sintaxă simplificată:

```
CREATE SNAPSHOT nume_instanceneu
[REFRESH[FAST|COMPLETE|FORCE] [START WITH dată] [NEXT dată]
 [WITH PRIMARY KEY|WITH ROWID]]
[FOR UPDATE]
AS subinterrogare
```

unde:

- clauza REFRESH - specifică modul și intervalul la care se va face reîmprospătarea automată a instantaneului:
 - FAST - va avea loc o reîmprospătare rapidă.
 - COMPLETE - va avea loc o reîmprospătare completă.
 - FORCE - Oracle decide în momentul în care are loc reîmprospătarea dacă aceasta poate să fie rapidă sau nu.

Dacă se omite una din opțiunile de mai sus, Oracle va considera implicit opțiunea FORCE.

- START WITH - specifică data la care va avea loc prima reîmprospătare automată.
- NEXT - specifică data la care va avea loc următoarea reîmprospătare automată. Această dată este folosită la calcularea intervalului dintre două reîmprospătări.

Cele două opțiuni de mai sus trebuie să conțină o dată calendaristică în viitor. Dacă se omit opțiunea START WITH, Oracle determină la crearea instantaneului momentul în care va avea loc prima reîmprospătare prin evaluarea opțiunii NEXT. Dacă se omite opțiunea NEXT, reîmprospătarea automată a instantaneului va avea loc doar o singură dată. În schimb, dacă sunt omise ambele opțiuni, reîmprospătarea automată nu va avea loc niciodată, aceasta făcându-se în mod manual prin utilizarea programatică a procedurii DBMS_SNAPSHOT.REFRESH().

- WITH PRIMARY KEY - specifică faptul că instantaneul va fi de tip cheie primară.
- WITH ROWID - specifică faptul că instantaneul va fi de tip ROWID.

- clauza FOR UPDATE - permite ca datele conținute de instantaneu simplu să poată fi modificate. Pentru mai multe informații, vezi documentația Oracle [26]/*Oracle8 Database/ Oracle8 Replication*.

- AS subinterrogare - specifică interogarea pe care se bazează instantaneul. Această subinterrogare se supune acelorași restricții ca și interogarea ce stă la baza unei vederi.

Următorul exemplu creează un instantaneu numit salariati_ro care să conțină toate înregistrările din tabela angajati și aflată în schema utilizatorului scott din baza de date denumită romania. Reîmprospătarea instantaneului va fi de tip rapid. Prima reîmprospătare a instantaneului va avea loc la o oră după ce instantaneul a fost creat, următoarele reîmprospătări făcându-se la interval de 7 zile.

```
CREATE SNAPSHOT salariati_ro
REFRESH FAST
START WITH sysdate +1/24
NEXT sysdate + 7
AS
SELECT * FROM scott.angajati@romania;
```

6.9.3. Modificarea Instantaneelor

Un instantaneu poate fi modificat într-unul din următoarele sensuri:

- Schimbarea caracteristicilor de stocare;

- Schimbarea modului și a intervalului la care poate avea loc reîmprospătarea automată.

Modificarea unui instantaneu se realizează cu ajutorul comenzi ALTER SNAPSHOT. Exemplul următor modifică modul de reîmprospătare al instantaneului de mai sus, adică aceasta va fi o reîmprospătare completă, precum și intervalul la care va avea loc:

```
ALTER SNAPSHOT salariati_ro
REFRESH COMPLETE
NEXT sysdate + 1;
```

6.9.4. Distrugerea instantaneelor

Ștergerea unui instantaneu se realizează cu comanda DROP SNAPSHOT. În cazul unui instantaneu simplu, în momentul în care instantaneul este șters, toate informațiile din jurnalul aferent instantaneului sunt și ele eliminate. Dacă jurnalul unui instantaneu este distrus, nu se va mai putea efectua o reîmprospătare rapidă iar instantaneul va trebui reîmprospătat complet.

Dacă tabelele master ale unui instantaneu sunt șterse, Oracle nu va distruge automat instantanele bazate pe aceste tabele. De aceea, atunci când se va încerca o nouă reîmprospătare, va surveni o eroare deoarece tabelele principale au fost eliminate.

6.10. Legăturile bazelor de date

Fiecare bază de date conținută într-o bază de date distribuită are propriul său nume, numit nume global al bazei de date, format prin prefixarea numelui de domeniu al rețelei pe care se află baza de date cu numele individual al bazei de date. Numele de domeniu și numele bazei de date este dat de parametrul de inițializare DB_DOMAIN, respectiv DB_NAME. Pot exista mai multe baze de date ce au același nume individual, dar fiecare bază de date trebuie să aibă propriul nume global. De exemplu, dacă domeniile de rețea romania.ro și europa.ro conțin fiecare o bază de date cu denumirea vanzari, aceste baze de date vor avea numele global vanzari.romania.ro respectiv vanzari.europa.ro.

Pentru a facilita cererile unei aplicații ce rulează pe o bază de date distribuită, Oracle utilizează *legăturile unei baze de date*. O legătură a unei baze de date este un obiect al bazei de date locale care permite accesul la obiecte dintr-o bază de date la distanță. Practic legătura definește calea către baza de date situată la distanță. Atunci când o instrucțiune conține o referire la numele global al unui obiect, sistemul Oracle încearcă să găsească o legătură la baza de date specificată în numele global al obiectului. Dacă în baza de date locală este găsită o legătură la baza de date specificată, se încearcă realizarea unei conexiuni. În caz contrar, nu este realizată conexiunea și instrucțiunea nu este executată.

Legătura unei baze de date este transparentă utilizatorului deoarece numele său este același cu numele global al bazei de date la care face referință. Dacă parametrul de inițializare GLOBAL_NAMES este setat pe valoarea TRUE, Oracle asigură că numele unor legături este același ca și numele global al bazei de date. Acesta este un mecanism vital pentru bazele de date distribuite, legăturile devenind astfel total transparente pentru utilizator.

Există trei tipuri de legături la baze de date:

- *private* – legătura este creată pentru un anumit utilizator și nu poate fi folosită decât de utilizatorul specificat. O legătură privată este mult mai sigură decât o legătură publică sau globală deoarece numai proprietarul legăturii sau subprogramele ce se găsesc în aceeași schemă cu legătura pot accesa baza de date corespunzătoare situată la distanță.
- *public* – legătura este creată pentru grupul de utilizatori PUBLIC și poate fi folosită de către toți utilizatorii bazei de date. Legăturile publice se folosesc atunci când mai mulți utilizatori necesită accesarea unei baze de date situată la distanță.
- *global* – legătura este creată și gestionată automat de către un serviciu de rețea în momentul în care rețeaua Oracle folosește numele globale ale unei baze de date. O legătură globală la o bază de date poate fi folosită de orice utilizator care specifică numele global al unei obiecte într-o instrucțiune. Gestionarea legăturilor globale este centralizată și simplă.

Tipul legăturilor dintr-o bază de date distribuită depinde de cerințele specifice aplicației ce utilizează sistemul.

6.10.1. Crearea legăturilor bazelor de date

Pentru a crea o legătură se folosește comanda CREATE DATABASE LINK. O sintaxă simplificată a acestei comenzi este următoarea:

```
CREATE [PUBLIC] DATABASE LINK nume_legătură
[CONNECT TO user IDENTIFIED BY parola]
```

unde:

- opțiunea PUBLIC face ca legătura creată să fie publică. Dacă această opțiune este omisă atunci legătura creată va fi privată;
- user și parola reprezintă numele utilizatorului și parola acestuia utilizată pentru conectarea la baza de date situată la distanță. Această opțiune se folosește pentru legături private.

De exemplu, următoarea comandă creează o legătură privată la baza de date cu numele global `vanzari.romania.ro`. Legătura are același nume ca și baza de date și poate fi folosită de utilizatorul `costel` identificat de parola `costica`:

```
CREATE DATABASE LINK vanzari.romania.ro
CONNECT TO costel IDENTIFIED BY costica
```

Odată creată legătura, utilizatorii pot accesa datele bazei de date respective prin folosirea numelor globale ale obiectelor. De exemplu, după crearea legăturii, utilizatorul `costel` poate interoga tabela `salariati` în următoarea manieră:

```
SELECT * FROM salariati@vanzari.romania.ro;
```

O legătură globală se creează cu ajutorul produsului Oracle Name Server. Pentru mai multe amănunte vezi documentația Oracle [26]/*Network/Net8 Administrator's Guide*. Legăturile nu se pot crea decât dacă utilizatorul deține privilegiul de sistem CREATE DATABASE LINK sau CREATE PUBLIC DATABASE LINK, iar în baza de date

situată la distanță privilegiul CREATE SESSION. Produsul Net8 trebuie instalat atât în nodul local cât și în nodul situat la distanță.

6.10.2. Stergerea legăturilor bazei de date

O legătură la o bază de date situată la distanță se poate șterge din baza de date locală cu ajutorul comenzi:

```
DROP DATABASE LINK nume_legătură
```

Nu se pot șterge legăturile la o bază de date aflate în schema altui utilizator. Acest lucru nu este posibil deoarece Oracle interprează, de exemplu, costel.legatura drept numele unei legături din schema proprie și nu ca legătura legatura din schema utilizatorului costel.

6.11. Dicționarul de date

Dicționarul de date este o componentă esențială a unei baze de date Oracle. Dicționarul de date reprezintă o mulțime de tabele care pot fi doar vizualizate (sunt read-only) și care furnizează informații despre baza de date. El este creat automat la crearea bazei de date și conține:

- definițiile tuturor obiectelor de schema din baza de date (tabele, vederi, indecsi, sinonime, sevențe, proceduri, funcții, pachete, trigger-e etc.);
- informații despre cât spațiu a fost alocat și cât spațiu este utilizat în prezent de obiectele din schema;
- valorile implicate (default) pentru coloane;
- informațiile despre constrângeri de integritate;
- numele utilizatorilor bazei de date;
- privilegiile și rolurile acordate fiecărui rol;
- alte informații generale despre baza de date.

Dicționarul de date este structurat în tabele și vederi, exact ca oricare alte date ale bazei de date. Toate tabelele și vederile dicționarului de date pentru o anumită bază de date sunt stocate în spațiul tabel SYSTEM al bazei de date respective și sunt proprietatea utilizatorului SYS. Din această cauză, asupra obiectelor din schema SYS nu trebuie niciodată efectuate operații de UPDATE, DELETE sau INSERT, deoarece acestea pot avea efecte distrugătoare asupra bazei de date.

Dicționarul de date este un element esențial pentru fiecare bază de date Oracle, dar este în același timp și un mijloc foarte important care poate fi utilizat de dezvoltatori de aplicații sau administratorul bazei de date pentru a afla informații despre aceasta. Deci, asupra obiectelor dicționarului de date există două moduri de acces:

- *de către utilizatorii bazei de date*. tabelele sau vederile dicționarului de date pot fi interogate de utilizatorii bazei de date prin comenzi SQL SELECT pentru a afla informații. Utilizatorii bazei de date nu pot modifica obiectele dicționarului de date, ci le pot doar vizualiza.

- *de către Oracle Server:* acesta modifică dicționarul de date pentru a reflecta schimbările din structura bazei de date. De asemenea, în timpul operațiilor bazei de date, serverul Oracle citește informații din dicționarul de date pentru a verifica dacă obiectele bazei de date există și dacă utilizatorii au acces la ele.

Oracle creează sinonime publice pentru multe din vederile dicționarului de date pentru a permite un acces convenabil la acestea utilizatorilor bazei de date. Pentru dezvoltatorii de aplicații care se referă la obiectele dicționarului de date, se recomandă ca aceștia să folosească sinonimele publice în loc de obiectele propriu-zise: este mai puțin probabil ca numele sinonimelor să se schimbe de la o versiune la alta.

Obiectele dicționarului de date se împart în următoarele două categorii:

- *tabele de bază:* acestea stochează informații despre baza de date asociată. În general, aceste tabele sunt folosite doar de Oracle, utilizatorii le accesează direct foarte rar deoarece informațiile conținute în acestea sunt greu de înțeles.
- *vederi accesibile utilizatorilor:* acestea prezintă informațiile din tabelele de bază într-un format ușor de înțeles pentru utilizatori. Acestea sunt folosite de utilizatorii bazei de date pentru a accesa informațiile din dicționarul de date.

Vederile dicționarului de date au nume care reflectă tipul de utilizare pentru care sunt destinate. Vederile sunt clasificate în trei grupe care se disting între ele prin prefixele USER, ALL și DBA:

- **USER_xxxxx:** Acestea conțin informații despre schema utilizatorului curent, incluzând obiectele schemei, privilegiile acordate de către utilizator, etc. De exemplu, următoarea interogare întoarce numele tuturor tabelelor conținute în schema curentă:

```
SELECT table_name FROM user_tables;
```

- **ALL_xxxxx:** Acestea conțin informații despre obiectele care pot fi accesate de către utilizatorul curent, adică obiectele din propria schemă plus obiectele pentru care are acordate privilegii de acces. De exemplu, următoarea interogare întoarce proprietarul și numele tuturor tabelelor care pot fi accesate de utilizatorul curent:

```
SELECT owner, table_name FROM all_tables;
```

- **DBA_xxxxx:** Acestea conțin informații despre toate obiectele bazei de date, deci ele sunt destinate pentru a fi folosite de către administratorul bazei de date; aceste vederi pot fi folosite doar de utilizatori care au acordat privilegiul de sistem SELECT ANY TABLE sau rolul de DBA. De exemplu:

```
SELECT owner, table_name FROM dba_tables;
```

Ca regulă generală, vederile a căror nume diferă doar prin prefix (de exemplu USER_TABLES, ALL_TABLES și DBA_TABLES) au coloane identice și prin urmare conțin informații similare, cu excepția faptului că vederile cu prefix USER nu conțin coloana OWNER.

O listă a vederilor dicționarului de date se găsește în Anexa 5. Următorul tabel rezumă principalele vederi ale dicționarului de date care conțin informații despre fișierele de date și structurile logice de stocare, utilizatorii, privilegiile și rolurile bazei de date precum și despre obiectele schemei. Sunt listate doar vederile cu prefixul DBA, însă în majoritatea cazurilor există și vederi similare cu prefixele USER și ALL.

Fisiere de date, spații tabel, segmente, extinderi	DBA_TABLESPACES, DBA_DATA_FILES, DBA_SEGMENTS, DBA_EXTENTS
Utilizatori, roluri și privilegii	DBA_USERS, DBA_ROLES, DBA_ROLE_PRIVS, DBA_COL_PRIVS, DBA_SYS_PRIVS, DBA_TAB_PRIVS
Toate obiectele bazei de date	DBA_OBJECTS, DBA_OBJECT_SIZE
Tabele	DBA_TABLES
Constrângeri	DBA_CONSTRAINTS
Vederi	DBA_VIEWS
Indeci	DBA_INDEXES, DBA_IND_COLUMNS
Clustere și clustere hash	DBA_CLUSTERS, DBA_CLU_COLUMNS, DBA_CLUSTER_HASH_EXPRESSIONS
Secvențe	DBA_SEQUENCES
Sinonime	DBA_SYNONYMS
Pachete, Proceduri și Funcții	DBA_SOURCE, DBA_ERRORS,
Triggere	DBA_TRIGGERS, DBA_TRIGGER_COLS
Instantanee	DBA_REGISTERED_SNAPSHOTS, DBA_REGISTERED_SNAPSHOT_GROUPS, DBA_SNAPSHOTS, DBA_SNAPSHOT_LOGS, DBA_SNAPSHOT_LOG_FILTER_COLS, DBA_SNAPSHOT_REFRESH_TIMES
Legături ale bazei de date	DBA_DB_LINKS
Tipuri de date	DBA_TYPES, DBA_TYPE_ATTRS, DBA_DEPENDENCIES, DBA_TYPE_METHODS, DBA_OBJECT_TABLES, DBA_METHOD_PARAM, DBA_METHOD_RESULTS

Alături de vederile menționate mai sus, Oracle mai păstrează niște vederi care conțin informații despre activitatea curentă a bazei de date. Acestea au prefixul V\$_ și sunt numite *tabele de performanță dinamică* (*dynamic performance tables*). De exemplu V\$_DATAFILE conține informații despre fișierele de date ale bazei de date. Aceste vederi nu trebuie în general accesate decât de administratorul bazei de date. și pentru aceste vederi sunt create sinonime publice, acestea fiind prefixate cu V\$ (deci sinonimul public pentru V\$_DATAFILE se va numi V\$DATAFILE).

În dicționarul de date există și vederi ale căror nume nu folosesc prefixele menționate mai sus. Printre acestea menționăm:

- DICTIONARY: Acesta conține toate tabelele, vederile și sinonimele din dicționarul de date.
- DICT_COLUMNS: Acesta conține toate coloanele din obiectele dicționarului de date.

De exemplu, dacă ne interesează să obținem informații referitoare la vederile ce conțin date despre tabelele bazei de date vom utiliza următoarea comandă:

```
SELECT * FROM dictionary WHERE table_name LIKE '%TABLE%';
```

- DUAL: Acesta este un mic tabel, având o singură coloană numită DUMMY de tip VARCHAR2 (1) și un singur rând conținând valoarea 'X'. El este folosit în interogări pentru a returna un rezultat și aceasta datorită faptului că într-o instrucțiune SELECT trebuie specificată neapărat și clauza FROM. Tabelul DUAL este pur și simplu un tabel standard Oracle care este utilizat ca un tabel de test. De exemplu, pentru a afișa data curentă se folosește următoarea interogare:

```
SELECT sysdate FROM dual;
```

Și pentru aceste vederi sunt create sinonime publice. De exemplu, vederea DICTIONARY are sinonimul public DICT.

Pentru obiectele unei baze de date există posibilitatea de a face anumite comentarii asupra lor prin inserarea unui text în dicționarul de date. Comentariul este creat prin comanda:

```
COMMENT ON {TABLE nume_object
           |COLUMN nume_object.nume_coloana}
IS 'text comentariu'
```

unde nume_object reprezintă numele unui tabel, vederi sau instantaneu. Comentariul se poate referi la **tabele**, **vederi**, **instantane** sau **coloane**.

Capitolul 7

Accesul concurrent la date și păstrarea consistenței acestora

Baza de date Oracle permite *acces concurrent*, adică poate fi accesată simultan de mai mulți utilizatori în cadrul mai multor sesiuni de lucru. O problemă fundamentală într-o bază de date cu acces concurrent este păstrarea consistenței datelor. Aceasta înseamnă pe de o parte că în cadrul fiecărei sesiuni de lucru, utilizatorul trebuie să aibă o "vedere" consistentă asupra bazei de date, incluzând modificările vizibile făcute de către alți utilizatori, iar pe de altă parte Oracle trebuie să împiedice modificările incorecte ale datelor, care ar putea compromite integritatea acestora.

Modalitatea cea mai simplă de a gestiona problema accesului concurrent la date ar fi ca fiecare utilizator să-și aștepte rândul pentru a accesa baza de date. Evident, această soluție este total neconvenabilă, scopul unui SGBD fiind de a reduce aceste așteptări astfel încât ele să fie inexistente sau neglijabile pentru alți utilizatori. Cu alte cuvinte, nu se pot face compromisuri nici în ceea ce privește consistența datelor, dar nici în ceea ce privește performanțele sistemului. Mecanismul prin care Oracle gestionează accesul concurrent la date folosește *un model de consistență multiversiune* și diverse tipuri de *blocări*, care vor fi discutate mai târziu în acest capitol. Aceste mecanisme au la bază conceptul de *tranzacție*.

7.1. Tranzacțiile și asigurarea consistenței la scriere

Conceptul de tranzacție este fundamental pentru modul în care Oracle asigură consistența datelor. O *tranzacție* este alcătuită din una sau mai multe instrucțiuni SQL. O tranzacție este cea mai mică unitate de lucru în Oracle, în sensul că pentru orice tranzacție, fie sunt executate toate schimbările făcute bazei de date de către tranzacție, fie nu este executată nici una dintre modificări. Cu alte cuvinte, o tranzacție nu poate fi executată parțial. La sfârșitul unei tranzacții, schimbările făcute bazei de date sunt fie *permanentizate (committed)*, fie sunt anulate, în acest ultim caz spunându-se că tranzacția a fost *derulată înapoi (rolled back)*. După ce o tranzacție a fost permanentizată sau derulată înapoi, următoarea tranzacție va începe cu următoarea instrucțiune SQL.

În timpul executării unei tranzacții, modificările făcute asupra bazei de date de către aceasta nu sunt vizibile altor sesiuni de lucru. Dacă tranzacția este permanentizată, atunci schimbările făcute de aceasta devin vizibile pentru celelalte sesiuni. Dacă tranzacția este derulată înapoi, atunci modificările asupra datelor sunt anulate, astfel încât datele afectate vor rămâne neschimbate, ca și când instrucțiunile SQL din tranzacție nu au fost niciodată executate.

Să luăm de exemplu o bază de date a unei bănci. Când un client transferă o sumă de bani dintr-un cont de depozit într-un cont curent, tranzacția poate să conțină trei operații

separate: retragerea sumei din contul de depozit, depunerea sumei în contul curent și înregistrarea operațiunii într-un jurnal:

```

UPDATE cont_depozit
SET balanta = balanta - 700
WHERE nr_cont = 87410;

UPDATE cont_depozit
SET balanta = balanta + 700
WHERE nr_cont = 87411;

INSERT INTO jurnal(nr_operatiune, suma, cont_debitor,
                   cont_creditor)
VALUES (jurnal_seq.NEXTVAL, 700, 87410, 87411);

COMMIT;

```

Pentru a asigura corectitudinea datelor, Oracle trebuie să garanteze că toate cele trei operații sunt efectuate. Dacă ceva neașteptat, de exemplu o defecțiune hard, împiedică executarea uneia dintre instrucțiuni, atunci celelalte instrucțiuni ale tranzacției trebuie anulate, adică tranzacția trebuie derulată înapoi. Deci, dacă una dintre instrucțiuni nu se poate executa cu succes, atunci celelalte trebuiește anulate pentru a menține consistența bazei de date.

O tranzacție începe cu prima comandă executabilă SQL și se încheie când apare una dintre următoarele comenzi sau evenimente:

- COMMIT [WORK] sau ROLLBACK [WORK]. Comanda COMMIT sau COMMIT WORK permanentizează modificările făcute de către tranzacție. Comanda ROLLBACK sau ROLLBACK WORK derulează înapoi tranzacția;
- o comandă DDL (CREATE, ALTER, DROP). Deci orice comandă DDL este implicit permanentizată, astfel încât orice tranzacție nu poate conține decât cel mult o comandă DDL;
- sfârșitul sesiunii curente de lucru, în acest caz făcându-se implicit o permanentizare;
- defecțiune hard sau soft, caz în care tranzacția este derulată înapoi pentru recuperarea datelor (rollback on recovery) și păstrarea integrității acestora.

După terminarea tranzacției, următoarea comandă executabilă SQL va începe în mod automat o nouă tranzacție.

7.1.1. Comanda AUTOCOMMIT

Dacă se folosește utilitarul SQL*Plus, există posibilitatea ca după fiecare comandă DML să aibă loc o permanentizare automată a datelor (un COMMIT implicit). Acest lucru se poate realiza folosind comanda SET AUTO[COMMIT], având sintaxa:

```
SET AUTO[COMMIT] [ON|OFF]
```

În cazul folosirii acestei comenzi cu opțiunea ON (SET AUTO ON sau SET AUTOCOMMIT ON) o tranzacție nu va putea conține decât cel mult o singură comandă

DML, iar instrucțunea ROLLBACK nu va mai avea nici un efect, datele fiind permanentizate implicit ori de câte ori este executată o comandă DML. Permanentizarea implicită a unei comenzi DML este anulată la executarea comenzi SET AUTO[COMMIT] cu opțiunea OFF (SET AUTO OFF sau SET AUTOCOMMIT OFF).

7.1.2. Puncte de salvare

În cazul tranzacțiilor mai lungi, care conțin multe instrucțuni SQL, pentru a împărți tranzacția în părți mai mici, pot fi declarați delimitatori intermediari, numite *puncte de salvare (savepoints)*. Un punct de salvare poate fi declarat folosind comanda SAVEPOINT, având sintaxa:

```
SAVEPOINT nume_punct_salvare
```

Declararea unor puncte de salvare în interiorul unei tranzacții permite ca la un moment ulterior să fie derulate înapoi toate instrucțiunile executate începând cu un punct de salvare specificat. Acest lucru se poate face folosind comanda ROLLBACK sau ROLLBACK WORK cu specificația TO [SAVEPOINT]:

```
ROLLBACK [WORK] TO [SAVEPOINT] nume_punct_salvare
```

De exemplu, să considerăm următoarea secvență de comenzi SQL:

```
INSERT INTO departament (cod_dept, cod_tara, nume_dept)
VALUES (dept_seq.NEXTVAL, 40, 'Proiectare');
```

```
SAVEPOINT alfa;
```

```
INSERT INTO departament (cod_dept, cod_tara, nume_dept)
VALUES (dept_seq.NEXTVAL, 40, 'Vanzari');
```

```
SAVEPOINT beta;
```

```
INSERT INTO departament (cod_dept, cod_tara, nume_dept)
VALUES (dept_seq.NEXTVAL, 40, 'IT');
```

În acest punct, anularea celei de-a treia inserări în tabelul departament se poate face folosind comanda:

```
ROLLBACK TO beta;
```

Anularea ultimelor două inserări se poate face folosind comanda:

```
ROLLBACK TO alfa;
```

Anularea tuturor celor trei inserări și încheierea tranzacției se face folosind comanda:

```
ROLLBACK;
```

7.2. Asigurarea consistenței cu ajutorul tranzacțiilor

Tranzacțiile furnizează utilizatorului bazei de date sau dezvoltatorului de aplicații capacitatea de a garanta consistența modificărilor operate asupra datelor. Pentru a asigura această consistență, comenzi SQL trebuie să fie grupate în mod logic în tranzacții. O tranzacție trebuie să fie o unitate logică de lucru, nici mai mult, nici mai puțin. Datele din baza de date trebuie să fie consistente înainte de începerea tranzacției și la sfârșitul acesteia. În plus, o tranzacție trebuie să cuprindă doar o singură modificare consistentă a datelor.

De exemplu, să considerăm exemplul transferului bancar de la începutul acestei secțiuni. Cele trei acțiuni efectuate (retragerea sumei din contul de depozit, depunerea ei în contul curent și înregistrarea în jurnal) trebuiau fie să fie toate executate, fie nici una. Pentru a asigura consistența datelor, orice altă acțiune fără legătură cu operațiunea dată (de exemplu, un nou depozit într-un alt cont) nu trebuie inclusă în aceeași tranzacție.

7.2.1. Modelul multiversiune și consistența la citire

Modelul multiversiune, furnizat de către Oracle, asigură *consistență la citire (read consistency)*, adică:

- Garantează că setul de date văzut de orice instrucțiune SQL este consistent și nu se schimbă în timpul execuției unei instrucțiuni; cu alte cuvinte - se spune că Oracle asigură o *consistență la citire la nivel de instrucțiune*,
- Operațiile de citire (SELECT) nu trebuie să vadă datele care sunt în proces de schimbare;
- Operațiile de scriere (INSERT, UPDATE, DELETE) nu trebuie să afecteze consistența datelor și să întrerupă sau să intre în conflict cu alte operații de scriere concurente.

Cea mai simplă modalitate de a ne imagina un sistem care asigură consistența la citire este de ne închipui că fiecare utilizator operează asupra unei copii proprii a bazei de date - de unde și numele de *model multiversiune*.

7.2.2. Implementarea modelului multiversiune

Consistența la citire este implementată de către Oracle prin păstrarea unei copii a datelor în segmentele de revenire. Când este efectuată o operație de INSERT, UPDATE sau DELETE asupra unui tabel, Oracle va face o copie a datelor înainte ca acestea să se modifice într-un segment de revenire. Toate operațiile de citire efectuate asupra tabelului în alte sesiuni de lucru vor vedea datele *însă cum erau ele înainte de schimbare* - deci vor vedea datele din segmentul de revenire. Înainte ca datele să fie permanentizate (prin COMMIT explicit sau implicit), doar utilizatorul din sesiunea curentă va vedea datele modificate, toți ceilalți vor vedea datele din segmentul de revenire. Dacă schimbările sunt permanentizate, atunci ele vor deveni vizibile și pentru ceilalți utilizatori. În acest caz, spațiul ocupat de datele vechi în segmentul de revenire este eliberat și poate fi utilizat din nou. Dacă tranzacția este derulată înapoi (prin ROLLBACK sau în cazul unei defecțiuni) atunci datele din segmentul de revenire sunt scrise înapoi în baza de date, iar ceilalți utilizatori văd în continuare datele inițiale, ca și cum modificările făcute de către tranzacție nu s-ar fi efectuat.

7.2.3 Tranzacții de citire și consistență la citire la nivel de tranzacție

În mod implicit consistența la citire asigurată de către Oracle este la nivel de instrucțiuțe, adică datele nu se schimbă în timpul efectuării unei instrucții. În acest caz, putem avea, în cadrul aceleiași sesiuni de lucru, două interogări identice care produc rezultate diferite - aceasta se poate întâmpla dacă între cele două interogări au fost permanentizate schimbările făcute de către o altă tranzacție. În anumite situații însă, dacă o tranzacție cuprinde mai multe interogări, se poate dori ca toate aceste interogări să aibă o vedere consistentă asupra datelor în raport cu același moment de timp. Cu alte cuvinte, interogările din această tranzacție nu vor simți efectul permanentizărilor făcute de către alte tranzacții după începerea tranzacției curente. În acest caz se spune că *tranzacția este de citire (read only transaction)* iar consistența la citire asigurată de către Oracle este la nivel de tranzacție (transaction-level read consistency). Consistența la citire la nivel de tranzacție este implementată similar cu cea la nivel de instrucțiuțe, adică folosind segmente de revenire.

Pentru a începe o tranzacție de citire se folosește comanda SET TRANSACTION READ ONLY. Această comandă trebuie să fie prima instrucțiuțe din tranzacție. După executarea acestei instrucții, toate permanentizările făcute de către alte tranzacții nu vor fi vizibile în tranzacția curentă. O tranzacție de citire nu poate conține decât interogări (instrucții SELECT); comenziile DML nu sunt permise iar comanda SELECT ... FOR UPDATE va produce o eroare. O tranzacție de citire se va termina atunci când se vor executa comenziile COMMIT [WORK], ROLLBACK [WORK] sau la executarea unei comenzi DDL - deoarece o comandă DDL realizează un COMMIT implicit. În timpul unei tranzacții de citire, alții utilizatori pot continua să interogheze și să modifice datele.

7.3. Blocări

Blocările (locks) sunt folosite de Oracle pentru a asigura integritatea datelor, permitând în același timp accesul concurent la date de către un număr infinit de utilizatori.

În principal, blocările folosite de Oracle sunt de două feluri:

- *Blocări de date* sau *blocări DML*: Aceste blocări protejează datele și le vom discuta pe larg în această secțiune;
- *Blocări de dicționar* sau *blocări DDI*: Acestea protejează definiția unui obiect al schemei (de exemplu a unui tabel) în timp ce o operație DDL acționează asupra acestuia sau face referire la acesta (după cum am menționat înainte, fiecare operație DDL permanentizează implicit tranzacția din care face parte, astfel încât blocarea este necesară doar pe durata unei astfel de operații). De exemplu, dacă un utilizator creează o procedură atunci toate obiectele la care se face referință în acea procedură vor fi blocate, prevenindu-se modificarea sau distrugerea lor înainte de închiderea compilării procedurii.

Comenziile SQL de interogare (SELECT fără clauza FOR UPDATE) nu provoacă nici un fel de blocare.

Din punct de vedere al resursei blocate, blocările DML pot fi de două feluri:

- *Blocări la nivel de rând*, atunci când blocarea afectează un singur rând;

- *Blocări la nivel de tabel*, atunci când blocarea afectează întreg tabelul.

Din punct de vedere al modului de declanșare a blocării, blocările DML sunt de două feluri:

- *Implicită*, atunci când blocarea este făcută în mod automat de către Oracle în urma efectuării unor operații de INSERT, UPDATE sau DELETE și nu necesită nici o acțiune din partea utilizatorului. Rândul asupra căruia se efectuează o astfel de operație este blocat pentru evitarea unor alte operații DML simultane asupra sa;
- *Explicită*, atunci când ele apar ca urmare a executării de către utilizator a următoarelor comenzi SQL:
 - SELECT cu clauza FOR UPDATE;
 - LOCK TABLE .

O blocare a unei resurse este obținută de către o tranzacție, deci blocarea va fi eliberată la încheierea tranzacției. Deci toate blocările obținute în timpul unei tranzacții sunt eliberate atunci când tranzacția este permanentizată sau derulată înapoi. În plus, toate blocările obținute după un punct de salvare sunt eliberate atunci când tranzacția este derulată înapoi până la acel punct de salvare.

7.3.1. Blocări la nivel de rând

Blocările la nivel de rând apar în mod implicit la efectuarea unor operații de INSERT, UPDATE și DELETE, cât și în mod explicit la execuțarea comenzi SELECT cu clauza FOR UPDATE. În această secțiune ne vom referi la blocările implicate la nivel de rând, comanda SELECT ... FOR UPDATE urmând să fie prezentată mai târziu.

În momentul efectuării unor operații de INSERT, UPDATE sau DELETE asupra datelor, rândul sau rândurile afectate de aceste operații sunt blocați. Blocarea se efectuează la nivel de rând, adică la nivelul cel mai de jos posibil, și nu la nivel de tabel, asigurându-se astfel cel mai bun acces concurrent posibil. Combinarea între modelul multiversiune descris mai sus și blocările la nivel de rând asigură faptul că utilizatorii nu intră în competiție pentru date decât dacă încearcă accesarea aceluiși rând. Mai precis, mecanismul folosit de Oracle pentru gestionarea concurenței asigură următoarele:

- Operațiile de citire (SELECT) nu trebuie să aștepte până la terminarea operațiilor de scriere (INSERT, UPDATE, DELETE) sau a altor operații de citire efectuate asupra acelorași rânduri;
- Operațiile de scriere nu trebuie să aștepte până la terminarea operațiilor de citire efectuate asupra acelorași rânduri, cu excepția situației când acest lucru este cerut explicit de o comandă SELECT ... FOR UPDATE;
- Operațiile de scriere trebuie doar să aștepte pentru alte operații de scriere care încearcă să modifice aceleași rânduri în tranzacții concurente.

La nivel de rând, blocările se pot face numai în modul *exclusiv* (X), adică un utilizator nu poate modifica un rând până ce tranzacția care l-a blocat nu s-a terminat (prin permanentizare sau derulare înapoi).

Dacă o tranzacție obține o blocare pentru un rând, atunci ca obține și o blocare la nivel de tabel pentru tabelul corespunzător. O blocare la nivel de tabel este de asemenea necesară pentru a preveni operații DDL care ar interacționa cu modificările de date din tranzacția curentă. Următoarea secțiune explică blocările la nivel de tabel.

7.3.2. Blocările la nivel de tabel

O tranzacție obține o blocare la nivel de tabel în mod implicit atunci când asupra tabelului este executată una dintre comenziile `INSERT`, `UPDATE` sau `DELETE` sau în mod explicit, prin comenziile `SELECT ... FOR UPDATE` și `LOCK TABLE`. Blocările la nivel de tabel au următoarele două scopuri: rezervarea accesului la tabel pentru tranzacția curentă și prevenirea de operații DDL care ar intra în conflict cu această tranzacție – de exemplu, asupra unui tabel nu pot fi executate operațiile `ALTER` sau `DROP` dacă există o tranzacție neterminată care deține asupra acestuia o blocare la nivel de tabel.

Blocările la nivel de tabel pot fi făcute în mai multe moduri, în funcție de caracterul mai mult sau mai puțin restricтив al blocării:

- *RS - row share*
- *RX - row exclusive*
- *S - share*
- *SRX - share row exclusive*
- *X - exclusive*

Modul de blocare a unui tabel determină modurile în care alte tranzacții pot bloca același tabel. Enumerarea de mai sus este făcută în ordinea crescătoare a caracterului restricтив al modului de blocare, de la cel mai puțin restricтив (RS) la cel mai restricтив (X). În continuare trecem în revistă fiecare dintre aceste moduri, arătând acțiunile care produc modul respectiv de blocare și ce acțiuni sunt permise sau nu în alte tranzacții concurente cu tranzacția care deține blocarea.

Mod de blocare RS la nivel de tabel

O blocare în mod RS la nivel de tabel arată că tranzacția care blochează tabelul a blocat rânduri din tabel și intenționează să le modifice. O blocare în mod RS se obține la executarea comenziilor `SELECT` cu clauza `FOR UPDATE` și `LOCK TABLE` cu opțiunea `ROW SHARE`. Modul de blocare RS este cel mai puțin restricтив dintre toate modurile de blocare, permitând gradul cel mai mare de acces concurrent pentru un tabel.

Operații permise. O blocare în mod RS permite acces (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) concurrent la tabel și blocarea concurrentă a tabelului de către altă tranzacție în orice mod, în afară de cel X.

Operații nepermise. O blocare în mod RS nu permite altor tranzacții concurente să blocheze tabelul în mod X.

Mod de blocare RX la nivel de tabel

O blocare în mod RX la nivel de tabel arată în general că tranzacția care deține blocarea a făcut una sau mai multe modificări asupra rândurilor din tabel. O blocare în mod RX este obținută la executarea comenziilor DML (INSERT, UPDATE și DELETE) și a comenziilor LOCK TABLE cu opțiunea ROW EXCLUSIVE. Blocarea în mod RX este ceva mai restrictivă decât blocarea în mod RS.

Operații permise. O blocare în mod RX permite acces (SELECT, INSERT, UPDATE, DELETE) concurrent la tabel și blocarea concurrentă a tabelului de către altă tranzacție în modurile RS și RX.

Operații nepermise. O blocare în mod RX nu permite altor tranzacții concurente să blocheze tabelul în modurile X, SRX și S.

Mod de blocare S la nivel de tabel

O blocare în mod S este obținută la executarea comenziilor LOCK TABLE cu opțiunea SHARE.

Operații permise. O blocare în mod S permite altor tranzacții doar interogarea (SELECT) tabelului și blocarea sa în modurile S și RS.

Operații nepermise. O blocare în mod S nu permite altor tranzacții concurente să efectueze operații de INSERT, UPDATE sau DELETE și să blocheze tabelul în modurile SRX și X.

Mod de blocare SRX la nivel de tabel

Modul de blocare SRX este mai restrictiv decât cel S. O blocare în mod SRX este obținută la executarea comenziilor LOCK TABLE cu opțiunea SHARE ROW EXCLUSIVE.

Operații permise. O blocare în mod SRX permite altor tranzacții doar interogarea (SELECT) tabelului și blocarea sa în modul RS. Asupra unui tabel nu poate deține simultan blocări în modul SRX decât cel mult o singură tranzacție.

Operații nepermise. O blocare în mod SRX nu permite altor tranzacții concurente să efectueze operații de INSERT, UPDATE sau DELETE și să blocheze tabelul în orice mod în afara de RS.

Mod de blocare X la nivel de tabel

Modul de blocare X este cel mai restrictiv mod de blocare. O blocare în mod X este obținută la executarea comenziilor LOCK TABLE cu opțiunea EXCLUSIVE.

Operații permise. O blocare în mod X permite altor tranzacții doar interogarea (SELECT) tabelului. Dacă o tranzacție blochează un tabel în modul X, atunci tabelul nu poate fi blocat în același timp de nici o altă tranzacție în nici un mod.

Operații nepermise. O blocare în mod SRX nu permite altor tranzacții concurente să efectueze operații de INSERT, UPDATE sau DELETE și să blocheze tabelul în orice mod.

Următorul tabel rezumă modul de blocare implicit precum și posibilitatea existenței simultane a mai multor tipuri de blocare la nivel de tabel, în urma executării comenzi SQL corespunzătoare:

Comanda SQL	Mod de blocare implicit	Moduri de blocare permise simultan				
		RS	RX	S	SRX	X
SELECT fără clauza FOR UPDATE	Nici unul	Da	Da	Da	Da	Da
INSERT	RX	Da	Da	Nu	Nu	Nu
UPDATE	RX	Da*	Da*	Nu	Nu	Nu
DELETE	RX	Da*	Da*	Nu	Nu	Nu
SELECT cu clauza FOR UPDATE	RS	Da*	Da*	Da*	Da*	Nu
LOCK TABLE ... în ROW SHARE MODE	RS	Da	Da	Da	Da	Nu
LOCK TABLE ... în ROW EXCLUSIVE MODE	RX	Da	Da	Nu	Nu	Nu
LOCK TABLE ... în SHARE MODE	S	Da	Nu	Da	Nu	Nu
LOCK TABLE ... în ROW SHARE EXCLUSIVE MODE	SRX	Da	Nu	Nu	Nu	Nu
LOCK TABLE ... în EXCLUSIVE MODE	X	Nu	Nu	Nu	Nu	Nu

Da* = Da, dacă nu există altă tranzacție care definește blocări la nivel de rând care intră în conflict cu blocarea la nivel de rând produsă de această comandă SQL; în caz contrar, tranzacția va aștepta eliberarea acestor blocări la nivel de rând.

În general, blocarea implicită la nivel de rând este suficientă în aplicații. Totuși, așa cum am menționat mai devreme, blocarea se poate face și în mod explicit, folosind comanda SELECT cu clauza FOR UPDATE (la nivel de rând și tabel) și comanda LOCK TABLE (la nivel de tabel). Aceste comenzi sunt prezentate în următoarele două secțiuni.

7.3.3. Comanda SELECT cu clauza FOR UPDATE

Folosirea clauzei FOR UPDATE într-o comandă SELECT determină blocarea rândurilor selectate în modul X și blocarea întregului tabel sau tabelelor pe care se face interogarea în modul RS. Deci comanda SELECT cu clauza FOR UPDATE nu modifică rândurile selectate ci doar le blochează. Această comandă este foarte folosită pentru "aducerea" unui rând mai devreme într-o tranzacție, înainte de a-l actualiza. La actualizarea rândurilor (prin comanda UPDATE) blocarea la nivel de rând rămâne neschimbată în timp ce modul de blocare al tabelului devine RX.

Sintaxa acestei comenzi este următoarea:

```
SELECT lista_coloane
  FROM tabel [,tabel] ...
 WHERE condiție
 FOR UPDATE [OF coloana [,coloana] ...] [NOWAIT]
```

Dacă se specifică NOWAIT și rândul sau rândurile selectate sunt deja blocați de altă tranzacție, atunci utilizatorul este înștiințat de acest lucru, returnându-i-se controlul. Dacă NOWAIT nu este specificat, atunci comanda așteaptă până când rândul este deblocat.

Să luăm de exemplu, tranzacția următoare:

```

SELECT salariu
FROM salariat
WHERE cod_salariat = 102
FOR UPDATE OF salariu;

UPDATE salariat
SET salariu = 3500
WHERE cod_salariat = 102;

COMMIT;

```

Atunci, la executarea primei comenzi, rândul cu cod_salariat=102 este blocat în mod X în timp ce tabelul salariat este blocat în mod RS. La execuțarea celei de-a doua comenzi, blocarea la nivel de rând se menține în timp ce blocarea la nivel de tabel este schimbată în modul RX. La executarea instrucției COMMIT, tranzacția este permanentizată și toate blocările sunt eliberate.

7.3.4. Comanda LOCK TABLE

Unul sau mai multe tabele pot fi blocați în oricare din modurile prezentate mai sus folosind comanda LOCK TABLE, având sintaxa:

```

LOCK TABLE nume_tabel [, nume_tabel] ...
IN mod_blocare MODE
[NOWAIT]

```

unde mod_blocare poate avea valorile ROW SHARE, ROW EXCLUSIVE, SHARE, ROW SHARE EXCLUSIVE sau EXCLUSIVE. Folosirea lui NOWAIT este opțională și are aceeași semnificație ca în cazul de mai sus.

În principal, blocarea manuală a unui tabel folosind comanda LOCK TABLE poate fi preferată în următoarele situații:

- Este necesară o "vedere" consistentă asupra mai multor tabele. În acest caz, tabelele pot fi blocați în modul S, împiedicând operațiile DML asupra acestuia;
- Se dorește împiedicarea altor utilizatori de a bloca tabelele asupra cărora operează tranzacția curentă. Acest lucru se poate face blocând tabelele în modul X;
- Se dorește ca o instrucție să nu aștepte pentru deblocarea unei resurse. În acest caz se poate folosi opțiunea NOWAIT.

Următorul tabel rezumă tipurile de blocare la nivel de rând și tabel obținute la executarea diverselor comenzi SQL.

Comanda SQL	Blocare la nivel de rând	Mod de blocare La nivel de tabel
SELECT fără clauza FOR UPDATE		
INSERT	X	RX
UPDATE	X	RX
DELETE	X	RX
SELECT cu clauza FOR UPDATE	X	RS
LOCK TABLE ... în ROW SHARE MODE		RS
LOCK TABLE ... în ROW EXCLUSIVE MODE		RX
LOCK TABLE ... în SHARE MODE		S
LOCK TABLE ... în ROW SHARE EXCLUSIVE MODE		RSX
LOCK TABLE ... în EXCLUSIVE MODE		X

7.3.5. Interblocarea

Datorită accesului concurent la date este posibil ca mai mulți utilizatori să se blocheze reciproc. Această situație se numește *interblocare (deadlock)*, pentru că fiecare dintre utilizatori așteaptă ca celălalt să elibereze resursa blocată. În cazul acesta problema nu se poate rezolva prin simpla așteptare, una din tranzacții trebuind să fie derulată înapoi.

Să luăm, de exemplu, următoarele două tranzacții care se execută simultan:

Tranzacția A	Tranzacția B
<pre>UPDATE salariat SET salariu=3500 WHERE cod_salariat=102;</pre>	<pre>UPDATE departament SET nume_dept='Proiectare' WHERE cod_dept=2 AND cod_tara=40;</pre>
<pre>UPDATE departament SET localitate='Ploiesti' WHERE cod_dept=2 AND cod_tara=40;</pre>	<pre>UPDATE salariat SET manager=101 WHERE cod_salariat=102;</pre>

La executarea primelor instrucțiuni din fiecare tranzacție nu este nici o problemă. Când însă tranzacțiile încercă să obțină blocări pentru instrucțiunile respective se va ajunge la interblocare deoarece tranzacția A trebuie să aștepte ca tranzacția B să deblocheze rândul din tabelul departament în timp ce tranzacția B trebuie să aștepte ca tranzacția A să deblocheze rândul din tabelul salariat. În general o interblocare poate să fie cauzată de mai mult de două tranzacții, de exemplu tranzacția 2 așteaptă după tranzacția 1, tranzacția 3 după tranzacția 2, etc., iar tranzacția 1 așteaptă după tranzacția *n*.

Detectarea Interblocării

Oracle detectază interblocările în mod automat. În acest caz, Oracle semnalează o eroare uneia dintre tranzacțiile implicate și derulează înapoi ultima instrucțiune din această

tranzacție. Acest lucru rezolvă interblocarea, deși cealaltă tranzacție poate încă să aștepte până la deblocarea resursei pentru care așteaptă. De obicei, tranzacția semnalată trebuie derulată înapoi în mod explicit.

Evitarea interblocării

Interblocările pot fi de obicei evitate dacă tranzacțiile care accesează aceleși tabele blochează aceste tabele în aceeași ordine, prin blocare implicită sau explicită. De exemplu, putem impune regula ca, atunci când este accesat atât un tabel master cât și un tabel detaliu, să fie blocat întâi tabelul master și după aceea cel de detaliu. Dacă astfel de reguli sunt bine alcătuite și aplicate, atunci probabilitatea de apariție a interblocărilor este foarte rară.

Capitolul 8

SQL

În acest capitol vor fi prezentate pe larg comanda de interogare a datelor SELECT, comenziile de manipulare a datelor INSERT, UPDATE, DELETE, precum și comanda TRUNCATE. Exemplile din acest capitol sunt realizate cu ajutorul utilitarului SQL*Plus.

8.1. Comanda SELECT

Interogarea datelor stocate în baza de date este considerată cea mai importantă facilitate a unui SGBD. În SQL ea se realizează prin intermediul comenzi SELECT. Comanda SELECT este folosită doar pentru interogarea datelor, ea nepotându-le modifica. Așa cum am văzut mai înainte, comanda SELECT implementează toți operatorii algebrei relaționale.

O instrucție SELECT cuprinde în mod obligatoriu cuvântul cheie FROM. Cu alte cuvinte, sintaxa minimală pentru comanda SELECT este:

```
SELECT atribute FROM obiect
```

După cuvântul cheie SELECT se specifică lista atributelor ce urmează să fie returnate ca rezultat al interogării, iar după cuvântul FROM se precizează obiectele (tabele, vederi, sinonime) din care se vor selecta aceste atrbute.

8.1.1. Atributele comenzi SELECT

În lista de atrbute pot apărea:

- toate coloanele din tabel sau vedere (în ordinea în care au fost definite în comanda CREATE TABLE/CREATE VIEW) prin utilizarea semnului *:

```
SQL> SELECT * FROM profesor;
```

COD	NUME	PRENUME	DATA_NAST	GRAD	SEF	SALARIU	PRIMA	COD_CATEDRA
100	GHEORGHIU	STEFAN	11-AUG-46	PROF		3000	3500	10
101	MARIN	VLAD	19-APR-45	PROF	100	2500		20
102	GEORGESCU	CRISTIANA	30-OCT-51	CONF	100	2800	200	30
103	IONESCU	VERONICA		ASIST	102	1500		10
104	ALBU	GHEORGHE		LECT	100	2200	2500	20
105	VOINEA	MIRCEA	15-NOV-65	ASIST	100	1200	150	10
106	STANESCU	MARIA	05-DEC-69	ASIST	103	1200	600	20

- numele coloanelor separate prin virgulă. Acestea vor apărea în rezultatul interogării în ordinea în care sunt specificate:

```
SQL> SELECT nume, prenume, salariu FROM profesor;
```

NUME	PRENUME	SALARIU
GHEORGHIU	STEFAN	3000
MARIN	VLAD	2500
GEORGESCU	CRISTIANA	2800
IONESCU	VERONICA	1500
ALBU	GHEORGHE	2200
VOINEA	MIRCEA	1200
STANESCU	MARIA	1200

- atribute rezultate din evaluarea unor expresii. Aceste expresii pot conține nume de coloane, constante, operatori sau funcții.

8.1.2. Operatori aritmetici

Operatorii aritmetici pot fi folosiți pentru a crea expresii având tipul de date numeric sau date calendaristice. Operatorii aritmetici sunt: + (adunare), - (scădere), * (înmulțire), / (impărțire). Ordinea de precedență a operatorilor poate fi schimbată cu ajutorul parantezelor.

De exemplu dacă în tabela profesor ne interesează să calculăm impozitul aferent salariilor, știind că acesta este de 38%, putem scrie următoarea interogare:

```
SQL> SELECT nume, salariu, salariu*0.38 FROM profesor;
```

NUMES	SALARIU	SALARIU*0.38
GHEORGHIU	3000	1140
MARIN	2500	950
GEORGESCU	2800	1064
IONESCU	1500	570
ALBU	2200	836
VOINEA	1200	456
STANESCU	1200	456

8.1.3. Aliasuri de coloane

Pentru exemplul anterior observăm că în momentul afișării rezultatelor, SQL*Plus utilizează numele coloanelor ca antet. Când acest lucru poate face dificilă înțelegerea rezultatelor, se poate schimba antetul prin atribuirea altor nume coloanelor (numite 'alias'-uri ale coloanei). Acest lucru se realizează specificând alias-ul după numele coloanei. În cazul în care alias-ul conține spații sau caractere speciale cum ar fi + sau -, acesta se va specifica între ghilimele.

În exemplul de mai jos alias-ul "DATA_NASTERE" conține spații deci este specificat între ghilimele în timp ce alias-ul impozit nu conține spații sau caractere speciale deci nu trebuie specificat obligatoriu între ghilimele.

```
SQL> SELECT nume, data_nast "DATA_NASTERE", salariu,
           salariu*0.38 impozit
      FROM profesor;
```

NUME	DATA NASTERE	SALARIU	IMPOZIT
GHEORGHIU	11-AUG-46	3000	1140
MARIN	19-APR-45	2500	950
GEORGESCU	30-OCT-51	2800	1064
IONESCU		1500	570
ALBU		2200	836
VOINEA	15-NOV-65	1200	456
STANESCU	05-DEC-69	1200	456

8.1.4. Operatorul de concatenare

Operatorul de concatenare, notat `||`, permite legarea coloanelor cu alte coloane, expresii aritmetice sau valori constante pentru a crea o expresie de tip sir de caractere. De exemplu, pentru a combina codul, numele si prenumele unui profesor, separate printr-un spatiu, se foloseste urmatoarea interogare:

```
SQL> SELECT cod || ' ' || nume || ' ' || prenume detalii
      FROM profesor;
```

DETALII

100	GHEORGHIU STEFAN
101	MARIN VLAD
102	GEORGESCU CRISTIANA
103	IONESCU VERONICA
104	ALBU GHEORGHE
105	VOINEA MIRCEA
106	STANESCU MARIA

8.1.5. Convertirea valorilor Null cu ajutorul functiei NVL

Dacă la o înregistrare pentru o anumită coloană valoarea este necunoscută sau neaplicabilă, atunci aceasta este Null. Această valoare nu trebuie confundată cu zero sau sirul de caractere format dintr-un spatiu. Așa cum am văzut în exemplele de până acum, dacă o anumită valoare este Null, SQL*Plus nu va afisa nimic. Pentru expresiile aritmetice, dacă una dintre valorile componente este Null, atunci și rezultatul expresiei este Null. De exemplu, pentru a calcula salariul total, ce reprezintă suma dintre coloanele salariu și prima putem folosi interogarea:

```
SQL> SELECT nume, salariu, prima,
           salariu+prima "SALARIU TOTAL"
      FROM profesor;
```

NUME	SALARIU	PRIMA	SALARIU TOTAL
GHEORGHIU	3000	3500	6500
MARIN	2500		
GEORGESCU	2800	200	3000
IONESCU	1500		

ALBU	2200	2500	4700
VOINEA	1200	150	1350
STANESCU	1200	600	1800

Observăm că pentru acele înregistrări care au avut valoarea Null în câmpul prima expresia ce calculează salariul total returnează tot valoarea Null.

Pentru a obține un rezultat diferit de Null, valorile Null trebuie convertite într-un număr (în cazul de față 0) înainte de a aplica operatorul aritmetic. Această convertire se poate realiza prin intermediul funcției NVL. Funcția NVL are două argumente. Dacă valoarea primului argument nu este Null, atunci NVL întoarce această valoare; altfel, ea întoarce valoarea celui de-al doilea argument. Cele două argumente pot avea orice tip de date. Dacă tipurile de date ale celor două argumente diferă, Oracle încercă să convertească al doilea argument la tipul de date al primului. De exemplu, pentru a putea calcula salariul total al tuturor cadrelor didactice, trebuie să convertim valoarea Null din coloana prima a tabelui profesor în valoarea 0 folosind NVL(prima, 0):

```
SQL> SELECT nume, salariu, prima,
      salariu+NVL(prima,0) "SALARIU TOTAL"
    FROM profesor;
```

NUME	SALARIU	PRIMA	SALARIU TOTAL
GHEORGHIU	3000	3500	6500
MARIN	2500		2500
GEORGESCU	2800	200	3000
IONESCU	1500		1500
ALBU	2200	2500	4700
VOINEA	1200	150	1350
STANESCU	1200	600	1800

8.1.6. Prevenirea selectării înregistrărilor dupicate

O comandă SELECT care nu cuprinde cuvântul cheie DISTINCT va afișa toate înregistrările care rezultă din interogare, indiferent dacă unele dintre ele sunt identice. De exemplu, interogarea de mai jos va returna următoarele rezultate:

```
SQL> SELECT grad FROM profesor;
```

GRAD

PROF
PROF
CONF
ASIST
LECT
ASIST
ASIST

În cazul folosirii cuvântului cheie DISTINCT înregistările după care sunt eliminate, afișându-se numai prima apariție a valorilor câmpurilor specificate în lista de atribute. De exemplu:

```
SQL> SELECT DISTINCT grad FROM profesor;
```

GRAD

ASIST
CONF
LECT
PROF

Dacă lista de atribute conține mai multe coloane, operatorul DISTINCT va afecta toate coloanele selectate. Următorul exemplu va afișa toate combinațiile de valori care sunt diferite pentru coloanele grad și cod_catedra.

```
SQL> SELECT DISTINCT grad, cod_catedra FROM profesor;
```

GRAD	COD_CATEDRA
-----	-----
ASIST	10
ASIST	20
CONF	30
LECT	20
PROF	10
PROF	20

8.1.7. Clauza ORDER BY

În mod normal, în urma interogării înregistrările rezultate apar în aceeași ordine în care au fost introduse în baza de date. Pentru a modifica ordinea de afișare se utilizează clauza ORDER BY, care sortează înregistrările după valorile din una sau mai multe coloane. Această clauză este urmată de numele coloanelor după care se va face sortarea. De asemenea, este posibil să se identifice coloana dintr-o clauză ORDER BY folosind în locul numelui coloanei un număr ordinal ce reprezintă poziția coloanei în rezultat (de la stânga la dreapta). Această facilitate face posibilăordonarea rezultatului interogării în funcție de un atribut al clauzei SELECT care poate fi o expresie complexă, fără a mai descrie acea expresie.

Nu există nici o limită a numărului de coloane în funcție de care se poate face sortarea. Nu este obligatoriu ca ordinea de sortare să se facă în funcție de o coloană care să fie afișată, dar în acest caz nu se mai poate folosi numărul de ordine al coloanei în loc de numele acestora. Înregistrările vor fi sortate mai întâi în funcție de primul câmp specificat după clauza ORDER BY, apoi, înregistrările care au aceeași valoare în acest prim câmp sunt sortate în funcție de valoarea celui de-al doilea câmp specificat după clauza ORDER BY, și a.m.d.

De exemplu, pentru a sorta ascendent înregistrările în funcție de impozitul pe salar, folosim interogarea:

```
SQL> SELECT nume, salariu*0.38 FROM profesor
      ORDER BY salariu*0.38;
```

care este echivalentă cu:

```
SQL> SELECT nume, salariu*0.38 FROM profesor ORDER BY 2;
```

NUME	SALARIU*0.38
VOINEA	456
STANESCU	456
IONESCU	570
ALBU	836
MARIN	950
GEORGESCU	1064
GHEORGHIU	1140

Înregistrările sunt sortate în mod implicit în ordine ascendentă (opțiunea ASC), afișarea în ordine descendentă făcându-se prin utilizarea opțiunii DESC. Observați că în momentul sortării valoarea Null este considerată cea mai mare, deci dacă sortarea este ascendentă este trecută pe ultima poziție și dacă sortarea este descendentă este trecută pe prima poziție. De exemplu:

```
SQL> SELECT grad, prima FROM profesor
      ORDER BY grad, prima DESC;
```

GRAD	PRIMA
ASIST	
ASIST	600
ASIST	150
CONF	200
LECT	2500
PROF	
PROF	3500

Se observă că în exemplul de mai sus înregistrările au fost mai întâi sortate ascendent (specificație implicită) în funcție de gradul didactic. Înregistrările cu același grad au fost apoi ordonate în funcție de cel de-al doilea criteriu de sortare, adică în funcție de prima primită cu specificația explicită de sortare descendentă.

8.1.8. Clauza WHERE

Clauza WHERE se folosește pentru a regăsi înregistrări ce corespund unei anumite condiții evaluată cu valoarea de adevară *True*, adică pentru a realiza anumite restricții de selecție. Astfel, clauza WHERE corespunde restricțiilor operatorilor din algebra relațională. Cu alte cuvinte, dacă o clauză ORDER BY este o clauză de sortare, clauza WHERE este o clauză de filtrare. Dacă nu se specifică o clauză WHERE, interogarea va întoarce ca rezultat toate rândurile din tabel. Alături de clauza FROM care este obligatorie, WHERE este cea mai folosită clauză a comenzii SELECT. Din punct de vedere sintactic, clauza WHERE este optională, dar atunci când este introdusă urmează imediat după clauza FROM:

```

SELECT      atribute
FROM        obiect
WHERE       condiție

```

Datorită existenței valorii Null, în SQL o condiție poate lua atât valorile *True* și *False* cât și valoarea *Necunoscut* (despre acest lucru vom discuta mai în detaliu în secțiunile următoare). O comandă SELECT cu clauza WHERE va returna toate înregistrările pentru care condiția are valoarea *True*. Condiția clauzei WHERE poate cuprinde numele unor coloane, constante, operatori de comparație sau operatori logici (NOT, AND, OR). Operatorii de comparație se pot împărți în două categorii: operatori relaționali și operatori SQL. Toți acești operatori sunt trecuți în revistă în continuare.

8.1.9. Operatori relaționali

Operatorii relaționali sunt:

=	egal
>	mai mare
>=	mai mare sau egal
<	mai mic
<=	mai mic sau egal
<> și !=	diferit

Cele două valori care sunt comparate trebuie să aparțină unor tipuri de date compatibile.

De exemplu, pentru a selecta toate cadrele didactice care nu aparțin catedrei cu codul 10 folosim următoarea interogare:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE cod_catedra <>10;
```

NUME	PRENUME
MARIN	VLAD
GEORGESCU	CRISTIANA
ALBU	GHEORGHE
STANESCU	MARIA

Șirurile de caractere și data calendaristică trebuie incluse între apostrofuri. De exemplu, pentru a selecta numai acele cadre didactice care au gradul didactic de profesor vom utiliza următoarea interogare:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE grad ='PROF';
```

NUME	PRENUME
GHEORGHIU	STEFAN
MARIN	VLAD

În cazul şirurilor de caractere, literele mici sunt diferite de literele mari. De exemplu, următoarea interogare nu va returna nici o înregistrare:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE grad='prof';
```

Toți operatorii de comparație pot fi folosiți atât pentru valori numerice cât și pentru şiruri de caractere sau date calendaristice. De exemplu, pentru a afla toate cadrele didactice care s-au născut înainte de 1 Ianuarie 1960 folosim interogarea:

```
SQL> SELECT nume, prenume, data_nast FROM profesor
      WHERE data_nast<'01-JAN-65';
```

NUME	PRENUME	DATA_NAST
GHEORGHIU	STEFAN	11-AUG-46
MARIN	VLAD	19-APR-45
GEORGESCU	CRISTIANA	30-OCT-51

În cazul şirurilor de caractere ordonarea se face după codul ASCII al acestora. De exemplu, pentru a afla toate cadrele didactice ale căror nume sunt în ordinea alfabetica după litera 'M' se poate folosi interogarea.

```
SQL> SELECT nume, prenume FROM profesor
      WHERE nume>='M';
```

NUME	PRENUME
MARIN	VLAD
VOINEA	MIRCEA
STANESCU	MARIA

Trebuie remarcat că interogarea de mai sus este corectă numai în cazul în care numele angajaților începe cu o literă mare, literele mici fiind în urma celor mari.

Există posibilitatea de a compara valoarea unei coloane cu valoarea altrei coloane pentru aceeași înregistrare. De exemplu, dacă dorim să selectăm acele cadre didactice care au primit primă mai mare decât salariul de bază vom avea:

```
SQL> SELECT nume, prenume, salariu, prima
      FROM profesor WHERE salariu<prima;
```

NUME	PRENUME	SALARIU	PRIMA
GHEORGHIU	STEFAN	3000	3500
ALBU	GHEORGHE	2200	2500

8.1.10. Operatori SQL

Există patru tipuri de operatori SQL, care pot opera cu toate tipurile de date:

1. BETWEEN...AND...
2. IN
3. LIKE
4. IS NULL

Operatorul BETWEEN...AND...

Operatorul BETWEEN...AND... permite specificarea unui domeniu mărginit de două valori între care trebuie să se afle valoarea testată. Domeniul de valori specificat este un interval închis iar limita inferioară trebuie specificată prima.

Astfel, dacă dorim selectarea celor cadre didactice care au salariul între 2000 și 3000 vom folosi comanda:

```
SQL> SELECT nume, prenume, salariu FROM profesor
      WHERE salariu BETWEEN 2000 AND 3000;
```

NUME	PRENUME	SALARIU
GHEORGHIU	STEFAN	3000
MARIN	VLAD	2500
GEORGESCU	CRISTIANA	2800
ALBU	GHEORGHE	2200

Operatorul IN

Operatorul IN permite specificarea unei liste de valori, valoarea testată trebuie să se afle printre valorile acestei liste.

De exemplu, dacă dorim selectarea celor cadre didactice care au gradul de conferențiar, lector sau asistent vom utiliza comanda:

```
SQL> SELECT nume, prenume, grad FROM profesor WHERE grad
      IN('CONF', 'LECT', 'ASIST');
```

NUME	PRENUME	GRAD
GEORGESCU	CRISTIANA	CONF
IONESCU	VERONICA	ASIST
ALBU	GHEORGHE	LECT
VOINEA	MIRCEA	ASIST
STANESCU	MARIA	ASIST

Operatorul LIKE

Operatorul LIKE permite specificarea unui anumit model de sir de caractere cu care trebuie sa se potriveasca valoarea testata. Acest operator se foloseste in mod special atunci cand nu se stie exact valoarea care trebuie cautata. Pentru a construi modelul dupa care se face cautarea pot fi folosite doua simboluri:

- % semnifica orice secventa de zero sau mai multe caractere
- _ semnifica orice caracter (care apare o singura data)

De exemplu, urmatoarea comanda SELECT va returna toate cadrele didactice al caror nume incepe cu litera 'G':

```
SQL> SELECT nume, prenume FROM profesor
      WHERE nume LIKE 'G%';
```

NUME	PRENUME
GHEORGHIU	STEFAN
GEORGESCU	CRISTIANA

Daca dorim selectarea celor cadre didactice al caror nume are litera 'O' pe a doua pozitia, indiferent de lungimea cuvantului, vom avea:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE nume LIKE '_O%';
```

NUME	PRENUME
IONESCU	VERONICA
VOINEA	MIRCEA

O problema intervine atunci cand sirul conține caracterele % sau _ (de exemplu sirul 'J_James') deoarece aceste caractere au semnificație predefinită. Pentru a schimba interpretarea acestor caractere se foloseste optiunea ESCAPE.

De exemplu, pentru a cauta toate titlurile de carte care incep cu caracterele 'J_' se poate folosi interrogarea:

```
SQL> SELECT titlu FROM carte
      WHERE titlu LIKE 'J/_%' ESCAPE '/';
```

In exemplul de mai sus optiunea ESCAPE identifica caracterul '/' ca fiind caracterul "escape". Deoarece in modelul folosit pentru LIKE acest caracter precede caracterul '_', acesta din urma va fi interpretat ca o simpla litera, fara alta semnificație.

Avantajul unei viteze mari de regasire ca urmare a indexarii este pierdut in momentul in care se cauta un sir de caractere care incepe cu _ sau % in intr-o coloană indexată.

Operatorul IS NULL

Operatorul **IS NULL** testează dacă o valoare este Null. Pentru a vedea utilitatea acestui operator să considerăm următoarele interogări:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE prima = NULL;
```

```
SQL> SELECT nume, prenume FROM profesor
      WHERE prima <> NULL;
```

Amândouă aceste interogări nu vor returna nici o înregistrare. Aceste lucru pare surprinzător la prima vedere deoarece ne-am fi așteptat ca prima interogare să returneze toate cadrele didactice care nu au primit primă, iar a doua toate cadrele didactice care au primit primă. În SQL însă, orice condiție care este formată dintr-un operator de comparație care are unul dintre termeni valoarea Null va avea ca rezultat valoarea Necunoscut, diferită de valoarea True (pentru care se face filtrarea). Pentru compararea cu Null se folosește operatorul special **IS NULL**.

Deci pentru a afla cadrele didactice care nu au primit primă se folosește interogarea:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE prima IS NULL;
```

NUME	PRENUME
MARIN	VLAD
IONESCU	VERONICA

La fel, pentru a afla cadrele didactice ale căror dată de naștere nu se cunoaște vom folosi următoarea interogare:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE data_nast IS NULL;
```

NUME	PRENUME
IONESCU	VERONICA
ALBU	GHEORGHE

8.1.11. Operatorii logici

Negarea operatorilor

În unele cazuri sunt mai ușor de căutat înregistrările care *nu* îndeplinesc o anumită condiție. Acest lucru se poate realiza folosind operatorul NOT. Operatorul NOT se poate folosi pentru negarea unei expresii logice (de exemplu expresii de tipul NOT *coloana=...*) sau pentru negarea operatorilor SQL în modul următor:

- NOT BETWEEN
- NOT IN
- NOT LIKE
- IS NOT NULL

De exemplu, pentru a selecta cadrelor didactice al căror nume nu începe cu litera 'G' se folosește interogarea:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE nume NOT LIKE 'G%';
```

NUME	PRENUME
MARIN	VLAD
IONESCU	VERONICA
ALBU	GHEORGHE
VOINEA	MIRCEA
STANESCU	MARIA

Pentru a selecta cadrele didactice care au primit prima se folosește interogarea:

```
SQL> SELECT nume, prenume FROM profesor
      WHERE prima IS NOT NULL;
```

NUME	PRENUME
GHEORGHIU	STEFAN
GEORGESCU	CRISTIANA
ALBU	GHEORGHE
VOINEA	MIRCEA
STANESCU	MARIA

Notă: Negarea unei expresii logice care are valoarea Necunoscut va avea tot valoare Necunoscut. De exemplu, o expresie de genul NOT coloana=NULL va avea valoarea Necunoscut, următoarea interogare nereturnând deci nici o înregistrare:

```
SQL> SELECT nume, prenume FROM profesor .
      WHERE NOT prima = NULL;
```

Condiții multiple de Interogare (operatorii AND și OR)

Operatorii AND și OR pot fi utilizati pentru a realiza interogări ce conțin condiții multiple. Expresia ce conține operatorul AND este adevărată atunci când ambele condiții sunt adevărate iar expresia ce conține operatorul OR este adevărată atunci când cel puțin una din condiții este adevărată. În aceeași expresie logică se pot combina operatorii AND și OR dar operatorul AND are o precedență mai mare decât operatorul OR, deci este evaluat mai întâi.

În momentul evaluării unei expresii, se calculează mai întâi operatorii în ordinea precedenței, de la cel cu precedența cea mai mare până la cel cu precedența cea mai mică. Dacă operatorii au precedență egală atunci ei sunt calculați de la stânga la dreapta.

Precedența operatorilor, pornind de la cea mai mare la cea mai mică este următoarea:

- toți operatorii de comparație și operatorii SQL: >, <, <=, >=, =, <>, BETWEEN...AND..., IN, LIKE, IS NULL;
- operatorul NOT;
- operatorul AND;
- operatorul OR.

Pentru a schimba prioritatea operatorilor se folosesc parantezele.

În exemplele de mai jos se observă modul de evaluare al expresiei în funcție de precedența operatorilor, precum și modul în care parantezele pot schimba acest lucru.

```
SQL> SELECT nume, prenume, salariu, cod_catedra
      FROM profesor WHERE salariu>2000 AND cod_catedra=10
      OR cod_catedra=20;
```

este echivalentă cu:

```
SQL> SELECT nume, prenume, salariu, cod_catedra
      FROM profesor WHERE (salariu>2000 AND cod_catedra=10)
      OR cod_catedra=20;
```

NUME	PRENUME	SALARIU	COD_CATEDRA
GHEORGHIU	STEFAN	3000	10
MARIN	VLAD	2500	20
ALBU	GHEORGHE	2200	20
STANESCU	MARIA	1200	20

dar este definită de:

```
SQL> SELECT nume, prenume, salariu, cod_catedra
      FROM profesor WHERE salariu>2000 AND (cod_catedra=10
      OR cod_catedra=20);
```

NUME	PRENUME	SALARIU	COD_CATEDRA
GHEORGHIU	STEFAN	3000	10
MARIN	VLAD	2500	20
ALBU	GHEORGHE	2200	20

8.1.12. Funcții

Funcțiile sunt o caracteristică importantă a SQL și sunt utilizate pentru a realiza calcule asupra datelor, a modifica date, a manipula grupuri de înregistrări, a schimba formatul datelor sau pentru a converti diferite tipuri de date. Funcțiile se clasifică în două tipuri:

1. Funcții referitoare la o singură înregistrare:

- funcții caracter;
- funcții numerice;
- funcții pentru data calendaristică și oră;
- funcții de conversie;
- funcții diverse.

2. Funcții referitoare la mai multe înregistrări:

- funcții totalizatoare sau funcții de grup.

Diferența dintre cele două tipuri de funcții este numărul de înregistrări pe care acționează. Funcțiile referitoare la o singură înregistrare returnează un singur rezultat pentru fiecare rând al tabelului, pe când funcțiile referitoare la mai multe înregistrări returnează un singur rezultat pentru fiecare grup de înregistrări din tabel.

O observație importantă este faptul că dacă se apelează o funcție SQL ce are un argument egal cu valoarea Null, atunci în mod automat rezultatul va avea valoarea Null. Singurele funcții care nu respectă această regulă sunt: CONCAT, DECODE, DUMP, NVL și REPLACE.

În continuare vom exemplifica și prezenta la modul general cele mai importante funcții, pentru sintaxă urmând să consultați Anexa 6.

8.1.13. Funcții referitoare la o singură înregistrare

Sunt funcții utilizate pentru manipularea datelor individuale. Ele pot avea unul sau mai multe argumente și returnează o valoare pentru fiecare rând rezultat în urma interogării.

Funcții caracter

Aceste funcții au ca argumente date de tip caracter și returnează date de tip VARCHAR2, CHAR sau NUMBER.

Cele mai importante funcții caracter sunt:

- CONCAT - returnează un sir de caractere format prin concatenarea a două siruri;
- LOWER - modifică toate caracterele în litere mici;
- UPPER - modifică toate caracterele în litere mari;
- LENGTH - returnează numărul de caractere dintr-un anumit câmp;
- REPLACE - caută într-un sir de caractere un subșir iar dacă îl găsește îl va înlocui cu un alt sir de caractere;
- SUBSTR - returnează un subșir de caractere având o anumită lungime începând cu o anumită poziție;

- TRANSLATE - căută într-un prim sir de caractere fiecare dintre caracterele specificate într-un al doilea sir, caracterele găsite fiind înlocuite de cele specificate într-un al trei-lea sir.

Exemplu de utilizare a funcției LENGTH:

```
SQL> SELECT LENGTH (nume) FROM profesor;
```

LENGTH (NUME)

9
5
9
7
4
6
8

Dintre cele mai importante funcții amintim:

- ROUND - rotunjește valorile la un anumit număr de poziții zecimale;
- TRUNC - trunchiază valorile la un anumit număr de poziții zecimale;
- CEIL - returnează cel mai mic întreg mai mare sau egal cu o anumită valoare;
- FLOOR - returnează cel mai mare întreg mai mic sau egal cu o anumită valoare;
- SIGN - returnează valoarea -1 dacă valoarea argumentului primit este mai mică decât 0, 1 dacă valoarea argumentului primit este mai mare decât 0 și 0 dacă valoarea argumentului primit este egală cu 0;
- SQRT - returnează rădâcina pătrată a argumentului primit;
- ABS - returnează valoarea absolută a argumentului primit;
- POWER - returnează valoarea unui număr ridicat la o anumită putere;
- MOD - returnează restul împărțirii a două numere;
- alte funcții matematice cum ar fi: LOG, SIN, TAN, COS, EXP, LN.

Funcții pentru dată calendaristică și oră

În Oracle datele de tip dată calendaristică sunt reprezentate sub un format numeric reprezentând: ziua, luna, anul, ora, minutul, secunda și secolul. Oracle poate manevra date calendaristice de la 1 ianuarie 4712 î.Cr până la 31 decembrie 4712 d.Cr. Modul implicit de afișare și introducere este sub forma: DD-MON-YY (ex. '31-Dec-99'). Aceasta categorie de funcții operează pe valori de tip dată calendaristică, rezultatul returnat fiind tot de tip dată calendaristică, excepție făcând funcția MONTHS_BETWEEN care returnează o valoare numerică.

Cele mai des întâlnite funcții sunt:

- ADD_MONTH - returnează o dată calendaristică formată prin adăugarea la data calendaristică specificată a unui anumit număr de luni;
- LAST_DAY - întoarce ca rezultat ultima zi a unei luni specificate;
- MONTHS_BETWEEN - returnează numărul de luni dintre două date calendaristice specificate;
- NEXT_DAY - returnează prima dată calendaristică ulterioară datei calendaristice specificate;
- SYSDATE - întoarce ca rezultat data calendaristică a sistemului.

Asupra datelor calendaristice se pot realiza operații aritmetice, cum ar fi scăderea sau adunarea, modul lor de funcționare fiind ilustrat în tabelul de mai jos:

Tip operand	Operatie	Tip operand	Tip rezultat	Descriere
data	+/-	număr	data	Adaugă/scade un număr de zile la o dată calendaristică
data	+/-	număr/24	data	Adaugă/scade un număr de ore la o dată calendaristică
data	+/-	număr/1440	data	Adaugă/scade un număr de minute la o dată calendaristică
data	+/-	număr/86400	data	Adaugă/scade un număr de secunde la o dată calendaristică

Tip operand	Operație	Tip operand	Tip rezultat	Descriere
data	-	data	număr zile	Scade două date calendaristice rezultând diferența în număr de zile. Dacă al doilea operand este mai mare decât primul numărul de zile rezultat este reprezentat de o valoare negativă.

De asemenea mai există funcțiile ROUND și TRUNC care rotunjesc, respectiv trunchiază data calendaristică. Aceste funcții sunt foarte folositoare atunci când se dorește compararea datelor calendaristice care au ora diferită. Exemplul următor rotunjește data de naștere a cadrelor didactice în funcție de an:

```
SQL> SELECT ROUND (data_nast, 'YEAR') "DATA"
      FROM profesor;
```

```
DATA
-----
01-JAN-47
01-JAN-45
01-JAN-52

01-JAN-66
01-JAN-70
```

Funcții de conversie

În general expresiile nu pot conține valori aparținând unor tipuri de date diferite. De exemplu, nu se poate înmulții 3 cu 7 și apoi aduna "ION". Prin urmare se realizează anumite conversii, care pot fi implicate sau explicate.

Conversiile implicate se realizează în următoarele cazuri:

- atribuirile de valori unei coloane (folosind comenzi INSERT sau UPDATE) sau atribuirilor de valori unor argumente ale unei funcții;
- evaluări de expresii.

Pentru atribuirile, programul Oracle efectuează în mod implicit următoarele conversii de tip:

- VARCHAR2 sau CHAR la NUMBER
- VARCHAR2 sau CHAR la DATE
- VARCHAR2 sau CHAR la ROWID
- NUMBER, DATE sau ROWID la VARCHAR2

Conversia la atribuire reușește în cazul în care Oracle poate converti tipul valorii atribuite la tipul destinației atribuirii.

Pentru evaluarea expresiilor, se realizează în mod implicit următoarele conversii de tip:

- VARCHAR2 sau CHAR la NUMBER
- VARCHAR2 sau CHAR la DATE
- VARCHAR2 sau CHAR la ROWID

De exemplu, pentru următoarea interogare se realizează conversia în mod implicit a constantei de tip CHAR, '10', la tipul NUMBER.

```
SQL> SELECT salariu + '10' FROM profesor;
```

```
SALARIU+'10'
```

```
-----
3010
2510
2810
1510
2210
1210
1210
```

Pentru conversiile explicite de tip, SQL pune la dispoziție mai multe funcții de conversie, de la un anumit tip de date la altul, după cum este arătat în tabelul de mai jos.

Tip convertit	CHAR	NUMBER	DATE	RAW	ROWID
Tip inițial					
CHAR	-	TO_NUMBER	TO_DATE	HEXTORAW	CHARTOROWID
NUMBER	TO_CHAR	-	TO_DATE(nr,'J')		
DATE	TO_CHAR	TO_DATE(date,'J')	-		
RAW	RAWTOHEX			-	
ROWID	RAWIDTOCHAR			-	

Cele mai uzuale funcții sunt:

- TO_CHAR - convertește un număr sau o dată calendaristică într-un sir de caractere;
- TO_NUMBER - convertește un sir de caractere alcătuit din cifre într-o valoare numerică;
- TO_DATE - convertește un sir de caractere sau un număr ce reprezintă o dată calendaristică la o valoare de tip dată calendaristică. De asemenea poate converti o dată calendaristică la un număr ce reprezintă data calendaristică Juliană.

Pentru a realiza conversia acestei funcții folosesc anumite păși de format. Pentru mai multe amănunte vezi Anexa 6.

Următorul exemplu va prelua data și ora curentă a sistemului din funcția SYSDATE și o va formața într-o dată scrisă pe litere ce va conține și ora în minute și secunde:

```
SQL> SELECT TO_CHAR(SYSDATE, 'DD MONTH YYYY HH24:MI:SS')
      data
      FROM dual;
```

```
DATA
```

```
-----
17 MAY 2000 17:03:38
```

Functii diverse

Acestea sunt în general funcții care acceptă ca argumente orice tip de dată.

Cele mai utilizate sunt:

- DECODE - Aceasta este una dintre cele mai puternice funcții SQL. Ea practic facilitează interogările condiționate, acționând ca o comandă 'if-then-else' sau 'case' dintr-un limbaj procedural. Pentru fiecare înregistrare se va evalua valoarea din coloana testată și se va compara pe rând cu fiecare valoare declarată în cadrul funcției. Dacă se găsesc valori egale, atunci funcția va returna o valoare aferentă acestei egalități, declarată tot în cadrul funcției. Se poate specifica că, în cazul în care nu se găsesc valori egale, funcția să întoarcă o anumită valoare. Dacă acest lucru nu se specifică funcția va întoarce valoarea Null.
- GREATEST - returnează cea mai mare valoare dintr-o listă de valori;
- LEAST - returnează cea mai mică valoare dintr-o listă de valori;
- VSIZE - returnează numărul de bytes pe care este reprezentată intern o anumită coloană;
- USER - returnează numele utilizatorului curent al bazei de date;
- DUMP - returnează o valoare ce conține codul tipului de date, lungimea în bytes, precum și reprezentarea internă a unei expresii.

Exemplul următor utilizează funcția DECODE pentru a returna o creștere a salariului cadrelor didactice cu grad de profesor, conferențiar și lector, restul salariilor rămânând nemodificate:

```
SQL> SELECT nume, grad, salariu,
      DECODE (grad, 'PROF', salariu*1.2,
              'CONF', salariu*1.15, 'LECT',
              salariu*1.1, salariu) "Salariu modificat"
    FROM profesor;
```

NUME	GRAD	SALARIU	Salariu modificat
GHEORGHIU	PROF	3000	3600
MARIN	PROF	2500	3000
GEORGESCU	CONF	2800	3220
IONESCU	ASIST	1500	1500
ALBU	LECT	2200	2420
VOINEA	ASIST	1200	1200
STANESCU	ASIST	1200	1200

8.1.14. Functii referitoare la mai multe înregistrări

Aceste funcții se mai numesc și funcții totalizatoare sau funcții de grup. Spre deosebire de funcțiile referitoare la o singură înregistrare, funcțiile de grup operează pe un set de mai multe înregistrări și returnează un singur rezultat pentru fiecare grup. Dacă nu este utilizată clauza GROUP BY, ce grupează înregistrările după un anumit criteriu, tabela este considerată ca un singur grup și se va returna un singur rezultat.

- COUNT - determină numărul de înregistrări care îndeplinesc o anumită condiție;
- MAX - determină cea mai mare valoare dintr-o coloană;
- MIN - determină cea mai mică valoare dintr-o coloană;
- SUM - returnează suma tuturor valorilor dintr-o coloană;
- AVG - calculează valoarea medie a unei coloane;
- STDDEV - determină abaterea sau deviația standard a unei coloane numerice;
- VARIANCE - returnează dispersia, adică pătratul unei deviații standard pentru o coloană numerică.

De exemplu:

```
SQL> SELECT MIN(salariu), MAX(salariu), AVG(salariu),
      COUNT(*)
      FROM profesor;
```

MIN(SALARIU)	MAX(SALARIU)	AVG(SALARIU)	COUNT (*)
1200	3000	2057.1429	7

Toate funcțiile de mai sus, cu excepția funcției COUNT, operează asupra unei coloane sau unei expresii, care este specificată ca parametru al funcției. În cazul funcției COUNT, argumentul acestuia nu contează, de obicei utilizându-se ca argument simbolul *.

Notă: Toate funcțiile de mai sus ignoră valorile Null, exceptie făcând funcția COUNT. Pentru a include în calcule și înregistrările cu valoarea Null se poate folosi funcția NVL.

Dacă nu este utilizată clauza GROUP BY, în lista de atrbute ale comenzi SELECT nu pot apărea funcții de grup alături de nume de coloane sau alte expresii care iau valori pentru fiecare înregistrare în parte. De exemplu, următoarea interogare va genera o eroare:

```
SQL> SELECT nume, MIN(salariu) FROM profesor;
ERROR at line 1:
ORA-00937: not a single-group group function
```

8.1.15 Pseudo-coloana ROWNUM

ROWNUM este o pseudo-coloană care numerotează rândurile selectate de o interogare. Astfel, pentru primul rând selectat pseudo-coloana ROWNUM are valoarea 1, pentru al doilea rând are valoarea 2, și.a.m.d. De exemplu, pentru a limita rândurile selectate de o interogare la maxim 5, se folosește următoarea comandă:

```
SELECT *
FROM profesor
WHERE ROWNUM<6;
```

Deoarece pseudo-coloana ROWNUM numerotează rândurile selectate, valorile sale vor trebui să înceapă tot timpul cu 1. Deci dacă în exemplul de mai sus condiția ar fi ROWNUM>6 sau ROWNUM=6 interogarea nu ar selecta nici un rând deoarece în acest caz condiția ar fi întotdeauna falsă. Pentru primul rând accesat de interogare ROWNUM va avea valoarea 1, condiția nu este îndeplinită și deci rândul nu va fi selectat. Pentru al doilea rând accesat de interogare ROWNUM va avea din nou valoarea 1, condiția nu este îndeplinită nici în acest caz și deci nici acest rând nu va fi selectat, s.a.m.d. Prin urmare nici unul din rândurile accesate nu vor satisface condiția conținută de interogare.

Pseudo-coloana ROWNUM se poate folosi atât în condiția unor comenzi UPDATE sau DELETE cât și pentru a atribui valori unice unei coloane, ca în exemplul de mai jos:

```
UPDATE vanzari
SET cod = ROWNUM;
```

Valoarea pseudo-coloanei ROWNUM este atribuită unui rând înainte ca acesta să fie sortat datorită unei clauze ORDER BY, de aceea valorile pseudo-coloanei nu reprezintă ordinea de sortare. De exemplu>

```
SQL> SELECT nume, prenume, ROWNUM FROM profesor
      ORDER BY salariu;
```

NUME	PRENUME	ROWNUM
VOINEA	MIRCEA	6
STANESCU	MARIA	7
IONESCU	VERONICA	4
ALBU	GHEORGHE	5
MARIN	VLAD	2
GEORGESCU	CRISTIANA	3
GHEORGHIU	STEFAN	1

8.1.16. Clauza GROUP BY

Clauza GROUP BY este utilizată pentru a împărți din punct de vedere logic un tabel în grupuri de înregistrări. Fiecare grup este format din toate înregistrările care au aceeași valoare în câmpul sau grupul de câmpuri specificate în clauza GROUP BY. Unele înregistrări pot fi excluse folosind clauza WHERE înainte ca tabelul să fie împărțit în grupuri.

Clauza GROUP BY se folosește de obicei împreună cu funcțiile de grup, acestea returnând valoarea calculată pentru fiecare grup în parte. În cazul folosirii clauzei GROUP BY, toate expresiile care apar în lista atributelor comenzi SELECT trebuie să aibă o valoare unică pentru fiecare grup, de aceea orice coloană sau expresie din această listă care nu este o funcție de grup trebuie să apară în clauza GROUP BY.

Prezentăm în continuare câteva exemple:

```
SQL> SELECT grad, AVG (salariu) FROM profesor
      GROUP BY grad;
```

GRAD	AVG (SALARIU)
ASIST	1300
CONF	2800
LECT	2200
PROF	2750

```
SQL> SELECT grad, MAX(salariu) FROM profesor
      WHERE prima IS NOT NULL GROUP BY grad;
```

GRAD	MAX (SALARIU)
ASIST	1200
CONF	2800
LECT	2200
PROF	3000

Următoarea interogare va genera o eroare deoarece în lista atributelor comenții SELECT există o coloană (nume) care nu apare în clauza GROUP BY:

```
SQL> SELECT nume, MIN(salariu) FROM profesor
      GROUP BY grad;
```

```
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

Comanda de mai sus este invalidă deoarece coloana nume are valori individuale pentru fiecare înregistrare, în timp ce MIN (salariu) are o singură valoare pentru un grup.

Clauza GROUP BY permite apelarea unei funcții de grup în altă funcție de grup. În exemplul următor, funcția AVG returnează salariul mediu pentru fiecare grad didactic, iar funcția MAX returnează maximul dintre aceste salarii medii.

```
SQL> SELECT MAX(AVG(salariu)) FROM profesor
      GROUP BY grad;
```

MAX (AVG (SALARIU))
2800

8.1.17. Clauza HAVING

Clauza HAVING este tot o clauză de filtrare ca și clauza WHERE. Totuși, în timp ce clauza WHERE determină ce înregistrări vor fi selecționate dintr-un tabel, clauza HAVING determină care dintre grupurile rezultate vor fi afișate după ce înregistrările din tabel au fost grupate cu clauza GROUP BY. Cu alte cuvinte, pentru a exclude grupuri de înregistrări se folosește clauza HAVING iar pentru a exclude înregistrări individuale se folosește clauza WHERE.

Clauza HAVING este folosită numai dacă este folosită și clauza GROUP BY. Expresiile folosite într-o clauză HAVING trebuie să aibă o singură valoare pe grup.

Atunci când se folosește clauza GROUP BY, clauza WHERE se utilizează pentru eliminarea înregistrărilor ce nu se doresc a fi grupate. Astfel, următoarea interogare este invalidă deoarece clauza WHERE încearcă să excludă grupuri de înregistrări și nu anumite înregistrări:

```
SQL> SELECT grad, AVG(salariu) FROM profesor
      WHERE AVG(salariu)>2000
        GROUP BY grad;
```

```
ERROR:
ORA-00934: group function is not allowed here
```

Pentru a exclude gradul didactic pentru care media de salariu nu este mai mare decât 2000 se folosește următoarea comandă SELECT cu clauza HAVING:

```
SQL> SELECT grad, AVG(salariu) FROM profesor
      GROUP BY grad
        HAVING AVG(salariu)>2000;
```

GRAD	AVG(SALARIU)
CONF	2800
LECT	2200
PROF	2750

Următoarea interogare exclude întâi cadrele didactice care nu au salariu mai mare decât 2500 după care exclude gradul didactic pentru care media de salariu nu este mai mare decât 2800.

```
SQL> SELECT grad, AVG(salariu) FROM profesor
      WHERE salariu > 2500
        GROUP BY grad
          HAVING AVG(salariu) > 2800;
```

GRAD	AVG(SALARIU)
PROF	3000

8.1.18. Regăsirea datelor din două sau mai multe tabele

O joncțiune este o interogare care regăsește înregistrări din două sau mai multe tabele. Capacitatea de a realiza o joncțiune între două sau mai multe tabele reprezintă una dintre cele mai puternice facilități ale unui sistem relațional. Legătura dintre înregistrările tabelelor se realizează prin existența unor câmpuri comune caracterizate prin domenii de definiție compatibile (chei primare sau străine). Pentru realizarea unei joncțiuni se folosesc comanda SELECT, precizând în clauza FROM numele tabelelor utilizate, iar în clauza WHERE criteriul de compunere.

Produsul a două sau mai multe tabele.

În cazul în care în interogare se specifică mai multe tabele și nu este inclusă o clauză WHERE, interogarea va genera produsul cartezian al tabelelor. Acesta va conține toate combinațiile posibile de înregistrări din tabelele componente. Astfel, produsul cartezian a două tabele care conțin 100, respectiv 50 de înregistrări va avea dimensiunea de 5.000 de înregistrări.

De exemplu, să considerăm tabela catedra cu următoarele 4 înregistrări:

COD_CATEDRA	NUME	PROFIL
10	INFORMATICA	TEHNIC
20	ELECTRONICA	TENHIC
30	AUTOMATICA	TENHIC
40	FINANTE	ECONOMIC

Atunci următoarea interogare va genera produsul cartezian al tabelelor, adică va avea ca rezultat $7 \times 4 = 28$ de rânduri ce vor conține toate combinațiile posibile de înregistrări din cele două tabele:

```
SQL> SELECT * FROM profesor, catedra;
```

Dacă în lista de atribuite ale comenzii SELECT sunt specificate coloanele selectate, atunci numele acestora trebuie să fie unic în cadrul tuturor tabelelor. Dacă există un nume de coloană care apare în mai mult de un tabel, atunci, pentru evitarea ambiguității, trebuie specificat și tabelul din care face parte coloana în cauză. De exemplu, în următoarea interogare pentru coloanele cod_catedra și nume trebuie specificate tabelele din care fac parte:

```
SQL> SELECT profesor.nume, prenume, catedra.cod_catedra,  
          catedra.nume  
     FROM profesor, catedra;
```

NUME	PRENUME	COD_CATEDRA	NUME
GHEORGHIU	STEFAN	10	INFORMATICA
MARIN	VLAD	10	INFORMATICA
GEORGESCU	CRISTIANA	10	INFORMATICA
IONESCU	VERONICA	10	INFORMATICA
ALBU	GHEORGHE	10	INFORMATICA
VOINEA	MIRCEA	10	INFORMATICA
STANESCU	MARIA	10	INFORMATICA
GHEORGHIU	STEFAN	20	ELECTRONICA
MARIN	VLAD	20	ELECTRONICA
GEORGESCU	CRISTIANA	20	ELECTRONICA
IONESCU	VERONICA	20	ELECTRONICA
ALBU	GHEORGHE	20	ELECTRONICA
VOINEA	MIRCEA	20	ELECTRONICA
STANESCU	MARIA	20	ELECTRONICA
GHEORGHIU	STEFAN	30	AUTOMATICA

MARIN	VLAD	30	AUTOMATICA
GEORGESCU	CRISTIANA	30	AUTOMATICA
IONESCU	VERONICA	30	AUTOMATICA
ALBU	GHEORGHE	30	AUTOMATICA
VOINEA	MIRCEA	30	AUTOMATICA
STANESCU	MARIA	30	AUTOMATICA
GHEORGHIU	STEFAN	40	FINANTE
MARIN	VLAD	40	FINANTE
GEORGESCU	CRISTIANA	40	FINANTE
IONESCU	VERONICA	40	FINANTE
ALBU	GHEORGHE	40	FINANTE
VOINEA	MIRCEA	40	FINANTE
STANESCU	MARIA	40	FINANTE

În general, pentru a scurta textul comenzi, în astfel de cazuri se folosesc de obicei alias-uri pentru numele tabelelor, care pot fi folosite în interogare. Astfel interogarea de mai sus se poate scrie:

```
SQL> SELECT p.nume, prenume, c.cod_catedra, c.nume
      FROM profesor p, catedra c;
```

În general, produsul cartezian este rar folosit, având o utilitate practică redusă.

Joncțiuni

Pentru a realiza o joncțiune între două sau mai multe tabele se utilizează clauza WHERE a interogărilor pe aceste tabele. În funcție de criteriul de compunere, se disting mai multe tipuri de joncțiuni:

1. joncțiuni echivalente (EQUI-JOIN) sau joncțiuni interne (INNER JOIN)
2. joncțiuni neechivalente
3. joncțiuni externe (OUTER JOIN)
4. auto-joncțiuni

1. Joncțiunile echivalente

O echijoncțiune conține operatorul egalitate (=) în clauza WHERE, combinând înregistrările din tabele care au valori egale pentru coloanele specificate.

De exemplu, pentru a afișa cadrele didactice și numele catedrei din care acestea fac parte se combină înregistrările din cele două tabele pentru care codul catedrei este același.

```
SQL> SELECT p.nume, p.prenume, c.nume
      FROM profesor p, catedra c
     WHERE p.cod_catedra=c.cod_catedra;
```

NUME	PRENUME	NUME
GHEORGHIU	STEFAN	INFORMATICA
IONESCU	VERONICA	INFORMATICA
VOINEA	MIRCEA	INFORMATICA

MARIN	VLAD	ELECTRONICA
STANESCU	MARIA	ELECTRONICA
ALBU	GHEORGHE	ELECTRONICA
GEORGESCU	CRISTIANA	AUTOMATICA

2. Juncțiuni neechivalente

Juncțiunile neechivalente sunt aceleia care nu folosesc în clauza WHERE operatorul egal. Operatorii cei mai utilizați în cazul juncțiunilor neechivalente sunt: <, >, <=, >=, <>, BETWEEN...AND....

Pentru a exemplifica un astfel de tip de juncție considerăm tabela gradsal ce conține pragul minim și pragul maxim al salariului dintr-un anumit grad de salarizare:

GRAD_SALARIZARE	PRAG_MIN	PRAG_MAX
1	500	1500
2	1501	2000
3	2001	2500
4	2501	3500
5	3501	10000

Evident, între tabelele profesor și gradsal nu are sens definirea unei juncțiuni echivalente deoarece nu există o coloană din tabela profesor căreia să-i corespundă o coloană din tabela gradsal. Exemplul următor ilustrează definirea unei juncțiuni neechivalente care evaluează gradul de salarizare a cadrelor didactice, prin încadrarea salariului acestora într-un interval stabilit de pragul minim și pragul maxim:

```
SQL> SELECT p.nume, p.grad, p.salariu, g.grad_salarizare
      FROM profesor p, gradsal g
     WHERE p.salariu BETWEEN g.prag_min AND g.prag_max;
```

NUME	GRAD	SALARIU	GRAD_SALARIZARE
IONESCU	ASIST	1500	1
VOINEA	ASIST	1200	1
STANESCU	ASIST	1200	1
MARIN	PROF	2500	3
ALBU	LECT	2200	3
GHEORGHIU	PROF	3000	4
GEORGESCU	CONF	2800	4

3. Juncțiuni externe

Dacă într-o juncție de tipul celor prezentate până acum una sau mai multe înregistrări nu satisfac condiția de compunere specificată în clauza WHERE, atunci ele nu vor apărea în rezultatul interogării. Aceste înregistrări pot apărea însă dacă se folosesc juncțiunea externă. Juncțiunea externă returnează toate înregistrările care satisfac condiția de juncție plus

acele înregistrări dintr-un tabel ale căror valori din coloanele după care se face legătura nu se regăsesc în coloanele corespunzătoare ale unei înregistrări din celalalt tabel.

Pentru a realiza o joncție externă între tabelele A și B ce returnează toate înregistrările din tabela A se utilizează semnul (+) în dreapta tabelului B. Pentru fiecare înregistrare din tabela A care nu satisface condiția de compunere pentru nici o înregistrare din tabela B, se va crea în tabela B o înregistrare nulă care va fi compusă cu înregistrarea din tabela A. Invers, pentru a realiza o joncție externă între tabelele A și B ce returnează toate înregistrările din tabela B, se utilizează semnul (+) în dreapta tabelului A.

În interogarea utilizată pentru a exemplifica joncțunea echivalentă, se observă că au fost selectate numai catedrele în care există cadre didactice. Pentru a afișa toate catedrele, indiferent dacă ele cuprind sau nu cadre didactice, se folosește următoarea interogare:

```
SQL> SELECT p.nume, p.prenume, c.nume
      FROM profesor p, catedra c
     WHERE p.cod_catedra(+) = c.cod_catedra;
```

NUME	PRENUME	NUME
GHEORGHIU	STEFAN	INFORMATICA
IONESCU	VERONICA	INFORMATICA
VOINEA	MIRCEA	INFORMATICA
MARIN	VLAD	ELECTRONICA
STANESCU	MARIA	ELECTRONICA
ALBU	GHEORGHE	ELECTRONICA
GEORGESCU	CRISTIANA	AUTOMATICA FINANTE

Se observă că ultima înregistrare (ce corespunde catedrei de finanțe care nu are în componență nici un cadru didactic) va avea coloanele corespunzătoare primului tabel completate cu Null.

Folosirea operatorului de joncție externă are următoarele restricții:

1. Operatorul (+) poate fi plasat în oricare parte a condiției din clauza WHERE, însă nu în ambele părți. Tabelul de partea căruia este amplasat acest operator va crea înregistrări nule care vor fi compuse cu înregistrările din celalalt tabel care nu satisfac condiția de compunere.
2. Dacă tabelele A și B au condiții multiple de joncție, atunci operatorul (+) trebuie utilizat în toate aceste condiții.
3. Într-o singură interogare nu se poate realiza o joncție externă a unui tabel cu mai multe tabele.
4. O condiție care conține operatorul (+) nu poate fi combinată cu o altă condiție ce utilizează operatorul IN.
5. O condiție care conține operatorul (+) nu poate fi combinată cu o altă condiție prin operatorul OR.

4. Auto-joncțiuni

Auto-joncțiunea reprezintă joncțiunea unui tabel cu el însuși. Pentru ca rândurile dintr-un tabel să poată fi compuse cu rânduri din același tabel, în clauza FROM a interogării numele tabelului va apărea de înai multe ori, urmat de fiecare dată de un alias.

De exemplu, pentru a selecta toate cadrele didactice care au un şef direct și numele acestui şef se folosește următoarea auto-joncțiune:

```
SQL> SELECT p.nume, p.prenume, s.nume, s.prenume
      FROM profesor p, profesor s
     WHERE p.sef=s.cod;
```

NUME	PRENUME	NUME	PRENUME
MARIN	VLAD	GHEORGHIU	STEFAN
GEORGESCU	CRISTIANA	GHEORGHIU	STEFAN
ALBU	GHEORGHE	GHEORGHIU	STEFAN
VOINEA	MIRCEA	GHEORGHIU	STEFAN
IONESCU	VERONICA	GEORGESCU	CRISTIANA
STANESCU	MARIA	IONESCU	VERONICA

Auto-joncțiunea poate fi folosită și pentru verificarea corectitudinii interne a datelor. De exemplu, este puțin probabil să existe două cadre didactice care au cod diferit dar în schimb au același nume, prenume și dată de naștere. Pentru a verifica dacă există astfel de înregistrări se folosește interogarea:

```
SQL> SELECT a.nume, a.prenume
      FROM profesor a, profesor b
     WHERE a.nume=b.nume AND a.prenume=b.prenume
       AND a.data_nast=b.data_nast AND a.cod<>b.cod;
```

8.1.19. Operatorii pentru mulțimi

Operatorii de mulțimi combină două sau mai multe interogări, efectuând operații specifice mulțimilor: reunire, intersecție, diferență. Acești operatori se mai numesc și *operatori verticali* deoarece combinarea celor două interogări se face coloană cu coloană. Din acest motiv, numărul total de coloane și tipurile de date ale coloanelor corespondente din cele două interogări trebuie să coincidă.

Există următorii operatori pentru mulțimi:

1. UNION – Returnează rezultatele a două sau mai multe interogări eliminând toate înregistrările dupicate;
2. UNION ALL – Returnează rezultatele a două sau mai multe interogări incluzând înregistrările dupicate;
3. INTERSECT – Returnează toate înregistrările distincte găsite în ambele interogări;
4. MINUS – Returnează toate înregistrările distincte care se găsesc în prima interogare dar nu și în a doua interogare.

Să considerăm de exemplu următoarele interogări:

```
SQL> SELECT grad, salariu FROM profesor
      WHERE cod_catedra = 10;
```

GRAD	SALARIU
PROF	3000
ASIST	1500
ASIST	1200

```
SQL> SELECT grad, salariu FROM profesor
      WHERE cod_catedra = 20;
```

GRAD	SALARIU
PROF	2500
LECT	2200
ASIST	1200

În continuare exemplificăm fiecare dintre operatorii pentru mulțimi aplicați acestor interogări:

```
SQL> SELECT grad, salariu FROM profesor
      WHERE cod_catedra = 10
      UNION
      SELECT grad, salariu FROM profesor
      WHERE cod_catedra = 20;
```

GRAD	SALARIU
ASIST	1200
ASIST	1500
LECT	2200
PROF	2500
PROF	3000

```
SQL> SELECT grad, salariu FROM profesor
      WHERE cod_catedra = 10
      UNION ALL
      SELECT grad, salariu FROM profesor
      WHERE cod_catedra = 20;
```

GRAD	SALARIU
PROF	3000
ASIST	1500
ASIST	1200
PROF	2500
LECT	2200
ASIST	1200

```
SQL> SELECT grad, salariu FROM profesor
  WHERE cod_catedra = 10
INTERSECT
SELECT grad, salariu FROM profesor
  WHERE cod_catedra = 20;
```

GRAD	SALARIU
ASIST	1200

```
SQL> SELECT grad, salariu FROM profesor
  WHERE cod_catedra = 10
MINUS
SELECT grad, salariu FROM profesor
  WHERE cod_catedra = 20;
```

GRAD	SALARIU
ASIST	1500
PROF	3000

Există următoarele reguli de folosire a operatorilor pentru mulțimi:

- interogările trebuie să conțină același număr de coloane;
- coloanele corespondente trebuie să aibă același tip de date;
- în rezultat vor apărea numele coloanelor din prima interogare, nu cele din a doua interogare chiar dacă aceasta folosește alias-uri; de exemplu:

```
SQL> SELECT cod FROM profesor
  MINUS
    SELECT sef FROM profesor;
```

COD
101
104
105
106

- clauza ORDER BY poate fi folosită o singură dată într-o interogare care folosește operatori de mulțimi; atunci când se folosește, ea trebuie poziționată la sfârșitul comenzi; de exemplu:

```
SQL> SELECT grad, salariu FROM profesor,
  WHERE cod_catedra = 10
UNION
SELECT grad, salariu FROM profesor
  WHERE cod_catedra = 20
ORDER BY Z;
```

GRAD	SALARIU
ASIST	1200
ASIST	1500
LECT	2200
PROF	2500
PROF	3000

- operatorii pentru mulțimi pot fi utilizati în subinterrogări;
- pentru a modifica ordinea de execuție este posibilă utilizarea parantezelor; de exemplu:

```
SQL> SELECT grad FROM profesor WHERE cod_catedra = 10
      INTERSECT
      SELECT grad FROM profesor WHERE cod_catedra = 20
      UNION
      SELECT grad FROM profesor WHERE cod_catedra = 30;
```

```
GRAD
-----
ASIST
CONF
PROF
```

```
SQL> SELECT grad FROM profesor WHERE cod_catedra = 10
      INTERSECT
      (SELECT grad FROM profesor WHERE cod_catedra = 20
      UNION
      SELECT grad FROM profesor WHERE cod_catedra = 30);
```

```
GRAD
-----
ASIST
PROF
```

8.1.20. Subinterrogări și operatorii ANY, ALL, EXISTS

O *subinterrogare* este o comandă SELECT inclusă în altă comandă SELECT. Rezultatele subinterrogării sunt transmise celeilalte interrogări și pot apărea în cadrul clauzelor WHERE, HAVING sau FROM. Subinterrogările sunt utile pentru a scrie interrogări bazate pe o condiție în care valoarea de comparație este necunoscută. Această valoare poate fi afișată folosind o subinterrogare. De exemplu:

```
SELECT coloane
      FROM tabel
     WHERE coloana=(SELECT coloane
                      FROM tabel
                     WHERE condiție)
```

Subinterrogarea, denumită și *interrogare interioară (inner query)*, generează valorile pentru condiția de căutare a instrucțiunii SELECT care o conține, denumită *interrogare exterioară (outer query)*. Instrucțiunea SELECT exterioară depinde de valorile generate de către interrogarea interioară. În general, interrogarea interioară se execută prima și rezultatul acesta este utilizat în interrogarea exterioară. Rezultatul interrogării exterioare depinde de numărul valorilor returnate de către interrogarea interioară. În acest sens, putem distinge:

- Subinterrogări care returnează un singur rând;*
- Subinterrogări care returnează mai multe rânduri.*

Din punct de vedere al ordinii de evaluare a interogărilor putem clasifica subinterrogările în:

1. *Subinterrogări simple* – în care interogarea interioară este evaluată prima, independent de interogarea exterioară (interogarea interioară se execută o singură dată);
2. *Subinterrogări corelate* – în care valorile returnate de interogarea interioară depind de valorile returnate de interogarea exterioară (interogarea interioară este evaluată pentru fiecare înregistrare a interogării exterioare).

Subinterrogările sunt îndeosebi utilizate atunci când se dorește ca o interogare să regăsească înregistrări dintr-o tabelă care îndeplinește o condiție ce depinde la rândul ei de valori din aceeași tabelă.

Notă: Clauza ORDER BY nu poate fi utilizată într-o subinterrogare. Regula este că poate exista doar o singură clauză ORDER BY pentru o comandă SELECT și, dacă este specificată, trebuie să fie ultima clauză din comanda SELECT. Prin urmare, clauza ORDER BY nu poate fi specificată decât în interogarea cea mai din exterior.

Subinterrogări care returnează un singur rând

În acest caz condiția, din clauza WHERE sau HAVING a interogării exterioare utilizează operatorii: =, <, <=, >, >=, <> care operează asupra unei subinterrogări ce returnează o singură valoare. Interogarea interioară poate conține condiții complexe formate prin utilizarea condițiilor multiple de interogare cu ajutorul operatorilor AND și OR sau prin utilizarea funcțiilor agregat.

Următoarea interogare selectează cadrele didactice care au salariul cel mai mic. Salariul minim este determinat de o subinterrogare ce returnează o singură valoare.

```
SQL> SELECT nume, prenume, salariu FROM profesor
      WHERE salariu = (SELECT MIN(salariu) FROM profesor);
```

NUME	PRENUME	SALARIU
VOINEA	MIRCEA	1200
STANESCU	MARIA	1200

Procesul de evaluare al acestei interogări se desfășoară astfel:

- Se evaluatează în primul rând interogarea interioară:
Valoarea obținută este $\text{MIN}(\text{salariu})=1200$
- Rezultatul evaluării interogării interioare devine condiție de căutare pentru interogarea exterioară și anume:

```
SELECT nume, prenume, salariu FROM profesor
WHERE salariu =1200;
```

În cazul în care interogarea interioară nu întoarce nici o înregistrare, interogarea exterioară nu va selecta la rândul ei nici o înregistrare.

Notă: Dacă care se utilizează operatorii: `=, <, <=, >, >=, <>` în condiția interogării exterioare, atunci interogarea interioară trebuie în mod obligatoriu să returneze o singură valoare. În caz contrar va apărea un mesaj de eroare, ca în exemplul următor:

```
SQL> SELECT nume, prenume, salariu
   FROM profesor
 WHERE salariu=(SELECT MIN (salariu)FROM profesor
 GROUP BY grad);
```

ERROR:

ORA-01427: single-row subquery returns more than one row

Subinterrogările pot fi folosite nu numai în clauza WHERE a interogării exterioare, ci și în clauza HAVING. Următoarea interogare afișează toate gradele didactice pentru care salariul minim este mai mare decât salariul mediu al tuturor cadrelor didactice.

```
SQL> SELECT grad FROM profesor
      GROUP BY grad
      HAVING min(salariu)>(SELECT avg(salariu)
                                FROM profesor);
```

GRAD

CONF
LECT
PROF

Subinterrogări care returnează mai multe rânduri

În cazul când interogarea întoarce mai multe rânduri nu mai este posibilă folosirea operatorilor de comparație. În locul acestora se folosește operatorul IN, care așteaptă o listă de valori și nu doar una.

Următoarea interogare selectează pentru fiecare grad didactic acele persoane care au salariul minim. Salariul minim pentru fiecare grad didactic este aflat printr-o subinterrogare, care, evident, va întoarce mai multe rânduri:

```
SQL> SELECT nume, salariu, grad FROM profesor
      WHERE (salariu, grad) IN
            (SELECT MIN (salariu), grad
              FROM profesor
              GROUP BY grad)
            ORDER BY salariu;
```

NUME	SALARIU	GRAD
VOINEA	1200	ASIST
STANESCU	1200	ASIST
ALBU	2200	LECT
MARIN	2500	PROF
GEORGESCU	2800	CONF

Notă: Spre deosebire de celelalte interogări de până acum, interogarea de mai sus compară perechi de coloane. În acest caz trebuie respectate următoarele reguli:

- coloanele din dreapta condiției de căutare sunt în paranteze și fiecare coloană este separată prin virgulă;
- coloanele returnate de interogarea interioară trebuie să se potrivească ca număr și tip cu coloanele cu care sunt comparate în interogarea exterioară; în plus, ele trebuie să fie în aceeași ordine cu coloanele cu care sunt comparate.

Alături de operatorul IN, o subinterrogare care returnează mai multe rânduri poate folosi operatorii ANY, ALL sau EXISTS. Operatorii ANY și ALL sunt prezentați în continuare, iar operatorul EXISTS va fi prezentat în secțiunea "Subinterrogări corelate".

Operatorii ANY și ALL sunt folosiți în mod obligatoriu în combinație cu operatorii relaționali =, !=, <, >, <=, >=; operatorii IN și EXISTS nu pot fi folosiți în combinație cu operatorii relaționali, dar pot fi utilizati cu operatorul NOT, pentru negarea expresiei.

Operatorul ANY

Operatorul ANY (sau sinonimul său SOME) este folosit pentru a compara o valoare cu *oricare* dintre valorile returnate de o subinterrogare. Pentru a înțelege modul de folosire a acestui operator să considerăm următorul exemplu ce afișează cadrele didactice ce câștigă mai mult decât profesorii care au cel mai mic salariu:

```
SQL> SELECT nume, salariu, grad FROM profesor
      WHERE salariu > ANY
            (SELECT DISTINCT salariu
              FROM profesor
             WHERE grad='PROF');
```

NUME	SALARIU	GRAD
GHEORGHIU	3000	PROF
GEORGESCU	2800	CONF

Interrogarea de mai sus este evaluată astfel: dacă salariul unui cadru didactic este mai mare decât cel puțin unul din salariile returnate de interogarea interioară, acea înregistrare este inclusă în rezultat. Cu alte cuvinte, >ANY însemnă mai mare decât minimul dintre valorile returnate de interogarea interioară, <ANY înseamnă mai mic ca maximul, iar =ANY este echivalent cu operatorul IN.

Notă: Opțiunea DISTINCT este folosită frecvent atunci când se folosesc operatorul ANY pentru a preveni selectarea de mai multe ori a unor înregistrări.

Operatorul ALL

Operatorul ALL este folosit pentru a compara o valoare cu *toate* valorile returnate de o subinterrogare. Considerăm următorul exemplu ce afișează cadrele didactice care câștigă mai mult decât asistenții cu salariul cel mai mare:

```
SQL> SELECT nume, salariu, grad FROM profesor
      WHERE salariu > ALL
            (SELECT DISTINCT salariu
              FROM profesor
             WHERE grad='ASIST');
```

NUME	SALARIU	GRAD
GHEORGHIU	3000	PROF
MARIN	2500	PROF
GEORGESCU	2800	CONF
ALBU	2200	LECT

Interrogarea de mai sus este evaluată astfel: dacă salariul unui cadru didactic este mai mare decât toate valorile returnate de interrogarea interioară, acea înregistrare este inclusă în rezultat. Cu alte cuvinte, `>ALL` înseamnă mai mare ca maximul dintre valorile returnate de interrogarea interioară iar `<ALL` înseamnă mai mic ca minimul dintre acestea.

Notă: Operatorul `ALL` nu poate fi utilizat cu operatorul `=` deoarece interrogarea nu va întoarce nici un rezultat cu excepția cazului în care toate valorile sunt egale, situație care nu ar avea sens.

Subinterrogările pot fi imbricate (utilizate cu alte subinterrogări) până la 255 de nivele, indiferent de numărul de valori returnate de fiecare subinterrogare. Pentru a selecta cadrele didactice care au salariul mai mare decât cel mai mare salariu al cadrelor didactice care aparțin catedrei de Electronică, vom folosi următoarea interrogare:

```
SQL> SELECT nume, prenume, salariu FROM profesor
      WHERE salariu >(SELECT MAX(salariu) FROM profesor
                        WHERE cod_catedra=
                            (SELECT cod_catedra
                              FROM catedra
                             WHERE nume='ELECTRONICĂ'));
```

NUME	PRENUME	SALARIU
GHEORGHIU	STEFAN	3000
GEORGESCU	CRISTIANA	2800

Subinterrogări corelate

În exemplele considerate până acum interrogarea interioară era evaluată prima, după care valoarea sau valorile rezultate erau utilizate de către interrogarea exterioară. Subinterrogările de acest tip sunt numite subinterrogări simple. O altă formă de subinterrogare o reprezintă *interrogarea corelată*, caz în care interrogarea exterioară transmite repetat căte o înregistrare pentru interrogarea interioară. Interrogarea interioară este evaluată de fiecare dată când este transmisă o înregistrare din interrogarea exterioară, care se mai numește și *înregistrare candidată*. Subinterrogarea corelată poate fi identificată prin faptul că interrogarea interioară nu se poate executa independent ci depinde de valoarea transmisă de către interrogarea exterioară. Dacă ambele interrogări accesează aceeași tabelă, trebuie să fie asigurate alias-uri pentru fiecare referire la tabela respectivă.

Subinterrogările corelate reprezintă o cale de a accesa fiecare înregistrare din tabel și de a compara anumite valori ale acesteia cu valori ce depind tot de ea.

Evaluarea unei subinterrogări corelate se execută în următorii pași:

1. Interrogarea exterioară trimite o înregistrare candidată către interrogarea interioară;
2. Interrogarea interioară se execută în funcție de valorile înregistrării candidate;
3. Valorile rezultate din interrogarea interioară sunt utilizate pentru a determina dacă înregistrarea candidată va fi sau nu inclusă în rezultat;
4. Se repetă procedeul începând cu pasul 1 până când nu mai există înregistrări candidate.

De exemplu pentru a regăsi cadrele didactice care câștigă mai mult decât salariul mediu din propria catedră, putem folosi următoarea interrogare corelată:

```
SQL> SELECT nume, prenume, salariu FROM profesor p
      WHERE salariu > (SELECT AVG(salariu) FROM profesor
      WHERE cod_catedra = p.cod_catedra);
```

NUME	PRENUME	SALARIU
GHEORGHIU	STEFAN	3000
MARIN	VLAD	2500
ALBU	GHEORGHE	2200

În exemplul de mai sus coloana interrogării exterioare care se folosește în interrogarea interioară este `p.cod_catedra`. Deoarece `p.cod_catedra` poate avea valoare diferită pentru fiecare înregistrare, interrogarea interioară se execută pentru fiecare înregistrare candidată transmisă de interrogarea exterioară.

Atunci când folosim subinterrogări corelate împreună cu clauza HAVING, coloanele utilizate în această clauză trebuie să se regăsească în clauza GROUP BY. În caz contrar, va fi generat un mesaj de eroare datorat faptului că nu se poate face comparație decât cu o expresie de grup. De exemplu, următoarea interrogare este corectă, ea selectând gradele didactice pentru care media salariului este mai mare decât maximul primei pentru același grad:

```
SQL> SELECT grad FROM profesor p
      GROUP BY grad
      HAVING AVG(salariu) > (SELECT MAX(prima) FROM profesor
      WHERE grad = p.grad);
```

Operatorul EXISTS

Operatorul EXISTS verifică dacă, pentru fiecare înregistrare transmisă de interrogarea exterioară, există sau nu înregistrări care satisfac condiția interrogării interioare, returnând interrogării exterioare valoarea True sau False. Cu alte cuvinte, operatorul EXISTS cere în mod obligatoriu corelarea interrogării interioare cu interrogarea exterioară. Datorită faptului că operatorul EXISTS verifică doar existența rândurilor selectate și nu ia în considerare numărul sau valorile atributelor selectate, în subinterrogare pot fi specificate orice număr de atribute; în particular, poate fi folosită o constantă și chiar simbolul * (desi acest lucru nu

este recomandabil din punct de vedere al eficienței). De altfel, EXISTS este singurul operator care permite acest lucru.

Următoarea interogare selectează toate cadrele didactice care au măcar un subordonat:

```
SQL> SELECT cod, nume, prenume, grad FROM profesor p
      WHERE EXISTS
            (SELECT '1' FROM profesor
             WHERE profesor.sef = p.cod)
                  ORDER BY cod;
```

COD	NUME	PRENUME	GRAD
100	GHEORGHIU	STEFAN	PROF
102	GEORGESCU	CRISTIANA	CONF
103	IONESCU	VERONICA	ASIST

La fel ca și operatorul IN, operatorul EXISTS poate fi negat, luând forma NOT EXISTS. Totuși, o remarcă foarte importantă este faptul că pentru subinterrogări, NOT IN nu este la fel de eficient ca NOT EXISTS. Astfel dacă în lista de valori transmisă operatorului NOT IN există una sau mai multe valori Null, atunci condiția va lua valoarea de adevar False, indiferent de celelalte valori din listă.

De exemplu, următoarea interogare încearcă să returneze toate cadrele didactice care nu au nici un subaltern:

```
SQL> SELECT nume, grad FROM profesor
      WHERE cod NOT IN (SELECT sef FROM profesor);
```

Această interogări nu va întoarce nici o înregistrare deoarece coloana sef conține și valoarea Null. Pentru a obține rezultatul corect trebuie să folosim următoarea interogare:

```
SQL> SELECT nume, grad FROM profesor p
      WHERE NOT EXISTS (SELECT '1' FROM profesor
                           WHERE sef=p.cod);
```

NUME	GRAD
MARIN	PROF
ALBU	LECT
VOINEA	ASIST
STANESCU	ASIST

În general, operatorul EXISTS se folosește în cazul subinterrogărilor corelate și este cîteodată cel mai eficient mod de a realiza anumite interogări. Performanța interogărilor depinde de folosirea indecșilor, de numărul rândurilor returnate, de dimensiunea tabeliei și de necesitatea creării tabelelor temporare pentru evaluarea rezultatelor intermediare. Tabelele temporare generate de Oracle nu sunt indexate, iar acest lucru poate degrada performanța subinterrogărilor dacă se folosesc operatorii IN, ANY sau ALL.

Subinterrogările mai pot apărea și în alte comenzi SQL cum ar fi: UPDATE, DELETE, INSERT și CREATE TABLE.

Așa cum am văzut, există în principal două moduri de realizare a interogărilor ce folosesc date din mai multe tabele: joncțiuni și subinterrogări. Joncțiunile reprezintă forma de interogare *relațională* (în care sarcina găsirii drumului de acces la informație revine SGBD-ului) iar subinterrogările forma *procedurală* (în care trebuie indicat drumul de acces la informație). Fiecare dintre aceste forme are avantajele sale, depinzând de cazul specific în care se aplică.

8.1.21. Operații pe tabele ce conțin informații de structură arborescentă

O bază de date relațională nu poate stoca înregistrări în mod ierarhic, dar la nivelul înregistrării pot exista informații care determină o relație ierarhică între înregistrări. SQL permite afișarea rândurilor dintr-o tabelă ținând cont de relațiile ierarhice care apar între rândurile tabelei. Parcurgerea în mod ierarhic a informațiilor se poate face doar la nivelul unei singure tabele. Operația se realizează cu ajutorul clauzelor START WITH și CONNECT BY din comanda SELECT.

De exemplu, în tabela profesor există o relație ierarhică între înregistrări datorată valorilor din coloanele cod și sef. Fiecare înregistrare aferentă unui cadru didactic conține în coloana sef codul persoanei căreia îi este direct subordonat. Pentru a obține situație ce conține nivelele ierarhice, vom folosi următoarea interogare:

```
SQL> SELECT LEVEL, nume, prenume, grad FROM profesor
      CONNECT BY PRIOR cod=sef
      START WITH sef IS NULL;
```

LEVEL	NUME	PRENUME	GRAD
1	GHEORGHIU	STEFAN	PROF
2	MARIN	VLAD	PROF
2	GEORGESCU	CRISTIANA	CONF
3	IONESCU	VERONICA	ASIST
4	STANESCU	MARIA	ASIST
2	ALBU	GHEORGHE	LECT
2	VOINEA	MIRCEA	ASIST

Explicarea sintaxei și a regulilor de funcționare pentru exemplul de mai sus:

- Clauza standard SELECT poate conține pseudo-coloana LEVEL ce indică nivelul înregistrării în arbore (cât de departe este de nodul rădăcină). Astfel, nodul rădăcină are nivelul 1, fiind acestuia au nivelul 2, ș.a.m.d.;
- În clauza FROM nu se poate specifica decât o tabelă;
- Clauza WHERE poate apărea în interogare pentru a restricționa vizitarea nodurilor (înregistrărilor) din cadrul arborelui;
- Clauza CONNECT BY specifică coloanele prin care se realizează relația ierarhică; acesta este clauza cea mai importantă pentru parcurgerea arborelui și este obligatorie;
- Operatorul PRIOR stabilește direcția în care este parcurs arborele. Dacă clauza apare înainte de atributul cod, arborele este parcurs de sus în jos, iar dacă apare înainte de atributul sef, arborele este parcurs de jos în sus;
- Clauza START WITH specifică nodul (înregistrarea) de început al arborelui. Ca punct de start nu se poate specifica un anumit nivel (LEVEL), ci trebuie specificată valoarea; această clauză este optională, dacă ea lipsește, pentru fiecare înregistrare se va parcurge arborele care are ca rădăcină această înregistrare.

În sintaxa interogării de mai sus, pentru a ordona înregistrările returnate, poate apărea clauza ORDER BY, dar este recomandabil să nu o folosim deoarece ordinea implicită de parcursere a arborelui va fi distrusă.

Pentru a elimina doar un anumit nod din arbore putem folosi clauza WHERE, iar pentru a elimina o întreagă ramură dintr-un arbore (o anumită înregistrare împreună cu fiile acesteia) folosim o condiție compusă în clauza CONNECT BY.

Următorul exemplu elimină doar înregistrarea cu numele 'GEORGESCU', dar nu și copiii acestea:

```
SQL> SELECT LEVEL, nume, prenume, grad FROM profesor
      WHERE nume != 'GEORGESCU'
      CONNECT BY PRIOR cod=sef
      START WITH sef IS NULL;
```

LEVEL	NUME	PRENUME	GRAD
1	GHEORGHIU	STEFAN	PROF
2	MARIN	VLAD	PROF
3	IONESCU	VERONICA	ASIST
4	STANESCU	MARIA	ASIST
2	ALBU	GHEORGHE	LECT
2	VOINEA	MIRCEA	ASIST

Pentru a elimina toată ramura care conține înregistrarea cu numele 'GEORGESCU' și înregistrările pentru subordonații acestea se folosește următoarea interogare:

```
SQL> SELECT LEVEL, nume, prenume, grad FROM profesor
      CONNECT BY PRIOR cod=sef AND nume != 'GEORGESCU'
      START WITH sef IS NULL;
```

LEVEL	NUME	PRENUME	GRAD
1	GHEORGHIU	STEFAN	PROF
2	MARIN	VLAD	PROF
2	ALBU	GHEORGHE	LECT
2	VOINEA	MIRCEA	ASIST

8.2 Comanda INSERT

Această comandă este utilizată pentru adăugarea unor rânduri noi într-o tabelă creată anterior sau în tabelele de bază ale unei vederi. Comanda INSERT poate fi utilizată în două moduri:

1. Pentru introducerea datelor într-un tabel, câte o înregistrare la un moment dat. În acest caz sintaxa este următoarea:

```
INSERT INTO tabela [(coloana1, coloana 2, ...)]
VALUES (valoare1, valoare2, ...)
```

În momentul inserării datelor, trebuie respectate următoarele reguli:

- Coloanele pot fi specificate în orice ordine, însă trebuie asigurată corespondența între coloane și valorile furnizate (coloanei 1 îi corespunde valoarea 1, coloanei 2 îi corespunde valoarea 2, și.a.m.d.) iar coloanelor nespecificate le va fi atașată valoarea Null;
- În cazul în care coloanele nu sunt specificate explicit, se impune să fie specificate valori pentru toate coloanele și ordinea acestor valori să coincidă cu cea în care coloanele au fost definite la crearea tabelei (dacă nu se cunoaște ordinea de declarare a coloanelor se poate folosi comanda DESCRIBE nume_tabela care va afișa lista coloanelor definite pentru tabela respectivă, tipul, lungimea și restricțiile de integritate);
- Valorile trebuie să aibă același tip de date ca și câmpurile în care sunt adăugate;
- Dimensiunea valorilor introduce trebuie să fie mai mică sau cel mult egală cu dimensiunea coloanei (un sir de 20 de caractere nu poate fi adăugat într-o coloană cu dimensiunea de 15 caractere);
- Valorile introduse trebuie să respecte restricțiile de integritate definite la crearea tablei (de exemplu, câmpuri definite ca NOT NULL sau UNIQUE).

Atunci când se inseră valori de tip date calendaristică în format predefinit (DD-MON-YY), sistemul presupune în mod automat secolul 20, ora 00:00:00 (miezul nopții)

Următoarea instrucție exemplifică introducerea unei noi înregistrări în tabela profesor:

```
SQL> INSERT INTO profesor (cod, nume, prenume, data_nast,
                               sef, salariu, cod_catedra)
      VALUES (107, 'POPESCU', 'SERGIU', '09-DEC-71',
              100, 1200, 20);
```

Se poate observa că valorile coloanelor grad și primă, care nu au fost specificate, vor fi Null.

În cazul în care nu se specifică implicit numele coloanelor, valorile trebuie introduse în ordinea în care au fost definite și nu se poate omite valoarea nici unei coloane. Următoarea instrucție va produce același efect ca cea de mai sus:

```
SQL> INSERT INTO profesor
      VALUES (107, 'POPESCU', 'SERGIU', '09-DEC-71',
              NULL, 100, 1200, NULL, 20);
```

2. Pentru introducerea datelor într-un tabel, prin copierea mai multor înregistrări dintr-un alt tabel sau grup de tabele; aceste înregistrări sunt rezultatul unei comenzi SELECT. În acest caz sintaxa este următoarea:

```
INSERT INTO tabela[(coloana1,coloana2,...)] comanda_select
```

Și în acest caz trebuie respectate regulile de inserare, singura diferență fiind faptul că valorile noi introduse sunt extrase cu ajutorul unei interogări, acest lucru creând posibilitatea de inserare a mai multor înregistrări în funcție de anumite condiții.

De exemplu, pentru a inseră în tabela nou_profesor, având coloanele cod, nume, prenume și data_nastere, înregistrările din tabela profesor care au gradul didactic de asistent se poate folosi următoarea instrucțiune:

```
SQL> INSERT INTO nou_profesor(cod, nume, prenume, data_nastere)
   SELECT cod, nume, prenume, data_nastere FROM profesor
   WHERE grad='ASIST';
```

8.3. Comanda UPDATE

Comanda UPDATE este folosită pentru a modifica valorile existente într-un tabel sau în tabelele de bază ale unei vederi și are următoarea sintaxă generală:

```
UPDATE tabela [alias]
SET atribuire_coloane, [atribuire_coloane, ...]
[WHERE condiție];
```

unde atribuire_coloane poate avea una dintre următoarele forme:

```
coloana = {expresie | (subinterrogare)}
sau
(coloana [,coloana] ...) = (subinterrogare)
```

Se observă că există două posibilități de modificare:

- furnizarea în mod explicit a fiecărei valori sau expresii pentru câmpurile ce trebuie modificate;
- obținerea valorilor cu ajutorul unei subinterrogări.

Comanda UPDATE modifică valorile înregistrărilor în funcție de condiția clauzei WHERE. În lipsa clauzei WHERE, vor fi actualizate toate înregistrările din tabelul dat.

Expresia furnizată ca nouă valoare a unei coloane poate cuprinde valorile curente ale câmpurilor din înregistrarea care este actualizată. De exemplu, pentru a mări salariul cu 20% și prima cu 100 pentru cadrele didactice ce au gradul de asistent, se va folosi următoarea comandă:

```
SQL> UPDATE profesor
   SET salariu=salariu*1.2, prima=prima+100
   WHERE grad='ASIST';
```

Pentru a exemplifică actualizarea datelor utilizând subinterrogări presupunem că mai avem o tabelă numită prima ce conține sumele de bani primite suplimentar de unele cadre didactice:

COD	PRIMA
102	100
103	200
102	50

Pentru a modifica datele din tabela **profesor** pe baza datelor din tabela **prima** se poate folosi următoarea comandă care conține o subinterrogare corelată și o subinterrogare imbricată:

```
SQL> UPDATE profesor
      SET prima=(SELECT SUM(prima)
                  FROM prima a
                  WHERE a.cod=profesor.cod)
        WHERE cod IN (SELECT cod FROM prima);
```

O altă posibilitate este ca sumele suplimentare conținute în tabela **prima** să fie adăugate la prima existentă în tabela **profesor**:

```
SQL> UPDATE profesor
      SET prima=(SELECT SUM (prima) + profesor.prima
                  FROM prima a
                  WHERE a.cod=profesor.cod)
        WHERE cod IN (SELECT cod FROM prima);
```

Să presupunem acum că toți asistenții sunt transferați la catedra din care face parte cadrul didactic cu codul 104 și vor primi același salariu cu acesta:

```
SQL> UPDATE profesor
      SET (cod_catedra,salariu)=(SELECT cod_catedra, salariu
                                    FROM profesor
                                    WHERE cod= 104)
        WHERE grad = 'ASIST';
```

8.4. Comanda DELETE

Comanda **DELETE** realizează ștergerea înregistrărilor dintr-o tabelă sau din tabelele de bază ale unei vederi în funcție de o anumită condiție și are următoarea sintaxă generală:

```
DELETE FROM tabela
[WHERE condiție]
```

Similar comenzi UPDATE, comanda DELETE șterge anumite înregistrări în funcție de condiția din clauza WHERE. În lipsa clauzei WHERE vor fi șterse toate înregistrările din tabelul dat. În această clauză pot fi incluse și subinterrogări.

De exemplu următoarea comandă șterge toate înregistrările pentru care gradul didactic este asistent:

```
SQL> DELETE FROM profesor
      WHERE grad='ASIST';
```

Notă: Comanda DELETE nu poate fi folosită pentru ștergerea valorii unui câmp individual (pentru aceasta folosiți comanda UPDATE) ci șterge înregistrări complete dintr-un singur tabel. În plus, comanda DELETE șterge numai înregistrări din tabel nu și tabelul. Pentru a șterge un tabel se folosește comanda DROP TABLE.

Un alt aspect important este faptul că, similar comenziilor INSERT și UPDATE, ștergerea înregistrărilor dintr-un tabel poate determina apariția unor probleme legate de integritatea referențială. Pentru a evita aceste probleme se pot defini constrângeri de integritate care împiedică operațiile de inserare, actualizare sau ștergere care ar distrugă integritatea referențială a datelor.

8.5. Comanda TRUNCATE

Pentru a șterge în mod rapid toate înregistrările dintr-o tabelă sau dintr-un cluster se poate folosi comanda TRUNCATE. Comanda TRUNCATE este mult mai rapidă decât comanda DELETE din următoarele motive:

- Comanda TRUNCATE este o comandă DDL, prin urmare se execută dintr-o singură tranzacție și deci nu folosește segmentul de revenire. Comanda trebuie folosită cu precauție deoarece nu mai poate fi derulată înapoi.
- Comanda TRUNCATE nu declanșează trigger-ul DELETE (vezi secțiunea 9.16).

Comanda are următoarea sintaxă generală:

```
TRUNCATE {TABLE tabel|CLUSTER cluster}
[ {DROP|REUSE} STORAGE ]
```

unde:

- Clauza TABLE specifică numele unei tabele iar clauza CLUSTER specifică numele unui cluster. După cum se observă din sintaxă, aceste două opțiuni sunt alternative, deci nu se poate specifica într-o comandă TRUNCATE ștergerea rândurilor dintr-o tabelă și dintr-un cluster în același timp. În cazul în care se specifică clauza TABLE, tabela la care se referă această clauză nu poate face parte dintr-un cluster. Comanda TRUNCATE se poate executa și asupra tabelelor organizate pe index. La trunchierea unei tabele, Oracle șterge automat datele din indecesii tabelei. În cazul în care se specifică clauza CLUSTER, clusterul la care se referă această clauză nu poate fi un cluster hash ci numai un cluster de index. De asemenea, la trunchierea unui cluster, Oracle șterge automat datele din indecesii tabelelor clusterului.
- Clauza DROP STORAGE eliberează spațiul alocat înregistrărilor șterse din tabel sau cluster. Clauza REUSE STORAGE păstrează spațiul alocat înregistrărilor șterse din tabel sau cluster. Acest spațiu care nu a fost dealocat poate fi reutilizat doar la operații de inserare sau modificare asupra tabelei sau clusterului. Aceste două opțiuni nu modifică efectul pe care îl are comanda TRUNCATE asupra spațiului eliberat de datele șterse din indecesii asociati. Opțiunea implicită este DROP STORAGE.

Stergerea înregistrărilor cu ajutorul comenzi TRUNCATE este mult mai avantajoasă decât eliminarea tabelului și recrearea lui ulterioră deoarece:

- Eliminarea tabelului face ca obiectele dependente de acesta să devină invalide, pe când în cazul folosirii comenzii TRUNCATE nu se întâmplă acest lucru;
- Comanda TRUNCATE nu necesită reacordarea de drepturi asupra tabelului așa cum se întâmplă dacă acesta a fost eliminat și apoi recreat;
- Eliminarea tabelului necesită recrearea indecșilor, constrângerilor de integritate, declanșatoarelor, precum și specificarea parametrilor de stocare.

De exemplu, dacă un utilizator execută comanda `SELECT COUNT (*) FROM nume_tabel`, iar această interogare returnează după un interval destul de îndelungat valoarea zero, se recomandă trunchierea tabelului cu eliberarea spațiului alocat înregistrărilor sterse.

Capitolul 9

PL/SQL

PL/SQL (Procedural Language/Structured Query Language) este o extensie a limbajului Oracle SQL. PL/SQL permite ca instrucțiunile SQL de interogare și manipulare a datelor să fie incluse în blocuri sau unități procedurale de cod, cu alte cuvinte, PL/SQL este un limbaj cu structură de bloc. Prin urmare programatorii pot realiza proceduri, funcții și blocuri ce combină instrucțiunile SQL cu instrucțiunile procedurale (așa cum sugerează și numele PL/SQL).

PL/SQL prezintă următoarele caracteristici:

1. Structură de bloc

Acest lucru permite ca programele să fie împărțite în blocuri logice bine definite și ușor de gestionat. De asemenea programatorul poate declara variabile locale blocului și poate manipula rutinele pentru tratarea erorilor (numite excepții) specifice blocului apelat.

2. Structuri de control

PL/SQL utilizează structuri de control ce permit controlarea cursului logic de executare a instrucțiunilor dintr-un bloc PL/SQL dând o flexibilitate mare în programare. Limbajul PL/SQL include structuri de control fundamentale (structura condițională, iterativă și sevențială). Această caracteristică PL/SQL permite gruparea comenzi și executarea acestora în mod controlat. Acest lucru nu se poate realiza în SQL deoarece acesta nu este un limbaj procedural.

3. Portabilitatea

Codul PL/SQL este compatibil cu orice sistem de operare sau platformă pe care rulează sistemul Oracle. Astfel programele se pot muta pe orice mediu gazdă care suportă Oracle și PL/SQL.

4. Integrarea

PL/SQL ocupă un rol central pentru Oracle Server și instrumentele Oracle. Variabilele și tipurile de date PL/SQL sunt compatibile cu cele SQL (adică tipuri de date ce corespund coloanelor din tabelele bazei de date Oracle). PL/SQL face legătura între accesul la tehnologia bazei de date și necesitatea programării procedurale.

5. Performanța

Utilizarea PL/SQL poate îmbunătăți performanțele unei aplicații. Performanțele diferă în funcție de mediul în care PL/SQL este utilizat.

9.1 Blocuri PL/SQL

Orice unitate PL/SQL cuprinde unul sau mai multe blocuri. Aceste blocuri pot fi complet separate sau imbricate, deci un bloc poate reprezenta doar o mică parte din alt bloc care la

rândul lui poate fi doar o parte din unitatea de cod. În general un bloc este sau un *bloc anonim* sau un *subprogram*.

Blocurile anonte

Sunt acele blocuri care nu au nume. Deoarece un bloc anonim nu are nume, acesta nu poate fi apelat din alte programe. Blocurile anonte sunt declarate într-un punct al aplicației de unde sunt lansate în execuție, moment în care controlul este transferat motorului PL/SQL. Înainte de fiecare execuție, blocurile anonte PL/SQL trebuie să fie compilate. Ele se pot regăsi în programe precompilate în SQL*Plus sau Server Manager. Declanșatoarele din Oracle Forms sunt alcătuite din astfel de blocuri.

Subprogramele

Sunt acele blocuri PL/SQL declarate sub formă de proceduri sau funcții. Deoarece aceste blocuri PL/SQL au asociat un nume ele pot fi apelate din alte subprograme. În cazul în care sunt declarate sub formă de funcții, la momentul apelării blocurile întorc în mod explicit un parametru. Dacă blocurile sunt declarate sub formă de proceduri, la momentul execuției nu se returnează în mod explicit nici un parametru. Oracle Forms permite declararea procedurilor și funcțiilor și apelarea lor dintr-un declanșator sau dintr-o altă procedură sau funcție.

Structura blocurilor PL/SQL

Așa cum este arătat în figura 9.1 un bloc PL/SQL este compus din trei părți:

- o parte declarativă;
- o parte executabilă;
- o parte de tratare a excepțiilor (în PL/SQL apariția unei erori este numită *exception*).

Un bloc PL/SQL conține în mod obligatoriu elementele BEGIN și END în care sunt incluse acțiunile ce urmează să fie executate. După elementul END trebuie pus obligatoriu ;.

<p>[DECLARARE declarații locale]</p> <pre>BEGIN Instrucțiuni [EXCEPTION Instrucțiuni] END;</pre>	<p>PROCEDURE nume [listă argumente] IS [declarații locale] BEGIN Instrucțiuni [EXCEPTION Instrucțiuni] END;</p>	<p>FUNCTION nume [listă argumente] RETURN tipul_dată_returnată IS [declarații locale] BEGIN Instrucțiuni; RETURN rezultat [EXCEPTION Instrucțiuni; RETURN rezultat;] END;</p>
--	---	---

Figura 9.1 – Structura blocurilor PL/SQL

Dintre cele trei secțiuni numai partea executabilă este în mod obligatoriu necesară. Ordinea celor trei părți este logică. Prima parte este cea de declarație, secțiune în care se definesc obiectele PL/SQL, cum ar fi variabilele ce urmează a fi referite în blocul PL/SQL. Secțiunea de declarație este optională. Odată declarate, obiectele pot fi manipulate în partea executabilă a blocului PL/SQL. Excepțiile apărute în timpul execuției sunt plasate în secțiunea de tratare a excepțiilor. În această secțiune se definesc acțiuni ce vor fi executate atunci când condițiile de apariție a unei erori sunt îndeplinite. Secțiunea de tratare a erorilor este optională și trebuie plasată înainte de END.

Sub-blocurile se pot imbrica în secțiunea executabilă sau în secțiunea de tratare a erorilor unui bloc PL/SQL, dar nu se poate realiza imbricarea în secțiunea declarativă. În schimb în secțiunea declarativă se pot defini subprograme locale ce pot fi apelate în blocul în care sunt definite.

9.2. Arhitectura PL/SQL

PL/SQL nu este un produs Oracle de sine stătător, ci o tehnologie asemănătoare cu un motor ce execută blocuri sau subprograme PL/SQL. Acest motor poate fi instalat în:

- Serverul bazei de date (Oracle Server)
- Instrumente Oracle (cum ar fi Oracle Forms sau Oracle Reports)

Cele două medii de programare sunt independente. Astfel, PL/SQL poate fi disponibil în serverul bazei de date dar nu și în instrumentele Oracle, sau invers. Figura 9.2 prezintă modul în care motorul PL/SQL procesează un bloc anonim.

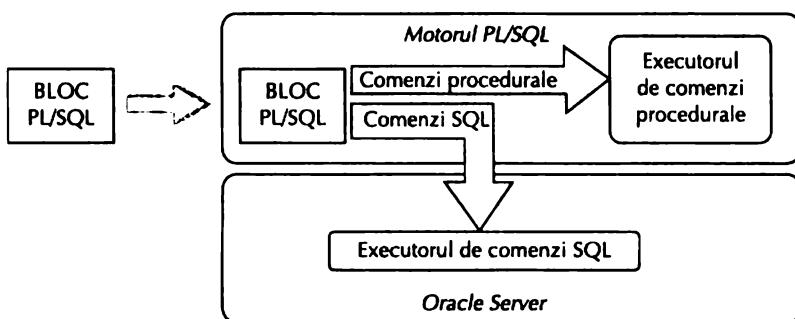


Figura 9.2 – Procesarea unui bloc anonim

Motorul execută comenziile procedurale dar comenziile SQL sunt trimise spre execuție către executorul de comenzi SQL (SQL Statement Executor) din Oracle Server.

În cazul în care instrumentele Oracle nu conțin un motor PL/SQL local, acestea trebuie să trimită spre procesare comenziile procedurale către Oracle Server. Prin urmare este nevoie de un singur transfer pentru a trimite blocul din aplicație către Server, ducând la îmbunătățirea performanței în special într-o rețea client-server.

În celălalt caz, atunci când motorul este instalat în instrumentele Oracle, comenziile procedurale sunt executate local, ceea ce înseanță reducerea cantității de prelucrat pentru Oracle Server precum și a numărului de cursoare de memorie necesare. Cu alte cuvinte, cea mai

mare cantitate de prelucrat este executată de către aplicația Oracle; mai mult, dacă blocul nu conține comenzi SQL atunci este executat local în întregime. Acest lucru este foarte util atunci când aplicația folosește controlul condițional sau iterativ. Oracle Forms utilizează în mod frecvent comenzi SQL doar pentru a testa valorile atribuite unor câmpuri sau pentru a realiza calcule simple.

Din punct de vedere al compatibilității PL/SQL cu limbajul SQL există următoarele reguli:

- PL/SQL furnizează toate comenziile limbajului de manipulare a datelor (DML) ale limbajului SQL, comanda de interogare SELECT (cu clauza INTO), comenziile pentru controlul tranzacțiilor, precum și funcțiile, pseudo-coloanele și operatorii SQL;
- Limbajul PL/SQL nu furnizează comenzi de definire a datelor.

9.3. Sintaxa de bază a PL/SQL

Deoarece PL/SQL reprezintă extensia procedurală a limbajului SQL, în general regulile de sintaxă care se aplică în SQL sunt aplicabile și în PL/SQL:

- comenziile pot fi scrise pe mai multe rânduri însă cuvintele cheie nu;
- unitățile lexicale (identificatori, nume de operatori, literalii) sunt despărțite prin unul sau mai multe spații, sau prin delimitatori ce nu pot fi confundați cu unități lexicale;
- nu se pot utiliza cuvinte rezervate ca identificatori, decât dacă sunt introduse între ghilimele (de exemplu "UPDATE");
- identificatorii pot conține maxim 30 de caractere și trebuie să înceapă cu un caracter alfabetic;
- literalii de tip caracter sau dată calendaristică trebuie incluși între apostrofuri;
- literalii de tip numeric se reprezintă prin valori simple (de exemplu -13.7) sau notație științifică (3E4, ceea ce înseamnă 3×10^4);
- comentariile pe mai multe linii se includ între simbolurile /* și */, iar comentariile mono-linie încep cu semnul dublă cratimă -- și se termină la sfârșitul liniei respective. Comentariile nu pot fi imbricate și nici nu se pot folosi comentarii mono-linie dacă blocul este procesat de un program de precompilare.

9.4. Referirea obiectelor

Regiunea unui program în care se poate referi un obiect se numește *domeniu de vizibilitate* al obiectului. Acest lucru este valabil pentru toate obiectele declarate în secțiunea DECLARE, inclusiv variabile, curse, exceptii definite de utilizator și constante. Blocul ce conține un obiect declarat, împreună cu sub-blocurile imbricate în acesta, reprezintă domeniul de vizibilitate al obiectului, deoarece obiectul poate fi accesat și din interiorul sub-blocurilor conținute în blocul unde acesta a fost declarat. Obiectele declarate în sub-blocuri sunt accesibile până în momentul când sub-blocul s-a terminat. În acest sens este recomandabilă definirea variabilelor cât mai local cu putință. Cu alte cuvinte, dacă o variabilă este folosită numai în blocul cel mai din interior, aceasta trebuie declarată în acel bloc.

Se pot defini obiecte care au același nume, dar numai dacă acestea se găsesc în blocuri diferite. În acest context numai obiectul declarat în blocul curent se poate folosi. Acest lucru nu înseamnă că obiectul cu același nume aflat în alt bloc decât cel curent nu mai există, ci faptul că obiectul nu are *vizibilitate* în blocul curent.

9.5. Variabile și constante

Instrucțiunile SQL și instrucțiunile procedurale pot folosi variabile și constante. Deoarece nu sunt permise referințe anticipate, variabilele și constantele trebuie declarate înainte ca acestea să fie referite în instrucțiuni. Așa cum am mai spus, variabilele și constantele se declară în secțiunea de declarație a unui bloc PL/SQL.

Orice constantă sau variabilă are un anumit tip de dată ce specifică un anumit format, constrângeri sau un anumit interval în care datele pot lua valori. PL/SQL dispune de o varietate de date predefinite printre care toate tipurile de date ce se regăsesc și în limbajul SQL (adică tipuri de date ce corespund coloanelor din tabelele bazei de date Oracle). În plus față de SQL, PL/SQL prevede câteva tipuri de date proprii cum ar fi: BOOLEAN, BINARY_INTEGER, RECORD, TABLE.

Tipurile de date se împart în mai multe categorii:

- *tipuri de date scalare* – sunt acele tipuri de date ce nu au componente interne, adică conțin valori atomice. Majoritatea tipurilor de date scalare din PL/SQL sunt similare tipurilor de date Oracle SQL, dar există și tipuri de date suplimentare (de exemplu BOOLEAN, BINARY_INTEGER etc.);
- *tipuri de date compuse* – sunt acele tipuri de date ce au componente interne care pot fi manipulate în mod individual;
- *tipuri de date referință* – sunt acele tipuri de date ale căror valori, numite *pointeri*, fac referință către obiecte din program;
- *tipuri de date LOB* - sunt acele tipuri de date ale căror valori, numite *locatori* (*locators*), specifică locația unor obiecte de dimensiuni mari (de exemplu imagini grafice).

Consultați Anexa 3 pentru o descriere completă a tipurilor de date PL/SQL.

Identifierul unei variabile sau constante trebuie să îndeplinească următoarele condiții:

- să fie compuse din litere, semnul dolar, liniuță de subliniere (underscore) și cifre;
- primul caracter să fie o literă;
- lungimea trebuie să fie de maxim 30 de caractere;
- nu se face distincție între litere mari și litere mici (nu este CASE SENSITIVE).

Declararea constanțelor

În cazul unei constante este obligatorie inițializarea acesteia cu o anumită valoare în momentul declarării. Această valoare nu poate fi modificată mai apoi în interiorul blocului. Sintaxa pentru declararea constanțelor este următoarea:

```
identifier CONSTANT tip_data := expresie
```

Exemplu:

```
pi      CONSTANT  NUMBER(9,5):=3.14159;
titlu  CONSTANT  CHAR(5):='Titlu';
```

Declararea variabilelor

În momentul declarării se poate inițializa în mod optional o variabilă cu o anumită valoare. Această valoare poate fi schimbată apoi prin atribuiri făcute variabilei în interiorul blocului. Sintaxa pentru declararea și inițializarea variabilelor este următoarea:

```
identifier tip_data [NOT NULL] [{:=|DEFAULT} expresie]
```

unde:

- Dacă la declararea variabilei se folosește constrângerea NOT NULL, atunci în mod obligatoriu trebuie atribuită o valoare variabilei în momentul declarării acesteia. Ca urmare a folosirii acestei constrângeri, variabila nu va putea avea valoarea Null.
- Pentru inițializarea variabilelor se poate folosi ori operatorul de atribuire, ori cuvântul cheie DEFAULT. Dacă nu se inițializează în momentul declarării, variabila va avea valoarea Null până în momentul când acesteia își atribue o valoare explicită. În momentul declarării unei variabile ce are tipul NATURALN sau POZITIVEN, trebuie specificată și o valoare inițială a acesteia. În caz contrar va apărea o eroare de compilare.

O observație foarte importantă este faptul că o variabilă nu trebuie să aibă același nume (identifier) cu numele unei coloane dintr-un tabel deoarece atunci când o comandă SQL folosește o variabilă PL/SQL ce are același nume cu numele unei coloane, sistemul Oracle recunoaște variabila drept nume de coloană. O soluție pentru rezolvarea conflictelor de nume este adoptarea unei convenții ce va utiliza prefixul v_ pentru reprezentarea variabilelor.

Exemplu:

```
v_salariu      NUMBER (10,2);
v_gasit        BOOLEAN DEFAULT FALSE;
v_nume          VARCHAR (25):='ION';
v_data          DATE DEFAULT SYSDATE;
v_total         NUMBER NOT NULL :=0;
```

Atribuirea variabilelor

Variabilelor PL/SQL li se pot atribui valori în interiorul blocului cu ajutorul instrucției de atribuire. În acest caz trebuie specificat numele variabilei ce va lua o nouă valoare, în stânga operatorului de atribuire :=. O altă metodă de atribuire a variabilelor este folosirea clauzei INTO a comenzi SELECT sau utilizarea cursoarelor, aşa cum vom vedea mai târziu în cuprinsul acestui capitol.

Exemplu:

```
v_nume:='DANIEL';
v_salariu:=salariu*12;--unde salariu este o variabilă existentă
v_data:='12-Nov-99';
```

Deși în momentul atribuirii variabilelor, PL/SQL realizează unele conversii implicate pentru tipurile de date, trebuie avută în vedere compatibilitatea variabilelor cu expresiile care le sunt atribuite. În acest sens se pot folosi funcțiile TO_CHAR, TO_DATE, TO_NUMBER.

9.5.1. Folosirea variabilelor PL/SQL pentru stocarea valorilor înregistrărilor dintr-un tabel

Atributul %TYPE

În momentul declarării variabilelor PL/SQL cu scopul de a stoca valorile unei coloane trebuie să avem grijă ca variabila să aibă același tip de dată cu coloana tabelului. Dacă nu se îndeplinește această condiție, va fi semnalată o eroare în momentul execuției. Utilizând atributul %TYPE putem atribui unei variabile tipul și mărimea coloanei așa cum este definită în dicționarul de date. PL/SQL determină tipul și mărimea variabilei în momentul când blocul este compilat, astfel încât aceasta va fi mereu compatibilă cu coloana respectivă.

Sintaxa pentru definirea acestor variabile este:

```
identificator nume_tabel.nume_coloana%TYPE
```

Exemplu:

```
DECLARE
    v_nume      angajati.nume%TYPE;
    v_salariu   angajati.salariu%TYPE;
BEGIN
    SELECT nume, salariu
    INTO v_nume, v_salariu
    FROM angajati;
    -- alte acțiuni
END;
```

Atributul %ROWTYPE

O variabilă de tip înregistrare PL/SQL conține o colecție de câmpuri diferite, fiecare câmp putând fi adresat în mod individual. Câmpurile dintr-o variabilă de tip înregistrare pot avea tipuri de date și mărimi diferite, precum coloanele dintr-un tabel. Variabila de tip înregistrare este foarte folosită atunci când se dorește extragerea unui rând dintr-un tabel în vederea procesării lui într-un program PL/SQL (în special în curse). Pentru a declara o variabilă de tip înregistrare ce se bazează pe o colecție de câmpuri dintr-o tabelă

sau vedere se folosește atributul %ROWTYPE. Câmpurile din variabila de tip înregistrare vor avea același nume ca și coloanele tabelului sau vederii.

Sintaxa pentru definirea acestor variabile este:

```
identificator referinta%ROWTYPE
```

unde **identificator** reprezintă numele ales pentru variabilă iar **referinta** reprezintă numele tabelului sau vederii pe care se bazează variabila. În momentul declarării variabilei verificat dacă referința este validă, adică existența tabelului sau vederii pe care se bazează variabila.

Exemplu:

```
DECLARE
    angajat_inreg angajati%ROWTYPE;
```

Pentru a popula o variabilă de tip înregistrare PL/SQL cu un rând al unui tabel, trebuie folosită o comandă **SELECT** ce conține în clauza **INTO** numele variabilei:

```
BEGIN
    SELECT * INTO angajat_inreg
    FROM angajati WHERE cod=101;
    -- alte acțiuni
END;
```

Putem de asemenea să atribuim unei variabile de tip înregistrare valorile unei alte variabile de tip înregistrare cu condiția ca tabelele pe care se bazează aceste variabile să aibă aceeași structură sau să se bazeze pe același tabel:

```
DECLARE
    inreg1      angajati%ROWTYPE;
    inreg2      angajati%ROWTYPE;
BEGIN
    inreg1:=inreg2;
    --alte acțiuni
END;
```

Atât în instrucțiunile procedurale, cât și în SQL câmpurile dintr-o variabilă de tip înregistrare se pot referi în mod individual:

```
angajati_inreg.bonus:=angajati_inreg.salariu/12;
```

În cazul instrucțiunii **INSERT** nu se poate specifica doar numele variabilei de tip înregistrare în lista valorilor ci va trebui specificat fiecare câmp individual din variabila de tip înregistrare. Prin urmare următoarea instrucțiune este ilegală:

```
INSERT INTO angajati (cod, nume, prenume, salariu, bonus)
VALUES (inreg1);
```

Pentru a putea insera o înregistrare în mod corect vom folosi:

```
INSERT INTO angajati (cod, nume, prenume, salariu, bonus)
VALUES (inreg1.cod, inreg1.nume, inreg1.prenume,
        inreg1.salariu, inreg1.bonus);
```

Referirea variabilelor non-PL/SQL

Pentru a putea referi variabilele non-PL/SQL (variabile declarate în programe de precompilare, câmpuri din aplicații Forms, variabile de legătură SQL*Plus) trebuie folosit înaintea numelui variabilelor semnul : cu scopul de a le distinge de variabilele PL/SQL la nivel de compilator.

Exemplu:

```
:global.var1:='Yes';
```

Variabilele non-PL/SQL se mai numesc și *variabilele programului gazdă* și sunt folosite pentru a transfera valori în ambele sensuri între programul sursă și PL/SQL.

9.6. Operatori PL/SQL

Operatorii logici, aritmetici și de concatenare utilizati de PL/SQL sunt aceeași ca cei utilizati de SQL. În plus, în PL/SQL există un operator exponential (**). Ordinea de execuție a operatorilor este prezentată în tabelul din figura 9.3. Ca și în SQL ordinea de execuție a operațiilor este controlată cu ajutorul parantezelor.

Operator	Operația
**, NOT	Ridicare la putere, negare logică
+, -	Identitate, negare (operatori unari)
*, /	Înmulțire, împărțire
+, -,	Adunare, scădere, concatenare
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparare
AND	AND logic
OR	OR logic

Figura 9.3 - Ordinea de execuție a operatorilor

9.7. Comenzile SQL și PL/SQL

Așa cum s-a arătat până acum, în PL/SQL sunt implementate comenziile SQL, singura excepție făcând comanda SELECT care trebuie să conțină o clauză suplimentară, clauza INTO, care nu există în SQL, și să îndeplinească anumite condiții.

Referințe PL/SQL în comenzi SQL

Comenzile SQL pot folosi variabile PL/SQL cu condiția ca acestea să nu aibă același nume cu numele coloanelor unui tabel, însă în cadrul comenziilor SQL nu se pot folosi funcții sau atribute specifice PL/SQL (de exemplu funcția SQLCODE sau SQLERRM)

Comenzi de manipulare a datelor

În PL/SQL se pot utiliza fără nici o restricție comenziile SQL de manipulare a datelor (INSERT, UPDATE, DELETE).

Comanda SELECT în PL/SQL

Comanda SELECT într-un bloc PL/SQL este denumită de către ANSI ca "SQL încapsulat" și asupra ei se aplică următoarea regulă:

O comandă SELECT trebuie să returneze o singură înregistrare. Returnarea mai multor înregistrări sau a nici uneia va genera o eroare.

Totuși, PL/SQL gestionează aceste erori în secțiunea de excepții a blocului și, așa cum vom vedea în continuare există posibilitatea folosirii acestor erori pentru a testa dacă o comandă SELECT returnează zero, una sau mai multe înregistrări. Clauza INTO este obligatorie și trebuie plasată între clauza SELECT și FROM. Această clauză este utilizată pentru a specifica numele variabilelor ce vor stoca valorile returnate de interogare. Pentru fiecare element selectat trebuie să existe o variabilă iar ordinea din clauza INTO trebuie să corespundă cu ordinea elementelor selectate. Variabilele din clauza INTO pot fi variabile PL/SQL (declarate în secțiunea DECLARE), variabile ale programului gazdă (variabile precedate de caracterul : numite și variabile de legătură) și variabile de substituție (precedate de caracterul &). Deoarece comanda SELECT trebuie să returneze o singură valoare clauzele GROUP BY și ORDER BY nu pot fi folosite.

Exemple referitoare la comanda SELECT...INTO au fost folosite în secțiunea anterioară pentru prezentarea atributelor %TYPE și %ROWTYPE.

9.8 Funcții predefinite în PL/SQL

Majoritatea funcțiilor SQL referitoare la o singură înregistrare sunt disponibile și în PL/SQL: funcții numerice, funcții caracter, funcții pentru conversii de tip, funcții pentru data calendaristică, funcții diverse. În plus PL/SQL prevede două funcții pentru raportarea erorilor, SQLCODE și SQLERRM, despre care vom discuta mai târziu în acest capitol.

Există și funcții care nu sunt disponibile în instrucțiuni procedurale:

- GREATEST, LEAST și DECODE;
- funcții de grup: AVG, MIN, MAX, COUNT, SUM, STDDEV și VARIANCE.

Funcțiile de grup se aplică numai grupurilor de înregistrări din tabele deci din acest motiv ele sunt disponibile numai în instrucțiunile SQL dintr-un bloc PL/SQL, și nu în cadrul instrucțiunilor procedurale. În cadrul instrucțiunii SELECT...INTO, funcțiile de grup trebuie folosite cu atenție deoarece, așa cum am mai menționat, această instrucțiune nu poate conține clauza GROUP BY.

9.9. Cursoare implicate

De fiecare dată când se execută o comandă SQL (SELECT...INTO, INSERT, UPDATE, DELETE), serverul bazei de date deschide o arie de memorie în care comanda este analizată și executată. Această arie este denumită *cursor*. În momentul când în partea executabilă a unui bloc se întâlnește o comandă SQL, PL/SQL gestionează în mod automat acest *cursor implicit*.

Pentru a evalua ce s-a întâmplat în momentul când cursorul implicit a fost utilizat, PL/SQL oferă programatorului câteva atribute: SQL%ISOPEN, SQL%ROWCOUNT, SQL%FOUND și SQL%NOTFOUND. Aceste atribute se folosesc în secțiunea de tratare a exceptiilor a unui bloc pentru a evalua rezultatul comenzilor de manipulare a datelor. Valoarea atributelor cursorului se referă întotdeauna la cea mai recent executată instrucțiune SQL. Semnificația atributelor cursoarelor implicate este descrisă în tabelul de mai jos.

Atribut	Tip dată	Descriere
%ROWCOUNT	Întreg	Returnează numărul de linii afectate de o instrucțiune INSERT, UPDATE sau DELETE. Atributul %ROWCOUNT returnează zero dacă instrucțiunea SQL nu afectează nici o valoare.
%FOUND	Boolean	Are valoarea True dacă o instrucțiune INSERT, UPDATE sau DELETE afectează una sau mai multe linii sau dacă instrucțiunea SELECT...INTO returnează o linie sau mai multe linii.
%NOTFOUND	Boolean	Are valoarea True dacă o instrucțiune INSERT, UPDATE sau DELETE nu afectează nici o linie sau în cazul în care instrucțiunea SELECT...INTO nu returnează nici o linie. Atributul %NOTFOUND are o valoare logică contrară atributului %FOUND.
%ISOPEN	Boolean	Acest atribut are valoarea False deoarece programul închide în mod automat cursorul implicit imediat după execuția instrucțiunii SQL asociate, iar atributele sunt folosite pentru a evalua ce s-a întâmplat după ce cursorul implicit a fost utilizat.

Notă: Atunci când comenziile INSERT, UPDATE și DELETE nu afectează nici o înregistrare, PL/SQL nu semnalează nici o eroare.

9.10. Controlul tranzacțiilor

Tranzacțiile pentru manipularea datelor încep cu prima comandă ce urmează instrucțiunea COMMIT sau ROLLBACK și se termină în momentul când următoarea instrucțiune COMMIT sau ROLLBACK este executată cu succes. Comenzile pentru controlul tranzacțiilor (COMMIT, ROLLBACK) sunt disponibile și în PL/SQL deși mediul gazdă poate impune unele restricții în utilizarea lor. Într-un bloc PL/SQL se pot include de asemenea și comenzi explicate de blocare (LOCK TABLE și SELECT...FOR UPDATE).

9.11. Structuri de control în PL/SQL

Fluxul secvențial de execuție a instrucțiunilor unui program PL/SQL poate fi modificat cu ajutorul structurilor de control. Limbajul PL/SQL furnizează patru structuri de control fundamentale: structura secvențială, condițională, iterativă și de salt necondiționat. Pe lângă aceste structuri de control, PL/SQL mai furnizează și instrucțiunea vidă.

Controlul secvențial

În mod preștabilit, blocurile PL/SQL sunt executate secvențial, de sus în jos. Procesarea începe cu instrucțiunea BEGIN și se termină cu instrucțiunea END. Prin introducerea structurilor de control condiționale, iterative sau de salt necondiționat se poate schimba cursul preștabilit de procesare al instrucțiunilor.

Controlul condițional

Pentru controlul condițional, adică pentru a efectua acțiuni alternative în funcție de o anumită condiție, este folosită instrucțiunea IF. Instrucțiunea IF este similară cu comenzi de control condițional din alte limbi procedurale. Sintaxa instrucțiunii IF este următoarea:

```
IF condiție THEN
    acțiuni;
  [[ELSIF condiție THEN
    acțiuni;]
  [ELSIF condiție THEN
    acțiuni;]
  ...]
  [ELSE acțiuni;]
ENDIF;
```

În funcție de modul de utilizare al instrucțiunii IF aceasta se poate clasifica în mai multe tipuri.

Instrucțiunea IF simplă

PL/SQL execută acțiunile ce urmează după instrucțiunea condițională numai dacă condiția este îndeplinită (are valoarea True). Condiția poate fi o *comparație* sau o *variabilă*

booleană. În cazul în care condiția are valoarea False sau Null, instrucțiunile nu sunt procesate, controlul fiind transferat următoarei instrucțiuni ce urmează după delimitatorul de sfârșit al instrucțiunii condiționale (END IF).

Exemplu:

```
IF v_nume = 'Ionescu' THEN
    v_bonus:=500000;
END IF;
```

Rezultatul unei comparații poate avea valoarea Null dacă una din valorile comparate conține valoarea Null. De exemplu, în următorul bloc PL/SQL, ambele variabile x și y sunt inițializate cu valoarea Null. Chiar dacă cele două variabile au aceeași valoare, condiția nu va fi îndeplinită deoarece valorile comparate au valoarea Null, și prin urmare instrucțiunea $x := x + 1$ nu va fi procesată.

```
DECLARE
    x      NUMBER:=null;
    y      NUMBER:=null;
BEGIN
    IF x=y THEN
        x:=x+1;
    END IF;
END;
```

Instrucțiunea IF-THEN-ELSE

Dacă condiția are valoarea False sau Null, atunci se execută acțiunile din clauza ELSE. Acest lucru asigură execuția selectivă a uneia dintre cele două seturi de acțiuni indiferent dacă este îndeplinită condiția sau nu. Fiecare instrucțiune IF-THEN poate să aibă o singură clauză ELSE și trebuie să se încheie cu END IF.

Exemplu:

```
IF v_nume = 'Ionescu' THEN
    v_bonus:=500000;
ELSE
    v_bonus:=0;
END IF;
```

Cele două seturi de acțiuni ale unei instrucțiuni condiționale pot conține la rândul lor alte instrucțiuni condiționale. Fiecare instrucțiune IF imbricată trebuie să se termine cu clauza END IF.

Instrucțiunea IF-THEN-ELSIF

Clauza ELSIF reprezintă o alternativă în utilizarea instrucțiunilor IF imbricate. Astfel, dacă acțiunea din clauza ELSE este o altă instrucțiune IF, este mult mai convenabil să se utilizeze clauza ELSIF.

Exemplu:

```

IF titlu='dna' THEN
    statut:='casatorita';
ELSIF titlu='dra' THEN
    statut:='necasatorita';
ELSE
    statut:='necunoscut';
END IF;

```

Instrucțiunea de mai sus este echivalentă cu următoarele instrucțiuni IF imbricate:

```

IF titlu='dna' THEN
    statut:='casatorita';
ELSE
    IF titlu = 'dra' THEN
        statut:='necasatorita';
    ELSE
        statut:='necunoscut';
    END IF;
END IF;

```

Controlul iterativ

Pentru a executa de mai multe ori o secvență de instrucțiuni se folosește controlul iterativ. Există mai multe instrucțiuni care sunt folosite în acest sens.

Instrucțiunea LOOP

Instrucțiunea LOOP reprezintă cel mai simplu ciclu iterativ și constă dintr-un grup de instrucțiuni care se repetă, incluse între delimitatorii LOOP și END LOOP. De fiecare dată când se ajunge la delimitatorul END LOOP controlul revine în punctul unde se află delimitatorul LOOP. Acest ciclu iterativ se poate executa la infinit dacă în interiorul lui nu există nici o instrucțiune de control care să oprească acest lucru. Pentru a opri acest ciclu trebuie plasată în interiorul ciclului instrucțiunea EXIT, instrucțiune ce transferă controlul instrucțiunii imediat următoare delimitatorului END LOOP. Instrucțiunea EXIT are următoarea sintaxă:

```
EXIT [eticheta] [WHEN condiție]
```

În interiorul ciclului instrucțiunea EXIT se poate folosi în două moduri:

- ca acțiune într-o instrucțiune condițională;

Exemplu:

```
LOOP
```

```

    i:=i+1;
    -- alte acțiuni
    IF i>=10 THEN
        COMMIT;

```

```

        EXIT;
    END IF;
END LOOP;
```

- ca instrucție de sine stătătoare, cu obligativitatea folosirii clauzei WHEN.

Exemplu:

```

LOOP
    i:=i+1;
    -- alte acțiuni
    EXIT WHEN i>=10;
END LOOP;
```

Instrucția FOR-LOOP

Instrucția FOR-LOOP permite parcurgerea unui ciclu de un număr specificat de ori. Sintaxa este următoarea:

```

FOR contor IN [REVERSE] limita_inf..limita_sup
LOOP
    --acțiuni
END LOOP;
```

Variabila contor este o variabilă de tip întreg a cărei valoare va fi incrementată sau decrementată la fiecare iterare a ciclului. De asemenea, variabila contor este declarată implicit (nu trebuie declarată de utilizator). Valorile limita_inf și limita_sup pot fi constante, variabile sau expresii, însă în urma evaluării trebuie să ia valori întregi. Ele reprezintă valoarea minimă, respectiv valoarea maximă pe care o poate lua contorul. Dacă la intrarea în ciclu în urma evaluării limita superioară este mai mică decât limita inferioară, instrucțiunile nu sunt executate.

În general variabila contor este inițializată la intrarea în ciclu cu valoarea minimă și este incrementată cu 1 până ajunge la valoarea maximă:

```

FOR i IN 1..100
LOOP
    -- contorul i ia valoarea 1 la prima iterare și 100 la ultima
END LOOP;
```

Instrucția FOR LOOP poate fi folosită cu opțiunea REVERSE, caz în care variabila contor este inițializată cu valoarea maximă și decrementată cu 1 până ajunge la valoarea minimă:

```

FOR n IN REVERSE 50..100
LOOP
    -- contorul n ia valoarea 100 la prima iterare și 50 la ultima
END LOOP;
```

Instrucțiunea WHILE

Structura WHILE-LOOP repetă un grup de instrucțiuni până în momentul în care o condiție devine falsă. Spre deosebire de instrucțiunea FOR-LOOP, în cazul instrucțiunii WHILE numărul de iterații ce urmează să fie efectuate de un ciclu este necunoscut până în momentul încheierii ciclului deoarece numărul de iterații depinde de valoarea pe care o ia condiția. În cazul în care condiția are valoarea True instrucțiunile sunt executate. În caz contrar, ciclul iterativ ia sfârșit iar controlul este transferat instrucțiunii imediat următoare delimitatorului END LOOP. Deoarece condiția este evaluată la începutul ciclului, este posibil ca instrucțiunile din interiorul ciclului să nu fie executate nici măcar o dată. Sintaxa instrucțiunii WHILE este următoarea:

```
WHILE condiție
LOOP
    -- acțiuni
END LOOP;
```

Condiția unei instrucțiuni WHILE poate fi o condiție compusă.

Exemplu:

```
WHILE nume LIKE 'F%' AND salariu > 1000000
LOOP
    --acțiuni
END LOOP;
```

Dacă variabilele implicate în condiție nu se schimbă în interiorul ciclului atunci condiția va avea valoarea True mereu și deci ciclul se va executa de un număr infinit de ori.

Instrucțiunea EXIT poate fi folosită și în cadrul instrucțiunii FOR sau WHILE pentru a permite părăsirea ciclului.

Salt necondiționat

Instrucțiunea GOTO execută un salt necondiționat la o etichetă, transferând controlul instrucțiunii sau blocului etichetat. Este recomandabil ca instrucțiunea GOTO să fie utilizată cât mai puțin deoarece contravine principiilor programării structurate. Instrucțiunea GOTO face obiectul următoarelor restricții:

- nu se poate utiliza pentru a transfera controlul unei instrucțiuni din interiorul unei structuri IF sau LOOP sau al unui sub-bloc;
- nu se poate utiliza pentru a transfera controlul în exteriorul unui subprogram;
- nu se poate utiliza pentru a transfera controlul din secțiunea de tratare a excepțiilor în blocul curent.

Sintaxa instrucțiunii GOTO este următoarea:

```
GOTO nume_eticheta
```

Instrucțiunea vidă (NULL)

Instrucțiunea NULL specifică în mod explicit faptul că nu va fi executată nici o acțiune iar controlul este transferat instrucțiunii imediat următoare.

9.12. Excepțiile

Excepția reprezintă o eroare sau un avertisment care este generat fie de severul Oracle, fie de aplicația utilizatorului. În cele mai multe limbaje de programare, mai puțin în cazul în care se dezactivează opțiunea de control al erorilor, o eroare de execuție, provocată de exemplu de o încercare de împărțire la zero, oprește procesarea normală a datelor și transferă controlul sistemului de operare. În PL/SQL mecanismul numit *tratarea excepțiilor* permite programului să își continue execuția și în prezența anumitor erori.

Un bloc se termină întotdeauna când apare o excepție, dar se pot executa acțiuni ulterioare apariției excepției într-o secțiune specială de tratare a excepțiilor. Astfel, dacă există acțiuni în secțiunea de tratare a excepțiilor, excepția (eroarea) nu se va propaga în afara blocului sau în mediul de execuție. Deși nu se pot anticipa toate erorile posibile, utilizatorul poate trata o parte din acestea în blocurile PL/SQL.

Există două tipuri de excepții:

1. *Predefinite* - sunt acele excepții predefinite de sistemul Oracle și sunt asociate unui cod specific de eroare. Aceste erori pot avea drept cauză căderi ale echipamentelor sau ale rețelei, erori de programare, erori legate de integritatea datelor și multe alte cauze. Deși fiecare eroare Oracle are asociat un cod specific, excepțiile trebuie referite numai prin nume. Despre modul în care aceste excepții pot fi tratate vom vorbi în continuare în acest capitol. În tabelul de mai jos sunt prezentate numele celor mai importante erori Oracle precum și codul de eroare asociat acestora:

Nume excepție	Eroarea Oracle	Cod eroare	Descriere
ACCES_INTO_NULL	ORA-06530	-6530	Atribuire de valori atributelor unui obiect neinitializat (initializarea unui obiect se face prin folosirea în atribuire a constructorului sau a unui alt obiect, vezi capitolul 10).
COLLECTION_IS_NULL	ORA-06531	-6531	Aplicarea unei metode diferite de EXISTS unui tabel imbricat sau VARRAY neinitializat (vezi capitolul 10).
CURSOR_ALREADY_OPEN	ORA-06511	-6511	Încercare de deschidere a unui cursor deja deschis.
DUP_VAL_ON_INDEX	ORA-00001	-1	Detectarea unei valori duplicate într-o coloană cu index unic.
INVALID_CURSOR	ORA-01001	-1001	Efectuarea unei operații ilegale asupra unui cursor.
INVALID_NUMBER	ORA-01722	-1722	Conversia unui sir de caractere într-un număr a eşuat.
LOGIN_DENIED	ORA-01017	-1017	Conectarea la serverul Oracle a eşuat datorită unui nume de utilizator invalid sau a unei parole invalide.

Nume excepție	Eroarea Oracle	Cod eroare	Descriere
NO_DATA_FOUND	ORA-01403	-1403	Comanda SELECT...INTO nu returnează nici o înregistrare.
NOT_LOGGED_ON	ORA-01012	-1012	A fost efectuat un apel al bazei de date fără conectarea prealabilă la serverul Oracle.
PROGRAM_ERROR	ORA-06501	-6501	Este generată atunci când PL/SQL întâlnește o problemă internă.
ROWTYPE_MISMATCH	ORA-06504	-6504	În cazul deschiderii unui cursor cu parametri, trebuie să fie efectuate asocieri între parametrii actuali și cei formalii, înainte de poziția acestora în sintaxa de definire a cursorului. Dacă parametrii actuali nu au același tip ca și parametrii formalii, se va declanșa această eroare.
STORAGE_ERROR	ORA-06500	-6500	Este generată atunci când PL/SQL nu are la dispoziție suficientă memorie sau există anumite probleme legate de aceasta.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533	Este generată atunci când este referit un element dintr-un tabel îmbricat sau VARRAY a căruia poziție este mai mare decât numărul de elemente al colecției (vezi capitolul 10).
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532	Este generată atunci când este referit un element dintr-un tabel îmbricat sau VARRAY a căruia poziție este în afara limitelor (vezi capitolul 10).
TIMEOUT_ON_RESOURCE	ORA-00051	-51	Pauză survenită în timp ce programul Oracle așteaptă o resursă.
TOO_MANY_ROWS	ORA-01422	-1422	O instrucțiune SELECT...INTO a returnat mai mult de o singură linie.
VALUE_ERROR	ORA-06502	-6502	A survenit o eroare matematică, de conversie, de truncare sau referitoare la o constrângere.
ZERO_DIVIDE	ORA-01476	-1476	Încercare de împărțire la zero.

2. *Definite de utilizator* - sunt acele excepții declarate în bloc de către utilizator. În mod uzual, PL/SQL va genera o excepție definită de utilizator dacă acest lucru este cerut în mod deliberat în interiorul blocului, putându-se face o asociere între codul erorilor și excepție. Un exemplu de excepție definită de utilizator poate apărea, în cazul unei bănci, în momentul când un client dorește retragerea unei sume ce depășește valoarea deținută în cont.

Pentru a exemplifica cele expuse mai sus vom considera următorul bloc:

```

BEGIN
  INSERT INTO angajati (cod, nume)
  VALUES (20, 'GEORGE');
  INSERT INTO angajati (cod, nume)
  VALUES (20, 'DUMITRU');
END;

```

228 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Blocul din exemplul de mai sus se sfărșește fără a trata excepțiile și va produce următoarea eroare datorată încercării de a introduce o valoare după un câmp ce este cheie primară a relației:

```
ora - 00001: unique constraint violated
```

Așa cum am mai spus, pentru a evita astfel de evenimente și pentru a preveni propagarea excepțiilor în mediul de execuție trebuie definite acțiuni referitoare la acestea în secțiunea de tratare a excepțiilor. Acțiunile pot consta din una sau mai multe instrucțiuni procedurale sau SQL. Tratarea unei excepții începe cu cuvântul cheie WHEN și se termină în momentul începerii tratării altrei excepții (apariția altui cuvânt WHEN) sau în momentul terminării blocului (apariția cuvântului END). Sintaxa pentru tratarea excepțiilor este următoarea:

```
WHEN nume_exceptie THEN acțiuni
```

Exemplu:

```
DECLARE
    v_nume      angajati.nume%TYPE;
    v_salariu   angajati.salariu%TYPE;
BEGIN
    SELECT nume, salariu INTO v_nume, v_salariu FROM angajati
    WHERE data_nastere BETWEEN '01-JAN-74' AND '31-DEC-74';
EXCEPTION
    WHEN no_data_found THEN
        INSERT INTO tabel_erori
        VALUES ('Nimeni născut în 74');
    WHEN too_many_rows THEN
        INSERT INTO tabel_erori
        VALUES ('Mai mult decât o persoană în 74');
END;
```

În exemplul de mai sus se tratează două excepții datorate faptului că o comandă SELECT în PL/SQL trebuie să returneze o singură înregistrare. Astfel, în cazul în care nici o înregistrare nu îndeplinește criteriul de selecție, în tabela tabel_erori va fi inserată o înregistrare ce va conține sirul de caractere 'Nimeni născut în 74' iar în cazul în care mai multe înregistrări vor îndeplini criteriul de selecție va fi inserată o înregistrare ce va conține sirul de caractere 'Mai mult decât o persoană în 74'.

Excepțiile pot fi activate explicit cu ajutorul comenzi RAISE nume_exceptie. Blocul PL/SQL de mai jos verifică dacă în tabela angajati mai există persoane cu numele 'IONESCU' și prenumele 'HORIA'. Înregistrarea va fi inserată numai dacă nu mai există nici o persoană cu acest nume și prenume, în caz contrar eroarea este înregistrată într-un tabel de erori.

```

DECLARE
    dummy VARCHAR2(1);
BEGIN
    SELECT 'x' INTO dummy
    FROM angajati
    WHERE nume='IONESCU' AND prenume='HORIA';
    RAISE too_many_rows;
EXCEPTION
    WHEN no_data_found THEN
        INSERT INTO angajati VALUES(101, 'IONESCU', 'HORIA');
    WHEN too_many_rows THEN
        INSERT INTO tabel_erori
        VALUES('Mai exista persoane cu acest nume si prenume');
END;

```

În cazul în care comanda SELECT nu va returna nici o înregistrare atunci controlul programului va fi transferat instrucțiunii de tratare a excepției WHEN no_data_found. În mod similar, dacă sunt returnate mai multe înregistrări atunci controlul programului va fi transferat instrucțiunii de tratare a excepției WHEN too_many_rows. Dacă însă exact o înregistrare este selectată, se va continua execuția programului cu următoarea instrucțiune. Aceasta va activa în mod explicit excepția too_many_rows, astfel încât și în acest caz controlul programului va fi transferat instrucțiunii de tratare a excepției WHEN too_many_rows.

Comanda SELECT și excepțiile

Principalele excepții ce apar ca urmare a rezultatului unei comenzi SELECT sunt no_data_found (nici o înregistrare returnată) și too_many_rows (mai mult de o înregistrare returnată). Exemplul anterior ilustrează modul de funcționare al acestor excepții. Se pot defini și alte excepții, fiecare cu propriul set de acțiuni. De fiecare dată când apare o excepție, PL/SQL va procesa numai secțiunea de tratare a excepțiilor aferentă acesteia.

Notă: PL/SQL nu va declanșa nici o eroare de tip no_data_found sau too_many_rows pentru comenzi de manipulare INSERT, UPDATE și DELETE.

Tratarea excepțiilor cu WHEN OTHERS

Secțiunea de tratare a excepțiilor execută acțiunile aferente fiecărei excepții specificate. Pentru a evita tratarea fiecărei excepții în parte, se folosește secțiunea de tratare WHEN OTHERS care va cuprinde acțiuni pentru orice excepție care nu a fost tratată, adică pentru capturarea excepțiilor neprevăzute sau necunoscute.

Exemplu:

```

DECLARE
    nr_cod INTEGER:=100;
BEGIN
    SAVEPOINT aici;
    INSERT INTO angajati (cod, nume, prenume)
    VALUES (nr_cod, 'GEORGESCU', 'RADU');

```

```

EXCEPTION
  WHEN dup_val_on_index THEN
    nr_cod:=nr_cod+1;
    ROLLBACK TO aici;
  WHEN OTHERS THEN
    INSERT INTO tabel_erori
      VALUES ('Eroare în timpul execuției blocului');
END;

```

Blocul PL/SQL anterior creează mai întâi un punct de salvare numit `aici`, după care încearcă să insereze în tabela `angajati` o înregistrare. Dacă tabela conține deja o înregistrare ce are valoarea din câmpul `cod` egală cu valoarea care se încearcă a fi introdusă, atunci va fi declanșată excepția `dup_val_on_index`. Această excepție este activată numai dacă coloana `cod` din tabelul `angajati` are definit un index unic, situație presupusă în acest exemplu. La tratarea acestei excepții se incrementează valoarea ce va fi introdusă în câmpul `cod`, după care tranzacția este reluată prin derularea înapoi până la punctul de salvare `aici`. În cazul în care se declanșează o altă excepție diferită de excepția `dup_val_on_index`, atunci în tabela `tabel_erori` va fi inserată o înregistrare ce va conține sirul de caractere 'Eroare în timpul execuției blocului'.

Secțiunea de tratare a excepțiilor `WHEN OTHERS` trebuie utilizată cu foarte mare atenție deoarece poate masca erori critice sau poate împiedica aplicația să răspundă în mod corespunzător. Este recomandabil ca această secțiune de tratare a erorilor să fie folosită cât mai puțin cu putință fiind de preferat referirea în mod explicit a fiecărei excepții în parte. În cazul în care secțiunea `WHEN OTHERS` este utilizată se recomandă folosirea acesteia, dacă este posibil, în blocul cel mai din interior. Un exemplu de folosire abuzivă a excepției `WHEN OTHERS` este următorul:

```

EXCEPTION
  WHEN OTHERS THEN
    NULL;
END;

```

Practic secțiunea de tratare `WHEN OTHERS` execută instrucțiunea `NULL` (instrucțiunea vidă). Prin executarea acestei excepții sunt ascunse toate erorile logice ale aplicației.

Funcții de gestionare a erorilor

La apariția unei excepții, căteodată este necesar să se cunoască codul de eroare asociat sau mesajul de eroare. Acest lucru este foarte important pentru tratarea excepțiilor cu `WHEN OTHERS`, atunci când este necesară alegerea unui anumit set de acțiuni în funcție de eroare.

În acest sens PL/SQL conține două funcții:

- `SQLCODE` - returnează codul erorii asociat excepției declanșate;
- `SQLERRM` - returnează un mesaj de eroare asociat excepției declanșate incluzând și numărul erorii.

Funcțiile SQLCODE și SQLERRM sunt folosite pentru a obține informații referitoare la cea mai recentă excepție.

Exemplul următor preia codul de eroare și mesajul excepției în cauză și le stochează în tabela erori ce conține câmpurile cod și mesaj:

```

DECLARE
    mesaj_eroare      VARCHAR2(60);
    cod_mesaj          NUMBER;
BEGIN
    .....
EXCEPTION
    WHEN OTHERS THEN
        cod_mesaj:=SQLCODE;
        mesaj_eroare:=SUBSTR(SQLERRM,1,60);
        INSERT INTO erori (cod, mesaj)
        VALUES (cod_mesaj, mesaj_eroare);
END;

```

Notă: Funcțiile SQLERRM și SQLCODE nu se pot utiliza direct ca parte a unei instrucțiuni INSERT sau SELECT. Așa cum este exemplificat mai sus, valorile funcțiilor trebuie atribuite unor variabile locale, funcția SQLERRM unei variabile de tip caracter (șirul de caractere se trunchiază la lungimea variabilei) iar funcția SQLCODE unei variabile de tip numeric.

Propagarea excepțiilor

Atunci când este întâlnită o eroare, PL/SQL căută în sub-blocul curent rutina de tratare a erorii, execută acțiunile din această secțiune, iar controlul este transferat celui mai apropiat bloc ce urmează după sfârșitul sub-blocului. Dacă PL/SQL întâlnește o excepție ce nu este tratată în blocul curent, atunci excepția se propagă. Acesta înseamnă că excepția se transmite și blocului căruia îi este transferat controlul, adică blocului imediat exterior, acest lucru realizându-se până în momentul când întâlnește un bloc ce conține o secțiune de tratare a excepției respective. Dacă nu există nici un bloc ce conține o secțiune de manipulare a excepției respective atunci în mediul programului gazdă va apărea o eroare datorată unei excepții ne tratate. În acest caz pe ecranul utilizatorului va fi afișat codul și mesajul de eroare. De exemplu, în Forms va apărea mesajul "FRM-40735 unhandled exception in...".

Această trecere de la secțiunea de tratare a excepțiilor a unui bloc interior la secțiunea de tratare a excepțiilor a unui bloc exterior ocolește executarea instrucțiunilor rămase în aceste blocuri, cu alte cuvinte PL/SQL nu procesează acțiunile dintre sfârșitul unui sub-bloc și cuvântul EXCEPTION al blocului căruia i se transferă controlul.

Avantajul acestui comportament este faptul că tratarea erorilor proprii unor instrucțiuni se poate face în blocul unde sunt apelate instrucțiunile, în timp ce tratarea excepțiilor generale aferente acestora se poate face în blocuri exterioare. Se recomandă tratarea excepțiilor proprii unor instrucțiuni în blocul unde acestea au fost apelate pentru a preveni

situația când o excepție este interceptată și tratată de o secțiune de tratare a excepțiilor necorespunzătoare. De asemenea, trebuie avută în vedere situația când acțiunile ce tratează o anumită excepție pot genera o altă excepție.

Exemplu:

```

DECLARE
    mesaj_eroare      VARCHAR2(60);
    cod_mesaj        NUMBER;
BEGIN
    DECLARE
        v1 NUMBER(4);
    BEGIN
        SELECT cod INTO v1 FROM angajati
        WHERE functia='PRESEDINTE';
    EXCEPTION
        WHEN too_many_rows THEN
            INSERT INTO mesaje
            VALUES ('Există mai mulți președinti');
    END;
    DECLARE
        v1 NUMBER(4);
    BEGIN
        SELECT cod INTO v1 FROM angajati
        WHERE functia='MANAGER';
    EXCEPTION
        WHEN too_many_rows THEN
            INSERT INTO mesaje
            VALUES ('Există mai mulți manageri');
    END;
    EXCEPTION
        WHEN OTHERS THEN
            cod_mesaj:=SQLCODE;
            mesaj_eroare:=SUBSTR (SQLERRM,1,60);
            INSERT INTO erori (cod, mesaj)
            VALUES (cod_mesaj, mesaj_eroare);
    END;

```

Exemplul anterior constă dintr-un bloc PL/SQL ce conține la rândul lui două sub-blocuri. Cele două sub-blocuri încearcă să selecteze din tabela `angajati` o înregistrare care are valoarea câmpului funcție egală cu '`PRESEDINTE`', respectiv '`MANAGER`'. Fiecare sub-bloc are propria secțiune de tratare a erorii `too_many_rows` în urma căreia se inserează în tabela `mesaje` sirul de caractere '`Există mai mulți președinti`', respectiv '`Există mai mulți manageri`'. La apariția într-un sub-bloc a unei alte erori diferită de eroarea `too_many_rows`, eroarea respectivă se propagă în blocul exterior unde este tratată cu ajutorul secțiunii `WHEN OTHERS`. În această secțiune se preia codul de eroare și mesajul excepției în cauză și se stochează în tabela `erori` ce conține câmpurile `cod` și `mesaj`.

Excepțiile pot fi generate și în partea declarativă a unui bloc prin inițializarea incorectă a unei expresii. De exemplu, următoarea declarație generează o excepție deoarece constanta limită nu poate stoca numere mai mari decât 999:

```

DECLARE
    limita CONSTANT NUMBER(3) := 5000; -- generează o excepție
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        -- nu poate trata excepția apărută în partea de
        -- declaratie

```

În exemplul de mai sus blocul curent nu poate trata excepția apărută în partea de declarație deoarece aceasta se propagă direct în blocul exterior.

Activarea excepțiilor

Excepțiile se pot activa în două moduri:

- Prin apariția unei erori Oracle, asociată cu o excepție, fapt care duce la activarea excepției respective în mod automat. De exemplu, dacă la inserarea unei înregistrări în baza de date apare eroarea ORA-0001 atunci PL/SQL activează excepția DUP_VAL_ON_INDEX;
- Prin activarea excepției explicit, utilizând comanda RAISE în interiorul blocului. Cu această comandă se pot activa excepții predefinite sau definite de utilizator. Unele instrumente Oracle, de exemplu Forms, au propriile excepții predefinite care pot fi activate cu comanda RAISE pentru a forța producerea anumitor evenimente. Exemplificarea comenzii RAISE va fi făcută în continuare în secțiunea ce tratează excepțiile definite de utilizator.

Excepții definite de utilizator

Sunt acele excepții declarate în secțiunea de declarație a blocului. Excepțiile definite de utilizator sunt folosite în situațiile când nu se mai dorește procesarea acțiunilor blocului, fiind activate atunci când se întâlnesc anumite condiții definite de utilizator. Sintaxa pentru declararea excepțiilor de acest gen este următoarea:

```
nume_excepție EXCEPTION
```

Pentru ca o excepție definită de utilizator să fie interceptată și tratată trebuie parcurs următoarele etape:

- generarea excepției în mod explicit în porțiunea executabilă a blocului, folosind comanda:

```
RAISE nume_excepție
```

- tratarea excepției în secțiunea de tratare a erorilor:

```
WHEN nume_excepție THEN
```

Exemplu:

```

DECLARE
    stoc_lipsă EXCEPTION;
    stoc      NUMBER(4);

```

```

BEGIN
    SELECT cantitate INTO stoc FROM produse WHERE cod=101;
    IF stoc < 10 THEN
        RAISE stoc_lipsa;
    END IF;
EXCEPTION
    WHEN stoc_lipsa THEN
        INSERT INTO alerta (cod_produs, cant)
        VALUES (101, stoc);
END;

```

În exemplul de mai sus se definește excepția `stoc_lipsa`, după care se selectează în variabila `stoc` valoarea câmpului `cantitate` din tabela `produse` pentru înregistrarea ce are codul 101. Dacă stocul acestui produs este mai mic decât 10, atunci se generează excepția `stoc_lipsa` folosind comanda `RAISE`. În secțiunea de tratare a excepției generate explicit se inserează în tabela `alerta` valoarea codului și a cantității care este sub limita admisă de politica companiei.

Asocierea unui cod de eroare Oracle cu un nume

Pentru a trata o eroare internă Oracle nenominalizată am putut vedea anterior cum se folosește clauza `WHEN OTHERS` drept detector universal de excepții. O altă metodă de a trata o eroare nenominalizată este cu ajutorul directivei de compilare `EXCEPTION_INIT`. PL/SQL conține nume predefinite pentru o parte din codurile interne ale excepțiilor, dar nu toate excepțiile au nume predefinite. Pentru a putea asocia un nume unui anumit cod intern de eroare se folosește directiva de compilare `EXCEPTION_INIT`. Acest lucru permite referirea oricărei excepții interne prin nume și scrierea unei rutine specifice pentru tratarea ei.

Această directivă de compilare trebuie să apară în porțiunea declarativă a unui bloc, pachet sau subprogram și are următoarea sintaxă:

```
PRAGMA EXCEPTION_INIT (nume_exceptie, numar_cod_eroare)
```

unde `nume_exceptie` este numele unei excepții declarate anterior. Deci această directivă de compilare trebuie să apară după ce a fost declarată o excepție, căreia i se asociază numele cu codul de eroare, așa cum este ilustrat în exemplul următor:

```

DECLARE
    detectare_deadlock EXCEPTION;
    PRAGMA EXCEPTION_INIT (detectare_deadlock, -60);
BEGIN
    ...
EXCEPTION
    WHEN detectare_deadlock THEN
        -- acțiuni specifice de tratare a excepției
    ...
END;

```

Executarea același set de acțiuni pentru excepții nominalizate explicit

Pentru a execuția același set de acțiuni în cazul mai multor excepții nominalizate explicit, în secțiunea de manipulare a excepțiilor se poate utiliza operatorul OR:

```
WHEN invalid_number OR value_error THEN
    INSERT INTO tabel_erori
    VALUES (...);
```

9.13 Utilizarea explicită a cursoarelor

Așa cum am mai spus, de fiecare dată când se execută o comandă SQL, serverul bazei de date deschide o arie de memorie în care comanda este analizată și executată. Un *cursor* este o construcție PL/SQL ce permite denumirea acestor arii de memorie și accesarea informațiilor referitoare la acesta. În sistemul Oracle există două tipuri de cursoare:

- *cursoare implicite* – sunt acele cursoare declarate în mod implicit de către PL/SQL pentru fiecare din instrucțiunile INSERT, UPDATE, DELETE și SELECT. Acest lucru survine indiferent dacă interogarea returnează una sau mai multe linii;
- *cursoare explicite* – sunt acele cursoare declarate de utilizator și folosite pentru procesarea rezultatelor comenziilor SELECT care returnează linii multiple.

În lipsa cursoarelor explicite, proiectantul de aplicații Oracle ar trebui să preia și să gestioneze în mod explicit fiecare linie selectată de interogare. Liniile multiple returnate de o interogare poartă numele de *setul activ de înregistrări*. Dimensiunea setului activ reprezintă numărul de linii care au rezultat în urma interogării.

Pentru a controla un cursor explicit trebuie parcuse următoarele etape:

1. Declararea cursorului;
2. Deschiderea cursorului;
3. Preluarea datelor în cursor;
4. Închiderea cursorului.

Numărul maxim de cursoare ce pot fi deschise simultan este stabilit de parametrul de inițializare OPEN_CURSOR. Nu există o limită a valorii pe care o poate avea acest parametru, singura limitare fiind datea de memorie disponibilă pentru gestionarea acestora.

Declararea cursoarelor

Declararea unui cursor constă în denumirea cursorului și asocierea unei interogări cursorului declarat. Interogarea asociată este analizată (sunt validate coloanele și tabelele) dar nu este executată. Numele care este atribuit unui cursor nu este o variabilă PL/SQL ci un identificator nedeclarat ce este folosit pentru a face referire la interogare. Numele unui cursor nu se poate folosi în expresii și nici nu se pot atribui valori numelui unui cursor. Sintaxa pentru declararea unui cursor explicit este următoarea:

```
CURSOR nume_cursor IS interogare
```

Interogarea asociată cursorului este o comandă SELECT ce nu include clauza INTO.

Exemplul 1:

```
DECLARE
    CURSOR c1 IS
        SELECT nume, prenume, salariu
        FROM angajati
        WHERE salariu > 1000000;
```

Exemplul 2:

```
DECLARE
    CURSOR c2 (v_sal NUMBER, v_functie VARCHAR2) IS
        SELECT nume, prenume, salariu
        FROM angajati
        WHERE salariu > v_sal AND functie = v_functie;
```

Deschiderea cursoarelor

Deschiderea cursoarelor explicite se realizează cu comanda OPEN și duce la executarea interogării și identificarea setului activ de înregistrări. Sintaxa este următoarea:

```
OPEN nume_cursor[(lista_argumete)]
```

Exemplu:

```
OPEN c1;
```

În cazul deschiderii unui cursor cu parametrii, trebuie să fie efectuate asocieri între parametrii actuali și cei formali ținând cont de poziția acestora în sintaxa de definire a cursorului. Astfel pentru cursorul c2 definit anterior putem folosi următoarea comandă de deschidere a cursorului:

```
OPEN c2 (1000000, 'ANALIST');
```

Parametrii actuali dintr-un cursor sunt în general constante dar pot fi și variabile PL/SQL sau variabile ale programului gazdă:

```
OPEN c2 (v_salariu, 'ANALIST');
-- unde v_salariu este o variabilă PL/SQL
```

Atunci când este executată comanda OPEN cursorul identifică acele linii care satisfac interogarea, adică liniile nu sunt citite până când nu se deschide cursorul. După deschidere cursorul este inițializat pe prima linie a setului activ.

În cazul în care cursorul este declarat cu clauza FOR UPDATE acesta nu își va pierde semnificația, adică în momentul deschiderii cursorului sistemul Oracle va bloca liniile care îndeplinesc criteriile de căutare ale interogării.

Preluarea datelor în cursor

Preluarea datelor în cursor este realizată cu comanda **FETCH**, comandă care preia liniile din setul activ de înregistrări una câte una și le introduce în variabile PL/SQL sau variabile ale programului gazdă. Trebuie să specifice variabile pentru fiecare coloană din interogarea asociată cursorului sau se poate folosi o singură variabilă de tip înregistrare. Aceste variabile trebuie declarate înainte de a utiliza comanda **FETCH**. Sintaxa comenzii **FETCH** este următoarea:

```
FETCH nume_cursor INTO variabile;
```

Exemplul 1:

```
DECLARE
    CURSOR c1 IS
        SELECT nume, prenume, salariu
        FROM angajati
        WHERE salariu > 1000000;
    .....
BEGIN
    OPEN c1;
    FETCH c1 INTO v_nume, v_prenume, v_salariu;
    .....
END;
```

Exemplul 2:

```
DECLARE
    CURSOR c1 IS
        SELECT nume, prenume, salariu
        FROM angajati
        WHERE salariu > 1000000;
    inreg c1%ROWTYPE;
    .....
BEGIN
    OPEN c1;
    FETCH c1 INTO inreg;
    .....
END;
```

De fiecare dată când este utilizată o comandă **FETCH** cursorul avansează la linia următoare a setului activ. Mai mult, singura cale de a parcurge setul activ este prin intermediul acestei comenzi. Dacă se dorește consultarea unei linii deja prelucrată anterior, trebuie să închideți și să redeschideți cursorul, după care să preluăți liniile una câte una până se ajunge la linia dorită. Dacă setul activ a fost parcurs, următoarea comandă **FETCH** va întoarce valori nule în variabile. Acest lucru nu va cauza o eroare. În schimb, dacă se mai încearcă executarea a încă unei comenzi **FETCH** se va genera o eroare.

Într-o comandă SQL ce este folosită în interiorul unui bloc se poate folosi clauza **WHERE CURRENT_OF** dacă se dorește o referire la linia curentă din setul activ al cursorului. Acest lucru permite modificări sau ștergeri ale liniei adresate, fără a se mai utiliza în mod explicit o referință, folosind **ROWID**. Pentru ca rândurile accesate de cursor să fie blocate în vederea modificării sau ștergerii lor, trebuie folosită clauza **FOR UPDATE** în interogarea aferentă cursorului.

Exemplu:

```

DECLARE
    CURSOR c1 IS
        SELECT nume, prenume, salariu
        FROM angajati
        WHERE salariu > 1000000
        FOR UPDATE OF salariu;
    inreg c1%ROWTYPE;
BEGIN
    OPEN c1;
    FETCH c1 INTO inreg;
    IF inreg.nume = 'ION' THEN
        DELETE FROM angajati WHERE CURRENT OF c1;
    END IF;
    -- alte acțiuni
END;

```

Închiderea cursoarelor

Închiderea cursoarelor se realizează cu comanda CLOSE. Această comandă închide explicit cursorul făcând posibilă redeschiderea lui. Comanda CLOSE are următoarea sintaxă:

```
CLOSE nume_cursor
```

Exemplu:

```
CLOSE c1;
```

Atunci când utilizatorul închide legătura cu serverul, cursorul se închide în mod implicit.

Atributele cursoarelor

Pentru a evalua ce s-a întâmplat în momentul când un cursor a fost utilizat, PL/SQL oferă programatorului patru atribute: %ISOPEN, %ROWCOUNT, %FOUND și %NOTFOUND. Aceste atribute se folosesc atât pentru cursoarele explicate cât și pentru cursoarele implicate (despre care am vorbit la începutul acestui capitol). În cazul folosirii atributelor unui cursor implicit, acestea sunt precedate de numele implicit al cursorului, adică SQL (exemplu: SQL%ISOPEN, SQL%ROWCOUNT, SQL%FOUND și SQL%NOTFOUND). Dacă se dorește folosirea atributelor unui cursor explicit, acestea trebuie să fie precedate la rândul lor de numele cursorului explicit (exemplu: c1%ISOPEN, c1%ROWCOUNT, c1%FOUND și c1%NOTFOUND). Semnificația fiecărui atribut al unui cursor explicit este descrisă în tabelul de mai jos.

Atribut	Tip dată	Descriere
%ROWCOUNT	Întreg	Conține numărul total de linii preluate până la un anumit moment de comanda FETCH. În momentul deschiderii cursorului acest atribut are valoarea zero, după prima preluare cu comanda FETCH are valoarea 1, s.a.m.d.

%FOUND	Boolean	Acest atribut are valoarea True dacă cea mai recentă comandă FETCH a preluat o linie din setul activ. După deschiderea cursorului și înaintea preluării cu comanda FETCH acest atribut are valoarea Null. După prima preluare, acest atribut va avea valoarea True până în momentul în care comanda FETCH nu mai returnează nici o linie.
%NOTFOUND	Boolean	Acest atribut are valoarea True în cazul în care cea mai recentă comandă FETCH nu a mai returnat nici o linie. Acest atribut este opusul logic al atributului %FOUND și este folosit pentru a părăsi ciclul atunci când nu mai este nici o linie de preluat din setul activ.
%ISOPEN	Boolean	Acest atribut are valoarea True în cazul în care cursorul este deja deschis.

Exemplu de utilizare a atributului %ISOPEN:

```
IF c1%ISOPEN THEN
    FETCH c1 INTO v_nume, v_prenume, v_salariu;
ELSE
    OPEN c1;
END IF;
```

În mod ușual atunci când se dorește procesarea liniiilor dintr-un cursor explicit se folosesc un ciclu iterativ în care este executată o instrucțiune FETCH. Eventual acest ciclu poate procesa toate rândurile din setul activ până când instrucțiunea FETCH setează atributul %NOTFOUND pe valoarea True, adică setul activ a fost parcurs. În momentul în care se setează atributul %NOTFOUND pe valoarea True, comanda FETCH va întoarce valori nule în variabile. Dacă în continuare se execută o altă comandă FETCH, atunci Oracle va genera eroarea "ORA-1002: Fetch out of sequence", eroare care va duce, în general, la terminarea execuției blocului datorită unei excepții neîntrătate. Este deci foarte important ca de fiecare dată când se preiau date din cursor să se testeze rezultatul acestei prelucrări înainte de a mai face referiri ulterioare la cursorul respectiv.

Exemplu:

```
OPEN c1;
LOOP
    FETCH c1 INTO v_nume, v_prenume, v_salariu;
    EXIT WHEN c1%NOTFOUND;
END LOOP;
CLOSE c1;
```

Ciclul iterativ destinat unui cursor (CURSOR FOR LOOP)

PL/SQL oferă programatorilor un tip special de ciclu iterativ aferent unui cursor explicit (*cursor for-loop*). Sintaxa pentru un astfel de ciclu este următoarea:

```
FOR nume_inregistrare IN nume_cursor[(parametri)] LOOP
    ...
END LOOP;
```

Un astfel de ciclu simplifică codificarea prin efectuarea automată a următoarelor operații:

- deschiderea cursorului la inițializarea ciclului;
- executarea unei instrucțiuni FETCH la fiecare parcurgere a ciclului;
- părăsirea ciclului atunci când toate liniile din setul activ au fost procesate;
- închiderea cursorului la părăsirea ciclului.

Un alt aspect foarte important este faptul că variabila nume_inregistrare nu trebuie declarată deoarece este internă ciclului și scopul ei durează atât timp cât ciclul este executat. În interiorul unui astfel de ciclu pot fi testate atributelor cursorului.

Exemplu:

```
DECLARE
    CURSOR c1 IS
        SELECT nume, prenume, salariu FROM angajati;
        ...
BEGIN
    FOR angajat_inreg IN c1 LOOP
        ...
        salariu_total:=salariu_total+angajat_inreg.salariu;
    END LOOP;
END;
```

O altă metodă eficientă de a simplifica codul este folosirea unui ciclu ce procesează liniile returnate de o interogare definită chiar la începutul ciclului. Practic, este definit un cursor chiar la intrarea în ciclu:

```
FOR angajat_inreg IN
  (SELECT nume FROM angajati WHERE salariu >1000000) LOOP
    IF angajat_inreg.nume = 'ION ' THEN...
END LOOP;
```

Deoarece cursorul creat este intern ciclului, acesta nu trebuie declarat. În acest caz nu se pot testa atributele cursorului deoarece acesta nu are nume.

Cursoare cu parametri

În interogarea asociată cursorului se pot face referiri la variabile cu condiția ca acestea să fie declarate înaintea declarării cursorului. Pe lângă variabile se pot folosi și parametri. Parametrii sunt folosiți pentru a transfera cursorului diverse valori în momentul când acesta este deschis. Valorile transferate pot fi utilizate de interogare la momentul execuției acesteia. Acest lucru înseamnă că programatorul poate deschide un cursor explicit de mai multe ori pe parcursul unui bloc, de fiecare dată rezultând un alt set activ. Sintaxa pentru definirea unui cursor explicit cu parametri este următoarea:

```
CURSOR nume_cursor (nume_parametru tip_data)
IS interogare
```

Tipurile de date ale parametrilor sunt aceleși ca cele ale variabilelor scalare, cu deosebirea că pentru parametri nu pot aplica constrângeri asupra tipului de date.

Exemplul următor ilustrează declararea cursorului c2 ce are parametri param1 și param2. Acești parametri permit transferarea valorii salariului precum și a funcției salariatului necesare în clauza WHERE. Cursorul poate fi deschis de mai multe ori dar cu valori diferite ale parametrilor, de exemplu OPEN c2 (90000, 'FUNCTIONAR') sau OPEN c2 (250000, 'MANAGER'), rezultând seturi active diferite.

```
DECLARE
    CURSOR c2 (param1 NUMBER, param2 VARCHAR2)
    IS SELECT nume, prenume, salariu, functie
       FROM angajati
      WHERE salariu > param1
        AND functie = param2;
```

9.14. Proceduri și funcții

După cum am văzut la inceputul capitolului, în PL/SQL se pot defini subprograme, acestea fiind blocuri PL/SQL cărora li se pot transmite parametrii și care pot fi apelate din alte programe. Există două feluri de subprograme PL/SQL: proceduri și funcții. O *funcție* este un bloc PL/SQL care acceptă orice număr de parametrii de intrare sau ieșire și returnează o valoare. Din acest motiv o funcție nu poate fi apelată decât ca parte a unei atribuiri. O *procedură* este un bloc PL/SQL care acceptă orice număr de parametrii de intrare sau ieșire dar nu returnează în mod explicit nici o valoare.

O procedură sau funcție are două părți: specificația și corpul. *Specificația* unei proceduri începe de la cuvântul cheie PROCEDURE și se sfârșește cu numele procedurii sau cu lista de parametrii. În cazul unei funcții, specificația începe de la cuvântul cheie FUNCTION și se sfârșește cu clauza RETURN, clauză care specifică tipul de dată al valorii returnate.

Corpul unei proceduri sau funcții începe cu cuvântul cheie IS și se termină cu cuvântul cheie END urmat în mod optional de numele acesteia. Corpul unei proceduri sau funcții are trei părți: partea declarativă, partea executabilă și partea de tratare a excepțiilor.

Partea declarativă conține declarații de tipuri, cursoare, constante, variabile, excepții și subprograme interne. Aceste obiecte sunt locale și există doar atât timp cât subprogramul este în execuție. Cuvântul cheie DECLARE, care introduce partea de declarații pentru un bloc anonim nu este utilizat în acest caz.

Partea executabilă conține instrucțiuni de atribuire, de control și de manipulare a structurilor Oracle și este localizată între cuvântul cheie BEGIN și cuvântul EXCEPTION (sau END). În cazul unei proceduri trebuie să existe cel puțin o instrucțiune în această secțiune. Dacă se dorește ca procedura să nu facă nimic se poate utiliza instrucțiunea NULL. În cazul unei funcții această secțiune trebuie să conțină cel puțin o instrucțiune RETURN (a nu se confunda instrucțiunea RETURN cu clauza RETURN din partea de specificație a unei funcții).

Partea de tratare a excepțiilor tratează excepțiile generate în timpul execuției subprogramului și este situată între cuvântul cheie EXCEPTION și END. Sintaxa unei funcții și sintaxa unei proceduri în PL/SQL sunt prezentate explicit mai jos:

Procedură	Funcție
<pre>PROCEDURE nume_procedură [listă parametrii] IS [declarații locale] BEGIN partea executabilă [EXCEPTION partea de manipulare a excepțiilor] END[nume_procedură]</pre>	<pre>FUNCTION nume_funcție [listă parametrii] RETURN tipul_datei IS [declarații locale] BEGIN partea executabilă [EXCEPTION partea de manipulare a excepțiilor] END[nume_funcție]</pre>

După cum se poate observa mai sus, declarațiile locale cât și partea de tratare a excepțiilor sunt opționale, însă partea executabilă este în mod obligatoriu necesară.

Funcțiile și procedurile pot fi declarate în orice bloc, subprogram sau pachet PL/SQL. Funcțiile și procedurile trebuie să fie declarate la sfârșitul secțiunii declarative, după toate celelalte elemente ale programului. Astfel, următoarea declarație este incorrectă deoarece procedura `calcul` trebuie să se găsească pe ultima poziție, înainte de declararea variabilei `suma` și a cursorului `c1`.

```
DECLARE
    PROCEDURE calcul (...) IS
        BEGIN
        ...
    END;
    suma NUMBER;
    CURSOR c1 IS SELECT * FROM angajati;
```

În continuare vom utiliza ca exemplu procedura `crestere_salariu`, care mărește salariul unui angajat cu o anumită valoare:

```
PROCEDURE crestere_salariu(cod_sal INTEGER, crestere INTEGER) IS
    salariu_curent      INTEGER;
    lipsa_salariu      EXCEPTION;
BEGIN
    SELECT sal INTO salariu_curent FROM angajati
    WHERE cod = cod_sal;
    IF salariu_curent IS NULL THEN
        RAISE lipsa_salariu;
    ELSE
        UPDATE angajati SET sal = sal + crestere
        WHERE cod = cod_sal;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO erori
        VALUES(cod_sal, 'Nu există nici un angajat cu
acest cod');
    WHEN lipsa_salariu THEN
        INSERT INTO erori
        VALUES (cod_sal, 'Salariul este necunoscut');
END crestere_salariu;
```

În momentul apelării, această procedură are ca parametrii codul angajatului și valoarea cu care crește salariul. Codul angajatului este folosit pentru a selecta salariul curent. Dacă nu există nici un angajat cu acest cod sau dacă salariul este necunoscut, se generează o excepție. Altfel, valoarea salariului crește cu valoarea parametrului *crestere* al procedurii.

O procedură este apelată ca orice instrucțiune PL/SQL. De exemplu, procedura *crestere_salariu* de mai sus poate fi apelată:

```
crestere_salariu (101, 50000);
```

În continuare vom utiliza ca exemplu funcția *sal_cunoscut*, care verifică dacă salariul unui anumit angajat este cunoscut sau nu:

```
FUNCTION sal_cunoscut (cod_sal INTEGER) RETURN BOOLEAN IS
    salariu_curent INTEGER;
    gasit BOOLEAN:=False;
BEGIN
    SELECT sal INTO salariu_curent FROM angajati
    WHERE cod=cod_sal;
    IF salariu_curent IS NOT NULL THEN
        gasit:=True;
    END IF;
    RETURN gasit;
EXCEPTION
    WHEN NO_DATA_FOUND THEN .
        INSERT INTO erori
        VALUES (cod_sal, 'Nu există nici un angajat cu
                        acest cod');
        RETURN gasit;
END sal_cunoscut;
```

În momentul apelării, această funcție are drept parametru de intrare codul angajatului. Codul angajatului este folosit în continuare pentru a selecta salariul curent. Dacă nu există nici un angajat cu acest cod, se generează o excepție iar funcția întoarce valoarea False. Altfel, dacă salariul este cunoscut funcția întoarce valoarea True iar dacă salariul este necunoscut funcția întoarce valoarea False.

O funcție este apelată ca parte a unei atribuirii. De exemplu, funcția *sal_cunoscut* de mai sus, poate fi apelată:

```
succes := sal_cunoscut (cod_s);
```

Instrucțiunea RETURN

Așa cum am observat în exemplul de mai sus, instrucțiunea RETURN determină întreruperea imediată a execuției subprogramului și revenirea în programul apelant. Execuția continuă în programul apelant cu instrucțiunea care urmează după apelul

procedurii. A nu se confunda instrucțiunea RETURN cu clauza RETURN, clauză care specifică tipul de dată returnat de funcție. Un subprogram poate conține mai multe instrucțiuni RETURN dar utilizarea mai multor puncte de ieșire dintr-un subprogram nu este recomandată.

În cazul unei proceduri, instrucțiunea RETURN nu poate conține nici o expresie deoarece dintr-o procedură nu se întoarce nici o valoare, această instrucțiune determinând întreruperea execuției înainte de a se ajunge la sfârșitul procedurii.

În cazul unei funcții, instrucțiunea RETURN trebuie să conțină în mod obligatoriu o expresie care este evaluată la momentul execuției instrucțiunii. Acest lucru este impus chiar de definiția unei funcții, care trebuie să întoarcă în mod obligatoriu o valoare. Valoarea rezultată în urma acestei evaluări este atribuită numelui funcției, care acționează în acest sens ca o variabilă ce are tipul specificat în clauza RETURN.

Parametrii subprogramelor

Procedurile și funcțiile pot avea sau nu parametrii. Parametrii sunt folosiți pentru a transfera informații între subprogrami și mediul apelant. Variabilele sau expresiile referite în lista de parametri la apelul unei proceduri se numesc *parametrii actuali*. De exemplu, următorul apel al procedurii *crestere_salariu*, descrisă anterior, conține parametrii actuali *cod_angaj* și *bonus*:

```
crestere_salariu (cod_angaj, bonus);
```

Variabilele declarate în specificația unui subprogram și referite în corpul acestuia se numesc *parametrii formali*. De exemplu, procedura *crestere_salariu* conține parametrii formali *cod_sal* și *crestere* declarati în specificația acesteia.

```
PROCEDURE crestere_salariu(cod_sal INTEGER, crestere INTEGER) IS
```

Este recomandabil să se utilizeze nume diferite pentru parametrii actuali și parametrii formali. În momentul apelării parametrii actuali sunt evaluati, iar valorile rezultate sunt atribuite parametrilor formalii corespondenți. Parametrul actual și parametrul formal corespunzător acestuia trebuie să aibă tipuri de date compatibile.

Asupra unui parametru formal nu se poate impune constrângerea NOT NULL. De asemenea, nu se pot specifica constrângerile asupra dimensiunii tipului de dată al parametrului. De exemplu, următoarea declarație este incorectă deoarece impune o constrângere asupra dimensiunii parametrului.

```
PROCEDURE insert_sal (cod_sal NUMBER(4)) IS ...
--incorrect deoarece trebuie să fie NUMBER
```

Există trei moduri de transmitere a parametrilor între mediul apelant și subprogram: IN (parametrii de intrare), OUT (parametrii de ieșire) și IN OUT (parametrii de intrare- ieșire). Semnificația fiecărui mod în parte este descrisă în continuare. În cazul în care modul de transmitere nu este specificat în mod explicit, valoarea default a acestuia este IN.

Modul de transmitere	Descriere
IN (default) (parametru de intrare)	<p>Transmite o valoare dinspre mediul apelant în subprogram.</p> <p>Parametrul formal nu poate fi schimbat în interiorul subprogramului, deci se poate spune că acționează ca o constantă.</p> <p>Parametrul actual poate fi o constantă, o variabilă inițializată sau o expresie.</p> <p>Parametrul actual este transferat prin valoare (printr-o copie a valorii)</p>
OUT (parametru de ieșire)	<p>Returnează o valoare din subprogram în mediul apelant.</p> <p>Parametrul formal se comportă ca o variabilă neinițializată. În interiorul subprogramului, parametrului OUT trebuie să îl se atribuie o valoare care se dorește să ajungă în mediul apelant.</p> <p>Parametrul actual trebuie să fie o variabilă, nu o constantă sau o expresie.</p> <p>Parametrul actual este transferat prin referință (printr-un pointer către valoarea parametrului de intrare)</p>
IN OUT (parametru de intrare-ieșire)	<p>Transmite o valoare dinspre mediul apelant în subprogram, și după execuția acestuia, returnează din subprogram în mediul apelant o valoare posibil diferită.</p> <p>Parametrul formal se comportă ca o variabilă inițializată. În interiorul subprogramului trebuie să îl se atribuie o valoare.</p> <p>Parametrul actual trebuie să fie o variabilă, nu o constantă sau o expresie.</p> <p>Parametrul actual este transferat prin referință (printr-un pointer către valoarea parametrului de intrare)</p>

Un subprogram poate să transmită anumite valori mediului apelant în două moduri:

- prin intermediul numelui unei funcții, deoarece aceasta trebuie să returneze în mod obligatoriu o valoare;
- prin intermediul parametrilor de ieșire OUT sau IN OUT.

În cazul în care este necesară transmiterea unei singure valori mediului apelant, este recomandată folosirea unei funcții prin intermediul numelui acesteia. În cazul în care este necesară transmiterea mai multor valori mediului apelant, este recomandată folosirea unei proceduri ce conține parametri de ieșire. Este recomandat să se evite folosirea parametrilor de ieșire OUT sau IN OUT în cazul unei funcții deoarece scopul unei funcții este de a returna o singură valoare.

În continuare vom prezenta exemple ce conțin anumite erori datorate modului greșit de folosire al parametrilor pentru fiecare mod de transmitere în parte.

Așa cum s-a menționat anterior, într-un subprogram un parametru IN acționează ca o constantă și prin urmare nu îl se pot atribui valori. Următorul exemplu va genera o eroare de compilare datorată încercării de a atribui unui parametru de intrare IN o anumită valoare:

```

PROCEDURE bonus(crestere INTEGER) IS
    bonus_minim CONSTANT INTEGER:= 100000;
BEGIN
    ...
    IF bonus_minim > creștere THEN

```

```

        crestere:=bonus_minim;
        -- in acest punct se va genera o eroare de compilare
    END IF;
    ...
END bonus;

```

Exemplul de mai jos ilustrează o procedură care are un parametru de intrare și un parametru de ieșire:

```

PROCEDURE calc_bonus(cod_angaj IN INTEGER, bonus OUT INTEGER) IS
    crestere INTEGER;
BEGIN
    SELECT sal * 0.30 INTO bonus FROM angajati
        WHERE cod = cod_angaj;
    crestere := bonus;
END calc_bonus;

```

Parametrul actual ce corespunde unui parametru formal OUT trebuie să fie o variabilă, și nu o constantă sau expresie. Prin urmare apelurile următoare sunt incorecte:

```

calc_bonus (101, 200000);
sau
calc_bonus (101, salariu + comision);

```

Ca și variabilele, un parametru formal OUT este inițializat cu valoarea Null. Prin urmare, înainte de a ieși dintr-un subprogram, în mod obligatoriu parametrilor formali OUT trebuie să li se atribuie o valoare. În caz contrar, parametrii actuali corespondenți acestora vor avea valoarea Null.

Un parametru IN OUT permite transferul unei valori în subprogramul apelat și returnează mediului apelant o altă valoare. În interiorul unui subprogram, un parametru IN OUT acționează ca o variabilă inițializată. Prin urmare, acestui parametru îi pot fi atribuite valori sau poate fi atribuit unor alte variabile. Cu alte cuvinte, un parametru formal IN OUT poate fi folosit ca o variabilă obișnuită.

9.14.1. Proceduri și funcții stocate

Subprogramele pot fi definite de orice utilitar Oracle care folosește limbajul PL/SQL. Utilitare precum Forms și Reports stochează aceste subprograme local, caz în care ele sunt accesibile numai aplicației și utilitarului respectiv. Pentru ca aceste subprograme să poată fi folosite de orice utilitar Oracle, ele trebuie stocate în baza de date. Astfel de proceduri sau funcții se numesc *proceduri* sau *funcții stocate*. Odată ce sunt compilate și stocate în dicționarul de date, procedurile și funcțiile devin obiecte ale bazei de date care pot fi accesate de orice număr de aplicații conectate la acea bază de date.

Acstea funcții și proceduri sunt stocate gata compilate. Deci, în momentul când sunt apelate, ele sunt încărcate și transmise motorului PL/SQL imediat. De asemenea, funcțiile și procedurile stocate beneficiază de capacitatea de a utiliza în comun memoria sistemului

Oracle. Doar o copie a oricărui subprogram trebuie să fie încărcată în memorie pentru mai mulți utilizatori.

9.14.2. Crearea, recompilarea, distrugerea și utilizarea subprogramelor

Crearea subprogramelor

Procedurile și funcțiile stocate pot fi create folosind comenzi SQL CREATE PROCEDURE și respectiv CREATE FUNCTION. O procedură sau funcție stocată nu poate fi modificată, ci trebuie să fie înlocuită cu o nouă definiție sau trebuie să fie distrusă. Opțiunea OR REPLACE (CREATE OR REPLACE PROCEDURE și CREATE OR REPLACE FUNCTION) permite înlocuirea definiției unei funcții sau proceduri fără a fi necesară distrugerea în prealabil a acesteia. Practic, această opțiune se folosește pentru a înlocui o versiune mai veche a unei proceduri cu o nouă versiune. Această înlocuire păstrează intacă toate privilegiile. În schimb, dacă distrugi procedura și o recreezi, privilegiile sunt distruse și trebuieesc re-acordate. Dacă comanda CREATE PROCEDURE sau CREATE FUNCTION este folosită fără opțiunea OR REPLACE pentru a crea o procedură sau funcție care există deja, va fi generat un mesaj de eroare.

Exemplul de mai jos creează funcția stocată select_manager ce returnează codul managerului unui salariat, iar în cazul în care salariatul este inexistent returnează valoarea -1.

```
CREATE OR REPLACE FUNCTION select_manager(v_cod IN NUMBER)
RETURN NUMBER IS
    v_manager    NUMBER(10) := -1;
BEGIN
    SELECT manager INTO v_manager FROM salariat
    WHERE cod_salariat = v_cod;
    RETURN v_manager;
EXCEPTION
    WHEN NO_DATA_FOUND THEN RETURN v_manager;
END select_manager;
```

Recompilarea subprogramelor

Procedurile și funcțiile existente pot fi recompilate folosind comenzi SQL ALTER PROCEDURE și respectiv ALTER FUNCTION. Recompilarea unei proceduri sau funcții nu modifică declarația sau definiția procedurii. Dacă recompilarea se încheie cu succes, procedura sau funcția devine validă iar la momentul execuției nu se mai recompilează. Aceasta duce la eliminarea procesărilor suplimentare și a erorilor de compilare la execuție.

Exemplu:

```
ALTER FUNCTION select_manager COMPILE;
```

Distrugerea subprogramelor

Pentru a distruge o procedură sau funcție se folosesc comenziile `DROP PROCEDURE` sau `DROP FUNCTION`.

Exemplu:

```
DROP FUNCTION select_manager;
```

Apelarea subprogramelor

Apelurile funcțiilor definite în blocuri PL/SQL și subprograme pot apărea în instrucțiuni procedurale, dar nu în comenzi SQL. De exemplu, următoarea comandă `INSERT` este ilegală:

```
INSERT INTO angajati(cod_salariat, nume, prenume,
                      data_nastere, manager)
VALUES(sal_seq.nextval, 'Ion', 'Călin', '11-JAN-67',
       select_manager (10));
```

Apelarea procedurilor sau funcțiilor se poate face în orice mediu în care este disponibil PL/SQL, precum SQL*Plus, Oracle Forms sau Oracle Reports. În SQL*Plus, procedurile și funcțiile pot fi apelate și în afara blocurilor PL/SQL, caz în care apelul va fi precedat de cuvântul `EXECUTE`, iar în cazul funcțiilor, atribuirea se va face într-o variabilă de legătură SQL*Plus. Exemplul următor ilustrează acest tip de apelare:

```
SQL> VARIABILE cod NUMBER
SQL> EXECUTE :cod:=select_manager (10);
```

O procedură sau funcție poate fi executată și de către alți utilizatori dacă aceștia au acest privilegiu. Privilegiul de execuție este acordat direct utilizatorilor și nu unor anumite roluri. Utilizatorul care are drepturi de execuție are implicit și drepturi asupra obiectelor accesate de procedura sau funcția respectivă, dar numai prin intermediul acesteia. Dacă nu se dorește ca utilizatorii să acceseze tabelele bazei de date în mod direct, se poate crea o aplicație în care aceștia să poată accesa tabelele bazei de date numai prin intermediul procedurilor sau funcțiilor. Acest lucru reprezintă o puternică facilitate ce duce la mărirea securității.

Vederea `USER_SOURCE` din dicționarul de date conține codul sursă al unei proceduri, funcții sau pachet stocat. Pentru a putea de exemplu să vizualizăm codul sursă al funcției `select_manager` vom folosi următoarea interogare:

```
SELECT text FROM user_source
WHERE name='select_manager' AND type='FUNCTION'
ORDER BY line;
```

Dacă se dorește ca utilizatorii să nu poată vizualiza codul sursă al unei proceduri sau funcții, se poate utiliza utilitarul Oracle Wrapper. Acest utilitar convertește codul sursă în format binar, fișierele rezultate având extensia *.plb.

Avantajele folosirii subprogramelor

Avantajele folosirii procedurilor și funcțiilor stocate sunt:

- Deoarece sunt stocate în baza de date pot furniza un control centralizat, sunt portabile și minimizează traficul de rețea între-un mediu client/server. Odată validate, ele pot fi folosite în mod repetat fără a fi necesară redistribuirea lor în rețea;
- Sunt lansate în execuție fără a se mai recompile;
- Deoarece funcțiile și procedurile au nume ele pot fi referite din alte unități de program, acest lucru permitând folosirea unui cod de mai multe ori fără a se mai rescrie;
- Prin utilizarea utilitarului PL/SQL Wrapper codul sursă poate deveni invizibil utilizatorului final prin scrierea blocurilor PL/SQL în format binar. Acest lucru duce la mărirea securității și permite protejarea informațiilor împotriva modificării;
- Accesul la baza de date se poate restrânge, utilizatorii neputând accesa datele decât prin intermediul procedurilor rezidente.

9.15. Pachete

Din punct de vedere literar un pachet reprezintă unul sau mai multe obiecte înșăurate într-o învelitoare. Un pachet în sine nu are nici o valoare, dar permite prezentarea unui produs într-o formă mai frumoasă. Un pachet Oracle are un înțeles similar, adică permite gruparea mai multor obiecte PL/SQL într-un tot unitar. Aceste obiecte pot fi constante, variabile, cursoare, funcții, proceduri și excepții. Cu toate că formatul unui pachet este asemănător cu formatul subprogramelor, spre deosebire de acestea, pachetul însuși nu poate fi apelat sau imbricat și nu își poate transfera parametrii.

Pachetele sunt formate din două părți: *specificația* (package specification) și *corpul pachetului* (package body).

- *Specificația* este partea pachetului care realizează interfața acestuia cu aplicația. Aici se declară variabilele, constantele, excepțiile, cursoarele și subprogramele (proceduri sau funcții) publice, adică obiectele care urmează a fi folosite în afara pachetului. Astfel, putem spune că specificația unui pachet conține declarații *publice*.
- *Corpul pachetului* conține detalii de implementare și declarațiile care nu pot fi văzute în afara pachetului. Declarațiile conținute de corpul unui pachet sunt declarații *private*. Corpul pachetului definește complet cursoarele, funcțiile și procedurile, implementând în acest fel specificația.

Uneori, corpul pachetului nu este necesar (nu există nici declarații locale și nici partea de implementare), deci poate lipsi. Acest lucru se întâmplă dacă specificația pachetului declară numai variabile, constante și excepții.

Corpul pachetului poate fi modificat sau înlocuit fără a fi necesară schimbarea specificației acestuia, adică schimbarea interfeței cu aplicația. Se poate astfel modifica definiția unui obiect din corpul pachetului, fără ca prin această Oracle să invalideze celealte obiecte care apelează obiectul respectiv sau care fac referire la el, cum s-ar întâmpla dacă ar modifica specificația pachetului.

Sintaxa specificației și a corpului unui program sunt prezentate în continuare.

Specificația pachetului:

```
PACKAGE nume IS
    declarații de tipuri și obiecte publice
    specificațiile subprogramelor publice
END [nume]
```

Corpul pachetului:

```
PACKAGE nume IS
    declarații de tipuri și obiecte private
    implementarea subprogramelor
[BEGIN
    instrucțiuni de inițializare]
END [nume]
```

9.15.1. Pachete stocate

În mod similar cu procedurile și funcțiile de sine stătătoare, pentru ca un pachet să poată fi utilizat de orice utilitar sau aplicație Oracle, este necesar ca acesta să fie stocat în baza de date după ce a fost în prealabil compilat. Astfel de pachete se numesc *pachete stocate*.

Specificația și corpul pachetului sunt stocate separat. Acest lucru face ca celelalte obiecte să depindă numai de specificație, nu și de corpul pachetului. Această separare permite modificarea definiției unui obiect din corpul pachetului, fără ca prin aceasta Oracle să invalideze celelalte obiecte care apelează obiectul respectiv sau care fac referire la el, cum s-ar întâmpla dacă s-ar modifica specificația pachetului.

9.15.2. Crearea, recompilarea, distrugerea și utilizarea pachetelor stocate

Crearea pachetelor

Specificația unui pachet se poate crea folosind comanda SQL CREATE PACKAGE, iar corpul unui pachet folosind comanda SQL CREATE PACKAGE BODY. Opțiunea OR REPLACE (CREATE OR REPLACE PACKAGE și CREATE OR REPLACE PACKAGE BODY) permite modificarea specificației sau corpului unui pachet fără a fi necesară distrugerea în prealabil a acestora.

În exemplul următor, se creează specificația pachetului gestiune_angajati ce cuprinde un tip de dată înregistrare, un cursor și două proceduri.

```
CREATE OR REPLACE PACKAGE gestiune_angajati IS
    TYPE tip_angaj IS RECORD(cod_angaj INTEGER, salariu REAL);
    CURSOR salariu_descresc RETURN tip_angaj;
```

```

PROCEDURE creștere_salariu(cod_sal INTEGER, crestere
INTEGER);
PROCEDURE concediaza_angajat (cod_sal NUMBER);
END gestiune_angajati;

```

Definițiile obiectelor publice declarate în specificația pachetului gestiune_angajati sunt conținute în corpul acestuia. Corpul pachetului se creează în modul următor:

```

CREATE OR REPLACE PACKAGE BODY gestiune_angajati IS
CURSOR salariu_descresc RETURN tip_angaj IS
    SELECT cod, sal FROM angajati ORDER BY sal DESC;
PROCEDURE creștere_salariu(cod_sal INTEGER, crestere
                             INTEGER) IS
    salariu_curent      INTEGER;
BEGIN
    SELECT sal INTO salariu_curent FROM angajati
    WHERE cod = cod_sal;
    UPDATE angajati SET sal = sal + crestere
    WHERE cod = cod_sal;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO erori
        VALUES(cod_sal, 'Nu există nici un angajat
                         cu acest cod');
    END creștere_salariu;
PROCEDURE concediaza_angajat (cod_sal NUMBER) IS
BEGIN
    DELETE FROM angajati WHERE cod = cod_sal;
    END concediaza_angajat;
END gestiune_angajati;

```

Pentru a face referire la obiectele unui pachet, trebuie folosită notația cu punct. Sintaxa generală este următoarea:

nume_pachet.nume_object

Așa cum s-a mai menționat, obiectele care sunt declarate în corpul pachetului nu pot fi utilizate decât în pachetul respectiv. De exemplu, codul PL/SQL din exteriorul pachetului nu poate face referire la nici una dintre variabilele care au fost declarate în corpul pachetului. Pentru ca variabila să fie vizibilă în exteriorul pachetului, aceasta trebuie declarată în partea de specificație și nu în corpul pachetului.

O observație foarte importantă este faptul că variabilele, constantele și cursoarele au un anumit domeniu de vizibilitate ce depinde de locul în care apare declarația. De exemplu, în cazul procedurilor sau funcțiilor, variabilele, constantele și cursoarele există numai pe durata apelului și își pierd valoarea atunci când execuția procedurii sau funcției se încheie. Dacă variabilele, constantele și cursoarele sunt declarate în specificația sau în corpul unui pachet, atunci valorile lor există pe toată durata sesiunii utilizatorului. În schimb, valorile lor se pierd atunci când pachetul este recompilat.

Recompilarea pachetelor

Pachetele existente pot fi recompilate folosind comanda SQL ALTER PACKAGE, având sintaxa:

```
ALTER PACKAGE nume_pachet COMPILE [PACKAGE|BODY]
```

În cazul în care după cuvântul cheie COMPILE apar opțiunile BODY, este recompilat doar corpul pachetului, iar în caz contrar (dacă apare opțiunea PACKAGE sau opțiunea este omisă) sunt recompilate atât specificația cât și corpul pachetului.

Distrugerea pachetelor

Distrugerea unui pachet se realizează cu comanda DROP PACKAGE, având sintaxa:

```
DROP PACKAGE nume_pachet [PACKAGE|BODY]
```

În cazul în care apare opțiunea BODY, este distrus doar corpul pachetului, iar în caz contrar (dacă apare opțiunea PACKAGE sau opțiunea este omisă) sunt distruse atât specificația cât și corpul pachetului.

Atunci când pachetul este distrus, toate obiectele dependente de acesta devin invalide. Dacă este distrus numai corpul unui pachet, nu și specificația acestuia, toate obiectele dependente rămân valide. În schimb, nu puteți apela nici una dintre procedurile și funcțiile rezidente declarate în specificația pachetului, până când nu recreați corpul pachetului.

Pentru a distruge un pachet, acesta trebuie să se găsească în schema dumneavoastră sau să posedăți privilegiul de sistem DROP ANY PROCEDURE.

Supraîncărcarea

Sistemul Oracle permite existența în același bloc, subprogram sau pachet a mai multor funcții sau proceduri cu același nume. Această tehnică este cunoscută sub numele de *supraîncărcare*. Supraîncărcarea se folosește atunci când se dorește executarea unei proceduri sau funcții de mai multe ori însă cu argumente de tipuri diferite. Oracle determină care dintre procedurile sau funcțiile cu același nume va fi apelată prin verificarea parametrilor formali ai acesteia.

Avantajele folosirii pachetelor stocate

Pachetele oferă următoarele avantaje:

- Funcțiile și procedurile stocate sunt organizate mai eficient prin gruparea acestora în funcție de anumite criterii;
- Pachetele permit supraîncărcarea procedurilor și funcțiilor, rezolvând conflictele de nume din cadrul unei scheme. Supraîncărcarea unei proceduri sau a unei funcții

înseamnă crearea în același pachet a mai multor proceduri sau funcții cu același nume, dar cu argumente diferite;

- Variabilele publice și cursoarele pachetelor persistă pe durata întregii sesiuni. Prin urmare, acestea se pot referi din orice mediu sau procedură;
- Gestionarea funcțiilor și a procedurilor stocate se realizează cu ușurință datorită facilității de a putea modifica corpul unui pachet fără a modifica și specificația acestuia;
- Se îmbunătățește securitatea funcțiilor și procedurilor stocate prin ascunderea codului sursă astfel încât utilizatorii nu pot avea acces la acesta;
- Privilegiul de executare al unui pachet include privilegiul de executare al tuturor funcțiilor și procedurilor conținute de acesta, prin urmare nu este necesară acordarea privilegiilor pentru fiecare funcție sau procedură în parte, permitând astfel acordarea de privilegii în mod eficient;
- Se îmbunătățește performanța prin încărcarea în memorie a întregului pachet în momentul primei apelări a acestuia. Prin urmare, apelările viitoare ale pachetului duc la îmbunătățirea timpului de răspuns datorită faptului că vor fi necesare mai puține operații de I/O.

Sfaturi în legătură cu utilizarea pachetelor stocate

Pentru a realiza o cât mai bună performanță la utilizarea pachetelor, în continuare prezentăm câteva sfaturi:

- În momentul creării pachetelor acestea trebuie să fie cât mai simple și generale pentru a putea fi reutilizate în aplicații viitoare;
- Evitați crearea pachetelor care reproduc funcționalitatea Oracle existente;
- Deoarece un pachet reflectă design-ul aplicației, creați corpul pachetului după ce ați conceput aplicația. În specificația pachetului trebuie să fie plasate numai acele obiecte care vreți să fie vizibile tuturor utilizatorilor;
- Pentru a reduce necesitatea recomplirii atunci când codul este modificat, plasați cât mai puține obiecte în specificația unui pachet. Modificarea corpului unui pachet nu necesită recomplirea procedurilor dependente. În schinib, modificarea specificației unui pachet face ca toate subprogramele rezidente care fac referire la pachetul respectiv să fie recomilate.

9.15.3. Pachete predefinite

Baza de date Oracle furnizează o serie de pachete predefinite ce ajută la realizarea aplicațiilor bazate pe PL/SQL. Numele și funcționalitatea celor mai importante pachete sunt prezentate în tabelul următor, pentru informații suplimentare privind funcțiile și procedurile conținute în pachetele utilizate mai frecvent urmând să consulta Anexa 7.

Nume pachet predefinit	Funcționalitate
DBMS_OUTPUT	Afișeați informații (în general, pe ecran) atunci când se execută un bloc sau un subprogram PL/SQL, ajutând la testarea și depanarea acestuia.

Nume pachet predefinitt	Functionalitate
DBMS_DDL	Recompilează proceduri, funcții sau pachete și analizează indecsi, tabele sau clusteri.
DBMS.Utility	Utilități DBA (analizează obiectele dintr-o anumită schema, verifică dacă serverul rulează în mod paralel și returnează timpul în sutini de secundă etc.).
DBMS_SESSION	Modifică anumite caracteristici ale sesiunii unui utilizator, setează rolul unui utilizator și reinicializează starea unui pachet.
DBMS_SQL	Accesează baza de date folosind SQL dinamic.
DBMS_TRANSACTION	Controlează și îmbunătășește performanțele tranzacțiilor bazei de date.
DBMS_PIPE	Operații de comunicare între două sau mai multe sesiuni conectate la aceeași instanță Oracle.
DBMS_ALERT	Avertizează o sesiune atunci când apare un eveniment în baza de date.
DBMS_LOCK	Permite folosirea exclusivă sau partajată a unei resurse.
DBMS_JOB	Permite execuția periodică a procedurilor stocate, oferind o cale de a gestiona procesele ce se execută în fundal.
DBMS_APPLICATION_INFO	Informă serverul de acțiunile întreprinse de o aplicație.
DBMS_SNAPSHOT	Conține proceduri pentru exploatarea instantaneelor.
DBMS_REFRESH	Administrează grupurile de reimprospătare.
DBMS_LOB	Permite accesul eficient la date de tip LOB.
UTL_FILE	Permite citirea și scrierea fișierelor text ale sistemului de operare.

9.16. Trigger (declanșatori)

Un *trigger (declanșator)* al bazei de date este un bloc PL/SQL stocat în baza de date, care este asociat unui tabel și care este executat în mod *implicit* ori de câte ori asupra tabelului asociat este lansată o anumită comandă DML (INSERT, UPDATE sau DELETE). Triggerul este invocat fie înainte, fie după executarea instrucțiunii DML, iar creatorul trigger-ului va specifica momentul executării declanșatorului în raport cu instrucțiunea DML corespunzătoare. O nouitate adusă de versiunea Oracle8 sunt trigger-ele INSTEAD OF, care se execută în locul instrucțiunilor DML efectuate asupra unei vederi; despre acestea vom vorbi ceva mai târziu, la sfârșitul acestei secțiuni, în ceea ce urmărează referindu-ne deocamdată numai la trigger-ele asociate tabelelor.

Deși prezintă similarități cu procedurile stocate, trigger-ele diferă de acestea prin următoarele:

- Un trigger este invocat în mod implicit la lansarea unei comenzi DML pe tabelul asociat, în timp ce o procedură este invocată în mod explicit. În plus, utilizatorul care lansează comanda DML nu are nevoie de nici un privilegiu pentru executarea trigger-ului. Practic triggerul este executat indiferent de utilizatorul care este conectat la baza de date sau de aplicația care este utilizată. Mai mult decât atât, utilizatorul respectiv nu are nici un control asupra execuției trigger-ului;
- Un trigger este stocat în baza de date ca text și compilat în momentul rulării. Din acest motiv este recomandat ca un trigger să fie alcătuit în principal din apeluri de proceduri stocate;
- Un trigger este asociat unui tabel, dar nu și unei vederi sau sinonim al tabelului. Practic, atunci când este executată o instrucțiune DML asupra unei vederi, sunt execuții declanșatori asociați tabelelor de bază ale vederii.

Scopurile pentru care sunt utilizați declanșatorii

Declanșatorii bazei de date garantează că o acțiune este executată în momentul când o tabelă este modificată. Declanșatorii suplimentează facilitățile standard Oracle prin furnizarea unui numitor elemente de gestiune a bazelor de date. De exemplu, un declanșator poate impune restricții asupra execuției operațiilor DML asupra unui tabel într-un anumit interval de timp, cum ar fi în afara orelor de program sau în week-end. Declanșatorii mai pot fi folosiți și în alte scopuri, cum ar fi:

- Impunerea respectării unor condiții complexe de integritate (referențială sau de comportament); dacă aceste constrângeri sunt prea complicate și nu pot fi definite prin intermediul constrângerilor de integritate ale tabelelor, ele pot fi implementate cu ajutorul trigger-elor;
- Popularea coloanelor redundante în cazul tabelelor denormalizate (de exemplu, calculul unui total);
- Transformarea într-un anumit format standard a datelor ce urmează a fi inserate în tabel; de exemplu, dacă se dorește conversia tuturor numerelor într-un format predefinit înainte de a fi inserate în tabel, trigger-ele pot furniza o conversie centralizată a acestora;
- Culegerea de informații statistice în legătură cu accesarea tabelelor;
- Întreținerea sincronizată a copiilor tabelelor situate în diferite noduri ale unei baze de date distribuite; deși există opțiuni de replicare Oracle, trigger-ele se pot folosi în mod extensiv pentru duplicarea datelor.

Declanșatorii trebuie să folosească numai atunci când este necesar. Utilizarea excesivă a declanșatorilor poate duce la o interdependență complexă care va face dificilă gestionarea unei aplicații mari. De exemplu, când un declanșator este invocat, acțiunile conținute de acesta pot invoca la rândul lor alt declanșator. Atunci când o comandă conținută de un declanșator invocă la rândul ei un alt declanșator, se spune că declanșatoarele sunt în *cascadă*. În acest sens trebuie acordată o foarte mare atenție folosirii declanșatorilor deoarece aceștia pot deveni *recursivi*. Acest lucru se întâmplă în cazul în care o comandă conținută de un declanșator invocă la rândul ei același declanșator, acesta executându-se până la terminarea resurselor de memorie.

Notă: Declanșatorii unei baze de date nu trebuie să confundați cu declanșatorii utilitarului Oracle Forms, aceștia din urmă fiind definiți, stocăți și execuți în mod diferit de către acest utilitar.

9.16.1. Declanșatorii și constrângerile de integritate

Majoritatea aspectelor privind integritatea datelor pot fi definite și impuse folosind constrângerile de integritate furnizate de către Oracle. Pentru condiții mai complexe, care nu pot fi definite în acest mod, se pot folosi trigger-ele bazei de date. Trigger-ele nu trebuie folosite însă pentru a simula constrângerile de integritate existente (NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK). Cel mai frecvent, trigger-ele se folosesc în următoarele cazuri:

- Pentru a impune integritatea referențială la modificarea sau ștergerea rândurilor din tabelele master prin atribuirea valorii Null sau unei valori implicate cheilor străine care fac referință la datele șterse sau modificate;
- Pentru a impune reguli de integritate referențială între tabele situate în diferite noduri ale unei baze de date distribuite;
- Pentru impunerea unor constrângeri complexe de comportament care nu pot fi definite utilizând constrângerea CHECK.

9.16.2. Sintaxa declanșatorilor și modul de creare

Un trigger este alcătuit din trei părți:

1. *evenimentul declansator* – instrucțiunea care a dus la invocarea declansatorului:
 - INSERT
 - UPDATE
 - DELETE
2. *condiția de declansare*
3. *corpuș declansatorului* – un bloc PL/SQL

Pentru crearea unui declansator al bazei de date asociat unei anumite tabele, se utilizează instrucțiunea CREATE TRIGGER. Sintaxa utilizată pentru a crea un declansator este următoarea:

```
CREATE [OR REPLACE] TRIGGER nume_declansator
{BEFORE|AFTER} eveniment_declansator ON nume_tabel
{REFERENCING {NEW AS|OLD AS} nume_calificativ}
[FOR EACH ROW]
[WHEN conditie]
[DECLARE
  Sectiune declarativa optională ce contine toate
  declaratiile tipurilor, variabilelor, constantelor si
  subprogramelor locale]
BEGIN
  Sectiunea executabila a declansatorului
[EXCEPTION
  Sectiune optională de tratare a exceptiilor]
END;
```

Următorul exemplu creează un declansator asociat tabelei salariaj:

```
CREATE OR REPLACE TRIGGER modificari_salariu
BEFORE INSERT OR UPDATE ON salariati --evenimentul declansator
FOR EACH ROW
WHEN (new.cod > 0) --restrictia de declansare
DECLARE
  dif_sal      NUMBER;
BEGIN
  dif_sal:=:NEW.sal-:OLD.sal;
  DBMS_OUTPUT.PUT_LINE('Salariu vechi: ' || :OLD.sal);
  DBMS_OUTPUT.PUT_LINE('Salariu nou: ' || :NEW.sal);
  DBMS_OUTPUT.PUT_LINE('Diferenta ' || dif_sal);
END;
```

După ce declanșatorul a fost creat, în momentul în care va fi lansată în execuție o comandă DML, ca de exemplu:

```
UPDATE salariati SET sal=sal+500.00
WHERE nrdept=10;
```

declanșatorul va fi executat în mod implicit. Prin urmare, pentru exemplul de mai sus, va fi afișat salariul vechi, salariul nou și diferența dintre cele două salarii pentru fiecare rând afectat de către comanda UPDATE. Pseudo-înregistrările :NEW și :OLD sunt utilizate de către declanșatoarele la nivel de rând pentru a accesa valorile noi și vechi ale coloanelor rândului curent procesat de către trigger. Mai multe detalii despre aceste pseudo-înregistrări vor fi prezentate ceva mai târziu, în secțiunea dedicată declanșatoarelor la nivel de rând.

Ca și în cazul procedurilor sau funcțiilor stocate, opțiunea OR REPLACE permite modificarea unui declanșator fără a fi necesară ștergerea în prealabil a acestuia.

În aceeași schemă nu pot exista doi declanșatori cu același nume. În schimb, numele declanșatorilor pot fi identice cu numele altor obiecte ale schemei, cum ar fi tabele, vederi sau proceduri. De exemplu, un tabel și un trigger pot avea același nume, dar este recomandabil să se evite această situație.

În continuare, pornind de la exemplul anterior, vom explica fiecare parte a unui declanșator precum și unele caracteristici ale acestora.

Evenimentul declanșator

Evenimentul declanșator este o comandă DML (INSERT, UPDATE sau DELETE) care face ca un declanșator să fie executat. În același declanșator al bazei de date poate fi inclusă orice combinație de evenimente declanșatoare, însă trebuie specificat cel puțin un eveniment. În exemplul de mai sus combinația de evenimente declanșatoare este:

```
INSERT OR UPDATE ON salariati
```

iar evenimentul care declanșează efectiv acest trigger este:

```
UPDATE ON salariati
```

Dacă în cazul operațiilor INSERT și DELETE condiția de declanșare nu poate face referință decât la întreaga înregistrare, în cazul operației UPDATE se poate face referire și la una sau mai multe coloane ale înregistrării. Prin urmare, în cazul comenzi UPDATE evenimentul declanșator al trigger-ului bazei de date poate fi definit sub forma:

```
UPDATE OF sal, nrdept ON salariati
```

În acest caz, orice comandă UPDATE care va modifica un alt câmp decât sal sau nrdept, nu va produce activarea declanșatorului.

Pentru a realiza un declanșator care să controleze operații DML multiple executate asupra aceleiași tabele dar totuși să execute blocuri de cod diferite, în funcție de instrucțiunea declanșatoare, se poate utiliza predicalele INSERTING, DELETING sau UPDATING. Pentru a înțelege modul de funcționare al acestor predicate, vom modifica

exemplul anterior astfel încât declanșatorul să execute blocuri diferite în funcție de instrucțiunea care a provocat declanșarea.

```

CREATE OR REPLACE TRIGGER modificari_salariu
BEFORE INSERT OR UPDATE ON salariati
FOR EACH ROW
WHEN (NEW.cod > 0)
DECLARE
    dif_sal      NUMBER;
BEGIN
    IF UPDATING THEN
        dif_sal:=:NEW.sal-:OLD.sal;
        dbms_output.put_line('Salariu vechi:'||:OLD.sal);
        dbms_output.put_line('Salariu nou:'||:NEW.sal);
        dbms_output.put_line('Diferenta'||dif_sal);
    END IF;
    IF INSERTING THEN
        dbms_output.put_line('Se inserează un nou angajat');
    END IF;
END;

```

Condiția de declanșare

Restricția de declanșare specifică o expresie booleană care trebuie să fie adevărată pentru ca declanșatorul să fie activat. Această condiție nu poate să fie o condiție PL/SQL, ci trebuie să fie o condiție SQL ce nu poate conține o subinterrogare. Spre deosebire de celelalte secțiuni ale declanșatorului, această secțiune poate lipsi. Restricția de declanșare este specificată cu ajutorul clauzei WHEN. În exemplul de mai înainte restricția de declanșare este:

NEW.cod > 0

În momentul în care clauza WHEN este inclusă, expresia este evaluată pentru fiecare înregistrare afectată. Dacă restricția de declanșare are valorile False sau Necunoscut, acțiunile conținute de un declanșator nu sunt executate. Evaluarea expresiei din clauza WHEN nu are nici un efect asupra execuției comenzi declanșatoare, adică comanda nu este anulată dacă expresia nu este adevărată.

De exemplu, pentru trigger-ul `modificari_salariu` de mai înainte, corpul declanșatorului nu va fi executat dacă noua valoare a câmpului cod are valoarea zero, Null sau negativă.

Clauza WHEN nu poate fi specificată decât pentru declanșatoare la nivel de rând (vezi secțiunea 9.16.3).

Corpul declanșatorului

Corpul declanșatorului este un bloc PL/SQL ce este executat atunci când se încearcă execuția unei comenzi cuprinse în evenimentul declanșator și când restricția de declanșare ia valoarea True (dacă aceasta există). Ca orice procedură stocată, corpul declanșatorului

poate conține comenzi SQL și PL/SQL, poate defini variabile, constante, curseare și exceptii, și poate apela alte proceduri stocate. În plus, pentru declanșatorii la nivel de rând (descriși în continuare) există câteva construcții speciale ce pot fi incluse în blocul PL/SQL:

- predicatele INSERTING, DELETING și UPDATING;
- nume de corelare pentru vechile și noile valori ale coloanelor rândului curent procesat de trigger;
- opțiunea REFERENCING.

Ultimele două construcții speciale sunt descrise în secțiunea ce tratează declanșatorii la nivel de rând.

Asupra corpului unui declanșator se aplică următoarele restricții:

- Nu poate conține o interogare sau o modificare a unui tabel aflat în plin proces de modificare în timp ce declanșatorul acționează. De exemplu, executarea unei instrucțiuni UPDATE asupra unui tabel nu va permite ca declanșatorul ce acționează ca rezultat al acestei instrucții să interogheze sau modifice tabelul respectiv;
- Nu poate conține o comandă de control a tranzacțiilor, cum ar fi COMMIT, ROLLBACK și SAVEPOINT;
- Nu poate conține nici o comandă DDL deoarece aceasta apelează în mod implicit comanda COMMIT înainte de se executa;
- Coloanele de tip LONG sau LONG ROW ale unei tabele nu pot fi folosite în interiorul unui declanșator decât dacă au fost convertite în prealabil în tipul CHAR sau VARCHAR2.

Dacă în timpul execuției unui trigger este declanșată o excepție, toate efectele corpului trigger-ului precum și comanda declanșatoare sunt derulate înapoi. Astfel, corpul unui trigger poate preveni execuția unei comenzi declanșatoare prin activarea unei excepții în interiorul acestuia. În acest context, excepțiile definite de utilizator sunt des folosite pentru a impune constrângeri de integritate sau autorizări complexe pentru securitatea datelor.

Dacă corpul declanșatorului necesită mai mult de 60 de linii de cod, este preferabil să împachetați codul într-o procedură stocată separat, și să apelați apoi această procedură din interiorul declanșatorului.

9.16.3 Tipuri de declanșatori

Așa cum am văzut în sintaxa comenzi CREATE TRIGGER, există anumite clauze folosite pentru a crea un anumit tip de declanșator. Din acest punct de vedere există mai multe tipuri de declanșatori:

- În funcție de numărul de ori de care este executat un declanșator
 - *la nivel de rând* (FOR EACH ROW) – declanșatorul se execută pentru fiecare rând în parte ce este afectat de instrucțiunea declanșatoare;
 - *la nivel de instrucțiune* – declanșatorul se execută o singură dată pentru instrucțiunea declanșatoare indiferent de numărul de linii afectate.

- În funcție de momentul în care declanșatorul este executat:
 - BEFORE – declanșatorul este executat înaintea executării instrucțiunii declanșatoare;
 - AFTER – declanșatorul este executat după executarea instrucțiunii declanșatoare

Prin urmare, tipurile posibile de declanșatori ce pot fi creați pentru o tabelă, în funcție de frecvența de execuție, instrucțiunea declanșatoare și momentul declanșării, sunt cele prezentate în tabelul de mai jos:

Nume	Funcție
BEFORE INSERT	ACTIONEAZĂ O SINGURĂ DATĂ, ÎNAINTEA EXECUȚIEI UNEI INSTRUCȚIUNI INSERT
BEFORE INSERT FOR EACH ROW	ACTIONEAZĂ ÎNAINTE DE CREAREA FIECĂREI NOI ÎNREGISTRĂRI
AFTER INSERT	ACTIONEAZĂ O SINGURĂ DATĂ, DUPĂ EXECUȚIA UNEI INSTRUCȚIUNI INSERT
AFTER INSERT FOR EACH ROW	ACTIONEAZĂ DUPĂ CREAREA FIECĂREI NOI ÎNREGISTRĂRI
BEFORE UPDATE	ACTIONEAZĂ O SINGURĂ DATĂ, ÎNAINTEA EXECUȚIEI UNEI INSTRUCȚIUNI UPDATE
BEFORE UPDATE FOR EACH ROW	ACTIONEAZĂ ÎNAINTE DE ACTUALIZAREA FIECĂREI ÎNREGISTRĂRI
AFTER UPDATE	ACTIONEAZĂ O SINGURĂ DATĂ, DUPĂ EXECUȚIA UNEI INSTRUCȚIUNI UPDATE
AFTER UPDATE FOR EACH ROW	ACTIONEAZĂ DUPĂ ACTUALIZAREA FIECĂREI ÎNREGISTRĂRI
BEFORE DELETE	ACTIONEAZĂ O SINGURĂ DATĂ, ÎNAINTEA EXECUȚIEI UNEI INSTRUCȚIUNI DELETE
BEFORE DELETE FOR EACH ROW	ACTIONEAZĂ ÎNAINTE DE ȘTERGerea FIECĂREI ÎNREGISTRĂRI
AFTER DELETE	ACTIONEAZĂ O SINGURĂ DATĂ, DUPĂ EXECUȚIA UNEI INSTRUCȚIUNI DELETE
AFTER DELETE FOR EACH ROW	ACTIONEAZĂ DUPĂ ȘTERGerea FIECĂREI ÎNREGISTRĂRI

În cazul în care sunt definiți mai mulți declanșatori de același tip, aceștia nu sunt executati într-o anumită ordine. Prin urmare trebuie avut în vedere faptul că integritatea bazei de date nu trebuie să depindă în acest caz de ordinea de execuție. Dacă ordinea de execuție este importantă, se recomandă scrierea într-un singur declanșator.

Declanșatorii la nivel de rând

Un declanșator la nivel de rând este executat de fiecare dată când un rând este afectat de instrucțiunea declanșatoare. De exemplu, dacă o comandă UPDATE actualizează 10 rânduri, declanșatorul este executat de 10 ori. Dacă instrucțiunea declanșatoare nu afectează nici o înregistrare, declanșatorul nu este executat de loc. Un declanșator la nivel de rând este creat prin specificarea opțiunii FOR EACH ROW. Dacă opțiunea nu este inclusă, declanșatorul este considerat a fi un declanșator la nivel de instrucțiune.

Declanșatorii la nivel de rând sunt foarte folositori atunci când acțiunile cuprinse în corpul declanșatorului depind de valorile rândurilor care sunt afectate de instrucțiunea declanșatoare.

Așa cum s-a mai menționat, declanșatorii la nivel de rând pot conține construcții speciale ce pot fi incluse în corpul acestuia: predicatele condiționale (**INSERTING**, **DELETING** și **UPDATING**), nume de corelare pentru vechile și noiile valori ale coloanelor rândului curent procesat de trigger, precum și opțiunea **REFERENCING**.

Nume de corelare pentru valorile noi și vechi ale coloanelor

În interiorul corpului unui declanșator, se pot accesa noiile și vechile valori ale coloanelor rândului curent procesat de trigger. În acest context există două nume de corelare pentru valorile fiecărei coloane ale tabelului ce este actualizat: un nume pentru vechile valori și un nume pentru noiile valori. Dacă o instrucție are nevoie de o valoare dintr-o înregistrare nouă sau actualizată, se utilizează :**NEW**. Dacă o instrucție are nevoie de valoarea unei coloane înainte ca ea să fi fost modificată, se utilizează :**OLD**. În funcție de tipul comenzii declanșatoare, anumite nume de corelare nu pot avea nici o semnificație. În cazul unei instrucții **INSERT**, vechile valori ale coloanelor au valoarea Null, în timp ce în cazul instrucției **DELETE** sunt Null noiile valori ale coloanelor.

Numele de corelare se pot utiliza atât în corpul declanșatorului cât și în clauzele opționale **WHEN** și **REFERENCING**. Cele două puncte care preced cuvintele cheie **NEW** și **OLD** sunt obligatorii atunci când acestea se utilizează în interiorul declanșatorului. Cele două puncte nu sunt permise atunci când se utilizează cuvintele cheie **NEW** și **OLD** în clauzele opționale **WHEN** și **REFERENCING**. Cuvintele cheie **NEW** și **OLD** nu pot fi utilizate cu coloane de tip **LONG** și **LONGRAW**.

Opțiunea **REFERENCING**

Opțiunea **REFERENCING** poate fi specificată în interiorul unui declanșator la nivel de rând pentru a evita conflictele de nume ce apar între numele de corelare și o tabelă a bazei de date ce este denumită **NEW** sau **OLD**. Conflictele de nume ce apar se datorează faptului că **NEW** și **OLD** nu sunt cuvinte rezervate, și prin urmare se pot crea tabele denumite **NEW** sau **OLD**, fapt care ar provoca ambiguități. În general această opțiune este destul de rar utilizată pentru că se evită crearea unor tabele denumite **NEW** sau **OLD**.

Presupunem că avem un tabel numit **new** ce conține câmpurile **camp1 (number)** și **camp2 (character)**. Următorul exemplu creează un declanșator asociat tabelului **new**, în care se evită conflictele de nume dintre numele tabelului și numele de corelare pentru valorile vechi și noi ale coloanelor rândului curent procesat de trigger:

```
CREATE OR REPLACE TRIGGER modificari_salariu
BEFORE ON new
REFERENCING NEW AS nou
FOR EACH ROW
BEGIN
    :nou.camp2 := TO_CHAR (:nou.camp1);
END;
```

În exemplul de mai sus, prin utilizarea opțiunii REFERENCING new AS NOU, noile valori ale coloanelor rândului curent procesat de trigger sunt referite utilizând numele de corelare nou, în loc de NEW.

Declanșatorii la nivel de instrucțiune

Un declanșator la nivel de instrucțiune este executat o singură dată pentru instrucțiunea declanșatoare, indiferent de numărul înregistrărilor afectate (chiar și în cazul în care nu este afectată nici o înregistrare). De exemplu, dacă o instrucțiune UPDATE actualizează 10 rânduri, declanșatorul este executat o singură dată.

Declanșatorii la nivel de instrucțiune sunt foarte folositori atunci când acțiunile cuprinse în corpul declanșatorului nu depind de valorile rândurilor care sunt afectate de instrucțiunea declanșatoare. De exemplu, dacă un declanșator realizează o constrângere de securitate referitoare la data și ora la care un utilizator poate modifica anumite tabele, atunci se va utiliza un declanșator la nivel de instrucțiune.

Declanșatorii BEFORE

Declanșatorii BEFORE sunt executati înaintea instrucțiunii declanșatoare. Acest tip de declanșator este în general folosit în următoarele situații.

- Pentru a modifica valorile coloanelor înaintea finalizării instrucțiunii declanșatoare INSERT sau UPDATE;
- Pentru a determina dacă instrucțiunea declanșatoare trebuie executată sau nu. În acest fel se pot elimina procesările inutile ale instrucțiunii declanșatoare.

Declanșatorii AFTER

Declanșatorii AFTER sunt executati după executarea instrucțiunii declanșatoare. Acest tip de declanșator este în general folosit în următoarele situații.

- Când instrucțiunea declanșatoare se dorește a fi executată indiferent de alte condiții;
- Pentru a executa acțiuni suplimentare față de acțiunile declanșatorului BEFORE, când acesta există.

Noile și vechile valori ale coloanelor rândului curent procesat de trigger sunt disponibile pentru declanșatorii la nivel de înregistrare, atât BEFORE cât și AFTER. Noile valori ale coloanelor pot fi modificate într-un declanșator la nivel de înregistrare BEFORE, însă nu pot fi modificate într-un declanșator la nivel de înregistrare AFTER deoarece instrucțiunea declanșatoare se execută înainte ca declanșatorul să fie invocat. Dacă un declanșator la nivel de înregistrare BEFORE schimbă valoarea :NEW.coloana, un declanșator la nivel de înregistrare AFTER invocat de aceeași instrucțiune declanșatoare vede modificările realizate de către declanșatorul BEFORE.

Declanșatorii la nivel de înregistrare AFTER sunt ceva mai eficienți decât declanșatorii BEFORE la nivel de înregistrare. Pentru un declanșator BEFORE la nivel de înregistrare,

blocul de date afectat de instrucțiunea declanșatoare trebuie citit logic de două ori: o dată pentru declanșator și o dată pentru instrucțiunea declanșatoare. În schimb, pentru un declanșator AFTER la nivel de înregistrare, blocul de date afectat de instrucțiunea declanșatoare trebuie citit logic o singură dată, atât pentru declanșator cât și pentru instrucțiunea declanșatoare.

Odinea de execuție a trigger-elor

Ordinea de execuție a trigger-elor depinde de tipul acestora. La apariția unei comenzi DML se execută următorul algoritm:

1. Se execută toți declanșatorii BEFORE la nivel de instrucțiune;
2. Pentru fiecare rând afectat de comanda DML:
 - a. Se execută toți declanșatorii BEFORE la nivel de rând;
 - b. Se blochează și modifică rândul afectat și se verifică constrângerile de integritate. Blocarea rămâne valabilă până în momentul în care tranzacția este permanentizată.
 - c. Se execută toți declanșatorii AFTER la nivel de rând;
3. Se verifică constrângerile de integritate amânate;
4. Se execută toți declanșatorii AFTER la nivel de instrucțiune.

Algoritmul de execuție a trigger-elor poate să prezinte mici variații sau să devină recursiv, în funcție de declanșatorii în cascadă și de tipul declanșatorilor lansați. O remarcă importantă este faptul că toate acțiunile și verificările efectuate ca urmare a apariției unei instrucțiuni declanșatoare trebuie să se deruleze cu succes. Dacă în interiorul unui trigger este generată o excepție care nu este tratată în mod explicit, triggerul este opriț și toate modificările efectuate de el sunt derulate înapoi.

Declanșatorii INSTEAD OF

O nouitate adusă de versiunea Oracle8 sunt declanșatorii INSTEAD OF. Spre deosebire de declanșatorii AFTER sau BEFORE, acestea se execută *în locul* instrucțiunii DML (INSERT, UPDATE, DELETE) specificate. Deci când o instrucțiune de actualizare activează un declanșator INSTEAD OF, în locul instrucțiunii originale de actualizare este executat declanșatorul respectiv. În plus, spre deosebire de declanșatorii AFTER sau BEFORE, declanșatorii INSTEAD OF se definesc pentru o *vedere*, nu pentru un tabel. Declanșatorii INSTEAD OF se pot defini atât pentru vederi relaționale cât și pentru vederi obiect. În mod implicit, declanșatoarele INSTEAD OF acționează la nivel de rând.

După cum s-a discutat în secțiunea 6.2.2, operațiile DML efectuate asupra unor vederi pot crea ambiguități în ceea ce privește actualizarea tabelelor de bază ale vederilor. În această categorie intră unele vederi bazate pe mai multe tabele, cât și vederi a căror interogare conține operatori cu mulțimi, funcții de grup, clauzele GROUP BY, CONNECT BY, START WITH și operatorul DISTINCT. Vederile obiect prezintă probleme suplimentare privind actualizarea, de exemplu vederile obiect sunt utilizate frecvent pentru a realiza o relație master/detaliiu, a căror actualizare creează ambiguități. Restricționarea operațiilor DML asupra tuturor acestor tipuri de vederi se datorează tocmai acestor ambiguități.

Declanșatorii INSTEAD OF furnizează o modalitate transparentă de actualizare a vederilor relaționale sau obiect care nu pot fi actualizate direct prin operații DML. Prin urmare, se pot scrie comenzi obișnuite INSERT, DELETE sau UPDATE asupra unei vederi, iar declanșatoarele INSTEAD OF acționează în vizibil utilizatorului în locul acestor comenzi, rezolvând problema ambiguității acestor comenzi.

Să considerăm următorul exemplu, în care vedere jucator_info este bazată pe tabelele jucator, club și turneu:

```

CREATE TABLE club(
cod_club          NUMBER(5)    PRIMARY KEY,
nume_club         VARCHAR2(10),
localitate        VARCHAR2(10));

CREATE TABLE jucator(
cod                NUMBER(7)    PRIMARY KEY,
nume              VARCHAR2(20),
cod_club          NUMBER(5)    REFERENCES club(cod_club));

CREATE VIEW jucator_info AS
SELECT j.cod, j.nume, c.cod_club, c.nume_club
FROM jucator j, club c
WHERE j.cod_club = c.cod_club;

CREATE OR REPLACE TRIGGER jucator_info_insert
INSTEAD OF INSERT ON jucator_info
FOR EACH ROW
DECLARE
    numar NUMBER(1);
BEGIN
    SELECT COUNT(*)
    INTO numar
    FROM DUAL
    WHERE EXISTS (SELECT 'x' FROM club
                  WHERE club.cod_club = :NEW.cod_club);
    IF numar = 0 THEN
        INSERT INTO club(cod_club, nume_club)
        VALUES (:NEW.cod_club, :NEW.nume_club);
    ELSE
        UPDATE club
        SET nume_club = :NEW.nume_club
        WHERE cod_club = :NEW.cod_club;
    END IF;
    SELECT COUNT(*)
    INTO numar
    FROM dual
    WHERE EXISTS
        (SELECT 'x' FROM jucator
         WHERE jucator.cod = :NEW.cod);
    IF numar = 0 THEN

```

```

    INSERT INTO jucator(cod, nume, cod_club)
    VALUES (:NEW.cod, :NEW.nume, :NEW.cod_club);
ELSE
    UPDATE jucator
    SET nume=:NEW.nume, jucator.cod_club=:NEW.cod_club
    WHERE cod = :NEW.cod;
END IF;
END;

```



Pentru fiecare rând inserat în vederea `jucator_info`, declanșatorul de mai sus testează mai întâi dacă există rândurile corespunzătoare în tabelele de bază. În funcție de rezultatul acestui test sunt inserate rânduri noi sau sunt actualizate cele existente.

9.16.4. Modificarea și stergerea unui declanșator

Modificarea unui declanșator constă în recomplilarca, activarea sau dezactivarea acestuia și se realizează prin comanda `ALTER TRIGGER`. Comanda are sintaxa:

```
ALTER TRIGGER nume_declansator {ENABLE|DISABLE|COMPILE}
```

Activarea și dezactivarea declanșatorilor

Un declanșator se poate afla în două stări: activat sau dezactivat. În momentul în care un declanșator a fost creat fără erori, el este în mod implicit activat, adică acesta se execută de fiecare dată când se îndeplinesc simultan două condiții: instrucțiunea declanșatoare este executată și restricția declanșatorului este adevărată. Dacă declanșatorul este dezactivat atunci declanșatorul nu se execută chiar dacă cele două condiții au fost îndeplinite. În anumite împrejurări se poate dori ca la un anumit moment declanșatorul să fie dezactivat temporar. Acest lucru se poate realiza cu ajutorul comenzii:

```
ALTER TRIGGER nume_declansator DISABLE
```

Pentru a activa un declanșator se va folosi aceeași comandă dar cu clauza `ENABLE`:

```
ALTER TRIGGER nume_declansator ENABLE
```

Deoarece un declanșator este asociat unui tabel, la nivelul tabelului se pot activa sau dezactiva toți declanșatorii asociați acestuia folosind comenziile:

```
ALTER TABLE nume_tabel DISABLE ALL TRIGGERS
ALTER TABLE nume_tabel ENABLE ALL TRIGGERS
```

Recompilarea unui declanșator

Pentru a evita recomplilarca unui trigger invalid în momentul rulării (care se face implicit), se poate folosi comanda:

```
ALTER TRIGGER nume_declansator COMPILE
```

266 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Această comandă recompilează în mod explicit un trigger indiferent dacă acesta este valid sau invalid. Recompilarea nu modifică definiția declanșatorului existent.

Ștergerea unui declanșator

În momentul în care un tabel este șters, în mod implicit sunt șterși și declanșatorii asociați acestuia. Dacă se dorește ștergerea unui anumit declanșator se folosește comanda:

```
DROP TRIGGER nume_declanșator
```

Capitolul 10

Orientare pe obiect în Oracle8

SGBD-urile *orientate pe obiect* sunt descendentele directe ale limbajelor de programare orientate pe obiect, precum SmallTalk sau Actor. Nevoia de stocare persistență, care a dus la apariția bazelor de date, a fost și motivul principal ce a dus la transformarea unor limbaje de programare orientate pe obiect în baze de date orientate pe obiect. Paradoxal, unul dintre argumentele generale în favoarea sistemelor orientate pe obiect este faptul că ele furnizează o modelare semantică mai puternică decât modelele de esență relațională, precum modelul entitate-legătură. Pe de altă parte însă, unul dintre cele mai puternice argumente împotriva bazelor de date orientate pe obiect este faptul că implementarea și limbajul lor de acces conțin elemente de nivel foarte scăzut (în genul pointerilor din C/C++). În esență, SGBD-urile orientate pe obiect recurg la un mod de interogare navigațional sau procedural, care prezintă similarități foarte puternice cu sistemele de gestiune pre-relaționale, precum sistemul ierarhic și rețea, spre deosebire de sistemele relaționale bazate pe SQL, unde modul de interogare este în general descriptiv și declarativ.

O caracteristică fundamentală a modelării orientate pe obiect este caracterul *natural* al acesteia, provenit din nivelul său ridicat de abstractizare. Cu alte cuvinte, lucruri reale pot fi modelate ca entități întregi, creând astfel o corespondență mai apropiată de lumea reală. De exemplu, într-o bază de date orientată pe obiect poate exista un obiect complex AUTOMOBIL, pe când într-o bază de date relațională datele despre un automobil vor fi stocate în mai multe tabele precum MARCA, MODEL, MOTOR, COMPONENTE, etc. rezultate în urma normalizării.

Pe scurt, care ar fi cele mai importante avantaje furnizate de tehnologia orientată pe obiect, în special în raport cu tehnologia relațională, dar și în general? Deși o parte au fost deja evidențiate până acum, le punctăm pe scurt pe cele mai importante:

- *modelarea* – la un nivel înalt de abstractizare;
- *reutilizarea* – codul poate fi reutilizat foarte ușor, mărindu-și astfel ciclul de viață;
- *complexitatea* – nu există limite pentru tipurile de date, putem avea date multimedia, tabele imbricate;
- *extensibilitatea* – există tipuri de date definite de utilizator care să reflecte exact modelul dorit;
- *performanța* – mai mare decât la sistemele bazate pe modelul relațional (deși greu de determinat, în mare parte ea depinzând de proiectarea logică și fizică a bazei de date, la segmente de aplicație compatibile s-a observat prin testare o mai mare apropiere de cerințele de performanță impuse de aplicație dacă s-a folosit Oracle8 orientat obiect decât în cazul folosirii doar a sistemului relațional).

10.1. Oracle8 și modelul relațional obiectual

Corporația Oracle nu abandonează rădăcinile relaționale ale bazelor sale de date odată cu apariția versiunii Oracle 8. Această versiune asigură în continuare toată tehnologia relațională, introducând în plus multe dintre caracteristicile tehnologici orientate pe obiect. De aceea

Oracle 8 poate fi numit un *SGBD relational obiectual (SGBDRO)*. Un astfel de SGBD este o fuziune a tehnologiilor relaționale și orientate pe obiect, permitând utilizatorilor să integreze gradat tehnologia orientată pe obiect cu aplicațiile relaționale deja existente.

Unul dintre marile avantaje ale unui SGBDR cu interfață orientată pe obiect, precum Oracle8, este faptul că programele de aplicație orientate pe obiect, scrise de exemplu în C++, pot comunica direct cu partea orientată pe obiect a SGBDR-ului. În caz contrar, legătura dintre modelul relațional și cel orientat pe obiect (care implică operații de descompunere și compunere) ar fi trebuit să fie făcută dinamic, în cadrul codului programului aplicație.

Așași, în Oracle8 utilizatorii pot defini tipuri adiționale de date – specificând atât structura datelor cât și tipurile de operații care pot fi efectuate asupra acestora – și pot folosi aceste tipuri în modelul relațional. Ca o consecință, în Oracle8 se pot utiliza obiecte în calitate de coloane ale unui tabel relațional sau pot fi definite tabele obiect, create după structura unui tip de obiect definit anterior. În acest fel, Oracle 8 stochează datele într-un mod natural, permitând în același timp și aplicațiilor extragerea acestora într-un mod natural. În plus, Oracle 8 furnizează noi tipuri de date care ușurează lucrul cu date complexe precum imagini, texte, grafice, video-clipuri, etc. În fine, în Oracle 8 se pot defini vederi obiect care reunesc într-un singur obiect datele din mai multe tabele, facilitând astfel combinarea modelului relațional cu cel obiectual.

În concluzie Oracle 8 facilitează integrarea dezvoltării orientate pe obiect cu bazele de date relaționale prin furnizarea unor facilități precum:

- **Tipuri de date definite de utilizator (opțiunea obiect):** Oracle 8 permite definirea de *tipuri obiect* și *tipuri colecție* (tipul VARRAY, tipul tabel). În plus, există un tip predefinit REF (referință de obiect) care permite ca la un obiect să se poată face referire într-un alt obiect sau tabel relațional.
- **Tipuri predefinite pentru obiecte de dimensiuni mari (LOB – large objects):** Oracle 8 include, pe lângă tipurile de bază deja cunoscute din versiunile anterioare, și tipuri de date mari (BLOB, CLOB, NCLOB, BFILE).
- **Vederi obiect.** O vedere obiect constituie o colecție de tabele normalizate reunite într-un tot unitar, permitând manipularea datelor din acestea ca pe niște obiecte.

10.2. Opțiunea obiect în Oracle 8

Oracle8 permite definirea de către utilizator a unor tipuri de date complexe care să reflecte realitatea ce trebuie modelată. Aceste tipuri pot fi folosite pentru crearea schemei bazei de date. În plus, aplicațiile pot accesa aceste date, fără a cunoaște detaliu privind modul lor de implementare.

Pentru a ilustra acest lucru să luăm următorul exemplu. Să considerăm că activitatea de vânzare a unei firme este organizată în comenzi. O comandă este făcută de un client și poate conține unul sau mai multe articole în diferite cantități, data comenzi, data de livrare, adresa de livrare, etc. În plus, firma are nevoie de anumite informații pe care aplicația trebuie să le obțină dinamic – de exemplu valoarea totală a comenziilor deja livrate sau care au mai rămas de livrat.

Pentru a modela această situație, în Oracle 8 poate fi definit *un tip obiect*, care servește ca un cadru pentru toate comenziile. Un tip obiect specifică elementele, numite atribute,

care constituie structura de date ce corespunde unei comenzi. Unele dintre atribute, precum lista articolelor comandate, pot fi la rândul lor obiecte structurate. Tipul obiect specifică de asemenea operațiile, numite metode, care pot fi executate asupra unei unități de date, precum determinarea valorii totale a unei comenzi.

Odată ce a fost creat acest tip obiect, pot fi create comenzi care corespund acestui cadru și pot fi stocate în coloanele unui tabel, tot la fel ca alte date precum numerele, sirurile de caractere sau datele calendaristice. În plus, comenziile pot fi stocate în *tabele obiect*, unde fiecare rând corespunde unei comenzi, iar coloanele sunt atributele comenzi.

Cum structura logică și comportamentul datelor corespunzătoare unei comenzi sunt definite în schema bazei de date, nu este necesar ca aplicația să cunoască detaliile acestora sau să reflecte schimbările lor. În plus, Oracle8 folosește informațiile despre tipurile obiect din schema bazei de date pentru a crește eficiența transmisiei. Dacă o aplicație client cere de la server datele despre comandă, ea poate primi toate datele într-o singură transmisie, aplicația putând apoi naviga prin datele respective, fără a-i fi necesară cunoașterea detaliilor de implementare și stocare și fără a fi necesară o nouă transmisie de la server.

Oracle furnizează un număr de facilități pentru folosirea tipurilor de date definite de către utilizator în programe de aplicație scrise în C/C++. Precompilatorii Oracle Pro*C/C++ (vezi documentația Oracle [26]/Application Development/Pro*C/C++ Precompiler Programmer's Guide) permit folosirea tipurilor definite de către utilizator în Programe scrise în C și C++, efectuând atât o verificare la compilare a acestor tipuri de date cât și convertirea lor în tipuri de date din C. *Oracle Call Interface (OCI)* (vezi documentația Oracle [26]/Application Development/Oracle Call Interface Programmer's Guide) furnizează facilități pentru accesarea și manipularea obiectelor din Oracle 8. *Object Type Translator (OTT)* (vezi documentația Oracle [26]/Application Development/Pro*C/C++ Precompiler Programmer's Guide) este un program care generează automat declarații în C pentru tipuri de date obiect definite în Oracle 8.

Tipurile de date ce pot fi definite de către utilizator în Oracle 8 se împart în următoarele două categorii:

- *tipuri obiect*;
- *tipuri colecție*.

Tipurile definite de utilizator pot conține tipuri de bază sau tipuri definite anterior. Fac excepție tipurile LONG, LONG RAW, ROWID și %TYPE.

Tipurile de date create de utilizator sunt obiecte ale schemei în care sunt create. Pentru crearea, modificarea și distrugerea acestor tipuri de date se folosesc comenziile CREATE TYPE, ALTER TYPE și respectiv DROP TYPE. Crearea unui tip de date nu presupune și alocarea de spațiu de stocare.

10.2.1. Tipul obiect

Tipurile obiect sunt abstracții ale entităților din lumea reală. Un tip obiect este un obiect din schema unui utilizator, care cuprinde următoarele trei componente:

- Un nume, care trebuie să identifice în mod unic tipul de obiect în cadrul schemei utilizatorului;
- *Atribute*, care alcătuiesc *structura* obiectului. Acestea pot avea tipuri de date predefinite sau alte tipuri definite de către utilizator. Atributele pentru un tip obiect sunt similare câmpurilor pentru o înregistrare sau coloanelor pentru un tabel. Spre deosebire însă de coloanele unui tabel, în Oracle 8 nu este posibilă definirea de constrângeri pe atributele unui tip obiect;
- *Metode*, care alcătuiesc *comportamentul* obiectului. Acestea sunt funcții sau proceduri scrise în PL/SQL și stocate în baza de date sau scrise într-un limbaj ca C și stocate extern. Metodele operează asupra atributelor obiectului și constituie operațiile care pot fi efectuate asupra unui obiect de acel tip.

Tipul obiect este construcția de bază a modelului orientat pe obiect, este şablonul după care se definesc obiecte. Tipul obiect în Oracle 8 este echivalentul unei clase din C++. Un obiect este o *instanță* a tipului obiect, o unitate structurată de date care respectă acest şablon. Similar cu o linie dintr-o tabelă, un obiect este data, iar tipul obiect este structura.

Exemplul care urmează ilustrează definirea unui tip obiect ale cărui atribuite au tipuri predefinite (*tip_adresa*), definirea unui tip obiect în care un atribut este de un tip obiect definit anterior (*tip_angajat*) și folosirea acestuia din urmă într-o coloană dintr-un tabel relational.

```

CREATE TYPE tip_adresa AS OBJECT(
    strada          VARCHAR2(20),
    numar           VARCHAR2(5),
    oras            VARCHAR2(20),
    tara             VARCHAR2(20));

CREATE TYPE tip_angajat AS OBJECT(
    cod_personal    NUMBER(9),
    nume            VARCHAR2(20),
    prenume          VARCHAR2(20),
    salariu         NUMBER(8),
    adresa          tip_adresa);

CREATE TABLE profesor(
    angajat        tip_angajat,
    titlu           VARCHAR2(10),
    functie         VARCHAR2(20),
    CONSTRAINT pk_cod PRIMARY KEY(angajat.cod_personal));

```

Deoarece în Oracle 8 nu este posibilă definirea de constrângeri pe atributele unui tip obiect, acestea sunt definite în momentul creării unui tabel. În exemplul de mai sus a fost definită constrângerea de cheie primară *pk_cod* care impune unicitatea codului unui angajat.

Ca și coloanele dintr-o tabelă, numele atributelor trebuie să fie unice în cadrul unui tip obiect, dar pot fi reutilizate de alte tipuri obiect. Tot ca și coloanele dintr-o tabelă, ele pot fi accesate prin numele lor folosind notația cu punct: *obiect.atribut*. În plus, pentru a

selecta valoarea unui atribut al unui obiect dintr-un tabel trebuie folosit un alias al tabelului. De exemplu:

```
SELECT p.angajat.nume, p.angajat.prenume FROM profesor p
WHERE p.angajat.cod_personal < 100;
```

10.2.2. Constructor

Atât atributele obiectului cât și metodele acestuia sunt specificate în declarația unui tip obiect. În plus, Oracle8 creează cel puțin o metodă pentru fiecare tip obiect, chiar dacă dezvoltatorul nu specifică nici o metodă ca parte a declarației tipului obiect respectiv. Această metodă este denumită *constructor*, adică o metodă care construiește un nou obiect după specificația tipului obiect. Ea are întotdeauna același nume ca tipul obiect, iar parametrii săi au aceleași tipuri cu atributele tipului obiect. Metoda constructor este o funcție, care întoarce ca valoare noul obiect.

De exemplu, expresia

```
adresa('CALEA VICTORIEI', '25B', 'BUCURESTI', 'ROMANIA')
```

reprezintă un obiect cu următoarele atribute:

strada	'CALEA VICTORIEI'
numar	'25B'
oras	'BUCURESTI'
tara	'ROMANIA'

Constructorul unui tip obiect poate fi folosit pentru popularea tabelelor relaționale ce conțin coloane de acel tip obiect, de exemplu:

```
INSERT INTO profesor(angajat, titlu, functie)
VALUES (tip_angajat(111,'GEORGESCU','BOGDAN',1000,
tip_adresa('TOPORASI','17','BUCURESTI','ROMANIA')),
'DOCTOR','PROFESOR');
```

10.2.3. Metode

În declarația unui tip obiect se pot specifica zero sau mai multe *metode*. Acestea se mai numesc și *metodele membre* ale tipului obiect, fiind specificate folosind cuvântul cheie MEMBER. Așa cum am menționat anterior, aceste metode sunt simple proceduri sau funcții PL/SQL care operează asupra atributelor tipului obiect. Aceste atribute pot fi folosite în cadrul definiției metodei, fără a apărea în mod explicit ca parametri ai acesteia în momentul apelării. La fel ca orice funcție sau procedură, o metodă poate avea zero sau mai mulți parametri. Tot la fel ca orice funcție sau procedură PL/SQL, trebuie creată o interfață, numită *specificația* metodei, și o implementare, numită *corpul* acesteia. În Oracle specificația metodei este făcută cu ajutorul construcției CREATE TYPE, iar implementarea cu ajutorul construcției CREATE TYPE BODY.

Exemplul următor recreează tipul obiect `tip_angajat`, adăugându-i metoda `salariu_miu`.

Notă: Un tip de dată nu poate fi șters sau recreat dacă este folosit în tabele sau alte tipuri de date. Prin urmare, mai întâi trebuie șters tabelul `profesor`, recreat tipul obiect `tip_angajat` cu ajutorul exemplului de mai jos iar în final recreat tabelul `profesor`.

```

CREATE OR REPLACE TYPE tip_angajat AS OBJECT(
cod_personal      NUMBER(9),
nume              VARCHAR2(20),
prenume            VARCHAR2(20),
salariu           NUMBER(8),
adresa             tip_adresa,
MEMBER FUNCTION categorie_salariu (limita1 IN NUMBER,
imita2 IN NUMBER) RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES (categorie_salariu, WNDS,
WNPS, RNPS, RNDS));

CREATE OR REPLACE TYPE BODY tip_angajat AS
MEMBER FUNCTION categorie_salariu(limita1 IN NUMBER,
imita2 IN NUMBER) RETURN NUMBER IS
    categorie NUMBER(1);
BEGIN
    IF salariu < limita1 THEN categorie := 1;
    ELSIF salariu < imita2 THEN categorie := 2;
    ELSE categorie := 3;
    END IF;
    RETURN categorie;
END categorie_salariu;
END;

```

Pentru ca o funcție membră a unui tip obiect să poată fi apelată în comenzi SQL, trebuie declarat nivelul de "puritate" al acesteia (adică măsura în care aceasta nu produce efecte secundare) folosind directiva PRAGMA RESTRICT_REFERENCES în specificația tipului obiect. Această directivă are ca parametrii, alături de numele funcției membre, una sau mai multe dintre următoarele valori: WNDS, WNPS, RNDS, RNPS, având următoarele semnificații:

- WNDS înseamnă "writes no database state", adică nu modifică tabelele bazei de date, mai exact funcția membră nu conține comenzi DML (INSERT, UPDATE, DELETE);
- WNPS înseamnă "writes no package state", adică funcția membră nu modifică valorile variabilelor din pachete;
- RNDS înseamnă "reads no database state", adică nu interoghează tabelele bazei de date, mai exact funcția membră nu conține comanda SELECT;
- RNPS înseamnă "reads no package state", adică funcția membră nu citește valorile variabilelor din pachete.

Dacă funcția membră `categorie_salariu` ar fi fost definită fără directiva `PRAGMA RESTRICT_REFERENCES` cu parametrul `WNDS`, atunci interogarea:

```
SELECT p.angajat.categorie_salariu(1000,2000) FROM profesor p;
```

ar fi returnat o eroare

"ORA-06571: Function CATEGORIE_SALARIU does not guarantee not to update database".

Notă: În cazul în care metoda membră a unui tip obiect este o procedură, nu se pun probleme nivelului de puritate deoarece nu poate fi apelată din comenzi SQL, ci numai de sine stătător.

O metodă a unui tip obiect poate fi folosită de către orice obiect de acel tip. De exemplu, dacă `x` și `y` sunt variabile PL/SQL, `x` de tipul `tip_angajat`, iar `y` de tip numeric, atunci este permisă atribuirea:

```
y := x.categorie_salariu(1000, 2000);
```

După această atribuire `y` va avea valoarea 1 dacă salariul angajatului reprezentat de variabila `x` este sub 1000, 2 dacă salariul este între 1000 și 2000 și 3 dacă salariul este peste 2000.

10.2.4. Metode de comparare

Metodele pot fi folosite pentru compararea obiectelor. Oracle oferă facilități pentru a compara două unități de date de un anumit tip predefinit (de exemplu, numere) și pentru a determina dacă unul este mai mare, egal sau mai mic decât celălalt. Pe de altă parte însă, Oracle nu poate compara două obiecte aparținând unui tip arbitrar definit de către utilizator, decât dacă cel care definește tipul respectiv oferă sistemului o ghidare în acest sens. Această ghidare se realizează prin definirea unor metode de comparare. Oracle furnizează două moduri de a defini o relație de ordine între obiectele de un anumit tip obiect: metode MAP și metode ORDER.

Metodele MAP folosesc capacitatea sistemului Oracle de a compara tipuri scalare predefinite. Acest tip de metodă utilizează atributele obiectului căruia îi este atașată pentru a calcula o valoare scalară. Din acest motiv, metodele MAP nu necesită parametri. Să presupunem, de exemplu, că vrem să definim un obiect dreptunghi, cu atributele lungime și latime. Atunci se poate defini o metodă MAP care returnează un număr, și anume aria dreptunghiului, adică produsul celor două attribute. Oracle poate compara două dreptunghiuri prin compararea ariilor lor.

```
CREATE OR REPLACE TYPE dreptunghi AS OBJECT(
    lungime NUMBER(10),
    latime NUMBER(10),
```

```

MAP MEMBER FUNCTION arie RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(arie, WNDS, WNPS, RNPS, RNDS));

CREATE OR REPLACE TYPE BODY dreptunghi AS
MAP MEMBER FUNCTION arie RETURN NUMBER IS
BEGIN
    RETURN lungime * latime;
END arie;
END;

```

Metodele ORDER sunt mai generale. O metodă ORDER permite ordonarea pentru obiectele unde proiecția într-un scalar este dificilă sau imposibilă. O astfel de metodă folosește logica sa internă pentru a compara două obiecte de un anumit tip obiect și returnează o valoare care codifică ordinea relației, de exemplu, poate returna -1 dacă primul obiect este mai mic, 0 dacă obiectele sunt egale și 1 dacă primul este mai mare. Ca toate metodele, și metodele ORDER acceptă ca argument implicit o instanță a tipului căruia îi sunt atașate; acest obiect este referit în interiorul metodei folosind cuvântul cheie SELF.

Să presupunem, de exemplu, că vrem să comparăm două adrese care conțin, ca mai sus, strada, numărul, orașul și țara, întâi după numele țării, apoi după numele orașului, străzii și în cele din urmă după număr. Acest lucru se poate face dacă adăugăm o metodă ORDER la definiția tipului obiect tip_adresa definit mai sus:

```

CREATE OR REPLACE TYPE tip_adresa AS OBJECT(
strada          VARCHAR2(20),
numar           VARCHAR2(5),
oras            VARCHAR2(20),
tara             VARCHAR2(20),
ORDER MEMBER FUNCTION ordine(a IN tip_adresa) RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(ordine, WNDS, WNPS, RNPS, RNDS));

CREATE OR REPLACE TYPE BODY tip_adresa AS
ORDER MEMBER FUNCTION ordine(a IN tip_adresa) RETURN NUMBER
NUMBER IS
    retval NUMBER(2);
BEGIN
IF SELF.tara < a.tara THEN retval := -1;
ELSIF SELF.tara > a.tara THEN retval := 1;
ELSE
    IF SELF.oras < a.oras THEN retval := -1;
    ELSIF SELF.oras > a.oras THEN retval := 1;
    ELSE
        IF SELF.strada < a.strada THEN retval := -1;
        ELSIF SELF.strada > a.strada THEN retval := 1;
        ELSE

```

```

        IF SELF.numar < a.numar THEN retval := -1;
        ELSIF SELF.numar > a.numar THEN retval := 1;
        ELSE retval := 0;
        END IF;
    END IF;
END IF;
RETURN retval;
END ordine;
END;

```

Definițiile tipurilor obiect pot conține o metodă MAP sau o metodă ORDER, dar nu pe amândouă.

Dacă un tip obiect nu conține nici o metodă de comparare, Oracle nu poate determina dacă un obiect de acel tip este mai mare sau mai mic decât altul de același tip. De exemplu, dacă x și y sunt variabile PL/SQL de tipul tip_angajat definit mai sus, atunci o comparație de tipul $x < y$ va genera un mesaj de eroare, cum ar fi "can not order objects without MAP or ORDER method". La fel se va întâmpla și dacă se va încerca ordonarea rezultatelor unei interogări (folosind clauza ORDER BY) după o coloană de un tip obiect care nu are definită o metodă de comparare.

Pentru exemplificare, în absența unei metode MAP sau ORDER următorul bloc PL/SQL va genera o astfel de eroare:

```

DECLARE
    d1 dreptunghi;
    d2 dreptunghi;
BEGIN
    d1:=dreptunghi(20,50);
    d2:=dreptunghi(10,90);
    IF d1 > d2 THEN
        DBMS_OUTPUT.PUT_LINE('Arie d1 > Arie d2');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Arie d1 < Arie d2');
    END IF;
END;

```

Pe de altă parte însă, chiar în absența metodelor de comparare, Oracle poate determina dacă două obiecte de un anumit tip sunt egale sau nu, comparându-le atributele corespunzătoare. Compararea se face în modul următor:

- Dacă toate atributele sunt ne-nule și egale, Oracle raportează că cele două obiecte sunt egale
- Dacă există un atribut pentru care cele două obiecte au valori ne-nule ne-egale, Oracle raportează că cele două obiecte nu sunt egale.
- Dacă nici unul din cazurile de mai sus nu este adevărat, atunci Oracle raportează că cele două obiecte nu pot fi comparate.

10.2.5. Tabele obiect

Un *tabel obiect* este un fel special de tabel care conține obiecte, furnizând o perspectivă relațională asupra atributelor acestor obiecte. Tabelele obiect sunt create cu ajutorul comenzi CREATE TABLE cu următoarea sintaxă simplificată:

```
CREATE TABLE nume_tabel OF tip_object;
```

De exemplu, următoarea comandă SQL definește un tabel obiect pentru obiecte de tipul *tip_angajat* definit mai înainte:

```
CREATE TABLE angajat OF tip_angajat;
```

Oracle permite ca un tabel să fie văzut în două moduri:

- Un tabel cu o singură coloană (tabel obiect), în care fiecare rând este un obiect de tipul obiect respectiv (*tip_angajat* în cazul nostru);
- Un tabel cu mai multe coloane, în care fiecare dintre atributele tipului obiect respectiv (*tip_angajat* în cazul nostru) ocupă o coloană.

De exemplu, asupra tabelului angajat se pot executa următoarele comenzi SQL:

```
INSERT INTO angajat VALUES
(111, 'GEORGESCU', 'BOGDAN', 1000, tip_adresa('TOPORASI',
'17', 'BUCURESTI', 'ROMANIA'));
```

```
SELECT VALUE(a) FROM angajat a
WHERE a.nume = 'GEORGESCU';
```

```
SELECT a.nume, a.prenume, a.adresa.oras FROM angajat a
WHERE a.nume = 'GEORGESCU';
```

Prima comandă de mai sus inserează un rând în tabelul angajat ca într-un tabel cu mai multe coloane. A doua comandă selectează date din tabelul angajat ca dintr-un tabel cu o singură coloană. În schimb, ultima comandă selectează datele din tabelul angajat ca dintr-un tabel cu mai multe coloane. Remarcă că pentru selectarea valorii unui atribut dintr-un tabel obiect trebuie folosit un alias și notația cu punct. Pentru a selecta toate valorile unui rând dintr-un tabel obiect se folosește funcția VALUE ce are ca argument alias-ul tabelului obiect.

În cele ce urmează, pentru obiectele care apar în tabele obiect vom folosi denumirea de *obiecte rând*, iar pentru obiectele care apar doar în coloanele unui tabel sau ca atrbute ale altor obiecte denumirea de *obiecte coloană*.

10.2.6. Tipul REF (referință)

În modelul relațional, relațiile mulți-la-unu erau exprimate prin cheile străine. Modelul relațional obiectual folosit de Oracle 8 furnizează o modalitate mai eficientă de exprimare a unei relații mulți-la-unu între obiectele rând. Oracle 8 atribuie fiecărui obiect rând creat un identificator unic, numit *identificatorul obiectului* (*Object Identifier, OID*). Acest identificator

este unic în cadrul sistemului, chiar și în afara Oracle. Oracle nu furnizează nici un fel de documentație privind structura internă a acestui identificator al obiectului.

Identifierul obiectului permite ca la acel obiect rând să se facă referire din alte obiecte sau din tabele relaționale. Această referință este reprezentată de un tip predefinit numit REF. Acesta conține o referință la un obiect rând de un tip obiect specificat.

Tipul REF poate fi folosit pentru a examina sau modifica obiectul la care acesta se referă. De asemenea, REF poate fi folosit pentru a obține o copie a obiectului la care se referă. Singurele schimbări care se pot face asupra unui REF sunt de a înlocui conținutul său cu o referință la un alt obiect de același tip sau de a-i atribui valoarea NULL.

La declararea unui tip de coloană, a unui element dintr-o colecție sau a unui atribut al unui tip obiect ca REF, se poate ca acesta să fie constrâns să conțină doar referințe la un anumit tabel obiect. Un astfel de REF este denumit un *REF cu scop*. Un REF cu scop necesită mai puțin spațiu de stocare și permite un acces mai eficient decât un REF fără scop.

Să considerăm următorul exemplu, în care sunt create două tabele obiect, care stochează date despre jucători de tenis și cluburile unde activează.

```
CREATE OR REPLACE TYPE tip_club AS OBJECT(
cod_club          NUMBER(7),
nume_club         VARCHAR2(20),
oras              VARCHAR2(15));

CREATE TABLE club OF tip_club;

CREATE OR REPLACE TYPE tip_jucator AS OBJECT(
cod_jucator        NUMBER(9),
nume               VARCHAR2(20),
prenume            VARCHAR2(20),
pozitie_atp       NUMBER(4),
club_ref           REF tip_club);

CREATE TABLE jucator OF tip_jucator
(SCOPE FOR (club_ref) IS club);
```

Primele două comenzi de mai sus creează un tip obiect tip_club și respectiv un tabel obiect club care conține rânduri de acest tip. A treia comandă creează un tip obiect tip_jucator în care unul dintre attribute, club_ref, este o referință la un obiect de tip tip_club. Ultima comandă creează un tabel obiect care conține rânduri de tip tip_jucator și precizează scopul coloanei club_ref, indicând că această coloană poate face referire doar la rândurile din tabelul club. Legătura unu-la-mulți dintre tabelele club și jucator este astfel realizată printr-o coloană de tip REF. Dacă în locul unei referințe să se folosesc tipul obiect propriu-zis (adică tipul tip_jucator ar avea un atribut de tipul tip_club și nu o referință la acesta), atunci ar apărea date redundante (informațiile despre un club ar fi stocate pentru fiecare jucător care activează la acel club) și anomalii specifice tabelelor ne-normalizate.

Este posibil ca un obiect identificat printr-un REF să nu mai fie disponibil – fie datorită stergerii obiectului, fie datorită unei schimbări de privilegii. Un astfel de REF este numit *dangling*. În limbajul SQL folosit de către Oracle 8 există un predicat (numit *IS_DANGLING*) care permite testarea disponibilității unui REF.

Accesarea unui obiect la care face referință un REF se numește *dereferențiere*. Aceasta se face de către operatorul DEREF. De exemplu:

```
SELECT j.nume, j.prenume, DEREF(j.club_ref)
FROM jucator j
WHERE j.pozitie_atp < 1000;
```

Dereferențierea unui obiect ne-disponibil va avea ca rezultat obiectul NULL.

Un REF la un obiect rând se poate obține selectând obiectul din tabelul său obiect și aplicând operatorul REF. De exemplu, următoarea comandă SQL inserează un nou jucător care activează la clubul cu codul 171.

```
INSERT INTO jucator (cod_jucator, nume, prenume,
                      pozitie_atp, club_ref)
SELECT 1273, 'POPA', 'VASILE', 789, REF(c)
FROM club c
WHERE c.cod_club = 171;
```

10.2.7. Tipurile colecție

Un tip colecție descrie o unitate de date formată dintr-un număr nedefinit de elemente, toate de același tip. Tipurile colecție din Oracle 8 sunt tipul *VARRAY* și *tipul tabel (tabel imbricat)*. Tipul VARRAY corespunde colecțiilor ordonate, în timp ce tipul tabel imbricat corespunde colecțiilor neordonate.

Amândouă tipurile colecție au metode constructor. Numele constructorului este numele tipului iar argumentul constructorului este lista noilor elemente ale colecției, separate prin virgulă. Metoda constructor este o funcție, care întoarce ca valoare noua colecție. Utilizarea metodei constructor pentru fiecare dintre cele două tipuri colecție va fi ilustrată mai târziu. O expresie constând din numele tipului respectiv urmat de paranteze fără nici un parametru - *nume_tip()* - reprezintă o apelare a metodei constructor pentru a crea o colecție nulă de acel tip. O colecție nulă nu este același lucru cu valoarea NULL.

10.2.8. Tipul VARRAY

Un vector (array) este o mulțime ordonată de elemente care aparțin aceluiași tip de dată. Fiecare element are un index, care este un număr ce corespunde cu poziția aceluiaj element în vector. Numărul de elemente din vector se numește dimensiunea acestuia. Oracle 8

permite definirea vectorilor de dimensiune variabilă, numiți VARRAY. La definirea unui tip de vector trebuie specificată dimensiunea sa maximă.

De, exemplu, comanda următoare creează un tip de vector de maximum 12 elemente, fiecare având tipul NUMBER(8, 2):

```
CREATE TYPE tip_rate_lunare AS VARRAY(12) OF NUMBER(8, 2);
```

Un tip de vector poate fi folosit ca:

- Tipul de dată al unei coloane dintr-un tabel relațional;
- Tipul unui atribut al unui tip obiect;
- Tipul unei variabile sau parametru PL/SQL sau tip de dată întors de către o funcție.

O unitate de date de tip vector este stocată de obicei în interiorul aceluiași spațiu-tabel cu celelalte date din tabelul din care face parte. Dacă însă ea este prea mare, Oracle o stochează ca pe o unitate de date de tip BLOB (vezi Anexa 3).

Tipul colecție este foarte potrivit pentru modelarea relațiilor unu-la-mulți unde numărul maxim de elemente din partea "mulți" a relației este cunoscut și unde ordinea acestor elemente este importantă. Exemplul următor creează un tabel care conține codul creditului și ratele lunare aferente acestuia:

```
CREATE TABLE rate_credit(
cod_credit      NUMBER(7),
rate_lunare     tip_rate_lunare);
```

Metoda constructor poate fi folosită pentru inserarea de date în acest tabel. De exemplu, prima dintre următoarele instrucțiuni inserează datele corespunzătoare pentru primele cinci luni, iar cea de-a doua inserează datele pentru douăsprezece luni, dar primele şase elemente au valoarea null.

```
INSERT INTO rate_credit(cod_credit, rate_lunare)
VALUES(1111, tip_rate_lunare (134.7, 156.8, 145.9, 123.7,
156.8));
```

```
INSERT INTO rate_credit(cod_credit, rate_lunare)
VALUES(1112, tip_rate_lunare (null, null, null, null,
null, null, 123.5, 134.7, 156.8, 145.9, 123.7, 156.8));
```

10.2.9. Tipul tabel imbricat

Tipul tabel este un tip definit de către utilizator care poate fi utilizat ca:

- Tipul de dată al unei coloane dintr-un tabel relațional;
- Tipul unui atribut al unui tip obiect;
- Tipul unei variabile sau parametru PL/SQL sau tip de dată întors de către o funcție.

Datorită faptului că tipul tabel poate fi folosit ca tip de dată pentru coloane din tabele relaționale, el se mai numește și tip *tabel imbricat (nested table)*.

Un tabel imbricat este un set de elemente neordonate, toate de același tip de date. El are o singură coloană și tipul acelei coloane este un tip predefinit sau un tip obiect. Dacă este de tip obiect, atunci tabelul poate fi văzut ca un tabel cu mai multe coloane, cu o coloană pentru fiecare dintre atributele tipului obiect.

Exemplul următor creează tipul obiect `tip_persoana` și tipul tabel imbricat `tip_tabel_persoana`:

```
CREATE TYPE tip_persoana AS OBJECT(
cod_personal    NUMBER(9),
nume            VARCHAR2(20),
prenume         VARCHAR2(20));

CREATE TYPE tip_tabel_persoana
AS TABLE OF tip_persoana;
```

Când un tip tabel imbricat apare ca tipul unei coloane dintr-un tabel relațional sau ca tipul atributului unui tip obiect folosit în construcția unui tabel obiect, Oracle stochează tabelul imbricat în afara tabelului gazdă, într-un *tabel de stocare*, folosind clauza STORE AS. Un exemplu care ilustrează folosirea unui tabel imbricat ca tipul coloanei unui alt tabel este dat mai jos.

Tabelele imbricate sunt foarte potrivite pentru modelarea relațiilor unu-la-mulți unde numărul maxim de elemente din partea "mulți" a relației nu este cunoscut și unde ordinea acestor elemente nu este importantă. De exemplu, scenariul de mai sus, despre jucătorii de tenis și cluburile lor, poate fi modelat în modul următor folosind tipul tabel imbricat:

```
CREATE TYPE tip_jucator AS OBJECT(
cod_jucator    NUMBER(9),
nume           VARCHAR2(20),
prenume        VARCHAR2(20),
pozitie_atp   NUMBER(4));

CREATE TYPE tip_tabel_jucator
AS TABLE OF tip_jucator;

CREATE TABLE club(
cod_club        NUMBER(7),
nume_club       VARCHAR2(20),
oras            VARCHAR2(15),
jucator         tip_tabel_jucator)
NESTED TABLE jucator STORE AS stocare_jucator;
```

Ultima linie din comanda de mai sus specifică `stocare_jucator` ca fiind tabelul de stocare pentru coloana `jucator` din tabelul `club`.

Metoda constructor pentru tipul tabel imbricat poate fi folosită pentru inserarea de date în tabelul de mai sus. De exemplu, următoarea comandă inserează un club la care activează doi jucători:

```
INSERT INTO club (cod_club, nume_club, oras, jucator)
VALUES(167, 'VOINICELUL', 'BUCURESTI',
      tip_tabel_jucator(tip_jucator(1123, 'ION', 'DUMITRU', 145),
      tip_jucator(2117, 'STOICA', 'DORU', 156)));
```

Interrogarea următoare returnează numele și prenumele jucătorilor care activează la clubul 'VOINICELUL'. Cuvântul cheie THE care prefixează subinterrogarea interioară indică serverului Oracle că rezultatul acesteia este un tabel imbricat și nu o valoare scalară.

```
SELECT nume, prenume
FROM THE(SELECT jucator
          FROM club
         WHERE nume_club = 'VOINICELUL');
```

Dintre avantajele tabelelor imbricate menționăm posibilitatea de a fi indexate (spre deosebire de tabelele obiect) și faptul că operațiile de join nu mai sunt necesare. Un dezavantaj al folosirii acestor tabele pentru modelarea relațiilor unu-la-mulți este faptul că, deși există o formă de ștergere în cascadă (ștergerea unui rând master va determina ștergerea tuturor rândurilor detaliu), nu pot fi definite restricții de integritate referențială.

10.2.10. Metode asociate colecțiilor

Colecțiile pot fi manipulate cu ajutorul următoarelor metode predefinite de tip funcție sau procedură:

Metoda	Descriere
COUNT	Returnează numărul de elemente al unei colecții.
DELETE	Șterge toate elementele dintr-o colecție.
DELETE (<i>m, n</i>)	Șterge toate elementele dintr-o colecție de tip tabel imbricat care au indicele cuprins în intervalul [<i>m, n</i>]. Dacă <i>m > n</i> metoda nu are nici un efect.
DELETE (<i>n</i>)	Șterge al <i>n</i> -lea element dintr-o colecție de tip tabel imbricat. Dacă elementul este NULL, metoda nu are nici un efect.
EXISTS (<i>n</i>)	Returnează valoarea TRUE dacă există al <i>n</i> -lea element al unei colecții.
EXTEND	Adaugă un element NULL într-o colecție de tip VARRAY. Într-o colecție de tip tabel imbricat nu se poate insera un element NULL.
EXTEND (<i>n</i>)	Adaugă <i>n</i> elemente NULL într-o colecție de tip VARRAY. Într-o colecție de tip tabel imbricat nu se pot insera elemente NULL.
EXTEND (<i>n, i</i>)	Adaugă <i>n</i> copii ale elementului <i>i</i> într-o colecție.
FIRST	Returnează primul element al unei colecții.
LAST	Returnează ultimul element al unei colecții.
LIMIT	Returnează numărul maxim posibil de elemente al unei colecții de tip VARRAY.
NEXT (<i>n</i>)	Returnează elementul care succede elementului al <i>n</i> -lea al colecției.

Metoda	Descriere
PRIOR (n)	Returnează elementul care precede elementul al <i>n</i> -lea al colecției.
TRIM	Sterge ultimul element al unei colecții.
TRIM (n)	Sterge ultimele <i>n</i> elemente ale unei colecții.

Exemplificarea modului de funcționare a funcției COUNT se găsește în secțiunea următoare.

10.2.11. Exemplu

Pentru a ilustra folosirea tipurilor de date prezentate până acum să considerăm scenariul de la începutul secțiunii 10.2, privind activitatea de vânzare a unei firme. Aceasta poate fi modelată folosind următoarele structuri.

```

CREATE OR REPLACE TYPE tip_adresa AS OBJECT(
    strada          VARCHAR2(20),
    numar           VARCHAR2(5),
    oras            VARCHAR2(20),
    tara             VARCHAR2(20));

CREATE OR REPLACE TYPE tip_lista_telefoane
AS VARRAY(10) OF VARCHAR(20);

CREATE OR REPLACE TYPE tip_client AS OBJECT
(cod_client        NUMBER(9),
nume              VARCHAR2(20),
adresa            tip_adresa,
lista_telefoane   tip_lista_telefoane,
MAP MEMBER FUNCTION map_client RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES (map_client, WNDS, WNPS, RNPS,
RNDS));

CREATE OR REPLACE TYPE BODY tip_client AS
MAP MEMBER FUNCTION map_client RETURN NUMBER IS
BEGIN
    RETURN cod_client;
END map_client;
END;

CREATE OR REPLACE TYPE tip_articol AS OBJECT
(cod_articol      NUMBER(7),
nume_articol     VARCHAR2(20),
pret              NUMBER(10, 2),
MAP MEMBER FUNCTION map_articol RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES (map_articol, WNDS, WNPS,
RNPS, RNDS));

```

```

CREATE OR REPLACE TYPE BODY tip_articol AS
MAP MEMBER FUNCTION map_articol RETURN NUMBER IS
BEGIN
  RETURN cod_articol;
END map_articol;
END;

CREATE OR REPLACE TYPE tip_art_com AS OBJECT(
articol_ref      REF tip_articol,
cantitate        NUMBER(3));

CREATE OR REPLACE TYPE tip_tab_art_com AS TABLE OF
tip_art_com;

CREATE OR REPLACE TYPE tip_comanda AS OBJECT
(cod_comanda      NUMBER(7),
client_ref       REF tip_client,
data_comanda    DATE,
data_livrare     DATE,
articole_comanda tip_tab_art_com,
adresa_livrare   tip_adresa,
MAP MEMBER FUNCTION map_comanda RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES (map_comanda, WNDS, WNPS,
RNPS, RNDS),
MEMBER FUNCTION valoare_totala RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES (valoare_totala, WNDS, WNPS));

CREATE OR REPLACE TYPE BODY tip_comanda AS
MAP MEMBER FUNCTION map_comanda RETURN NUMBER IS
BEGIN
  RETURN cod_comanda;
END;
MEMBER FUNCTION valoare_totala RETURN NUMBER IS
  i      INTEGER;
  a      tip_articol;
  a_c    tip_art_com;
  total NUMBER := 0;
  BEGIN
    FOR i IN 1..SELF.articole_comanda.COUNT LOOP
      a_c :=SELF.articole_comanda(i);
      SELECT DEREF(a_c.articol_ref) INTO a
      FROM DUAL;
      total := a_c.cantitate * a.pret;
    END LOOP;
    RETURN total;
  END valoare_totala;
END;

```

```

CREATE TABLE client_tab OF tip_client
(PRIMARY KEY(cod_client));

CREATE TABLE articol_tab OF tip_articol
(PRIMARY KEY(cod_articol));

CREATE TABLE comanda_tab OF tip_comanda
(PRIMARY KEY(cod_comanda),
SCOPE FOR(client_ref) IS client_tab)
NESTED TABLE articole_comanda STORE AS stocare_art_com;

```

10.3. Tipuri de date LOB

O mare parte din eficiența bazelor de date se datorează modului de gestionare a tipurilor de date de bază, precum numere, șiruri de caractere sau date calendaristice. De exemplu, există facilități pentru compararea și ordonarea datelor, facilități de accesare rapidă prin indexare și alte optimizări. Tipurile de date mai complexe - precum imagini, texte, grafice, video-clipuri, etc. – sunt mai greu de reprezentat folosind aceste tipuri de bază. Oracle8 facilitează lucrul cu astfel de tipuri de date prin introducerea unor tipuri de date LOB (Large Objects) ale căror valori specifică locația unor obiecte de dimensiuni mari. Există mai multe tipuri de date LOB: BFILE, BLOB, CLOB și NCLOB.

Obiectele ale căror locație este specificată folosind aceste tipuri de date pot fi stocate în fișiere externe, în interiorul rândului sau în exteriorul acestuia. Coloanele de tip BLOB, CLOB, NCLOB ale unui tabel stochează datele în baza de date, adică în interiorul sau în exteriorul unui rând, pe când pentru o coloană de tip BFILE datele sunt stocate în afara bazei de date sub formă de fișiere.

Mai multe amănunte despre tipurile de date LOB se găsesc în Anexa 3.

10.4. Vederi obiect

Tot la fel cum o vedere este un tabel virtual, o vedere obiect este un tabel obiect virtual. Oracle 8 furnizează vederi obiect ca extensii ale mecanismului relațional. Prin folosirea vederilor obiect se pot crea tabele obiect virtuale din date stocate în coloanele tabelelor relaționale sau tabelelor obiect din baza de date. Vederile obiect pot fi create pe baza uneia sau mai multor tabele sau vederi relaționale, tabele obiect sau a altor vederi obiect.

Vederile obiect ale tabelelor relaționale oferă utilizatorilor și dezvoltatorilor avantajul de a folosi date relaționale în aplicații orientate pe obiect, fără a fi necesară convertirea acestor date. Astfel, asupra datelor mai vechi din tabele relaționale pot fi folosite tehniciile programării orientate pe obiect, permitându-se astfel dezvoltarea în paralel atât a aplicațiilor relaționale existente cât și a unor aplicații noi, orientate pe obiect. Putem astfel spune că vederile obiect sunt cele care facilitează migrarea graduală de la tehnologia relațională către tehnologia orientată obiect. Pentru utilizator, întregul sistem apare ca

fiind orientat pe obiect, deși în realitate datele sunt relaționale. Oricum acest lucru e transparent. Desigur, deoarece obiecte sunt un caz particular al vederilor în general, ele oferă și același avantaj pe care îl oferă orice altă vedere: vederi diferite ale acelorași date pentru utilizatori diferiți și scopuri diferite.

Conceptual, procedura de definire a unei vederi obiect cuprinde următorii pași:

- Definirea unui tip obiect care va reprezenta tipul rândurilor din vedere obiect;
- Definirea vederii obiect, care trebuie să specifică atât o interogare pe una sau mai multe tabele (cuprinzând câte o coloană pentru fiecare dintre atributele tipului obiect de bază) cât și un identificator de obiect pe baza atributelor tipului obiect de bază, pentru a permite identificarea fiecărui obiect (rând) din vedere obiect de către date de tip REF.

Acest identificator corespunde cu identificatorul unic al obiectului (OID) generat de Oracle pentru rânduri din tabele obiect. Pe de altă parte însă, în cazul vederilor obiect, declarația acestuia trebuie să specifică ceva care este unic în cadrul datelor din tabelele de bază (de exemplu o cheie primară). Dacă vedere obiect este bazată pe un singur tabel obiect sau pe o singură vedere obiect, atunci nu este necesară specificarea identificatorului de obiect, Oracle folosind identificatorul de obiect din tabelul obiect sau vedere obiect originală.

Să considerăm, de exemplu, următoarele două tabele relaționale:

```
CREATE TABLE club(
cod_club          NUMBER(7)    PRIMARY KEY,
nume_club         VARCHAR2(20),
oras              VARCHAR2(15));

CREATE TABLE jucator(
cod_jucator        NUMBER(9)    PRIMARY KEY,
nume               VARCHAR2(20),
prenume            VARCHAR2(20),
pozitie_atp       NUMBER(4),
cod_club           NUMBER(7)    REFERENCES club(cod_club));
```

Comenzile SQL de mai jos creează o vedere obiect pe baza acestor două tabele relaționale:

```
CREATE OR REPLACE TYPE tip_club AS OBJECT(
cod_club          NUMBER(7),
nume_club         VARCHAR2(20),
oras              VARCHAR2(15));

CREATE OR REPLACE TYPE tip_jucator AS OBJECT(
cod_jucator        NUMBER(9),
nume               VARCHAR2(20),
prenume            VARCHAR2(20),
pozitie_atp       NUMBER(4),
club               tip_club);
```

286 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

```
CREATE OR REPLACE VIEW jucator_club OF tip_jucator
WITH OBJECT OID(cod_jucator)
AS SELECT j.cod_jucator, j.nume, j.prenume, j.pozitie_atp,
tip_club(c.cod_club, c.nume_club, c.oras)
FROM jucator j, club c
WHERE j.cod_club = c.cod_club;
```

Din punctul de vedere al utilizatorului, vederea obiect creată mai sus se va comporta ca un tabel obiect în care rândurile sunt de tipul tip_jucator. La fel ca în cazul tabelelor obiect, fiecare rând va avea un identificator de obiect. Oracle construiește acest identificator pe baza cheii specificate. În vederi obiect bazate pe un singur tabel, acesta este, în cele mai multe cazuri, cheia primară a tabelului. Dacă vederea este bazată pe mai multe tabele - ca în exemplul de mai sus - cheia furnizată trebuie să fie unică pentru datele selectate din toate tabelele, astfel încât să identifice în mod unic rândurile din vederea obiect.

Datele din rândurile vederilor obiect pot fi accesate de programul de aplicație la fel ca orice alt obiect din baza de date. În cadrul comenziilor SQL, vederile obiect pot fi utilizate la fel ca tabelele obiect, de exemplu ele pot apărea în lista unei instrucțiuni SELECT, în clauza WHERE, în operații DML (cu anumite restricții, specifice vederilor). De asemenea, este posibilă definirea unei vederi obiect pe baza unei alte vederi obiect. Chiar dacă provin din mai multe tabele, datele dintr-un rând al unei vederi obiect traversează rețeaua în cadrul unei singure operații.

Asupra unei vederi obiect se pot efectua operații DML (INSERT, DELETE, UPDATE) la fel ca asupra unui tabel obiect. La fel ca și în cazul vederilor relaționale, Oracle execută aceste comenzi dacă nu există nici o ambiguitate privind datele afectate de aceste comenzi. Aceste restricții au fost descrise pe larg pentru vederile relaționale (vezi secțiunea 6.2), ele vizează vederi care cuprind operatori pe mulțimi (UNION, MINUS, INTERSECT), funcții de grup (COUNT, SUM, MAX, etc.), clauza GROUP BY, operatorul DISTINCT, care conțin expresii sau sunt bazate pe mai multe tabele.

O modalitate alternativă de a modifica datele dintr-o vedere sunt trigger-ele INSTEAD OF. Acestea se execută ori de câte ori un program aplicație încearcă să execute o instrucție DML, în locul acesteia. Trigger-ele INSTEAD OF sunt descrise pe larg în secțiunea 9.16.

Capitolul 11

Oracle Forms

Oracle Forms este un 4GL (limbaj de generația a IV-a) puternic, utilizat pentru dezvoltarea aplicațiilor interactive. Orice aplicație creată cu *Oracle Developer* este prin natura ei o aplicație multi-utilizator, bazată pe tehnologia client/server, independentă de platformă.

Oracle Forms constituie coloana vertebrală a pachetului de programe *Oracle Developer*, fiind utilizat pentru realizarea *interfejelor interactive* ale aplicațiilor. Deși este un mediu îndeajuns de complet pentru a realiza aplicații pentru baze de date puternice fără utilizarea altor instrumente, *Oracle Forms* permite apelarea atât a altor componente *Oracle Developer*, cât și a programelor scrise în C sau a altor aplicații.

În directorul destinat instrumentelor *Oracle Developer* sunt create trei icoane pentru *Oracle Forms*: Form Builder, Form Compiler și Forms Runtime. Dezvoltarea unei aplicații se realizează cu Form Builder, care poate apela pentru compilare și execuție celelalte două componente (Form Compiler și Forms Runtime), nefiind necesară apelarea separată a acestora.

Cu *Oracle Forms* pot fi create patru obiecte de bază numite module:

- *formă*;
- *meniu*;
- *bibliotecă PL/SQL*;
- *bibliotecă de obiecte*.

Fiecare modul este alcătuit din mai multe obiecte. Cu ajutorul instrumentului *Form Builder* se pot gestiona și combina cele patru tipuri de module astfel încât să poată fi creată o aplicație complexă.

Meniurile și formele sunt utilizate de obicei în orice aplicație. Un meniu funcționează numai dacă este atașat la o formă. Pe de altă parte însă, un meniu poate fi partajat de mai multe forme, iar o formă poate utiliza mai multe meniuri sau combinații între acestea. Bibliotecile sunt simple depozite de cod PL/SQL care pot fi utilizate de aplicații.

Codul sursă al unei forme, meniu sau biblioteci poate fi păstrat într-un fișier și/sau într-o tabelă a bazei de date Oracle. Pentru a stoca codul sursă într-o bază de date, la instalarea produsului Oracle Developer trebuie activată opțiunea Oracle Developer Database Tables. Această opțiune va face posibilă crearea unor tabele speciale unde se pot stoca formele, meniurile și bibliotecile.

Stocarea codului sursă într-un fișier este în general mai ușoară și mai intuitivă decât stocarea în tabele, dar stocarea codului sursă într-o tabelă a unei baze de date va permite unui modul să facă referire la codul din alt modul. În timp ce sistemul de stocare în fișiere permite gestionarea diferitelor părți de cod sursă printr-o ierarhie de directoare, sistemul de stocare în baza de date permite o partajare mai ușoară a modulelor între diferite platforme,

iar modificarea și vizualizarea codului poate fi controlată cu strictețe. Dacă este instalat PVCS (Polytron Version Control System) atunci acest sistem de control al versiunii poate fi utilizat în cadrul Oracle Forms. După instalarea instrumentului PVCS trei elemente de meniu sunt disponibile în submenuul *File/Administration: Check-In, Check-Out și Source Control Options*. Aceste opțiuni sunt valabile pentru toate mediile de dezvoltare din Oracle Developer – Forms, Reports, Graphics și Procedure Builder. Toate funcțiile de gestiune a codului sursă sunt accesate din aceste meniuuri.

11.1. Form Builder

Form Builder este mediul în care aplicațiile sunt create, testate și depanate. El furnizează o serie de instrumente de design cu ajutorul căror se poate crea și modifica funcționalitatea sau aspectul unei aplicații:

- Object Navigator
- Editorul de proprietăți (Property Palette)
- Editorul de Aranjare (Layout Editor)
- Editorul PL/SQL (PL/SQL Editor)
- Editorul de Meniuri (Menu Editor)
- Object Library

Toate aceste instrumente sunt disponibile în meniul *Tools*.

Cea mai importantă componentă a acestuia este *Object Navigator*. Acesta permite accesul la diverse componente necesare pentru realizarea formelor, meniurilor și bibliotecilor, precum și la obiectele bazei de date cum ar fi declanșatoarele (*triggers*) și procedurile stocate. Deoarece *Object Navigator* apare și în alte componente ale pachetului *Oracle Developer*, cum ar fi *Oracle Graphics* și *Oracle Reports*, se impune o cunoaștere cât mai bună a facilităților oferite de acesta.

Object Navigator permite o navigare facilă între elementele ce aparțin formelor, meniurilor sau bibliotecilor. Un obiect poate fi editat, modificat sau creat în întregime de către *Object Navigator*. Fereastra principală a *Object Navigator* prezintă fiecare obiect (instanță) sau tip de obiect (clasa de obiecte), ce poate fi precedat de unul dintre simbolurile , sau . Semnele și apar atunci când un obiect sau tip de obiecte are elemente (copii) care îl aparțin: semnul indică faptul că obiectul sau tipul obiect este deja expandat, adică elementele copii pot fi vizualizate, iar semnul apare atunci când acesta este restrâns, adică elementele copii sunt ascunse. Expandarea, respectiv restrângerea unui obiect sau tip de obiecte se obține făcând de două ori clic pe semnul , respectiv . Semnul apare atunci când un obiect sau tip de obiect poate avea elemente copii, dar acestea nu există deocamdată. Atunci când un obiect nu este precedat de nici unul dintre aceste semne, atunci înseamnă ca el nu poate avea elemente copii, adică el este la nivelul cel mai de jos al ierarhiei.

O formă este realizată prin crearea și editarea obiectelor. *Object Navigator* utilizează icoane pentru a indica un obiect care are *proprietăți*. O proprietate este un parametru care descrie un obiect. Fiecare tip de obiect are o listă diferită de proprietăți care poate fi

modificată utilizând *Editorul de Proprietăți (Property Palette)*. Editorul de Proprietăți poate fi accesat făcând clic pe obiectul ce se dorește a fi modificat și selectând opțiunea *Tools → Properties Palette* din meniul principal.

Obiectele sunt create selectând opțiunea *Navigator → Create* din meniul principal. Dacă un tip de obiecte nu conține nici o instanțiere – este precedat de simbolul - în momentul expandării tipului de obiect va fi creat automat un obiect copil (o instanță) pentru acesta. Stergerea unui obiect se poate face selectând obiectul și alegând opțiunea *Navigator → Delete* din meniul principal sau apăsând tastă *Delete*. Obiectele pot fi create și sterse selectând opțiuni din meniul *Navigator* sau utilizând bara de butoane din sferastră *Object Navigator* (vezi figura 11.1).

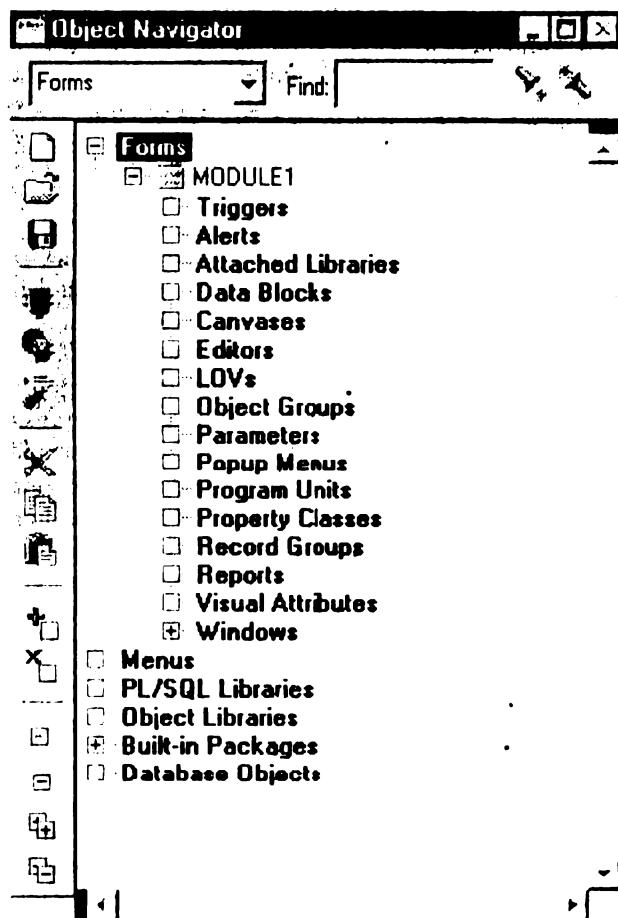


Figura 11.1-Object Navigator

11.2. Obiecte și tipul acestora

11.2.1. Forme

O formă (*form*) este un modul de bază creat cu *Oracle Forms* și reprezintă o colecție de mai multe obiecte. Forma apare în *Object Navigator* ca obiect părinte al tuturor obiectelor care sunt construite în cadrul ei. O parte din obiectele conținute într-o formă se regăsesc și în module de tip meniu sau biblioteci. Pe de altă parte, forma are și proprietăți proprii, cum ar fi titlul formei (*Title*), ce meniu va fi utilizat (*Menu Module*), dacă forma are o infășurare tridimensională (*Use 3D Controls*), etc.

Fișierele care conțin codul sursă al unei forme au extensia *.fmb* (form binary). Acestea sunt transformate de compilator în fișiere executabile care au extensia *.fmx*. De reținut că aceste fișiere nu se pot executa direct, ci doar prin intermediul modulului *Forms Runtime*. Modificarea sau schimbarea unei aplicații este posibilă doar dacă este disponibilă forma binară (*fmb*) și într-un fișier, și într-o tabelă a bazei de date.

11.2.2. Declanșatoare (Triggers)

Un *declanșator* (*trigger*) este un simplu bloc PL/SQL ce se execută ca răspuns la un eveniment. Cei care programează sub mediul Microsoft Windows sunt deja familiarizați cu evenimente precum apăsarea pe mouse (*mouse-click*) sau apăsarea unei taste (*key-pressed*). În *Oracle Forms* există peste o sută de evenimente predefinite care pot lansa un declanșator, adică execuția unui bloc PL/SQL, atunci când are loc evenimentul respectiv. Pentru multe dintre evenimente există procese隐式 (de exemplu tasta "↓" va realiza navigarea la următoarea înregistrare), dar acestea pot fi modificate (dezactivate, amplificate, înlocuite) prin intermediul declanșatoarelor - de exemplu procesul atașat apăsării pe tasta "↓" poate fi redefinit prin intermediul declanșatorului *KEY-DOWN*. Declanșatoarele pot fi desințate la nivel de formă, bloc (înregistrare) sau element (câmp). În plus, un declanșator poate cauza executarea unui alt declanșator sau a unei alte unități de program.

Fiecare trigger este denumit după evenimentul care îl declanșează. Există cinci prefixe pentru trigger-ele evenimentelor predefinite: *key-*, *on-*, *post-*, *pre-* și *when-*.

Trigger-ele key- sunt cel mai ușor de înțeles. Fiecare trigger este declanșat ca răspuns la apăsarea *logică* a unei taste. Apăsarea logică a unei taste permite tastei fizice să aibă o utilizare diferită de cea logică. De exemplu, tasta fizică tradițională pentru apelarea Help-ului este F1, dar unele aplicații utilizează o tastă diferită pentru aceasta. Aplicația poate declanșa trigger-ul asociat evenimentului Help utilizând trigger-ul *KEY-HELP* fără a avea nevoie să știe care este tasta fizică asociată (legătura dintre apăsarea fizică a tastei și apăsarea logică poate fi definită diferit pentru fiecare platformă și poate fi modificată cu ajutorul componentei *Oracle Terminal*).

Oricum, trigger-ele *key-* trebuie folosite cu atenție. În primul rând, un trigger *key-realizat* de un utilizator scrie funcția de bază a tastei. De exemplu, în momentul scrierii unui trigger *KEY-EXIT* acesta trebuie să includă comanda *EXIT* (care este

funcționalitatea implicită în absența trigger-ului), altfel apăsarea tastei *Exit* nu va realiza închiderea formei. Neincluderea funcționalității隐式的 pot cauza confuzii. În al doilea rând, deși trigger-ele key- pot fi utilizate pentru a schimba funcționalitatea tastelor (de exemplu, schimbarea tastei *Help* cu tasta *Exit*), este mai bine să se modifice legătura dintre tasta fizică și funcționalitatea ei logică folosind Oracle Terminal decât să se realizeze acest lucru prin program. În al treilea rând, într-un mediu cu interfață grafică există mai multe căi de a activa un anumit proces. Apăsarea unei taste este una dintre aceste căi. De exemplu, help-ul poate fi în mod ușor apelat prin apăsarea tastei *Help* sau prin selectarea opțiunii din meniu. Programatorul trebuie să se asigure că funcționalitatea este aceeași indiferent de modul în care help-ul este apelat. De obicei, un trigger trigger key- nu este cea mai bună alegere pentru realizarea acestui lucru.

Trigger-ele on- sunt de obicei numite *declanșatoare de tranzacții* (*transactional triggers*). Multe dintre aceste trigger-e sunt generate automat de către *Oracle Forms*. De exemplu, declanșatorul ON-CHECK-DELETE-MASTER este generat în momentul creării unei relații între blocuri. Deși acest declanșator poate fi creat și întreținut manual, acest lucru aduce mai multe necazuri decât avantaje. Alte declanșatoare on- pot fi folosite pentru a scrie comportamentul predefinit al *Oracle Forms* pentru a-l adapta cerințelor aplicației. Declanșatorul ON-ERROR este de obicei folosit de către programatori pentru a prezenta informații de eroare unui utilizator. Alte declanșatoare de tranzacții, cum ar fi ON-INSERT sau ON-LOCK sunt rar utilizate, dar pot fi folosite în anumite circumstanțe, ca de exemplu atunci când sunt folosite alte baze de date decât cele din Oracle.

Trigger-ele pre- și post- sunt activate înainte, respectiv după un eveniment. După cum sugerează numele declanșatoarele pre- și post- nu rescriu funcționalități predefinite în *Oracle Forms*, ci le completează; acestea pot fi extinse folosite pentru inițializări sau ștergeri.

Trigger-ele when- sunt foarte utilizate în *Oracle Forms*. Nu există nici o funcționalitate predefinită pentru evenimentele care declanșează trigger-ele when- astfel încât scrierea trigger-elor when- nu redefineste nici o altă funcție. Trigger-ul WHEN-VALIDATE-ITEM este unul dintre cele mai frecvent utilizate trigger-e și este folosit pentru a valida datele într-o formă. Declanșatorul WHEN-BUTTON-PRESSED definește funcționalitatea unui buton al unei forme, fiind activat atunci când este apăsat acel buton.

După cum am menționat înainte, trigger-ele pot fi definite la trei nivele: la nivel de formă, la nivel de bloc și la nivel de element. Când definim un declanșator în editorul PL/SQL, câmpul "Type:" va fi completat cu valoarea "Trigger". Cele două câmpuri "Object:" din cadrul acestui editor vor fi completeate după cum urmează. Primul câmp "Object:" va conține "(form level)" pentru un declanșator la nivel de formă, adică acest declanșator va fi activat indiferent de poziția cursorului. Primul câmp "Object:" poate conține de asemenea un nume de bloc. Acesta este un declanșator de la nivelul acelui bloc, care se activează numai atunci când acel bloc este blocul curent. Al doilea câmp "Object:" poate conține un element aparținând blocului ce apare în primul câmp "Object:". Acesta este un declanșator la nivelul de element (câmp), care se va activa numai atunci când elementul curent este acel element. Nivelul unui declanșator este reprezentat în *Object Navigator* de către poziția acestuia: declanșatoarele de la nivelul formei apar sub obiectul formă, declanșatoarele de la nivelul blocului apar sub fiecare bloc individual și declanșatoarele de la nivelul elementului apar sub fiecare element.

Declanșatoarele pot fi definite la fiecare din cele trei nivele ierarhice. Fereastra proprietăților unui declanșator (accesată prin selectarea declanșatorului din *Object Navigator* și a opțiunii *Tools → Property Palette* din meniul principal) conține o proprietate numită *Functional/Execution Hierarchy*. Valoarea implicită a acestei proprietăți este *override*, adică declanșatorul de la nivelul cel mai de jos va suprascrie funcția declanșatorului de la nivelul cel mai înalt. De exemplu, dacă sunt definite trei declanșatoare WHEN-BUTTON-PRESSED, unul atașat unui buton, unul atașat blocului în care se găsește butonul și unul atașat formei, atunci, când butonul este apăsat, se va activa declanșatorul de la nivelul elementului (butonului), celelalte nefiind activate. Dacă însă, în loc de *override*, proprietatea *Functional/Execution Style* se va seta pe una dintre opțiunile *before* sau *after*, atunci fiecare declanșator va fi activat înainte, respectiv după declanșatorul de la nivelul superior – în cazul nostru, pentru opțiunea *before*, vor fi activate în ordine declanșatoarele de la nivelul butonului, blocului și al formei, iar pentru opțiunea *after* activarea declanșatoarelor va fi făcută în ordine inversă. Nu toate declanșatoarele pot fi definite la toate nivelurile, de altfel acest lucru nu ar avea nici sens. În *Object Navigator* declanșatoarele sunt prezentate ca aparținând formei, blocului sau elementului, în funcție de nivelul la care sunt definite.

Blocul PL/SQL care definește declanșatorul poate fi executat cu succes sau nu. Un declanșator eșuează când blocul PL/SQL generează o excepție specifică *form_trigger_failure*. Multe declanșatoare se comportă diferit în funcție de succes sau eșec. De exemplu, declanșatorul WHEN-VALIDATE-ITEM generează excepția specifică *form_trigger_failure* și eșuează pentru a arăta că data este invalidă și nu va fi acceptată.

În timpul oricărei acțiuni a utilizatorului pot fi activate mai multe declanșatoare. De exemplu, atunci când utilizatorul apasă tasta de salvare (COMMIT), se va activa în primul rând declanșatorul KEY-COMMIT. Să presupunem că declanșatorul KEY-COMMIT a fost definit astfel:

```
COMMIT;
MESSAGE ('Salvare incheiata');
```

Declarația COMMIT va lansa o complicată serie de evenimente, activând declanșatoare ca PRE-COMMIT, POST-DATABASE-COMMIT și POST-FORMS-COMMIT, care, la rândul lor, pot activa alte declanșatoare. După toate acestea, controlul va fi returnat declanșatorului KEY-COMMIT pentru a afișa mesajul.

Sfera de acțiune corespunzătoare declanșatorului și selectarea declanșatorului cel mai potrivit sunt cruciale pentru realizarea unor forme corecte. Înțuitiv putem avea un păienjeniș de declanșatoare, este vital să selectăm declanșatorul corect atunci când îl definim. Arta selectării declanșatorului corect constă în definirea corectă a momentului în care se dorește ca respectivul cod PL/SQL să fie executat.

De exemplu, este o diferență foarte mare între execuția unui cod PL/SQL atunci când utilizatorul apasă tasta de salvare (COMMIT) și execuția codului PL/SQL la începutul operațiunii de salvare din *Oracle Forms*. Declanșatorul KEY-COMMIT nu va fi activat dacă salvarea este efectuată prin executarea instrucției COMMIT în alt declanșator, dar declanșatorul PRE-COMMIT va fi activat.

Declanșatorul *user-named* este o moștenire de la primele versiuni de *Oracle Forms* și este încă păstrat pentru asigurarea compatibilității. Acest declanșator este un bloc de cod PL/SQL care nu este asociat nici unui eveniment predefinit. Din punct de vedere al funcționalității, declanșatorul *user-named* este echivalent cu o *unitate de program*. Din punct de vedere al stilului programării, folosirea unităților de program este preferabilă folosirii declanșatoarelor *user-named*.

11.2.3. Ferestre de avertizare (Alerts)

O *fereastră de avertizare (alert)* este în esență o casetă definită de utilizator, care conține o icoană, un mesaj și unul până la trei butoane. Fereastra de avertizare se folosește pentru a furniza informații adiționale sau pentru a semnala întâlnirea unei erori, permitând utilizatorului să ia sau nu o anumită decizie în funcție de butoanele pe care le deține fereastra de avertizare. Afisarea ferestrelor de avertizare se realizează folosind în program funcția `SHOW_ALERT`.

Ferestrele de avertizare sunt utile pentru a opri lucrul utilizatorului și a-i atrage atenția. Când ferestrele de avertizare sunt afișate folosind funcția `SHOW_ALERT`, va fi returnat un număr care indică ce buton a fost apăsat de utilizator pentru a îndepărta fereastra de alertă. În acest fel, alertele pot fi folosite pentru a adresa întrebării utilizatorului. Pentru că fereastra de avertizare este o intrerupere, nu trebuie să se abuzeze de folosirea ei.

11.2.4. Biblioteci atașate (Attached library)

O *bibliotecă atașată (attached library)* este o colecție de cod PL/SQL stocat. Bibliotecile atașate pot fi extremitate folositoare pentru stocarea celor mai frecvente și comune funcții sau proceduri precum și a funcțiilor sau procedurilor specifice aplicației. Avantajul bibliotecilor atașate față de unitățile de program constă în faptul că blocul de cod partajat de mai multe forme poate fi păstrat într-un singur loc și modificat fără să fie necesară recompilarea fiecărei forme. Mai mult decât atât, bibliotecile atașate asigură programatorul că toate modulele aplicației folosesc aceleși funcții sau proceduri. O bibliotecă este creată utilizând obiectul *Libraries* din *Object Navigator*. Pentru a utiliza funcțiile stocate într-o bibliotecă, forma trebuie să aibă atașată acea bibliotecă prin utilizarea obiectului *Attach Libraries*, vezi secțiunea "Crearea și utilizarea bibliotecilor" din acest capitol.

11.2.5. Blocurile de date (Data Blocks)

Un *bloc* este în *Oracle Forms* o structură logică similară unei tabele dintr-o bază de date. El conține un set de elemente care sunt similare coloanelor, și sunt aranjate sub formă de înregistrări, care sunt similare rândurilor dintr-o tabelă. Un bloc poate să corespundă sau nu unei tabele din baza de date. Un bloc ce corespunde unei tabele din baza de date poate fi utilizat pentru inserarea, ștergerea și actualizarea rândurilor tabelei pe baza înregistrărilor din bloc. Blocurile care nu corespund nici unei tabele sunt în general utilizate pentru a stoca informații de control generice cum ar fi criterii de interogare, contoare, indicatori condiționali sau alte informații ce pot fi asociate mai multor înregistrări sau unor activități.

Blocurile ce corespund unei tabele sunt cunoscute sub denumirea *blocuri ale bazei de date (database blocks)* și au specificată valoarea Yes pentru parametrul *Database/Database Data Block*. Un bloc al bazei de date poate fi direct asociat cu o tabelă (în acest caz se spune că blocul este *bazat pe acea tabelă*) sau poate să fie asociat unor proceduri PL/SQL stocate prin intermediul cărora sunt interogate și modificate date din baza de date.

Un bloc bazat pe tabelă are câteva funcții proprii care nu cer o programare adițională. Fără nici un alt cod adițional, un bloc bazat pe tabelă poate fi folosit pentru a introduce criterii de interogare prin plasarea lor în *mod interogare (query mode)*. În mod interogare, utilizatorul poate introduce criterii de interogare în elementele blocului pentru a restricționa rândurile ce sunt returnate la execuția interogării. Oracle Forms va procesa interogarea, returnând rândurile corespunzătoare din baza de date în înregistrări ale blocului. Acest comportament implicit poate fi schimbat în totalitate sau lărgit prin utilizarea declanșatoarelor. Nu este necesar ca toate elementele dintr-un bloc bazat pe tabelă să fie direct conectate cu tabela de bază. Astfel de elemente, care nu se bazează pe tabelă, pot fi utilizate pentru a prelua și afișa coloane care rezultă din calcul sau alte informații. Pe de altă parte însă, elementele dintr-un bloc bazat pe tabelă nu pot fi conectate direct la o altă tabelă.

În loc de a fi direct asociat cu un tabel, există posibilitatea ca un bloc al bazei de date să fie populat folosind o procedură PL/SQL stocată. Acest lucru este posibil prin selectarea valorii *Procedure* pentru proprietatea *Database/Query Data Source Type* a blocului. Numele procedurii PL/SQL folosite pentru populare este specificat de valoarea proprietății *Database/Query Data Source Name*. De asemenea, proprietatea *Database/Query Data Source Arguments* va specifica numele și tipul argumentului procedurii de interogare, iar proprietatea *Database/Query Data Source Columns* numele și tipurile coloanelor asociate acesteia. Folosirea unei proceduri PL/SQL permite populări ale blocului mult mai complicate decât o simplă comandă SQL, aşa cum este ilustrat de exemplul de mai jos care extrage datele dintr-un tabel utilizând o interogare ierarhică.

De asemenea, pot fi specificate proceduri pentru inserare, actualizare, ștergere și blocare a datelor pentru un bloc de date (folosind grupul de proprietăți *Advanced Database* ale blocului) sau se pot folosi inserările, actualizările și ștergerile implicate rezultate din conectarea blocului la tabelul de bază, independent de modul de interogare a înregistrărilor (acest lucru se realizează prin selectarea valorii *Table* pentru proprietatea *Advanced Database/DML Data Target Type*, numele tabelului fiind apoi indicat ca valoare a proprietății *Advanced Database/DML Data Target Name*).

Procedurile PL/SQL interacționează cu blocul de date prin parametrii de tip referință de cursor (REF CURSOR) sau tabele indexate de înregistrări (TABLE OF tip_record INDEX BY BYNARY_INTEGER), vezi Anexa 3. O referință de cursor returnează o referință către un cursor deschis ce poate fi utilizat pentru interogare de către bloc, dar nu poate fi folosit pentru operații DML. Tabelele indexate de înregistrări sunt mulțimi de înregistrări ce sunt returnate în totalitate formei. Tabelele indexate de înregistrări pot fi folosite într-un bloc atât pentru interogare cât și pentru operații DML. Cu toate că sunt foarte eficiente, tabelele de dimensiuni mari cer o cantitate mare de memorie clientului. Blocurile de date asociate cu un tabel sau o procedură PL/SQL pot fi create manual sau folosind *Data Block Wizard*.

Pentru a ilustra folosirea unei proceduri pentru interogarea datelor să considerăm următorul exemplu:

```

CREATE TABLE element(
cod_element VARCHAR2(10)      NOT NULL,
cod_parinte VARCHAR2(10) );

CREATE OR REPLACE PACKAGE element_pkg IS
TYPE tip_element IS RECORD(
    cod_element          element.cod_element%TYPE,
    cod_parinte          element.cod_parinte%TYPE,
    cod_radacina         element.cod_element%TYPE);
TYPE tip_tab_element IS TABLE OF tip_element INDEX BY
BINARY_INTEGER;
END;

CREATE OR REPLACE PACKAGE BODY element_pkg IS
BEGIN
    NULL;
END;

CREATE OR REPLACE PROCEDURE interogare_ierarhica (e IN
OUT element_pkg.tip_tab_element) IS
i INTEGER;
BEGIN
i := 0;
FOR c_parinte IN (
    SELECT cod_element
    FROM element
    WHERE cod_parinte IS NULL
    ORDER BY cod_element)
LOOP
    FOR c_copil IN (
        SELECT cod_element, cod_parinte
        FROM element
        CONNECT BY cod_parinte = PRIOR cod_element
        START WITH cod_element = c_parinte.cod_element)
    LOOP
        i := i+1;
        e(i).cod_element := c_copil.cod_element;
        e(i).cod_parinte := c_copil.cod_parinte;
        e(i).cod_radacina:= c_parinte.cod_element;
    END LOOP;
    END LOOP;
END;

```

Codul de mai sus descrie un tabel cu date ierarhice (în care fiecare element poate avea un element părinte) și o procedură folosită pentru interogarea ierarhică a acestuia. Această procedură se poate folosi ca sursă de interogare a unui bloc care va conține două coloane ale tabelului, precum și un element ce nu aparține tablei de bază, cod_radacina (acesta va indica elementul de la nivelul cel mai de sus pentru fiecare ierarhie). Observați că pachetul element_pkg există numai pentru a furniza pentru tipul tip_tab_element, care este

folosit de procedură. Valoarea proprietății *Database/Query Data Source Type* va fi *Procedure*, iar valoarea proprietății *Database/Query Data Source Name* va fi *interrogare_ierarhica*. Proprietatea *Database/Query Data Source Arguments* va specifica că nume al parametrului, TABLE ca tip al său și element_pkg.tip_tab_element ca nume al acestui tip. Coloanele cod_element, cod_parinte și cod_radacina și tipurile acestora vor fi specificate de proprietatea *Database/Query Data Source Columns*. Cel mai ușor mod de a construi acest bloc este de a folosi *Data Block Wizard*. Dacă se dorește ca operațiile de inserare, modificare și ștergere pentru bloc să fie cele implicate asociate tabelului element, atunci proprietăților *Advanced Database/DML Data Target Type* și respectiv *Database/DML Data Target Name* vor fi atribuite valorile TABLE și respectiv element.

Un bloc ce nu se bazează pe o tabelă a bazei de date sau pe o procedură PL/SQL este cunoscut sub denumirea de *bloc neasociat bazei de date (non-database block)* sau *bloc de control (control block)*. Blocurile neasociate bazei de date nu au funcționalități proprii precum cele bazate pe tabelă, dar pot fi întrebuită în multe cazuri, ca de exemplu pentru stocarea variabilelor locale sau a unor informații generice de control. Un bloc de control poate fi bazat de asemenea pe declanșatori tranzacționali, utilizati în mod tipic numai pentru surse de date diferite de Oracle. Declanșatorii tranzacționali înlocuiesc funcționalitatea predefinită asociată în mod normal unui bloc de date.

11.2.6. Elemente (Items)

Elementele sunt componentele vizuale ale unei forme. Fiecare element aparține unui bloc, care prin esență este o colecție de elemente. Într-un bloc PL/SQL din cadrul unei forme, un element poate fi accesat folosindu-se sintaxa :bloc.element (două puncte, următe de numele blocului și numele elementului, acestea separate prin punct). Un element poate fi atât un câmp cât și un buton sau alt mecanism de control. Există nouă tipuri diferite de elemente: *text item*, *chart item*, *check box*, *display item*, *image*, *button*, *radio group*, *list item* și *custom item*.

Un *text item* este un câmp folosit pentru introducerea, regăsirea și editarea cu ajutorul tastaturii a textelor de tip sir de caractere, număr sau dată. Un *display item* este similar cu acesta, excepție făcând faptul că datele nu pot fi manipulate ci numai afișate, iar utilizatorul nu poate naviga prin acest tip de element.

Elementul *image* este o imagine bitmap în formatul grafic propriu al sistemului Oracle. Dacă imaginea este stocată și regăsită dintr-o bază de date, atunci câmpul respectiv trebuie să fie de tip LONG RAW. Imaginea poate de asemenea să fie extrasă dintr-un fișier cu ajutorul procedurii predefinite READ_IMAGE_FILE(nume_fisier, tip_fisier, id_element). Avantajul stocării unei imagini în baza de date este faptul că aceasta poate fi regăsită utilizând Oracle Forms pe orice platformă. De exemplu, o imagine stocată utilizând Oracle Forms în Windows poate fi regăsită utilizând Oracle Forms în orice versiune a sistemului de operare Windows sau chiar și pe un calculator Macintosh fără a schimba absolut nimic. Din punct de vedere al performanței însă, regăsirea unei imagini de dimensiuni considerabile într-o bază de date poate fi costisitoare în comparație cu un fișier local.

Un *check box* este un element de control folosit des în mediul Windows, care arată ca o casetă pentru bifare de pe un buletin de vot. Un *check box* este utilizat pentru a alege una din două valori fixe. Cele două valori sunt specificate în soia de proprietăți a elementului. Proprietatea

Functional/Check Box Mapping of Other Values determină comportamentul programului Oracle Forms în momentul extragerii unor valori din baza de date care nu sunt egale cu nici una din valorile specificate. Cele două valori sunt specificate în proprietatea *Functional/Value when Checked* (valoarea corespunzătoare bifării), respectiv *Functional/Value when Unchecked* (valoarea corespunzătoare nebifării) din foaia de proprietăți a elementului. Proprietatea *Functional/Check Box Mapping of Other Values* poate fi setată să afișeze aceste valori ca fiind bifate sau nebifate, sau poate fi setată pe valoarea *not allowed*. Aceasta înseamnă că un rând cu o valoare care nu se află în proprietatea *Functional/Value when Checked* sau *Functional/Value when Unchecked* nu va fi adus în momentul când se interoghează baza de date și nu se va semnala nici o eroare. Această opțiune trebuie utilizată cu atenție. De exemplu, când un element check box așteaptă valoarea "Y" sau "N" și opțiunea *not allowed* este specificată, atunci rândurile ce au valoarea Null nu pot fi aduse din baza de date.

Un *radio group* este un element de interfață care permite selectarea unei variante dintr-un număr de opțiuni fixe și care se exclud una pe alta. Fiecare opțiune este reprezentată de un singur *radio button*, un *radio group* fiind deci alcătuit din mai multe *radio button*. Fiecare *radio button* în sine funcționează similar unui check box, adică permite alegerea unei singure valori dintre două valori specificate. Doar un *radio button* al unui *radio group* poate fi selectat la un moment dat. Când este creat un *radio group*, *radio button*-ul este creat individual în Object Navigator ca și copil al *radio group-ului*. Deoarece valorile *radio button-ului* sunt selectate la momentul proiectării, *radio button-ul* este în principal folosit atunci când variantele sunt fixe.

Un *button* este un element de control folosit frecvent în mediul Windows pe care un utilizator poate să "apeze" (să-l activeze) apăsând pe mouse sau pe o combinație de taste. Spre deosebire de celelalte elemente, un buton nu poate avea la bază o coloană a unei tabele. El poate fi doar apăsat și nimic mai mult. Butonul este unul dintre cele mai puternice elemente din Oracle Forms, deoarece în momentul apăsării butonului se lansează declanșatorul WHEN-BUTTON-PRESSED, care poate face aproape orice, de la salvarea datelor până la lansarea altelor forme. Butoanele sunt deseori utilizate în barele de instrumente (toolbars) pentru a accesa mai ușor cele mai utilizate funcții (de exemplu salvarea datelor, introducerea sau executarea unei interogări, navigarea la înregistrarea anterioară sau următoare, etc.). Butoanele pot fi activate sau dezactivate prin program pentru a arăta utilizatorului care funcții sunt disponibile la un moment dat.

Un *chart item* este o arie dreptunghiulară utilizată pentru a crea un spațiu necesar afișării obiectelor create de către Oracle Graphics.

Un *custom item* este un obiect OLE sau VBX pe o platformă ce suportă control de tip OLE2 sau VBX.

Pentru a fi afișate, elementele sunt în mod normal atribuite unui *canvas* (pentru informații despre canvas vezi secțiunea 11.2.8), dar un element poate fi atribuit și aşa-numitului "null canvas" (adică practic nu aparține nici unui canvas), ceea ce face ca elementul să devină invizibil pentru utilizator. Elementele care nu aparțin nici unui canvas sunt folositoare ca variabile locale formei, care pot fi folosite de orice unitate de program sau de orice alt bloc PL/SQL din cadrul acesteia.

11.2.7. Relații (Relations)

O relație master/detail asociază un bloc master și un bloc detaliu, reflectând relația determinată de cheia primară și cheia străină între tabelele pe care se bazează blocurile. Fiecarui rând dintr-un bloc master îi corespund zero, unul sau mai multe rânduri din blocul detaliu prin-o condiție de joncțiune.

Obiectul relație coordonează automat relația master/detail dintr-o blocuri. Atunci când o relație este creată, sunt create în mod automat și declanșatoarele și procedurile PL/SQL necesare pentru a forța coordonarea dintre cele două blocuri. Codul generat variază în funcție de particularitățile relației.

O relație poate fi creată în mod automat la crearea unui bloc folosind facilitatea Data Block Wizard, prin selectarea unui bloc master pe care se bazează blocul curent. Această opțiune este disponibilă numai atunci când blocul master există deja în formă și când relația cheie primară/cheie străină între tabele este definită în baza de date.

Nu există restricții asupra relațiilor care sunt create manual în Object Navigator. Blocurile master și detaliu sunt specificate împreună cu condiția de joncțiune, cu modul în care se realizează ștergerile din blocul master (în cascadă, izolate sau ne-izolate) și cu modul în care se realizează coordonarea dintre blocul master și blocul detaliu – această coordonare se poate realiza în momentul în care blocul master se modifică sau în momentul în care se navighează la blocul detaliu (coordonare întârziată). Atunci când proprietățile relației se schimbă, declanșatoarele generate automat se regenerează. Relația apare în Object Navigator ca un obiect copil al blocului master.

Cele trei moduri de realizare a ștergerii din blocul master au următoarele semnificații:

- **Ne-izolate (Non Isolated):** împiedică ștergerea unei înregistrări din blocul master atunci când există o înregistrare asociată în blocul detaliu; acesta este valoarea implicită.
- **Izolate (Isolated):** permite ștergerea unei înregistrări din blocul master, fără a afecta înregistrările din blocul detaliu.
- **În cascadă (Cascading):** permite ștergerea unei înregistrări din blocul master și șterge toate înregistrările asociate din blocul detaliu. Dacă există mai multe relații master/detaliu înlănuite, atunci doar rândurile din primul bloc de detaliu sunt șterse, opțiunea "în cascadă" netransmițându-se automat celorlalte nivele din lanț. Trebuie menționat că dacă relația cheie primară/cheie străină este definită în baza de date cu opțiunea de ștergere în cascadă, atunci proprietatea de ștergere în cascadă la nivel de bloc nu trebuie utilizată deoarece va rezulta o eroare.

11.2.8. Canvas-uri (Canvases)

Canvas-urile sunt suprafețe vizuale pe care sunt plasate elementele, astfel încât acestea pot fi manipulate sau vizualizate de către utilizatorul final. Canvas-urile pot fi suprapuse, permitând astfel crearea unor suprafețe complexe și modificabile ce conțin elemente. Canvas-urile nu sunt ferestre; ele sunt vizualizate prin intermediul ferestrelor, care sunt definite separat. Relația dintre ferestre și canvas-uri este aceea că într-o fereastră apar unul sau mai multe canvas-uri. Canvas-urile reprezintă suprafețele pe care apar elementele, iar

ferestrele sunt arii de vizualizare pentru unul sau mai multe canvas-uri. Un canvas poate fi de unul din următoarele cinci tipuri: cuprins, stivuit, bară de instrumente verticală, bară de instrumente orizontală sau multi-pagină.

Canvas-ul cuprins (content canvas) este cel mai simplu tip de canvas. Acesta este canvas-ului de bază al unei ferestre și reprezintă o simplă suprafață de o lungime și lățime date, pe care sunt plasate elementele. Un canvas cuprins poate fi mai mic sau mai mare decât fereastra în care apare. Dacă canvas-ul cuprins este mai mare decât fereastra în care apare, atunci fereastra trebuie să aibă specificate bare de defilare (scroll bars) pentru a permite utilizatorului să vizualizeze toată aria vizibilă a canvas-ului cuprins. Într-o fereastră pot apărea mai multe canvas-uri cuprins, dar numai unul singur este vizibil la un moment dat.

Un *canvas stivuit (stacked canvas)* poate fi plasat deasupra unui canvas cuprins sau stivuit. Un canvas stivuit conține proprietăți adiționale (poziție, lățime, lungime) referitoare la aria din canvas care este vizibilă pe ecran (*viewport*), acesta putând fi mai mică decât aria totală a canvas-ului. De asemenea, pot fi specificate barele de defilare orizontală sau verticală, ce permit utilizatorului să vizualizeze toată aria canvas-ului stivuit.

Un *canvas bară de instrumente verticală sau orizontală (vertical toolbar canvas sau horizontal toolbar canvas)* este un tip special de canvas, proiectat cu scopul expres de a crea bare de instrumente. O bară de instrumente este cunoscută majorității utilizatorilor Windows drept o colecție de butoane ce apare în unele aplicații de-a lungul marginii unei ferestre. În Oracle Forms, un canvas bară de instrumente reprezintă un loc unde programatorul plasează butoane sau alte elemente ce trebuie să fie accesibile tot timpul în cadrul unei anumite ferestre. Un canvas bară de instrumente este conectat la o fereastră prin specificarea acesteia în proprietățile ferestrei. Spre deosebire de canvas-urile cuprins sau stivuit, care nu pot fi asociate decât unei singure ferestre, canvas-ul bară de instrumente poate fi asociat mai multor ferestre.

Un *canvas multi-pagină (tab-canvas)* este un canvas care conține mai multe pagini, fiecare cu câte o etichetă. Dintre aceste pagini, utilizatorul poate selecta la un moment dat numai una. Elementele care sunt plasate pe un *tab-canvas* sunt localizate utilizând o combinație formată din numele canvas-ului și eticheta paginii. Deoarece părțile vizibile ale unui *tab-canvas* sunt paginile, pentru a fi vizibile, elementele trebuie plasate pe una dintre pagini. Figura 11.2 prezintă în aceeași fereastră un canvas cuprins, un canvas stivuit, un canvas multi-pagină și un canvas bară de instrumente verticală.

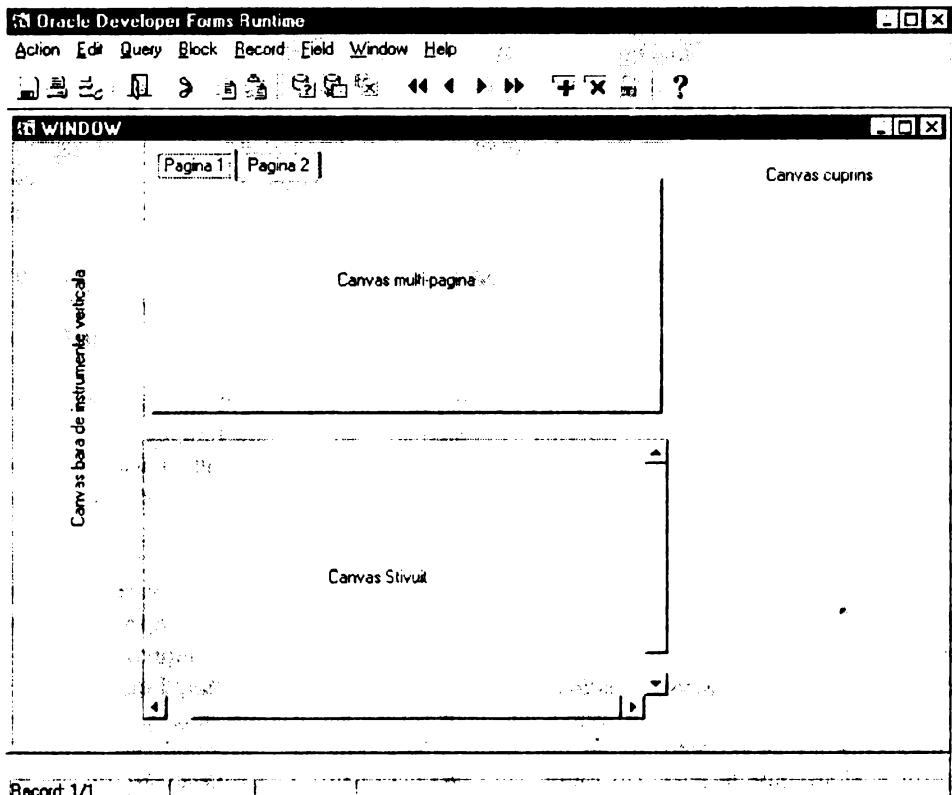


Figura 11.2 - Canvas cuprins, stivuit, bară de instrumente și multi-pagină

Atunci când canvas-urile se suprapun, canvas-ul care conține elementul curent al formei va fi afișat întotdeauna deasupra. Toate celelalte canvas-uri pot fi programate să fie afișate sau să fie ascunse. Numai un singur canvas cuprins poate fi vizibil într-o fereastră, dar canvas-urile stivuite pot apărea în orice număr.

11.2.9. Editoare (Editors)

Un *editor* este un editor de text care apare pe ecran în cadrul unei aplicații și oferă o modalitate mai convenabilă de a vedea sau edita un text lung într-un text item. Un editor este definit prin specificarea unor proprietăți cum ar fi poziția, mărimea, titlul, fontul. Un editor poate fi utilizat pentru mai multe elemente ale formei, dar datorită posibilității limitate de control prin program, de obicei se creează editoare individuale pentru fiecare în parte. Când un editor este asociat elementului curent, editorul se va afișa utilizând funcția predefinită SHOW_EDITOR sau atunci când utilizatorul apasă tasta *Edit*.

11.2.10. Liste de valori (LOVs)

LOV reprezintă prescurtarea expresiei *list of values* (*listă de valori*), care este folosită pentru a automatiza sau valida selectarea unor valori dintr-o listă. Figura 11.3 arată un

exemplu de listă de valori aşa cum apare într-o aplicație. Datele selectate într-o listă de valori sunt specificate de către un *grup de înregistrări* (vezi secțiunea următoare). O listă de valori poate fi creată manual sau folosind un LOV Wizard. Grupul de înregistrări pe care se bazează lista de valori trebuie să existe deja la crearea listei de valori sau poate fi creat odată cu aceasta dacă se folosește LOV Wizard.

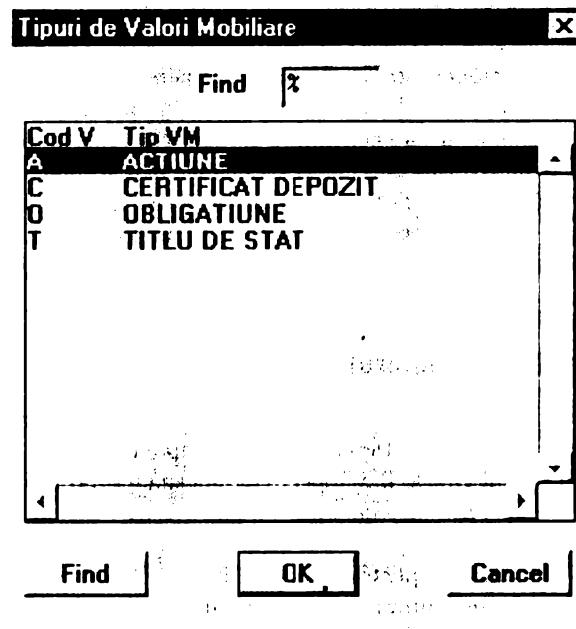


Figura 11.3 - Listă de valori

Un LOV poate fi specificat în lista de proprietăți a unui element prin proprietatea *List of Values/List of values*. Când aplicația este lansată în execuție, indicatorul "List of Values" apare în bara de stare în momentul când cursorul este mutat pe un element pentru a indica că acesta are asociată o listă de valori.

Elementele returnate de către un LOV sunt mai departe returnate în elementele specificate în setul de proprietăți *Functional/Column Mapping* din foaia de proprietăți a LOV-ului. O listă de valori poate avea mai multe coloane vizibile sau ascunse și poate returna valori în orice element valid de pe o formă, indiferent dacă acel element are definit sau nu un LOV în propria foaie de proprietăți. Cu alte cuvinte, proprietatea *List of Values/List of values* a unui element determină dacă utilizatorul poate vedea lista de valori prin apăsarea tastei *List of values* (în mod normal aceasta este F9 în Windows), dar nu are implicații asupra locului unde vor ajunge valorile din listă.

Elementele au și o altă proprietate, *List of Values/Validate from list*, care permite ca valorile unui element să fie validate după valorile returnate de LOV-ul specificat în proprietatea *List of Values/List of values*. Când această proprietate este setată pe valoarea *True*, orice valoare introdusă într-un element va face ca lista de valori să fie consultată

pentru validare de către Oracle Forms. Dacă valoarea unui element apare în lista de valori, aceasta va fi considerată validă. Dacă valoarea unui element nu apare în lista de valori, va apărea lista de valori pentru ca utilizatorul să selecteze din aceasta o valoare validă; lista va insista să apară până în momentul în care este introdusă o valoare validă.

11.2.11. Grupuri de obiecte (Object Groups)

Un *grup de obiecte* (*object group*) este o colecție de elemente dintr-o formă. Grupurile de obiecte nu sunt necesare pentru a crea sau utiliza o formă, nu pot fi referite prin program și nici nu aduc vreo contribuție la funcționalitatea formei. Un grup de obiecte este utilizat pentru a grupa obiecte într-o formă astfel încât acestea să poată fi copiate sau referite într-o altă formă sau bibliotecă de obiecte fără a selecta fiecare obiect în parte. Un grup de obiecte permite deci gruparea unor componente folosite dintr-o formă în module logice și reutilizarea lor ca o unitate în alte forme.

11.2.12. Parametri (Parameters)

Un *parametru* este o variabilă utilizată pentru transmiterea datelor între două forme sau între o formă și un alt modul Oracle Developer (de exemplu o formă și un raport). Atunci când o formă (sau alt modul Oracle Developer) este apelată dintr-o altă formă, parametrii sunt transmiși ca parte a unei *liste de parametri*.

Prin definirea unor parametri pentru o formă folosind Object Navigator, forma se va așteaptă să primească o valoare pentru fiecare dintre parametrii, iar în cazul în care valoarea nu va fi transmisă, parametrul va primi o valoare implicită. Cu alte cuvinte, parametrii definiți în cadrul unei forme folosind Object Navigator reprezintă *parametrii de intrare* pentru forma respectivă, ai căror valori sunt transmise din forma apelantă. Pe de altă parte, *parametrii de ieșire* (adică cei din forma apelantă) sunt creați prin crearea de liste de parametri, folosind funcția CREATE_PARAMETER_LIST. Parametrii sunt adăugați listei utilizând funcția ADD_PARAMETER. Lista este transferată altor forme (sau altui instrument Oracle Developer) utilizând funcțiile RUN_PRODUCT, OPEN_FORM, CALL_FORM sau NEW_FORM. Toți parametrii din listă trebuie să fi fost deja definiți ca parametri în forma apelată utilizând Object Navigator, altfel Oracle Forms va genera o eroare.

Următorul exemplu creează o listă de parametri, adaugă la aceasta doi parametrii și apoi transmite această listă unei forme folosind procedura OPEN_FORM:

```

DECLARE
    list_id      PARAMLIST;
BEGIN
/* creeaza lista de parametri "parametri_intrare" */
    lista_id := CREATE_PARAMETER_LIST('parametri_intrare');
/* adauga doi parametrii la lista, specificand pentru
fiecare numele, tipul si valoarea */
    ADD_PARAMETER(lista_id, 'TARA', TEXT_PARAMETER, 'ROMANIA');
    ADD_PARAMETER(lista_id, 'ORAS', TEXT_PARAMETER, 'BUCURESTI');

```

```
/* cheama forma, transmitand lista de parametri */
OPEN_FORM('forma_oras', ACTIVATE, NO_SESSION, list_id);
END;
```

Parametrii interni, adică cei definiți în formă utilizând Object Navigator, alcătuiesc o listă predefinită a formei, numită listă implicită (default). Lista implicită de parametri poate fi transmisă unei alte forme ca parametru de ieșire, adică la fel ca o listă de parametri creată prin utilizarea funcției CREATE_PARAMETER_LIST. Parametrii care nu se află în forma ce primește lista implicită de parametri trebuie să fie eliminate din listă prin utilizarea funcției DELETE_PARAMETER, pentru a evita eroarea generată de Forms. Exemplul următor apelează forma forma_noua cu lista implicită de parametrii din care a fost sters parametrul p1.

```
DECLARE
    lista_id PARAMLIST := GET_PARAMETER_LIST('default');
BEGIN
    DELETE_PARAMETER(lista_id, 'p1');
    OPEN_FORM('forma_noua', ACTIVATE, NO_SESSION, lista_id);
END;
```

Parametrii sunt transmiși numai într-o singură direcție, adică valoarea unui parametru se transmite de la parametrii de ieșire la cei de intrare, dar orice schimbare a valorii parametrilor de intrare nu are influență asupra parametrilor de ieșire ai formei apelante (pentru a transmite valori în ambele direcții trebuie utilizate variabile globale).

Un parametrul poate fi accesat în PL/SQL ca o variabilă de legătură a formei, sub forma :PARAMETER.nume_parametru.

11.2.13. Meniuri atașate (Popup Menus)

Meniurile atașate (popup menus) sunt meniuri dependente de context care apar în mediul Windows la apăsarea butonului drept al mouse-ului. În Forms, aceste meniuri pot fi atașate la elemente sau canvas-uri ale formei (folosind proprietatea *Functional/Popup Menu* a acestora). Meniurile "popup" pot conține un număr de elemente de meniu care pot executa un cod PL/SQL, ele fiind similare din acest punct de vedere modulelor de meniu, care vor fi prezentate într-o secțiune ulterioară. Deoarece un meniu "popup" poate fi atașat la mai multe elemente sau canvas-uri, el nu are întrinsec conotația de context. Cu alte cuvinte, dacă un element de meniu trebuie să știe din care element sau canvas a fost apelat, trebuie scris un cod sursă care să transmită această informație. Un meniu "popup" diferă de un modul meniu prin faptul că este parte dintr-o formă și nu prezintă caracteristici întrinseci de securitate.

11.2.14. Unități de program (Program Units)

Unitățile de program (program units) sunt de patru tipuri: *proceduri (procedures)*, *funcții (functions)*, *specificații de pachete (package specifications)* și *corpuri de pachete (package bodies)*. În momentul când se creează o unitate de program utilizând Object Navigator, sunt specificate numele și tipul acesteia, iar apoi va apărea editorul PL/SQL ce va conține un cadru pentru tipul de unitate de program selectată (numele și tipul unității de program, cuvântul cheie

BEGIN - dacă este cazul - și cuvântul cheie END urmat de ;). Unitățile de program din Form Builder sunt similare funcțional cu procedurile, funcțiile și pachetele stocate. Diferența constă în faptul că, în timp ce acestea din urmă sunt obiecte ale bazei de date și deci pot fi apelate de orice utilitar Oracle, pentru orice aplicație, unitățile de program din Form Builder pot fi definite doar de către Form Builder și numai în interiorul modulului în care au fost definite.

După cum s-a mai discutat în capitolul 9, procedurile și funcțiile sunt tipuri de subprograme definite de utilizator, adică blocuri PL/SQL cu parametri optionali de intrare și ieșire care pot fi apelați din orice alt bloc PL/SQL în cadrul aceluiași modul. Unitățile de program sunt proiectate pentru a se putea evita scrierea acelorași linii de cod de mai multe ori în cadrul aceluiași modul. Cele două tipuri de subprograme definite de utilizator, *procedurile (procedures)* și *funcțiile (functions)*, acționează în mod similar, dar sunt diferite din punct de vedere sintactic. O procedură returnează orice număr de valori prin modificarea valorilor parametrilor transferați prin lista sa de argumente. O funcție, din contră, returnează o singură valoare într-o variabilă de același tip cu funcția atunci când este apelată. O procedură este apelată prin utilizarea numelui procedurii într-un bloc PL/SQL, ca în exemplul:

```
nume_procedura (argumente);
```

iar o funcție este apelată ca parte a unei expresii:

```
:variabila := nume_functie(argumente);
```

Decizia de a folosi o procedură sau o funcție se ia în funcție de valorile care se doresc și returnate. Dacă un subprogram definit de utilizator returnează valori multiple, atunci trebuie folosită o procedură. Dacă un subprogram definit de utilizator returnează o singură valoare atunci poate fi folosită fie o procedură, fie o funcție, dar o funcție este în general o alegere mai bună deoarece codul PL/SQL este mai ușor de citit și de înțeles. Dacă nu se returnează nici o valoare, este mai bine să se aleagă o procedură decât o funcție care va returna o valoare fictivă.

Pachetele reprezintă un grup de subprograme definite de utilizator. Un pachet este compus din două părți, *specificația* și *corpul pachetului*. Specificația pachetului, care reprezintă interfața acestuia, declară constante, tipuri, variabile, exceptii, funcții și proceduri. Specificația pachetului nu conține cod sursă sau definiții de proceduri sau funcții, aceste definiții sunt cuprinse în corpul pachetului, care reprezintă implementarea pachetului. Declarațiile din specificația pachetului sunt publice și pot fi referite sau apelate prin prefixarea numelui obiectului cu numele pachetului, ca în exemplul următor:

```
:variabila:=nume_pachet.nume_functie(argumente);
```

```
nume_pachet.nume_procedura(argumente);
```

Obiectele declarate în corpul pachetului sunt private și vizibile numai în acel pachet. Dacă se dorește, de exemplu, manipularea unei variabile definite în corpul pachetului, trebuie folosită o procedură sau funcție declarată în specificație.

Crearea pachetelor este mai dificilă și necesită mai mult timp decât crearea funcțiilor și procedurilor, dar ele au o serie de avantaje față de subprogramele individuale. În primul rând

subprogramele dintr-un pachet pot partaja ușor variabilele ce sunt definite în specificația pachetului. În al doilea rând, prin aranjarea într-un pachet a unităților de program care depind unele de altele, se delimită clar că unitățile de program (din pachet) trebuie să rămână împreună. Un pachet face puțin probabilă posibilitatea ca o unitate de program necesar să fie uitată.

Oracle Forms conține un set de *pachete predefinite* (*built-in packages*) care pot fi vizualizate folosind Object Navigator. Funcțiile și procedurile cele mai des folosite sunt grupate în pachetul STANDARD. Atunci când acestea sunt apelate, nu este necesar ca numele lor să fie precedate de numele pachetului din care fac parte (STANDARD).

11.2.15. Clase de proprietăți (Property Classes)

Clasele de proprietăți (*property classes*) sunt seturi de proprietăți care pot fi moștenite de alte obiecte. Astfel se permite utilizarea seturilor comune de proprietăți precum și schimbarea în masă a proprietăților. Acest lucru se realizează prin schimbarea valorilor proprietăților în acea clasă de proprietăți din care acestea sunt moștenite. Clasele de proprietăți pot de asemenea moșteni valori din alte clase de proprietăți.

Un element moștează valorile proprietăților și declanșatoarele dintr-o clasă de proprietăți prin specificarea numelui clasei de proprietăți în proprietatea *Subclass Information*. Proprietățile moștenite sunt marcate cu semnul =. Un element poate suprascrie orice proprietate pe care acesta îi moștează de la clasa de proprietăți prin specificarea unei valori diferite în locul valorii moștenite de la clasa. Proprietățile care sunt suprascrise nu mai sunt marcate cu semnul =. Pentru a schimba o proprietate suprascrisă înapoi la o proprietate moștenită, se va apăsa butonul [*Inherit*] din editorul de proprietăți.

În momentul când este creată, o clasă de proprietăți nu are nici o proprietate. Fiecare proprietate și valoarea asociată acesteia sunt adăugate folosind butonul [*Add Property*]. Pentru a crea în mod rapid o clasă de proprietăți bazată pe o mulțime de proprietăți a unui element, se vor selecta proprietățile dorite și se va apăsa butonul [*Property Class*], în acest mod fiind creată o clasă nouă de proprietăți care cuprinde toate proprietățile selectate.

11.2.16. Grupuri de înregistrări (Record Groups)

Un *grup de înregistrări* (*record group*) este foarte asemănător cu o vedere sau cu o tabelă a unei baze de date, diferența fiind faptul că acesta este stocat în memoria rezervată Oracle Forms și nu pe disc. Un grup de înregistrări poate fi manipulat prin program sau se poate baza pe orice interogare asupra bazei de date. Grupurile de înregistrări sunt adesea utilizate pentru LOV-uri. Grupurile de înregistrări pot fi transmise prin referință de la o formă la alte instrumente *Oracle Developer* printr-un parametru.

Deși conceptual un grup de înregistrări poate părea similar unui bloc, acestea sunt obiecte distințe, proiectate pentru scopuri distințe. Un bloc este o colecție de elemente, în timp ce un grup de înregistrări este o colecție de date.

11.2.17. Rapoarte (Reports)

Un *raport* (*report*) în Form Builder este un obiect care realizează integrarea cu un modul raport (construit folosind Oracle Reports), permitând dezvoltatorului unui aplicație în Forms să introducă parametri în paleta de proprietăți asociată acestui obiect, cum ar fi numele fișierului modulului raport și modul său de execuție. Rapoartele pot fi lansate în execuție utilizând funcția predefinită `RUN_REPORT_OBJECT`. Rapoartele nu există în versiunile mai vechi de Forms Builder, apelarea unui modul raport făcându-se prin intermediul procedurii `RUN_PRODUCT`.

11.2.18. Atribut vizual (Visual Attribute)

Un *atribut vizual* (*visual attribute*) este o colecție de proprietăți de afișare la care se poate face referință în loc de a specifica direct fiecare proprietate a unui obiect. Avantajul atributelor vizuale constă în faptul că prin ele se poate asigura consistența diferențelor tipuri de elemente prin schimbarea proprietăților atribuției vizuale în loc de schimbarea proprietăților fiecarui element în parte. Atributele vizuale pot fi conținute în clase de proprietăți.

11.2.19. Ferestre (Windows)

O *fereastră* este un element esențial al unei interfețe grafice. Ferestrele sunt suprafețe rectangulare ce comunică informații utilizatorului și conțin elemente ce pot fi manipulate.

O formă conține o fereastră specială cunoscută ca *fereastră MDI*. MDI reprezintă prescurtarea de la *interfață pentru documente multiple* (*multiple document interface*) și este utilizată în aplicații pentru a gestiona ferestre multiple. Fereastra MDI conține în mod uzual o opțiune *Window* în meniu său, care enumera ferestrele care sunt copii ai ferestrei MDI, și oferă opțiuni de gestionare pentru aceste ferestre, cum ar fi *Tile* (ferestrele sunt afișate alăturat) și *Cascade* (ferestrele sunt suprapuse). În Oracle Forms, fereastra MDI poate afișa canvas-uri bară de instrumente verticală sau orizontală, prin setarea proprietăților *Physical/Form Vertical Toolbar Canvas* sau *Physical/Form Horizontal Toolbar Canvas* pentru formă. Fereastra MDI nu poate fi manipulată în mod direct în Form Builder, și nu apare ca fereastră în Object Navigator.

Ferestrele pot fi mutate, aranjate și redimensionate de către utilizator. Acest lucru nu produce nici o modificare a vreunui canvas pe care îl poate afișa o fereastră, exceptie făcând faptul că aria de afișaj poate crește sau se poate micșora. Cu alte cuvinte, când un utilizator redimensionează o fereastră în Forms, elementele afișate în acea fereastră păstrează aceleasi dimensiuni, iar canvas-ul devine mai mult sau mai puțin vizibil.

11.3. Utilizarea variabilelor

În Oracle Forms există mai multe tipuri de variabile. *Variabilele locale* (*local variables*) sunt create și inițializate în secțiunea `DECLARE` a unui bloc PL/SQL și sunt disponibile numai în blocul PL/SQL în care au fost declarate. Variabilele locale pot fi definite în orice

declanșator sau unitate de program. Datorită scopului și consumului mic de resurse, variabilele locale sunt tipul preferat de variabile în orice situație. Când este nevoie de partajarea datelor sau stocarea permanentă a valorilor, valorile variabilelor locale pot fi ușor copiate în orice alt tip de variabilă.

Un element nebazat pe o tabelă poate fi afișat sau ascuns. Așa cum s-a discutat mai devreme, un element poate fi ascuns prin atribuirea valorii *Null* proprietății *Physical/Canvas*, ceea ce înseamnă că elementul nu este atribuit nici unui canvas. Un element ascuns nebazat pe o tabelă poate fi folosit ca o variabilă în cadrul unei forme, pentru a transmite o valoare între două sau mai multe declanșatoare sau unități de program PL/SQL. Folosirea ca variabilă a unui element nebazat pe tabelă dintr-un bloc bazat pe tabelă are avantajul că acesta deține o instanță pentru fiecare înregistrare din acel bloc. Cu alte cuvinte, atunci când se atribuie sau se preia o valoare dintr-un unui element nebazat pe tabelă al unui bloc bazat pe tabelă, se atribuie sau se preia valoare respectivă numai pentru înregistrarea curentă. Acest lucru poate fi folosit de exemplu în calculul subtotalurilor și al valorilor bazate pe valorile înregistrărilor. Un element nebazat pe tabelă dintr-un bloc nebazat pe tabelă poate fi tratat ca un simplu spațiu de stocare. Acesta diferă față de un element nebazat pe tabelă dintr-un bloc bazat pe tabelă doar prin faptul că fiecare valoare nu este asociată unei anumite înregistrări. Elementelor nebazate pe o tabelă le sunt atribuite tipuri de date explicite și sunt disponibile atâtă timp cât forma rulează, făcând dintr-un element nebazat pe tabelă un bun candidat la stocarea valorilor partajate de declanșatoare și funcții folosite în coordonarea operațiilor dinsău formă. Observați că unitățile de program PL/SQL stocate în biblioteci nu pot face referință la un element al formei, astfel că valorile respective trebuie transmise ca parametrii.

Variabilele globale (global variables) sunt variabile speciale, create în momentul când li se atribuie valori. De exemplu, instrucțiunea PL/SQL :GLOBAL.costel := 'Dummy'; creează o variabilă globală numită costel. Variabilele globale sunt disponibile pentru toate instanțele instrumentelor client Oracle ce operează pe o anumită mașină. Prin urmare, o variabilă globală poate fi utilizată pentru transmiterea de valori între aplicații sau între instanțe ale aplicațiilor. Totuși, este preferabil să se folosească parametri ori de căte ori este posibil, deoarece la utilizarea variabilelor globale apar problemele următoare:

- Variabilele globale au mereu lungimea de 255 bytes și pot stoca numai date de tip text. Pentru a stoca informații numerice sau de tip dată calendaristică este necesară o conversie de tip, care poate cauza probleme.
- Variabilele globale alocă memorie pentru stocarea variabilelor până în momentul când sunt șterse *explicit*. În particular, pe o platformă Windows, utilizarea variabilelor globale poate cauza fragmentarea memoriei și poate stingeri utilizarea memoriei sistemului.

În general este bine să nu se utilizeze variabile globale decât dacă nu există alte opțiuni sau dacă necesitățile specifice ale aplicației conduc la utilizarea de variabile globale. Atunci când sunt utilizate, fiecare variabilă globală trebuie să fie ștersă în mod explicit și memoria alocată ei eliberată, utilizând funcția ERASE, de exemplul ERASE ('GLOBAL.costel'). Unul dintre avantajele variabilelor globale este posibilitatea de a examina valorile variabilelor globale în momentul depanării formei. Aceasta se face cel mai bine atribuind valori variabilelor globale numai în punctele strategice de testare, astfel încât toate referințele la variabilele globale să poată fi șterse atunci când forma este finalizată.

Variabilele de sistem sunt variabile speciale utilizate pentru a seta sau regăsi valorile parametrilor folosiți de Oracle Forms, cum ar fi :SYSTEM.SUPPRESS_WORKING, care poate avea valoarea True sau False și este folosită pentru a suprima mesajele de lucru ale Oracle Forms, sau :SYSTEM.MOUSE_ITEM, care returnează elementul în care se găsește cursorul mouse-ului, în caz că există un astfel de element. Marea majoritate a variabilelor de sistem pot fi doar citite, nu și modificate.

Parametrii sunt utilizati în mod explicit pentru transmiterea valorilor între module Oracle Developer. În PL/SQL, un parametru este accesat la fel ca o variabilă globală, fiind însă prefixat de cuvântul PARAMETER. În instrumentele Oracle Developer, altele decât Oracle Forms, un parametru este accesat prin nume, fără a utiliza nici un prefix.

11.4. Crearea și utilizarea meniurilor

Un meniu este un modul separat care este creat folosind Oracle Forms. Un meniu este un set de elemente de meniu definite în mod ierarhic care pot fi selectate de utilizator. Precum o formă, un modul meniu trebuie generat înainte de a fi folosit. Atunci când este stocat într-un fișier, codul sursă al meniului are extensia .mmb, iar meniul executabil are extensia .mmx. Un meniu nu poate fi rulat fără o formă.

Meniurile sunt editate și aranjate vizual în *Editorul de Meniuri (Menu Editor)*. Editorul de meniuri simulează funcționarea meniului aşa cum ar funcționa într-o formă. Editorul de meniuri este reprezentat în figura 11.4.

Un element al unui meniu poate fi de următoarele tipuri: *Plain, Check, Magic, Radio, Separator*. Tipul unui meniu este indicat de către valoare proprietății *Functional/Menu Item Type*. Valoarea implicită a acestei proprietăți este *Plain*. *Check* arată că elementul de meniu poate avea două valori, bifat și nebifat, iar *Radio* este folosit atunci când elementul face parte dintr-un *Radio Group*, numai una dintre elementele grupului putând fi activă. Starea elementelor de tip *Check* sau *Radio* poate fi setată, respectiv preluată, folosind procedura predefinită *SET_MENU_PROPERTY*, respectiv funcția predefinită *GET_MENU_PROPERTY*. *Magic* indică una dintre următoarele elemente de meniu predefinite: *Cut, Copy, Paste, Clear, Undo, Help, About, Quit, Window și Page Setup*. Elementele de tip *Magic* sunt afișate în stilul specific platformei pe care este executată forma, cu acceleratorii corespunzători (tastele asociate) și hiperurile deja definite. Tipurile de elemente de meniu "magice" *Copy, Paste, Clear, Window, Quit și Page Setup* au funcții predefinite, pe când pentru celelalte trebuie definite funcțiile asociate. Elementele de tip *Separator* sunt linii sau alte elemente cosmetice, folosite pentru a delimita celelalte elemente din meniu.

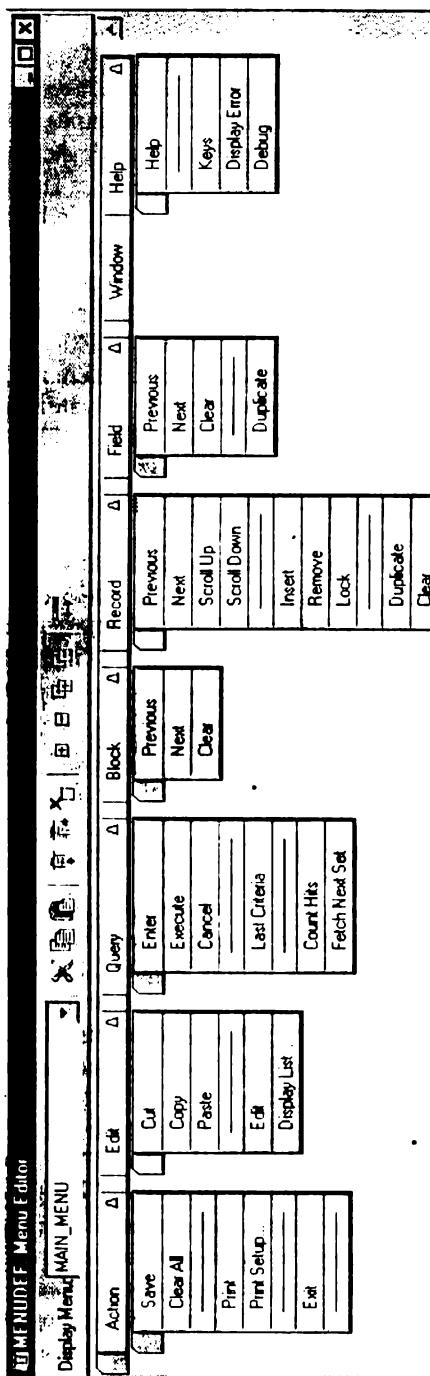


Figura 11.4. Meniul implicit din Menu Editor

Orice element al unui meniu, în afară de cele de tip *Separator*, poate avea o comandă asociată, care este executată atunci când un utilizator selectează un element de meniu. Tipul comenzii asociate este specificat de proprietatea *Functional/Command Type* și poate avea valorile: *Null*, *Menu*, *PL/SQL*, *Plus*, *Form* și *Macro*. Un element de meniu *Null* nu face nimic și este folosit pentru elemente de tip *Separator*. Un element de meniu *Menu* indică faptul că elementul de meniu este elementul părinte al unui submenu. Tipul de element de meniu *PL/SQL* este un tip de comandă ce execută un bloc PL/SQL. Un bloc PL/SQL poate realiza orice acțiune, cum ar fi lansarea în execuție a unor forme, rapoarte, programe externe sau orice operație PL/SQL validă. Codul blocului PL/SQL este specificat de proprietatea *Functional/Menu Item Code* a elementului de meniu. În momentul utilizării PL/SQL într-un meniu, este important de observat că meniul operează ca un modul *extern* și nu este o parte a formei. Din acest motiv, blocul PL/SQL nu poate face referință în mod direct la obiectele din formă. Tipurile de comenzi *Plus*, *Form* și *Macro* nu trebuie utilizate niciodată. Ele există numai pentru compatibilitate cu versiuni precedente și nu sunt necesare.

Atunci când se dezvoltă un meniu, structura ierarhică a meniului este reprezentată și în Object Navigator și în Menu Editor.

Pentru ca o formă să utilizeze un anumit meniu, numele acestuia modul de meniu va fi specificat ca valoare a proprietății *Functional/Menu Module* a acelei forme. Când forma este rulată, meniul va apărea ca parte a formei. Modulul de meniu va fi căutat fie printre fișierele sistemului, fie în baza de date, în funcție de valoarea proprietății *Functional/Menu Source*. Atunci când pentru proprietatea *Functional/Menu Module* este folosită valoarea *DEFAULT*, va fi utilizat meniul intern implicit.

Rolurile meniului și securitatea

Meniurile pot fi conectate cu rolurile bazei de date pentru a furniza diferite meniuri și opțiuni pentru utilizatorii cărora le-au fost acordate diverse roluri în baza de date. Pentru aceasta, proprietatea *Menu Security/Use Security* a modulului de meniu trebuie să aibă valoarea *Yes*, iar lista de roluri a bazei de date care au acces la acest modul de meniu va fi specificată de proprietatea *Menu Security/Module Roles*. Lista rolurilor trebuie introdusă manual, vezi figura 11.5; pentru a fi introduse ca roluri ale modului meniu, nu este necesar ca rolurile respective să existe în baza de date.

Pentru a specifica apoi care dintre aceste roluri trebuie să aibă acces la un anumit element de meniu, se folosește proprietatea *Menu Security/Item Roles* a acestui element de meniu. Pentru a utiliza elementul de meniu, este necesar ca utilizatorului să îl se acorde măcar unul dintre rolurile selectate pentru această proprietate. Așa cum este arătat în figura 11.6, pot fi selectate roluri multiple din lista de roluri specificată în proprietatea *Menu Security/Module Roles* a modulului de meniu. În Microsoft Windows tastele Ctrl și Shift sunt folosite în combinație cu mouse-ul pentru a selecta mai multe elemente pentru *Item Roles*.

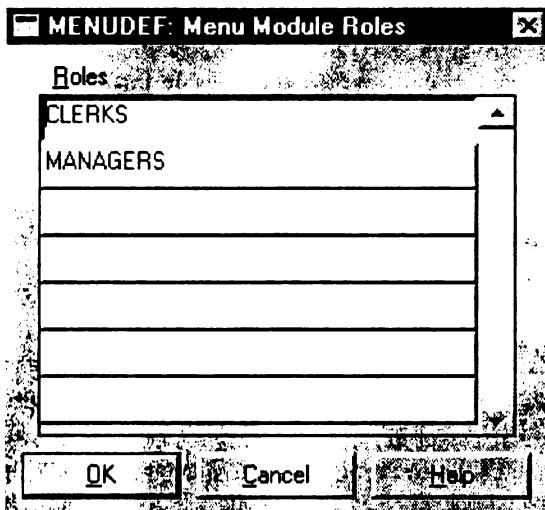


Figura 11.5 - Roulurile asociate modulului meniu

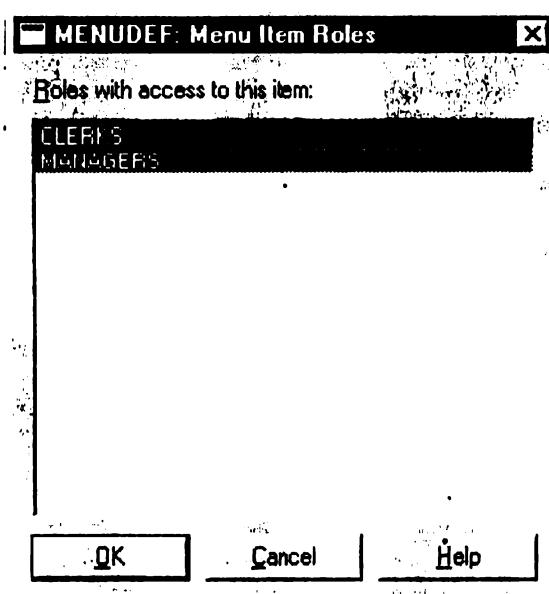


Figura 11.6 - Roulurile asociate elementelor de meniu

Dacă proprietatea elementului de meniu *Menu Security/ Display without Privilege* are valoarea *No* atunci elementul meniului nu va fi afişat pentru un utilizator căruia nu i s-a acordat nici unul dintre rolurile necesare. Valoarea *Yes* a acestei proprietăți indică faptul că elementul va fi afişat, dar nu va fi disponibil utilizatorilor care nu au privilegiul corespunzător.

Roulurile bazei de date pot fi utilizate pentru securitatea meniului indiferent dacă meniu este rulat dintr-un fișier sau din baza de date. Oracle Developer folosește vederea

FRM50_ENABLED_ROLES (creată în script-ul frm60bld.sql) pentru a determina dacă un utilizator are privilegiile adecvate, astfel că toți utilizatorii trebuie să aibă accesul SELECT la această vedere pentru a rula orice tip de meniu atunci când proprietatea meniului *Menu Security/Use Security* are valoarea Yes. Altfel va rezulta eroarea "FRM-10256: User is not authorized to run Oracle Forms Menu".

11.5. Crearea și utilizarea bibliotecilor

11.5.1. Biblioteci PL/SQL

Bibliotecile PL/SQL (PL/SQL libraries) conțin unități de program PL/SQL care pot fi utilizate de către orice instrument sau modul din utilitarele Oracle Developer. Pentru a folosi unități de program PL/SQL dintr-o bibliotecă PL/SQL, biblioteca trebuie atașată la modulul care va apela unitățile de program. Bibliotecile PL/SQL pot fi atașate la forme, meniuri, alte biblioteci PL/SQL și, de asemenea, la afișaje din Oracle Reports sau Oracle Graphics.

O bibliotecă PL/SQL poate fi stocată într-un fișier sau în baza de date. Atunci când este stocată în fișiere, extensia utilizată pentru biblioteca PL/SQL ce conține cod sursă este .pll. Spre deosebire de modulele de forme și modulele de meniuri, pentru a putea fi folosite, nu este necesar ca bibliotecile PL/SQL să fie generate pentru a crea un fișier executabil separat. Fișierul .pll folosit în Form Builder conține atât codul sursă necesar dezvoltării cât și codul executabil necesar în momentul rulării. Când o bibliotecă PL/SQL este atașată unor forme, meniuri sau alte biblioteci PL/SQL în Form Builder, este atașat și fișierul .pll corespunzător. Pentru a putea rula aplicația din Form Builder, este necesară doar compilarea prealabilă a subprogramelor și pachetelor din librărie. Atunci când aplicația este distribuită beneficiarului, se poate genera un fișier .plx din fișierul .pll. Fișierul .plx nu include și componenta de cod sursă care face parte din fișierul .pll. La execuție, Form Builder caută un fișier .plx cu numele corespunzător, dacă nu găsește acest fișier, atunci caută un fișier .pll cu același nume.

O bibliotecă PL/SQL poate conține orice funcții, proceduri sau pachete PL/SQL. Unitățile de program dintr-o bibliotecă PL/SQL nu pot face referință în mod direct la obiectele din modulul la care sunt atașate (formă, meniu, raport sau altă bibliotecă PL/SQL). Ideal, transferul datelor între unitățile de program dintr-o bibliotecă PL/SQL și modulul la care aceasta este atașată se face numai prin intermediul parametrilor (IN, OUT și RETURN - în cazul funcțiilor). Acest lucru face ca biblioteca PL/SQL să fie generică și să poată fi apelată din orice modul care poate transmite corect parametrii către funcțiile din bibliotecă.

Unitățile de program dintr-o bibliotecă PL/SQL pot transmite *indirect* date între biblioteca PL/SQL și modulul la care este aceasta atașată, utilizând funcțiile predefinite NAME_IN și COPY. Funcția predefinită NAME_IN returnează un sir de caractere reprezentând valoarea unei variabile sau a unui câmp din modulul la care biblioteca este atașată. Funcția predefinită COPY salvează valoarea șirului de caractere într-o variabilă sau câmp din modulul la care biblioteca este atașată. Dacă acest nivel de interacțiune cu modulul gazdă este necesar, este în general mai bine să se utilizeze o unitate de program PL/SQL direct în modulul gazdă decât să se utilizeze o bibliotecă PL/SQL.

11.5.2. Biblioteci de obiecte

Bibliotecile de obiecte (object libraries) reprezintă un mijloc de reutilizare a obiectelor în mai multe forme. Spre deosebire de un meniu sau o bibliotecă PL/SQL, o bibliotecă de obiecte nu poate fi asociată explicit cu o formă și nu poate fi afișată deoarece este un depozit de obiecte și nu poate fi accesată în momentul rulării.

La cel mai simplu nivel, o bibliotecă de obiecte reprezintă un container ușor de manuit în care obiectele pot fi plasate sau din care pot fi copiate. Pentru a plasa un obiect și subobiectele sale într-o bibliotecă de obiecte, trebuie creată o bibliotecă de obiecte folosind Object Navigator. O bibliotecă de obiecte conține pagini (*library tabs*) pe care obiectele pot fi aduse prin selectarea obiectului și mutarea acestuia din modulul Forms, din Object Navigator, în fereastra bibliotecii de obiecte. Obiectul este copiat în biblioteca de obiecte, care este stocată într-un fișier cu extensia *.olb*. Fereastra bibliotecii de obiecte poate fi apelată folosind opțiunea *Object library* din meniul *Tools*.

Pentru a folosi într-o formă obiectele dintr-o bibliotecă de obiecte, obiectul este selectat și mutat în formă din fereastra bibliotecii de obiecte în poziția corespunzătoare din Object Navigator. Un obiect poate fi *copiat* sau *subclasat* dintr-o bibliotecă de obiecte. Obiectele copiate sunt simplu copiate în formă, fără nici o altă referință la biblioteca de obiecte. Subclasând un obiect, se copiază obiectul în formă dar se face referire la obiectul original ca obiect părinte. Proprietățile obiectului pot fi suprascrise utilizând paleta de proprietăți. Proprietățile suprascrise pot fi aduse la valorile originale ale obiectului părinte folosind butonul *[Inherit]* din paleta de proprietăți. Dacă obiectul părinte este modificat, proprietățile ne-suprascrise ale obiectului fiu se vor schimba la următoarea încărcare a formei în Form Builder. Forma va trebui să fie regenerată pentru ca schimbările să aibă loc în forma executabilă. Un obiect care este subclasat va afișa propriul obiect părinte în elementul *Subclass Information* din paleta de proprietăți. Dacă obiectul original pe care se bazează un obiect este șters, informațiile moștenite se vor pierde pentru obiectele fiu. Obiectele fiu vor exista însă în continuare ca obiecte independente.

Pentru a edita un obiect dintr-o librărie de obiecte, trebuie mai întâi ca acesta să fie copiat într-o formă, editat și apoi copiat înapoi în librăria de obiecte.

Un obiect dintr-o bibliotecă de obiecte poate fi marcat ca *SmartClass*. Aceasta înseamnă că obiectul trebuie să fie subclasat, nu copiat, atunci când este folosit (totuși acest lucru nu este forțat de Form Builder). Marcarea unui obiect ca *SmartClass* se poate face prin selectarea opțiunii corespunzătoare din meniul ferestrei bibliotecii de obiecte. Identificarea unui obiect ca *SmartClass* se realizează cu ajutorul unui semn ce apare alături de element în fereastra obiectului.

Cum un obiect poate conține la rândul lui alte obiecte (de exemplu un bloc conține mai multe elemente), prin plasarea obiectului părinte într-o bibliotecă de obiecte, sunt plasate în bibliotecă și toate obiectele fiu ale acestuia. Atunci când obiectul părinte este copiat sau subclasat într-o formă, toți fiu lui vor fi în mod similar copiați sau referiți. Obiectele care sunt subclasate ca obiecte fiu ale unui alt obiect, nu au informații despre subclasele individuale în paleta lor de proprietăți, moștenirea fiind exprimată doar pentru obiectul părinte.

11.6. Rularea unei forme

O formă poate fi lansată în execuție din interiorul lui *Form Builder* sau folosind utilitarul *Form Runtime*. Atunci când este rulată, o formă are un număr de funcții predefinite care sunt disponibile fără a fi nevoie ca programatorul să scrie vreo linie de cod. Multe din aceste funcții pot fi apelate prin apăsarea tastelor predefinite. O listă a tastelor disponibile poate fi vizualizată folosind tasta *Show Keys*, adică combinația **CTRL-F1**.

O funcție importantă este *afișarea erorilor* (*Display Error*). Această funcție poate fi apelată prin combinația **SHIFT-F1**. Dacă survine o eroare a bazei de date în timpul execuției unei forme, utilizând această funcție se poate vedea detalii despre instrucțiunea care a cauzat eroarea. Această funcție este foarte folositoare pentru detectarea și corectarea erorilor, alături de depanatorul integrat (*debugger*). O altă funcție este *Meniu Block*, o listă a blocurilor de date ale formei în care se poate naviga.

Utilizatorul poate naviga printre câmpuri, înregistrări și blocuri de date folosind atât mouse-ul cât și tastatura. Modul de navigare este specificat de valoarea proprietății *Navigation/ Navigation Style* a unui bloc, care indică efectul folosirii tastei *Next Item* (Tab) atunci când cursorul se află în ultimul element navigabil: pentru valoarea *Same Record* cursorul este mutat în primul element navigabil al aceleiași înregistrări, pentru valoarea *Change Record* cursorul este mutat în primul element navigabil al înregistrării următoare, iar pentru valoarea *Change Block* cursorul este mutat în primul element navigabil al blocului următor.

De asemenea, de multe ori sunt create butoane pentru deplasarea între înregistrări deoarece tastele predefinite nu sunt întotdeauna înțelese imediat de utilizatori. O bară de instrumente "smartbar" construită în Forms pune la dispoziție săgeți pentru deplasarea printre înregistrări.

O formă construită cu Form Builder poate fi rulată pe Web fără modificări, cu condiția ca ea să nu conțină comenzi, funcții sau proceduri care sunt specifice platformei.

Modul interogare

Unul dintre punctele puternice ale Oracle Forms este capacitatea integrată de interogare prin exemple, în aşa numitul *mod interogare* (*query mode*). O formă poate fi comutată în modul interogare prin utilizarea tastei *Enter Query* (F7) sau prin program. În modul interogare se poate introduce criteriul de interogare, completând în mod corespunzător câmpurile după care se dorește să se facă filtrarea. Criteriul de interogare poate include caracterele de înlocuire (*wildcards*) din SQL % și _ ce pot fi folosite pentru crearea de şabloane. Criteriul de interogare poate deveni foarte complex prin folosirea operatorilor de comparație pentru orice combinație de câmpuri. Acest lucru se poate realiza prin introducerea unei variabile de legătură (aceasta trebuie prefixată de două puncte) în unul din câmpurile aflate pe formă. În această situație, la execuția interogării apare o fereastră unde utilizatorul poate introduce orice criteriu arbitrar referitor la câmpuri (vezi figura 11.7).

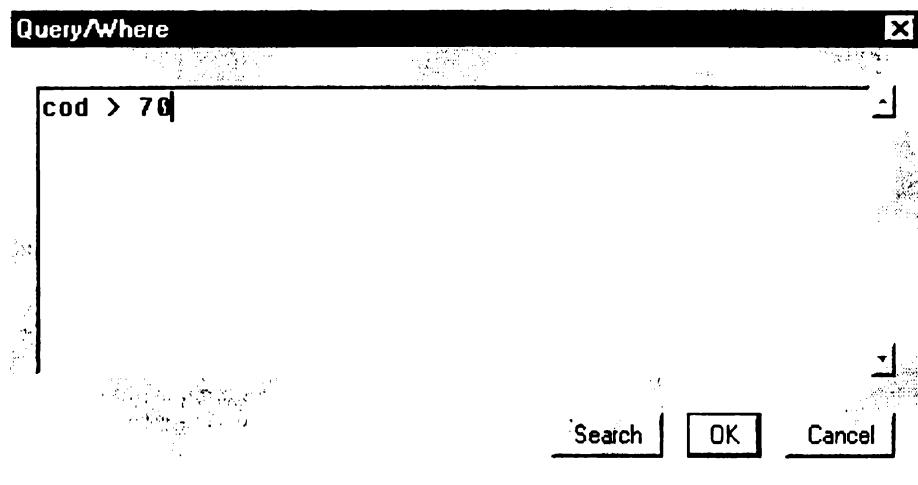


Figura 11.7 - Fereastra de introducere a criteriului de interogare

Când o formă este în modul interogare, dezvoltatorul de aplicații poate exercita doar un control limitat asupra formei. Deoarece nu au fost introduse date, multe funcții (ca de exemplu NEXT_RECORD) nu au sens și multe trigger-e nu se declanșează. Unele trigger-e pot fi specificate să se declanșeze sau nu atunci când forma este în modul interogare.

Executarea unei interogări se face prin apăsarea tastei *Execute Query* (F8) sau prin program. Când este executată o interogare, toate înregistrările returnate pot fi parcuse de la un capăt la altul și vizualizate sau modificate.

Capitolul 12

Aplicație în Forms

Pentru a construi o aplicație rapid și eficient, modelul bazei de date ce stă la baza aplicației trebuie să fie solid construit înainte de lansarea lui *Oracle Forms*.

Aceasta înseamnă că datele sunt modelate și normalizate mai întâi și structurile bazei de date sunt create înainte de încercarea construirii de aplicații. Deoarece o aplicație este construită pe baza unor structuri de date, proiectarea atență a acestora ajută în construirea unei aplicații solide. Pe de altă parte, dacă structurile de date ar fi incorrect construite, aplicația ar avea o bază nesigură.

Prezentăm în cele ce urmează o aplicație relativ simplă pentru o agendă. Aplicația va ține evidența contactelor, precum și unul sau mai multe numere de telefon pentru fiecare contact. Se vor folosi două tabele, CONTACTE și CONTACT_TELEFOANE, legate între ele printr-o relație cheia primară și cheia secundară.

Comenzile DDL folosite pentru crearea structurii bazei de date a aplicației sunt prezentate în cele ce urmează:

```
CREATE TABLE contacte (
    contact_id      NUMBER(9,0)      NOT NULL,
    nume            VARCHAR2(20)     NOT NULL,
    zip_cod         NUMBER(9,0),
    prenume          VARCHAR2(20),
    salut            VARCHAR2(10),
    linie_adresa_1  VARCHAR2(40),
    linie_adresa_2  VARCHAR2(40),
    oras             VARCHAR2(35),
    departament      VARCHAR2(2),
    adresa_email    VARCHAR2(30));

CREATE TABLE contact_telefoane (
    contact_id      NUMBER(9,0)      NOT NULL,
    numar_telefon   VARCHAR2(20)     NOT NULL,
    tip_numar_telefon  VARCHAR2(10)  NOT NULL );

CREATE SEQUENCE contact_id_seq;

ALTER TABLE contacte
    ADD CONSTRAINT contact_id_pk
    PRIMARY KEY (contact_id);

ALTER TABLE contact_telefoane
    ADD CONSTRAINT contact_id_fk
```

```

FOREIGN KEY (contact_id)
REFERENCES contacte (contact_id);

ALTER TABLE contact_telefoane
ADD CONSTRAINT tip_numar_telefon_ck
CHECK (tip_numar_telefon IN
('FAX', 'ACASA', 'SERVICIU', 'PAGER', 'CELULAR')) ;

```

Sunt câteva lucruri importante de spus despre declarațiile DDL de mai sus. În primul rând, constrângerile de cheie primară și cheie străină sunt impuse la nivelul bazei de date. Acest lucru ajută aplicația în două sensuri importante: întâi serverul bazei de date asigură că aceste constrângerile sunt respectate, eliberând astfel aplicația de o povară, și, în al doilea rând, *Form Builder* poate folosi aceste constrângerile pentru a construi relații între blocurile bazate pe aceste tabele.

După rularea comenziilor DDL de mai sus se lansează *Form Builder*. Dacă nu ați anulat sereașura de dialog "Wellcome" (Fig. 12.1), selectați opțiunea "Build a new form manually" pentru a construi o nouă formă manual. Un nou modul, numit "MODULE1", este automat creat după intrarea în *Form Builder*, și apare în Object Navigator. Numele și titlul formei pot fi schimbată în ceva diferit de "MODULE1" prin selectarea modulului "MODULE1" și folosirea opțiunii din meniul principal *Tools>Property Palette*, proprietățile *General/Name* și *Functional/Title*.

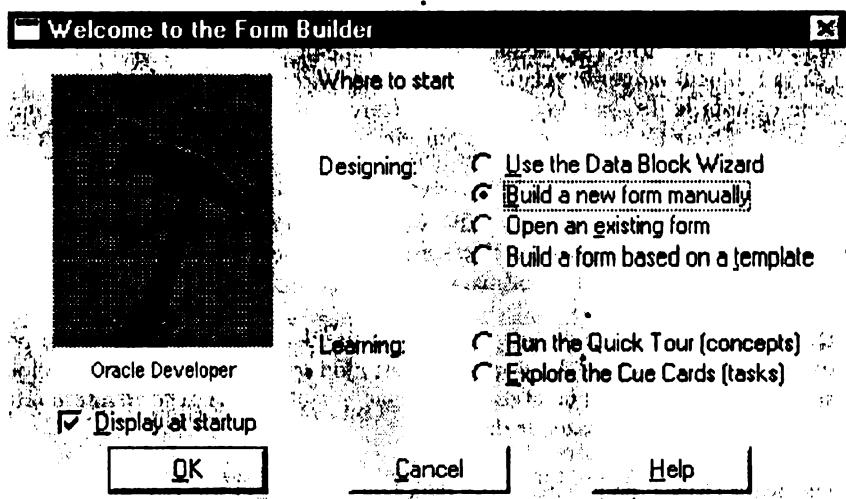


Figura 12.1 Dialogul inițial din Form Builder

Primul pas în crearea aplicației este folosirea opțiunii din meniu *Tools>Data Block Wizard* pentru deschiderea fereastră de creare a unui nou bloc. Se selectează "Table or View" pentru stabilirea tipului blocului de date. Pe următoarea fereastră de dialog, ilustrată în Fig. 12.2, va apărea un câmp cu eticheta "Table or View", având în dreapta un buton cu eticheta [Browse].

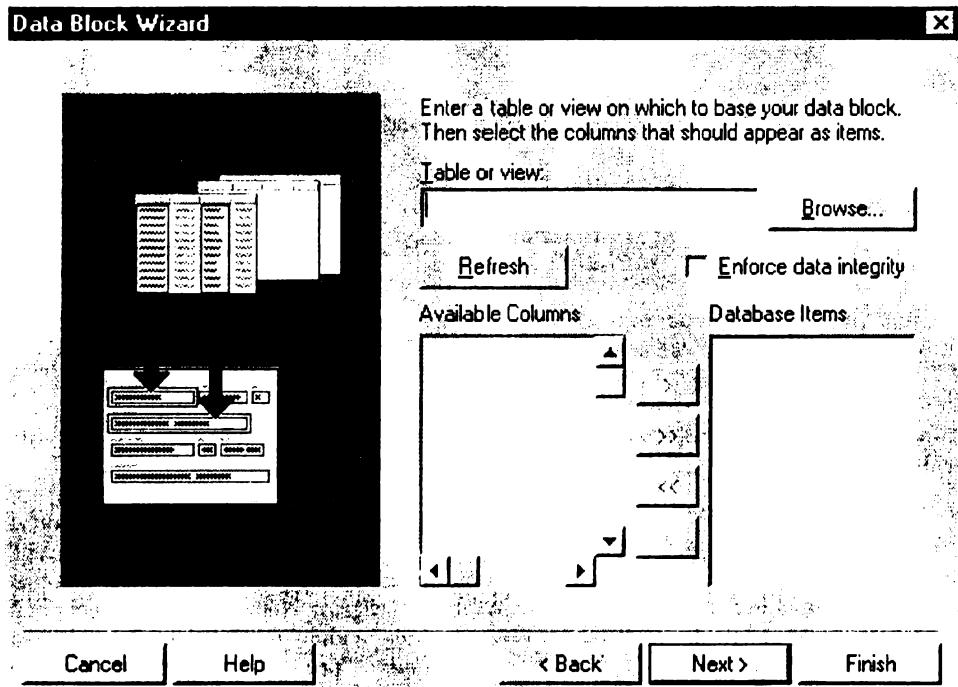


Figura 12.2 Data Block Wizard

Apăsarea butonului [Browse] va stabili în primul rând o conexiune la baza de date dacă aceasta nu este încă stabilită. Apoi va apărea o fereastră de dialog conținând lista tabelelor bazei de date din care urmează să se facă selecția. Tabelul CONTACTE va fi primul selectat pentru că va fi blocul *master* în relația noastră *master/detail*. Apăsând butonul [>>] se vor copia toate câmpurile disponibile ale tabelului (care apar în lista "Available Columns") în lista de selecție "Database Items" (care conține elementele blocului care sunt direct asociate coloanelor din baza de date).

Bifarea căsuței "Enforce data integrity" va avea ca rezultat atribuirea valorii "True" pentru proprietatea *Data/Required* a fiecărui element asociat unei coloane definită NOT NULL sau care face parte din cheia primară a tabeliei.

Notă: Numele blocului nu trebuie neapărat să fie identic cu cel al tabeliei, și poate să schimbe mai târziu prin *Object Navigator*. În acest mod se permite ca mai multe blocuri să fie construite pe baza aceluiși tabel dacă e necesar, și ajută forma să se adapteze mult mai ușor la schimbările structurii bazei de date.

Se selectează [Next], apoi "Create the data block, then call the Layout Wizard", astfel încât la selectarea butonului [Finish] va fi apelat *Layout Wizard*. Se selectează apoi canvasul ("New Canvas") de tip "Content", ceea ce va duce la crearea unui nou canvas de tip "Content" pe care vor fi afișate elementele blocului. Apăsarea butonului [>>] duce la copierea tuturor elementelor disponibile (din lista "Available items") în lista de selecție "Displayed Items" (care conține elementele ce vor fi afișate și deci vor fi vizibile utilizatorului) ca în Fig. 12.3.

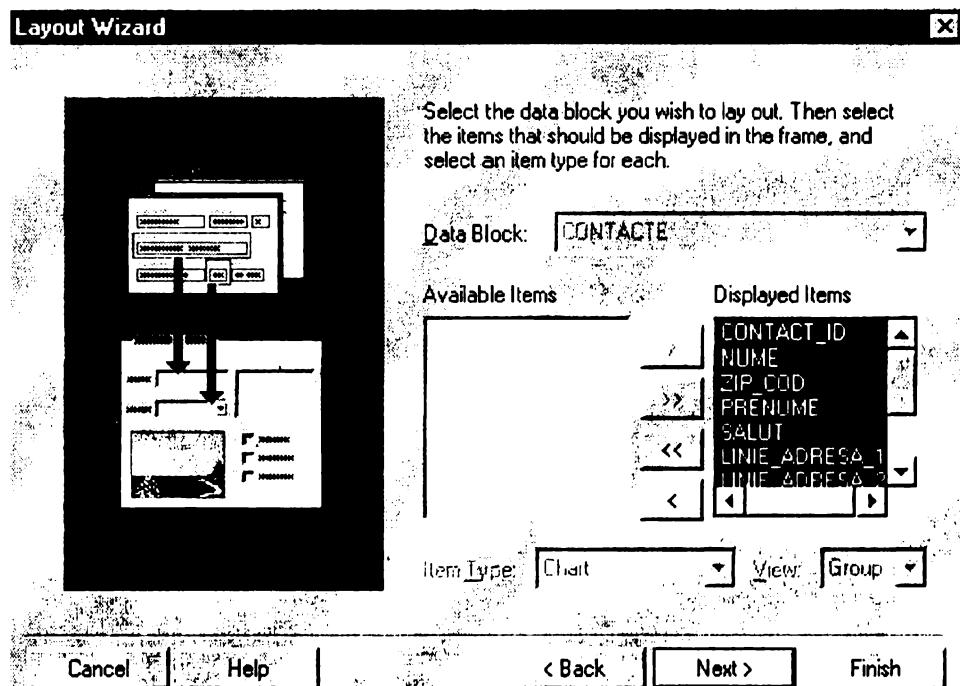


Figura 12.3 Selectarea elementelor ce urmează a fi afișate folosind Layout Wizard

Următoarea fereastră de dialog permite ca etichetele, înălțimile și lățimile inițiale ale elementelor să fie ajustate – vezi Fig. 12.4 - modificându-se în mod corespunzător și proprietățile respective (*Prompt/Prompt*, *Physical/Width*, *Physical/Height*) din fereastra *Property Palette* a formei. Valorile inițiale a acestor proprietăți reprezintă întotdeauna un bun punct de plecare pentru stabilirea valorilor dorite. Evident, aceste valori pot fi modificate și mai târziu, folosind fereastra *Property Palette*.

Mutându-ne pe următoarea fereastră avem posibilitatea să alegem stilul de aranjare (layout) al ecranului: "Form" sau "Tabular". Tipul "Form" va aranja fiecare element unul sub celălalt, în ordinea în care apar în baza de date, pentru fiecare element eticheta acestuia fiind afișată în stânga sa (tipul "Tabular" va poziționa toate elementele pe același rând, așezând eticheta deasupra elementului). Reașezarea elementelor pe ecran va putea fi făcută ulterior folosind utilitarul *Layout Editor*.

Următoarele opțiuni se referă la titlul cadrului construit ("Frame Title"), numărului înregistrărilor afișate pe ecran ("Number of Records Displayed"), și distanța dintre înregistrări ("Distance Between Records"). Pentru exemplul nostru, titlul va fi "Contracte", numărul de înregistrări afișate 1, iar distanța dintre înregistrări 0 (această valoare nu contează dacă doar o singură înregistrare este afișată pe ecran, aşa cum este cazul de față). Când se apasă pe butonul [Finish], se creează aranjamentul și se afișează *Layout Editor* ca în Fig. 12.5.

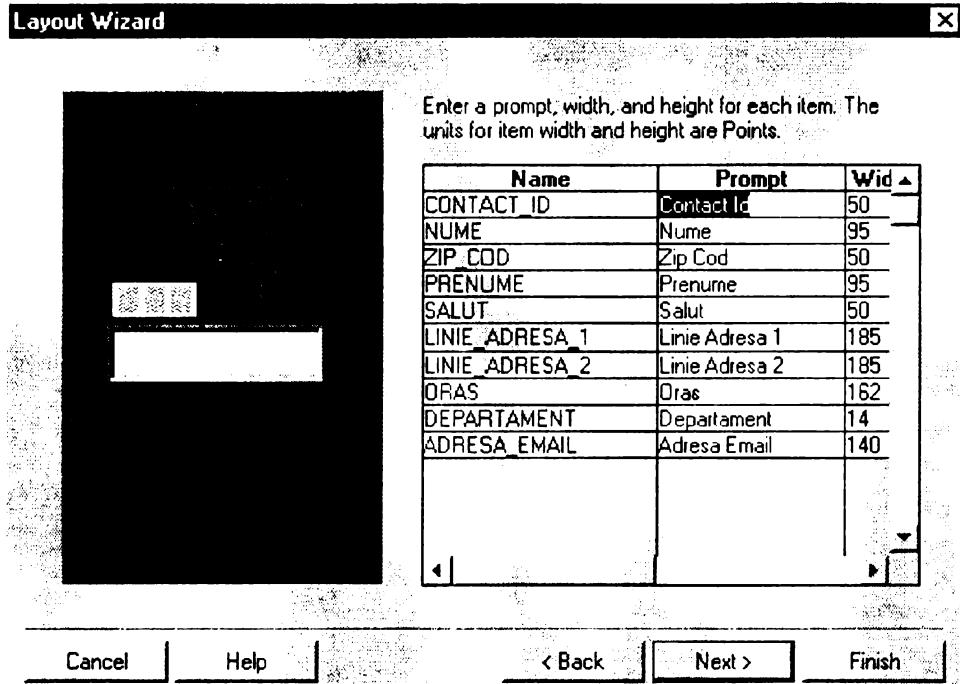


Figura 12.4 Fereastra pentru ajustarea etichetelor și mărimii elementelor

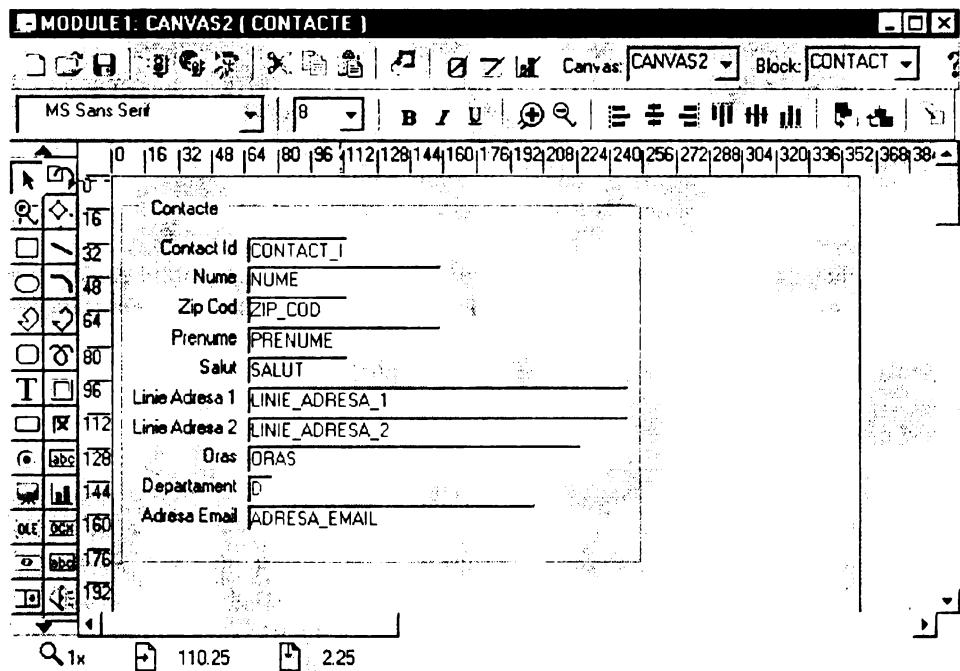


Figura 12.5. Înfișarea inițială a blocului creat.

Deși *Layout Editor* aranjează elementele din bloc în ordinea în care apar coloanele corespondente în definiția tabelului, ele pot fi rearanjate în orice ordine, poziția lor pe ecran fiind independentă de ordinea lor din baza de date. De multe ori, ordinea inițială a elementelor este schimbată datorită necesităților aplicației. În acest exemplu, cheia primară și coloanele NOT NULL au fost plasate primele.

Textul pentru etichetă (prompt) este generat într-un font predefinit (pentru platformele Microsoft acesta fiind "Ms Sans Serif"), dar acesta poate fi schimbat ulterior. Schimbarea fontului se poate face repede selectând toate textele etichetelor și folosind opțiunea *Format>Font...* din meniu pentru schimbarea atributelor tuturor elementelor selectate în același timp. Selectarea mai multor elemente se poate face prin apăsarea butonului stâng al mouse-ului într-un loc fără elemente, tragerea unui dreptunghi în jurul elementelor ce urmează să fie selectate și eliberarea butonului, ori prin apăsarea tastei Shift și selectarea fiecărui element în parte.

Notă: Când elementele se rearanjează pe ecran trebuie să fie de asemenea rearanjate și în Object Navigator. Ordinea elementelor din Object Navigator determină ordinea de navigare a elementelor folosind tastele Tab și Shift-Tab.

Frame-ul din jurul elementelor este șters prin selectarea lui cu un singur clic al mouse-ului și prin selectarea opțiunii *Edit>Clear* din meniu (sau prin apăsarea tastei Delete). Apoi elementele pot fi rearanjate și aliniate în subgrupuri logice.

Deoarece eticheta elementului este o proprietate atașată elementului, ea nu poate fi manipulată independent, fiind dependentă de poziția elementului. Proprietățile care descriu legătura dintre un element și etichetă apar în fereastra Property Palette a elementului, grupate sub *Prompt*. O etichetă poate fi înlăturată prin stergerea textului din proprietatea *Prompt/Prompt* a elementului. Selectarea textului etichetei în Layout Editor va permite mutarea acestuia sau de element; selectarea unui element în Layout Editor va muta atât elementul cât și textul etichetei sale.

La aranjarea elementelor pe ecran se poate folosi și opțiunea de meniu *Arrange>Align Objects*. Mai întâi se selectează obiectele ce urmăreză să fie aranjate, apoi se folosește această opțiune de meniu pentru afișarea cutiei de dialog *Alignment Settings* (Fig. 12.6). Folosirea acestei opțiuni permite alinierea precisă a elementelor, ceea ce economisește timp și îmbunătățește aspectul aplicației. Operația de aliniere poate fi ușor înșelătoare și necesită ceva timp pentru ca utilizatorul să se obișnuiască cu ea. Opțiunea *Align Left* de exemplu, va alinia toate obiectele selectate cu cel mai din stânga obiect selectat. Dacă se dorește alinierea marginilor stângi ale obiectelor față de un alt obiect care nu e cel mai din stânga, mai întâi trebuie să se traseze toate celelalte obiecte în dreapta obiectului luat ca punct de referință.

O opțiune folosită pentru aranjarea obiectelor este *Arrange>Group*. Când elementele sunt colectate într-un grup, grupul este tratat ca un singur obiect de către Layout Editor. Deci, ori de câte ori grupul este mutat în Layout Editor, spațiul din interiorul grupului și poziția elementelor în cadrul grupului vor fi păstrate.

Pentru o aliniere exactă a elementelor, se selectează elementul și se folosesc săgețile. Aceasta permite mutarea elementului către o unitate, ceea ce e mult mai precis față de folosirea mouse-ului.

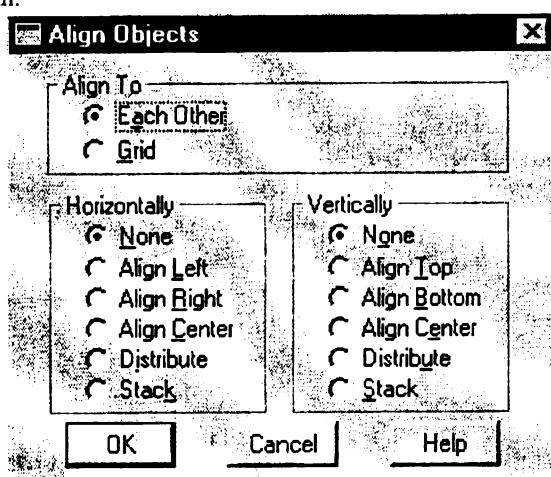


Figura 12.6 Opțiunea din meniu de aranjare a obiectelor

Poziția exactă și mărimea pe ecran a elementului pot fi determinate prin vizualizarea și modificarea proprietăților lui. Unitățile folosite sunt cele specificate de către proprietatea formei *Physical/Coordinate System* (Aceste unități pot fi schimbate oricând, dar precizia poate fi pierdută la schimbarea în unități superioare). Pentru vizualizarea proprietăților unui element afișat în Layout Editor, se face dublu clic pe element. Rearanjarea elementelor pe canvas este redată în Fig. 12.7.

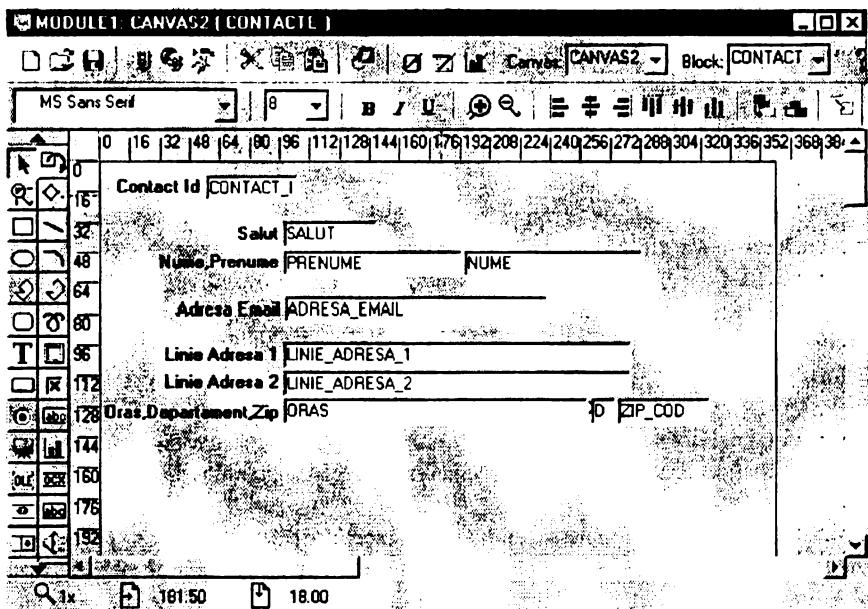


Figura 12.7 Rearanjarea elementelor pe canvas

Se adaugă apoi blocul detaliu al formei și legătura *master/detail* dintre blocul existent și acesta. Ca și înainte, pentru a crea blocul se folosește facilitatea *Tools>Data Block Wizard* (Fig. 12.8). Blocurile pot fi de asemenea create prin selectarea clasei *Data Blocks* în Object Navigator și prin selectarea fie a opțiunii *Navigator>Create* din meniu, fie a opțiunii *Create* din bara de instrumente a Object Navigator, ceea ce permite, de asemenea, folosirea facilității Data Block Wizard. Deoarece Data Block Wizard creează automat elemente și relații, este mult mai ușoră folosirea acestuia decât crearea manuală a blocului.

Notă: Dacă un bloc este selectat în Object Navigator, Data Block Wizard va presupune că schimbă proprietățile blocului selectat, și nu faptul că se creează un bloc nou.

Prin apăsarea butonului [Browse], tabelul *CONTACT_TELEFOANE* poate fi selectat ca tabel de bază pentru noul bloc. Butonul [>>] copiază toate coloanele disponibile în lista "Database Items", ca în Fig. 12.8.

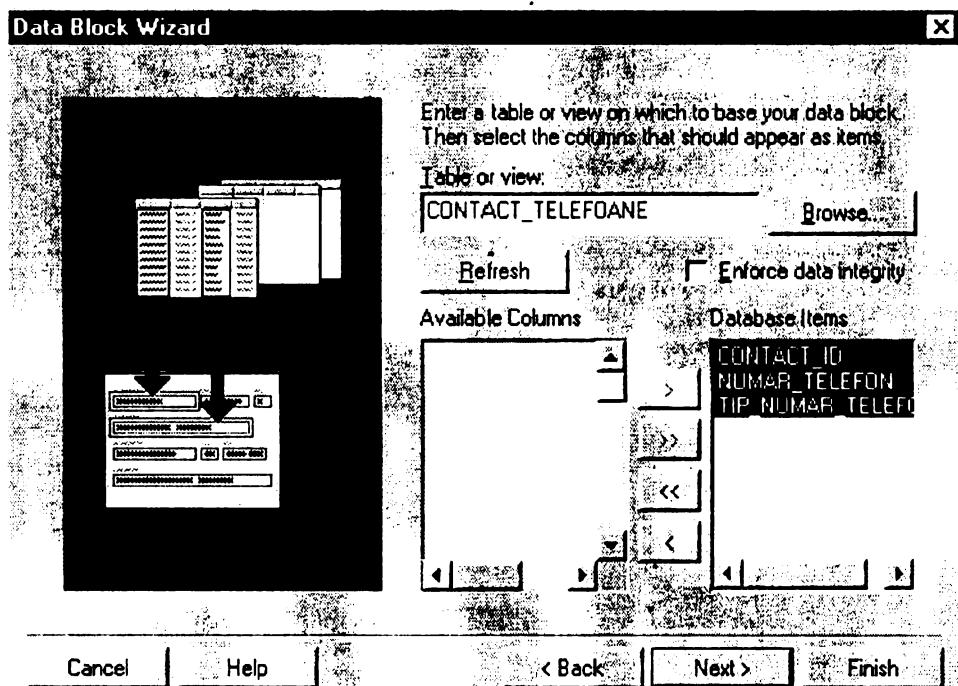


Figura 12.8 Data Block Wizard

Deoarece acesta este al doilea bloc de date al formei, în Data Block Wizard va apărea o nouă fereastră pentru crearea legăturii dintre blocuri. Prin selectarea butonului [Create Relationship] se afișează o listă cu legături cheia primară/cheia străină ce au fost stabilite în baza de date. În acest caz, trebuie să fie selectată legătura stabilită de către constrângerea *CONTACT_ID_FK* a cheii străine. Legătura corectă dintre blocurile de date este automat încărcată din baza de date. Fig. 12.9 arată acest dialog.

După crearea blocului CONTACT_TELEFOANE, este lansat Layout Editor pentru a plasa elementele din blocul de date în canvas. Afisarea coloanei CONTACT_ID ar fi redundanță, de aceea nu este selectată în lista elementelor afisate pe canvas ("Displayed Items"). Selecțiile sunt arătate în Fig. 12.10.

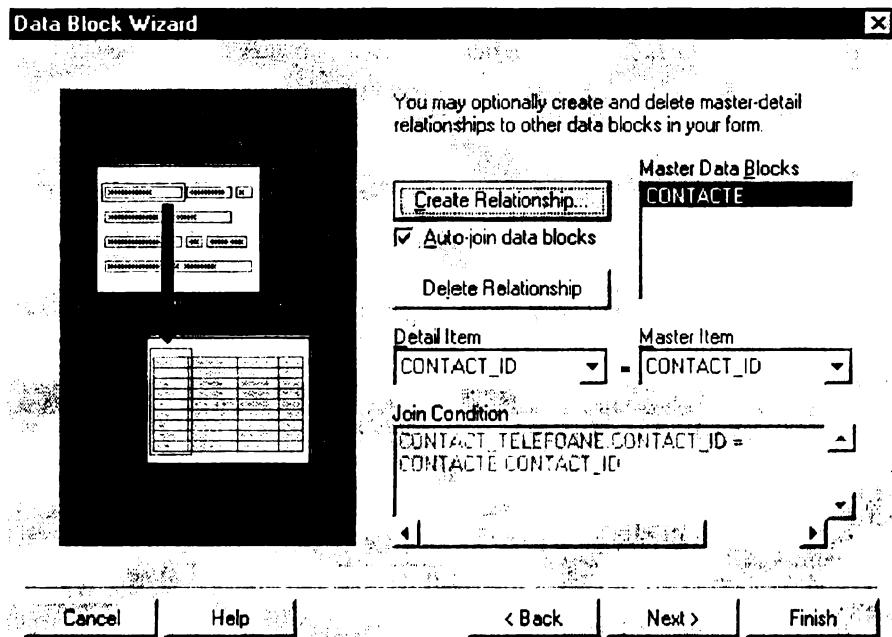


Figura 12.9 Definirea unei relații

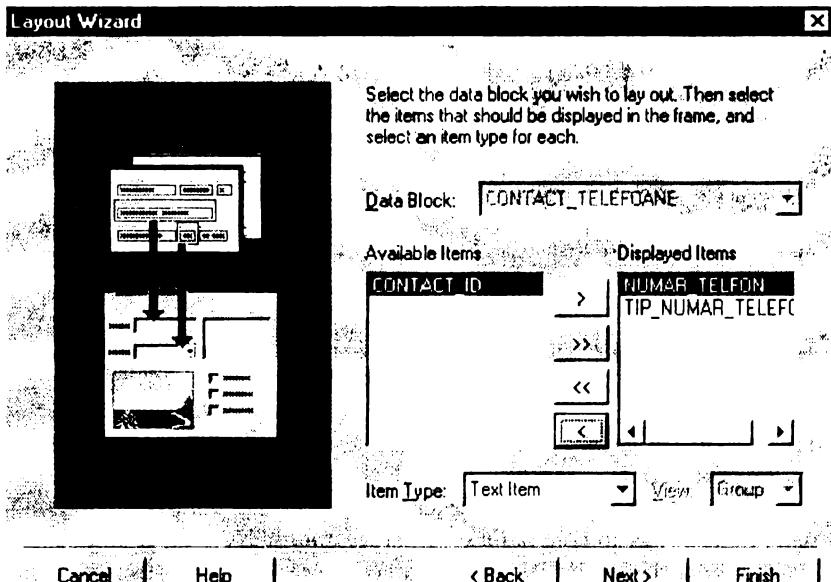


Figura 12.10 Adăugarea elementelor din blocul Contact_Telefoane în Layout Editor

De această dată trebuie aleasă o aranjare a elementelor (layout) de tip "Tabular", iar numărul înregistrărilor afișate trebuie să fie 3. Deoarece fiecare persoană de contact poate avea mai multe numere de telefon, aplicația permite spațiu pentru ca 3 numere de telefon să fie afișate pe ecran în același timp. Deși doar trei numere de telefon sunt afișate simultan, un număr arbitrar de numere de telefon pot fi introduse pentru fiecare contact. Deoarece pot fi introduse mai multe înregistrări decât cele afișate pe ecran, blocului îi va trebui atașat un scroll bar – aceasta se realizează bifând căsuța respectivă din fereastra de dialog. La apăsare butonului [Finish], elementele vor fi aranjate pe CANVAS2 ca și în Fig. 12.11.

Figura 12.11 Layout implicit pentru CONTACT_TELEFOANE

Elementul CONTACT_ID al blocului CONTACT_TELEFOANE este un câmp ascuns. Fiind coordonat cu elementul CONTACT_ID al blocului CONTACTE, afișarea lui nu este necesară.

Noul bloc trebuie de asemenea să fie rearanjat. Fiecare element din blocul CONTACT_TELEFOANE are trei apariții, care din punctul de vedere al aranjării sunt tratate ca un singur element. Spațiul dintre înregistrări poate fi ajustat folosind proprietatea *Records/Distance Between Records* din foaia de proprietăți a elementului (Dacă se dorește, se poate schimba și numărul de înregistrări afișate pe ecran, prin intermediul proprietății *Records/Number of Records Displayed* a blocului).

Constrângerea TIP_NUMAR_TELEFON_CK definită la nivelul bazei de date permite introducerea a doar cinci valori pentru câmpul TIP_NUMAR_TELEFON: "FAX", "ACASA", "SERVICIU", "PAGER", "CELULAR". În plus, valorile respective trebuie scrise cu literă mare (upper case) pentru a fi acceptate de către baza de date (deoarece constrângerea în baza de date este "case sensitive"). Pentru ca aceste restricții să nu incomodeze utilizatorul și pentru a preveni introducerea unor valori incorecte, este posibil ca utilizatorului să i se dea posibilitatea să selecteze valorile respective dintr-o listă în loc să le introducă de la tastatură. Acest lucru se realizează schimbând tipul elementului TIP_NUMAR_TELEFON în *List Item*. Pentru a realiza această schimbare, se deschide foaia de proprietăți a elementului și se schimbă valoarea proprietății *General/Item Type* în "List Item".

De asemenea, proprietatea *Functional/Elements in List* a elementului specifică lista pe care o va vedea utilizatorul și valorile care vor fi introduse sau extrase din baza de date. Elementele listei ("List Elements") sunt cele vizualizate de utilizator, pentru fiecare dintre acestea existând o valoare asociată (specificată în "List Item Value"), ce reprezintă valoarea introdusă sau extrasă din baza de date atunci când pe ecran apare elementul respectiv (vezi Fig.12.12). Se observă că deoarece valorile listei de elemente nu sunt niciodată văzute de către utilizatorul aplicației, ele ar fi putut să fie foarte bine doar niște coduri numerice sau prescurtări.

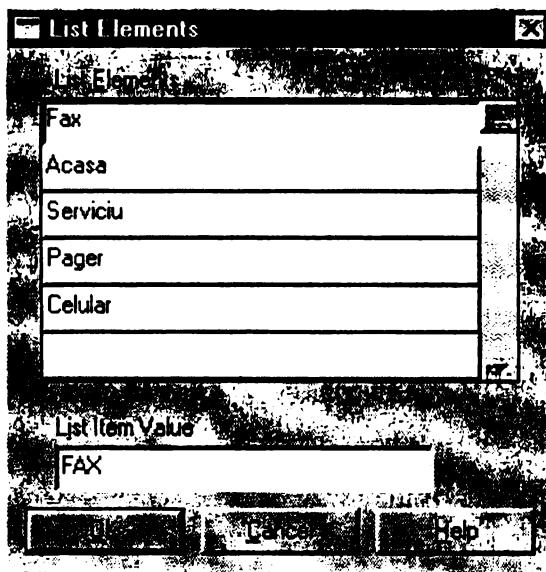


Figura 12.12 – Specificarea unui element din listă

Deoarece constrângerea de la nivelul bazei de date garantează că nici o altă valoare nu poate fi stocată în baza de date, aplicația nu trebuie să prevadă această posibilitate. Dacă însă constrângerea nu ar fi existat, s-ar fi putut folosi proprietatea *Functional/Mapping of Other Values* a elementului pentru a specifica o valoare din baza de date care să fie folosită pentru cazul când în câmpul respectiv este returnată o altă valoare decât cele specificate în listă. Dacă nu este specificată nici o valoare pentru această proprietate, atunci pentru cazul când valorile returnate nu aparțin listei se folosesc valoarea proprietății *Data/Initial Value*, care specifică în același timp și valoarea inițială pentru înregistrările nou create.

Valorile specificate pentru fiecare dintre aceste două proprietăți trebuie să fie atât valori valide ale bazei de date căști să fie incluse în lista de valori a elementului. Dacă nici una din aceste două proprietăți nu are atribuită o valoare, atunci forma nu va extrage din baza de date înregistrările pentru care câmpul în cauză are alte valori decât cele specificate în listă.

În mod implicit, Layout Editor creează un *cadrul* (*frame*) pentru fiecare bloc de date, astfel încât dacă acesta este mutat în Layout Editor, elementele blocului se mută în același timp cu cadrul. Cadrul are propria lui foaie de proprietăți care poate fi modificată. Proprietățile ce sunt grupate sub titlul *Physical* modifică aspectul cadrului, și pot fi folosite pentru obținerea rapidă a aspectului dorit. Alte cadre pot fi repede construite folosind obiectul *frame* din Layout Editor.

În acest punct aplicația este funcțională. Programul poate fi acum generat și rulat folosind opțiunea *File>Run* din meniul principal. Implicit, opțiunile din *Tools>Options* sunt configurate astfel încât un nodul va fi generat înainte de rulare și nu va fi salvat înainte de a fi generat. Astfel, când este selectată opțiunea de meniu *File>Run*, un fișier cu extensia *.fmx* este creat pentru forma executabilă în directorul de lucru. Acest fișier are numele modulului formei și va suprascrie fără nici o avertizare orice alt fișier executabil al formei cu același nume sau în acel director.

Dacă forma este lăsată, se observă că titlurile celor două ferestre ale formei sunt "WINDOW1" (pentru fereastra WINDOW1) și respectiv "Oracle Developer Forms Runtime" (pentru fereastra MDI). Acestea sunt valorile implicate pentru aceste titluri și în mod normal trebuie schimbată în funcție de aplicație. Pentru a modifica titlul ferestrei WINDOW1 se modifică în mod corespunzător proprietatea *Functional/Title* a ferestrei (în cazul de față, acesta este "Contacte"). Dacă această proprietate are valoare nulă, atunci titlul ferestrei va fi specificat de către proprietatea *General/Name*.

Deoarece aplicația rulează în mod curent sub Microsoft Windows, aplicația are o fereastră părinte, ceea ce înseamnă că fereastra MDI sau Multiple Document Interface Window. Aplicațiile Microsoft Windows folosesc interfața MDI pentru a crea aplicații folosind ferestre multiple într-un standard. Oracle Forms aderă de asemenea la acest standard. Schimbarea proprietăților ferestrei MDI a aplicației poate fi făcută prin program, această fereastră putând fi setată prin identificatorul său, FORMS_MDI_WINDOW:

```
SET_WINDOWPROPERTY(FORMS_MDI_WINDOW, TITLE,
                    'Aplicatie in Forms');
```

Observați că, întrucât fereastra MDI există doar în mediul Windows, linia de cod de mai sus are sens (dacă aplicația rulează în acest mediu). Evident, procedura predefinită SET_WINDOW_PROPERTY poate fi folosită pentru schimbarea proprietăților oricărei ferestre a formei în modul rulat după schimbarea titlurilor se găsește în figura 12.13.

Linia de cod de mai sus este cel mai bine plasată în declanșatorul WHEN-NEW-FORM-INSTANCE. Această declanșator este în mod curent folosit pentru inițializarea operațiilor într-o formă și este cel mai potrivit pentru a atribui valori proprietăților prin program. Pentru a crea un declanșator WHEN-NEW-FORM-INSTANCE, se selectează *Triggers* în Object Navigator, și apoi se folosește opțiunea *Navigator>Create* din meniu sau opțiunea *Create* din bara de instrumente a Navigator pentru a selecta și crea declanșatorul.

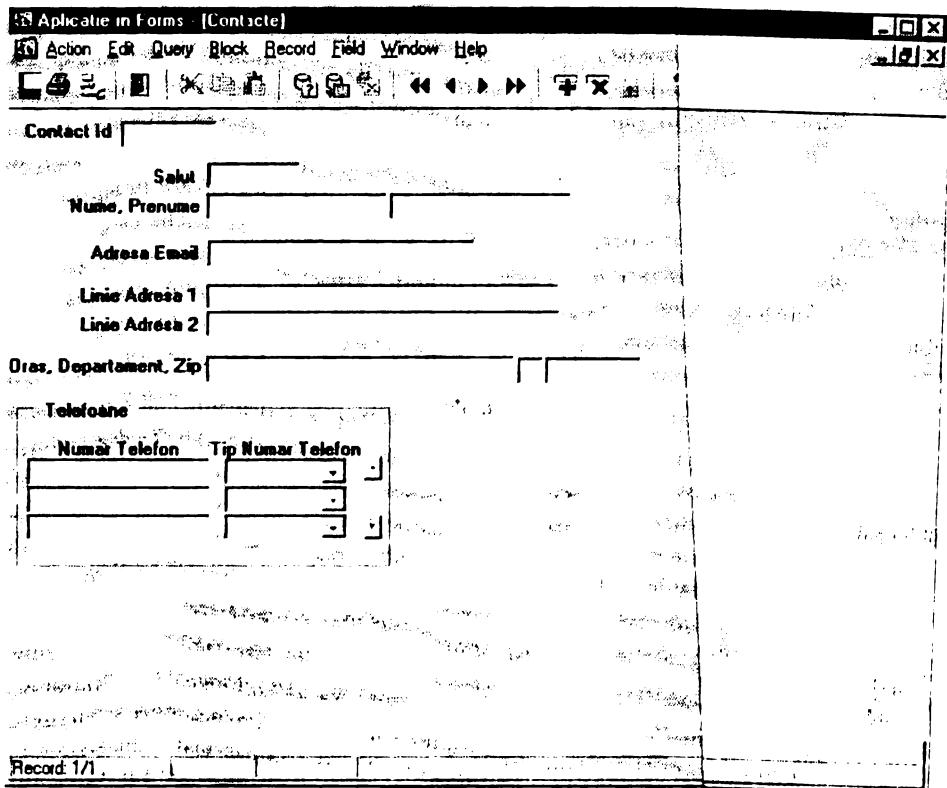


Figura 12.13 Forma la rulare (Client/Serve)

Crearea unui nou declanșator lansează cutia de dialog de selecție al declanșatorului, care conține o listă cu declanșatoare. Un declanșator poate fi căuta sau editat folosind câmpul *Find*. Selectarea unui declanșator lansează editorul PL/SQL unde sunt editați toți declanșatorii și unitățile de program. Odată ce a fost creat, un deșifrator poate fi editat prin localizarea lui cu Object Navigator și dublu clic cu mouse sau prin selectarea opțiunii din meniu *Tools>PL/SQL Editor*.

Tot la rularea formei se observă că nu este posibilă navigație în afara câmpului CONTACT_ID (cursorul rămâne întuit în acesta) atât timp cât acest câmp nu este introdus nimic. Acest lucru este valabil pentru orice câmp "obbligato" (pentru care valoarea proprietății *Data/Required* este *True*). Dacă acest lucru este inconvenient pentru utilizator, el poate fi evitat dacă pentru proprietatea *Functional/Desrequired Enforcement* a formei este aleasă valoarea *True*. În acest caz, va fi permisă garea liberă între elementele înregistrării, verificarea elementelor obligatorii făcând-o la "validarea" înregistrării (la ieșirea din aceasta sau la salvarea ei).

O altă problemă care poate apărea la rularea formei este a navigarea prin intermediul tastaturii, cursorul poate sări dintr-un câmp într-altele mod aparent aleator. Aceasta se întâmplă pentru că elementele sunt parcurse în ceea ce în care apar în Object Navigator. Acest comportament este ușor corectabil prin reordinea elementelor în

Object Navigator în ordinea în care apar pe ecran. Aceasta constă în selectarea fiecărui element și așezarea lui într-o poziție diferită în *Object Navigator*.

După rearanjarea elementelor, utilizatorul va fi capabil să navigheze folosind tastatura în cadrul primului bloc, în ordine, dar când sfârșitul blocului este atins, cursorul se va înapoia la primul element din bloc. Aceasta poate părea puțin bizar unui utilizator care nu e familiar cu blocurile, mai ales dacă mulți nu vor ști să caute tasta de navigare la următorul bloc. Pentru a face aplicația mai clară, acest comportament poate fi schimbat prin schimbarea valorii proprietății de bloc *Navigation/Navigation Style* în "Change Data Block" pentru ambele blocuri, CONTACTE și CONTACT_TELEFOANE.

Un alt lucru care mai rămâne de făcut este legat de câmpul CONTACT_ID. Nu are sens ca utilizatorul să trebuiască să editeze acest câmp, având în vedere că există o secvență a bazei de date creată pentru acest scop. Proprietății *Data/Initial Value* a elementului CONTACT_ID îl poate fi dată valoarea :SEQUENCE.CONTACT_ID_SEQ.NEXTVAL, astfel încât elementul va fi inițializat cu valori din secvența bazei de date. Deoarece secvența se va incrementa de fiecare dată când va fi încărcată valoarea implicită, este probabil ca să existe "spații" între valorile salvate în baza de date pentru acest câmp. În mod normal, aceasta nu constituie o problemă.

Pentru că utilizatorul poate modifica valoarea câmpului CONTACT_ID înainte de salvarea înregistrării, este posibil să se încerce salvarea unei înregistrări conținând un CONTACT_ID deja alocat. Deși baza de date va preveni acest lucru, mesajul de eroare generat ("FRM-40508: Unable to insert records") dă puține informații și este prea general pentru utilizator. Schimbarea proprietății *Database/Enforce Primary Key* a blocului CONTACTE precum și a proprietății *Database/Primary Key* a elementului CONTACT_ID în Yes, va spune formei să verifice unicitatea câmpului înainte de inserarea înregistrării. În acest caz, eroarea returnată pentru o valoare duplicată va fi "FRM-40600:Record has already been inserted", ceea ce dă mai multe informații și este deci mai folositoare pentru utilizator.

Ultima problemă funcțională de corectat este ștergerea unei înregistrări. Deoarece există o relație *master/detail* între blocuri, în mod implicit Oracle Forms va preveni ștergerea unei înregistrări master până când toate înregistrările detail vor fi șterse, returnând mesajul "Can not delete master record when matching detail records exist". Acest lucru este coordonat de către legătura ce aparține blocului CONTACTE. Această legătură are o proprietate numită *Functional/Delete Record Behaviour*, care în mod implicit are valoarea *Non Isolated*. Schimbând valoarea acestei proprietăți în *Cascading*, se modifică declanșatorii de coordonare și comportamentul formei astfel încât atunci când înregistrarea master este ștearsă, vor fi de asemenea șterse și toate înregistrările detaliu.

În acest punct, aplicația este completă și funcțională. Evident, întotdeauna se pot aduce îmbunătățiri, dar nu există defecți majore în proiectarea interfeței. Forma poate fi de asemenea rulată în modul Web.

Anexa 1

Produse Oracle8

Oracle Enterprise Manager

Oracle Enterprise Manager (OEM) este un pachet de utilitare cu interfață grafică pentru administrarea bazei de date. Componentele principale ale OEM sunt următoarele:

- *Backup Manager* oferă posibilitatea efectuării operațiilor de arhivare și recuperare asociate cu baza de date;
- *Data Manager* este un utilitar folosit pentru transferul, exportul, importul și încărcarea datelor. Comanda Export este folosită pentru a extrage date într-un format independent de sistemul Oracle. Datele exportate pot fi încărcate într-o altă bază de date Oracle sau în aceeași bază de date prin folosirea comenzi Import. De asemenea comanda Export se poate folosi pe post de backup logic al bazei de date. Utilitarul Loader este folosit pentru a introduce date în baza de date Oracle din fișiere text sau alte fișiere compatibile cu acest utilitar;
- *Instance Manager* oferă posibilitatea manevrării instanțelor, sesiunilor utilizatorilor și tranzacțiilor nesigure. Cu acest utilitar este posibilă pornirea sau închiderea instanțelor Oracle, precum și manevrarea instanțelor multiple într-o rețea cu server paralel. De asemenea, este posibilă modificarea parametrilor de inițializare și selectarea fișierului de inițializare sau a configurației folosite la pornirea instanței;
- *Lock Manager* vizualizează blocajele dintr-o instanță. Este un utilitar folosit pentru analizarea sesiunilor suspendate și pentru alte situații similare;
- *Oracle Expert* oferă posibilitatea reglării instanței și a performanțelor bazei de date. Acest utilitar generează o listă de recomandări ce pot fi implementate automat pentru a îmbunătăți performanțele;
- *Performance Manager* monitorizează performanța unei instanțe Oracle, prin diferite reprezentări grafice ce conțin statistici de performanță;
- *Schema Manager* oferă posibilitatea efectuării operațiilor de tip Data Definition Language (DDL), cu care se pot crea, modifica, șterge și vizualiza obiecte ale bazei de date cum ar fi tabele, indecsi, clustere, declanșatoare și secvențe;
- *Security Manager* oferă posibilitatea efectuării sarcinilor de administrare a utilizatorilor cum ar fi adăugarea, modificarea și ștergerea utilizatorilor, precum și a drepturilor și profilurilor acestora;
- *Software Manager* ajută la administrarea unui mediu distribuit precum și la automatizarea sarcinilor de administrare a bazei de date;
- *SQL Worksheet* se comportă asemănător cu o sesiune SQL*Plus. Prin urmare, se poate folosi pentru a introduce și a executa comenzi SQL;
- *Storage Manager* oferă posibilitatea efectuării sarcinilor de manevrare a spațiului unei baze de date cum ar fi crearea, modificarea și ștergerea spațiilor de tabel. De asemenea se pot crea sau șterge segmente de rollback;

- *Tablespace Manager* ajută la vizualizarea spațiului folosit la nivelul obiectelor dintr-un spațiu tabel. De asemenea se pot obține informații despre spațiul folosit și spațiul liber din baza de date;
- *TopSession Monitor* oferă posibilitatea monitorizării sesiunilor utilizatorilor activi precum și vizualizarea folosirii resurselor. Această informație poate fi folosită pentru a descoperi performanțele slabe;
- *Oracle Trace* dă posibilitatea modificării comenzi interne SQL pentru a îmbunătăji performanțele sistemului.

SQL*Plus

SQL*Plus este un utilitar care permite utilizatorilor lansarea de comenzi SQL asupra bazei de date, cât și lansarea unor comenzi PL/SQL precum crearea procedurilor, funcțiilor și a pachetelor stocate și a declanșatoarelor bazei de date. Totuși, nu este posibilă pornirea sau oprirea unei instanțe Oracle folosind SQL*Plus.

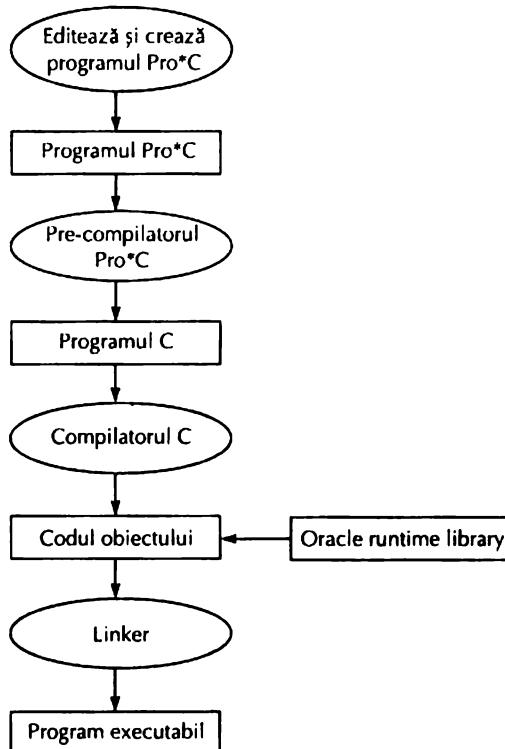
SQL*Plus dă posibilitatea utilizatorului să efectueze următoarele tipuri de comenzi SQL:

- Crearea, modificarea și ștergerea obiectelor bazei de date (operații DDL);
- Interogarea datelor pentru a selecta sau extrage datele depozitate;
- Introducerea, actualizarea și ștergerea datelor (operații DML);
- Accesul și transferul datelor între baze de date;
- Posibilitatea utilizatorului de a introduce date în mod interactiv ;
- Funcții DBA sau sarcini de administrare a bazei de date cum ar fi administrarea utilizatorilor (crearea, modificarea și ștergerea utilizatorilor), administrarea spațiilor tabel (crearea, modificarea și ștergerea spațiilor tabel), precum și operațiile de arhivare și recuperare.

În completarea acestor funcții SQL de bază, SQL*Plus mai oferă numeroase funcții de editare și formatare ce dau posibilitatea utilizatorilor de a tipări sub forma unor rapoarte rezultatele obținute din interogări.

Precompilatoare

Un compilator al unui limbaj de generația a treia nu recunoaște comenziile SQL folosite pentru a realiza interfața cu RDBMS. Prin urmare, dacă este nevoie de puterea și flexibilitatea unui limbaj cum ar fi C, C++, Fortran sau COBOL și se dorește ca acestea să realizeze o interfață cu Oracle8 RDBMS, atunci este nevoie de o unealtă ce poate converti expresiile SQL în cerințe pe care compilatorul unui limbaj le poate înțelege. Precompilatoarele Oracle ajută la încorporarea de instrucțiuni SQL în programe scrise în alte limbi de nivel înalt, numite limbi gazejdă. Oracle oferă mai multe precompilatoare cum ar fi Pro*C, Pro*Cobol, Pro*Fortran și Pro*Pascal. Așa cum este prezentat în figura de mai jos, un program de precompilare citește codul sursă structurat și generează un fișier sursă pe care îl poate procesa compilatorul unui astfel de limbaj.



Oracle Developer

Oracle Developer este un pachet de utilitare care permite dezvoltarea aplicațiilor ce pot accesa o bază de date Oracle. Cu ajutorul utilitarelor conținute în Oracle Developer se pot crea formulare, rapoarte, grafice, interogări și proceduri. De asemenea dă posibilitatea dezvoltării aplicațiilor de Web deja existente sau a altora noi. Spre deosebire de alte limbaje convenționale, Oracle Developer nu necesită scrierea unui program întreg și continuu care să manipuleze datele, el necesitând doar scrierea unei părți de cod, restul fiind generat la crearea în mod vizual a diverselor obiecte.

Oracle Developer cuprinde următoarele componente:

- **Project Builder.** Cu acest instrument se pot urmări și controla documentele aplicației, fișierele codului sursă, graficele, formularele, rapoartele, interogările și aşa mai departe;
- **Oracle Forms.** Formularele și meniurile sunt unele dintre cele mai ușoare și mai populare modalități pentru utilizatori de a interacționa cu baza de date. Oracle Forms realizează prin componentele sale (*Form Builder*, *Form Runtime*) interfața dintre utilizatorul unei aplicații și baza de date. Oracle Forms este bazat pe programarea orientată pe eveniment, fapt ce permite scrierea de cod PL/SQL atașat unui eveniment pentru a defini acțiunea care se va executa la apariția evenimentului respectiv. De asemenea Oracle Forms utilizează și conceptul de programare orientată pe obiect, permisând moștenirea de obiecte (moștenirea anumitor caracteristici), încapsularea (definirea unor caracteristici proprii fiecărei clase de obiecte), polimorfismul (scrierea unor funcții);

- **Oracle Reports.** Este un instrument ce dă posibilitatea generării de rapoarte prin componenta *Report Builder* precum și execuția acestuia prin componenta *Report Runtime*. Rapoartele pot fi vizualizate pe ecran sau transmise automat la imprimantă. De asemenea se pot introduce rapoarte în alte utilitare online cum ar fi browser-ele, graficele și formularele de Web;
- **Oracle Graphics.** Este evident că prezentarea datelor într-un mod grafic și vizual este mult mai eficientă decât prezentarea acestora în formă brută. Oracle Graphics este instrumentul prin care se pot genera grafice corespunzătoare rezultatelor unor interogări. Se poate folosi constructorul de grafice pentru a realiza afișarea acestora în mod interactiv. De asemenea, constructorul de grafice care va permite includerea de grafice în formulare și rapoarte. Oracle Graphics folosește aceeași structură ca și alte componente ale Oracle Developer: generare – prin componenta *Graphic Builder* și execuție – prin componenta *Graphic Runtime*;
- **Query Builder.** Acest utilitar permite utilizatorului să interacționeze cu tabelele bazei de date în mod tabular pe ecran. Tabelele implicate în interogare sunt disponibile pe ecran și utilizatorii pot construi interogarea cu ajutorul mouse-ului. Rezultatele interogării sunt de asemenea afișate pe ecran;
- **Schema Builder.** Aceasta este o unealtă grafică DDL (data definition language) care se poate folosi pentru a crea, modifica și șterge obiecte alcătuiti de date cum ar fi tabele, indecsi, clustere, secvențe;
- **Procedure Builder.** Cu ajutorul acestui instrument se pot construi proceduri în mod interactiv. Procedure Builder este un instrument care prin componente sale (Program Unit Editor, Stored Program Unit Editor, Database Trigger Editor, PL/SQL Interpreter) este folosit pentru a crea, edita, depana și compila programe PL/SQL. De exemplu, componenta Stored Program Unit Editor permite crearea, vizualizarea, modificarea și ștergerea unităților de program (proceduri, funcții, pachete) rezidente în baza de date la care este conectat Oracle Procedure Builder. Pentru crearea, modificarea sau ștergerea unui declanșator este utilizată componenta Database Trigger Editor;
- **Translation Builder.** Acest utilitar permite extragerea unor siruri traductibile din resurse Oracle sau non-Oracle și efectuarea tranzacției dorite. De exemplu se pot traduce fișiere Microsoft Windows (*.rc) și fișiere HTML în fișiere de resurse Oracle.

În mod tradițional, Oracle Developer operează pe o arhitectură client/server, unde utilitarele folosite de client și aplicația se află pe un calculator (un terminal), iar serverul bazei de date se află pe alt calculator (serverul). Aplicația Oracle Developer și serverul bazei de date sunt independente una față de alta, făcând astfel mai ușoară întreținerea lor. Comunicarea dintre cele două se face prin Net8 (sau versiunea anterioară SQL*Net) care trebuie instalat atât pe serverul bazei de date cât și pe terminalul client.

Odată cu dezvoltarea Web-ului, Oracle a introdus o arhitectură pe trei nivele cu un server adițional ce rulează codul aplicației. Astfel, cele trei nivele ale arhitecturii sunt serverul bazei de date, serverul de aplicație și terminalul. Serverul de aplicație rulează aplicația, împreună cu Oracle Developer și un server de web. Terminalul va trebui să aibă doar un browser de web cu capacitatea Java. Comunicația dintre serverul bazei de date și serverul de aplicație se face prin intermediul Net8 (sau SQL*Net), aceste două nivele implementând arhitectura client/server clasica. Cele trei nivele sunt complet independente

una de alta. Un important avantaj al Oracle Developer este faptul că se folosește același software pentru a crea atât aplicații client-server cît și versiuni web ale acestor aplicații.

Oracle Designer

Oracle Designer este un pachet de utilitare ce permite dezvoltarea de aplicații complexe. Utilitarele conținute în acest pachet permit descrierea și analiza modelului, proiectarea acestuia, generarea automată a bazei de date, programarea sa, generarea de comenzi SQL din diagrama entitate/relație, generarea de aplicații obiect. Printre componentele acestui pachet de utilitare se numără: Process Modeling, System Modeling, Design Wizard, System Design, Generation, Repository Administration. Database Designer, o versiune mai simplistă a Oracle Designer, permite realizarea unei descrieri grafice a bazei de date și generează comenzi SQL de definire a datelor (comenzi DDL) utilizând diagrama creată. Database Designer este un instrument ce nu poate fi accesat de mai mulți utilizatori.

Alte produse Oracle

Oracle Power Objects

Acest instrument permite dezvoltarea aplicațiilor ce utilizează un număr relativ mic de resurse ale sistemului. Oracle Power Objects este conceptual similar cu Oracle Developer, dar nu conține multe din facilitățile acestuia.

Oracle Con Text

Dacă este integrat cu orice sistem text, Oracle Con Text poate analiza, filtra sau reduce un text pentru a spori viteza de citire a acestuia, verifică erorile gramaticale, calitatea și stilul.

Media Server

Oracle Media Server furnizează o bibliotecă de funcții multimedia de o varietate foarte mare. Media Server se ocupă de stocarea, regăsirea și gestionarea filmelor, muzicii, fotografiilor sau articolelor.

Oracle Spatial Data

Această opțiune permite stocarea datelor geografice și spațiale într-o bază de date. Cu ajutorul acesteia complexitatea gestionării spațiului de stocare este redusă iar performanțele sistemului cresc semnificativ.

Oracle Web Server

Oracle Web Server a fost proiectat pentru a permite accesul utilizatorilor Web la o bază de date Oracle. Prin urmare, informațiile sunt extrase direct dintr-o bază de date Oracle și nu din simple fișiere tradiționale, mărindu-se performanța și funcționalitatea server-ului Web. Datorită dezvoltării comerțului electronic, începând cu Oracle8i arhitectura client/server a fost înlocuită de arhitectura Web Server.

Internet Commerce Server

Internet Commerce Server este un set complet de unele proiectate pentru a crea, executa și administra un sistem Oracle ce este utilizat în comerțul pe Web. În acest fel sistemul poate furniza servicii robuste și sigure din punct de vedere al securității datelor.

Anexa 2

Comenzi frecvent utilizate în administrarea bazelor de date și utilitarele cu care se pot executa

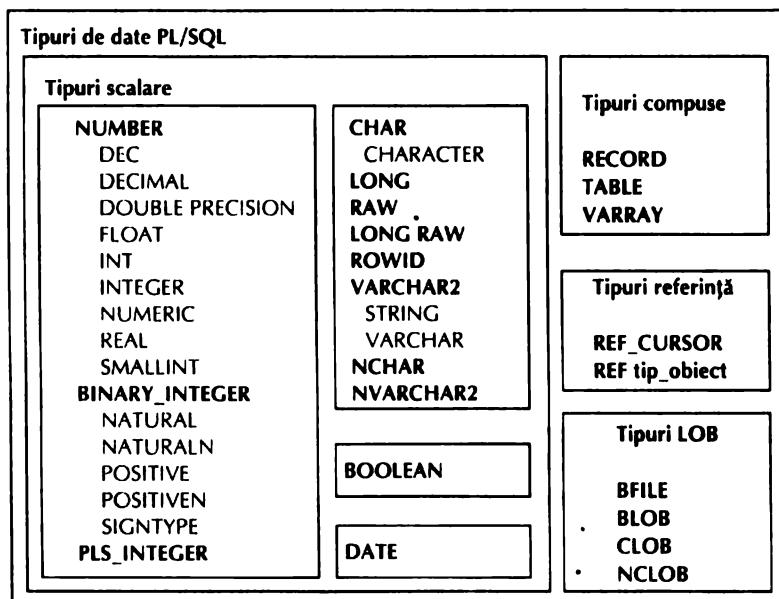
Comanda	Utilitar recomandat a fi utilizat pentru execuția comenziilor	Alte utilitare în care se poate executa comanda
Crearea unei baze de date	Oracle Database Assistant	<ul style="list-style-type: none">▪ ORADIM80 și Server Manager (SVRMGR30)▪ SQL Worksheet
Stergerea unei baze de date	Oracle Database Assistant	ORADIM80 și Server Manager (SVRMGR30)
Stergerea unui serviciu al bazei de date	Oracle Database Assistant	ORADIM80
Pornirea unei baze de date	Instance Manager	<ul style="list-style-type: none">▪ ORADIM80 și Server Manager (SVRMGR30)▪ SQL Worksheet
Oprirea unei baze de date	Instance Manager	<ul style="list-style-type: none">▪ ORADIM80 și Server Manager (SVRMGR30)▪ Control Panel▪ SQL Worksheet
Exportarea datelor	Data Manager	Export Utility (EXP80)
Importarea datelor	Data Manager	Import Utility (IMP80)
Încărcarea datelor	Data Manager	SQL*Loader (SQLLDR80)
Salvarea datelor dintr-o bază de date (back-up)	Backup Manager	<ul style="list-style-type: none">▪ Recovery Manager (RMAN80)▪ NT Backup Manager▪ OCOPY80
Recuperarea datelor dintr-o bază de date (recover)	Backup Manager	<ul style="list-style-type: none">▪ Recovery Manager (RMAN80)▪ NT Backup Manager▪ OCOPY80
Autentificarea administratorilor bazei de date și a utilizatorilor	Security Manager	<ul style="list-style-type: none">▪ Server Manager (SVRMGR30)▪ SQL*Plus▪ Sistemul de operare Windows NT▪ SQL Worksheet
Acordarea rolurilor	Security Manager	<ul style="list-style-type: none">▪ User manager
Crearea obiectelor bazei de date	Schema Manager	<ul style="list-style-type: none">▪ Server Manager▪ SQL*Plus

Anexa 3

Tipuri de date PL/SQL

Tipurile de date din PL/SQL se împart în mai multe categorii: scalare, compuse, referință și LOB. Un *tip scalar* nu are componente interne, adică conține valori atomice. Un *tip compus* are componente interne care pot fi manipulate individual. Un tip referință păstrează valori, numite pointeri, care reprezintă adresa unor alte elemente ale programului. Un tip LOB păstrează valori care specifică locația unor obiecte de dimensiune mare.

Unele tipuri de date din PL/SQL conțin *subtipuri*. Un subtip reprezintă un tip de date asupra căruia se aplică constrângeri astfel încât rezultă un subset de valori. În figura următoare sunt prezentate tipurile de date din PL/SQL (cu caractere îngroșate) precum și subtipurile acestora, acolo unde este cazul.



Pe lângă aceste de tipuri de date mai există și un alt tip de date scalar, MLSLABEL, ce este folosit cu o versiune Oracle cu un grad mare de securitate, Trusted Oracle.

1. Tipuri de date scalare

Dintre tipurile scalare PL/SQL, următoarele sunt și tipuri Oracle SQL, adică tipuri de date folosite pentru coloanelor din tabelele Oracle: NUMBER, VARCHAR2, CHAR, LONG, RAW, LONG RAW, ROWID, NVARCHAR2, NCHAR, DATE. În unele cazuri totuși, tipurile de date

PL/SQL dîsferă de corespondențele lor SQL prin dimensiunea maximă permisă – de exemplu, în PL/SQL lungimea maximă pentru o variabilă de tip VARCHAR2 este de 32767 bytes, pe când o coloană de tipul VARCHAR2 poate avea maxim 4000 bytes. Atunci când există astfel de diferențe, ele vor fi evidențiate.

Tipurile de date scalare se împart în patru categorii:

a. Tipuri de date ce stochează valori numerice

- **Tipul de dată NUMBER**

Tipul de dată NUMBER este folosit pentru stocarea numerelor reale și are următoarea sintaxă:

```
NUMBER [(precizie[, scală])]
```

Dacă nu este specificată scala, valorile numerice sunt rotunjite la numere întregi. De exemplu NUMBER(9, 2) înseamnă reprezentarea numărului pe 9 cifre cu 2 zecimale. Variabilele de tip NUMBER au o precizie de la 1 la 38 de cifre și o scală de la -84 la 127.

Tipul de date NUMBER are următoarele subtipuri:

- DEC;
- DECIMAL;
- DOUBLE PRECISION;
- FLOAT;
- INTEGER;
- INT;
- NUMERIC;
- REAL;
- SMALLINT.

- **Tipul de dată BINARY_INTEGER**

Este un tip de dată utilizat pentru a stoca numere întregi cu semn situate în intervalul -2147483647 ... 2147483647.

Tipul de date BINARY_INTEGER are următoarele subtipuri:

- NATURAL;
- NATURALN;
- POSITIVE;
- POSITIVEN;
- SIGNTYPE.

Subtipurile NATURAL și POSITIVE restrâng domeniul numerelor întregi la numere pozitive iar subtipurile NATURALN și POSITIVEN previn, în plus, atribuirea valorilor Null. Subtipul SIGNTYPE restrânge domeniul variabilelor la valorile -1, 0, și 1.

- **Tipul de dată PLS_INTEGER**

Este utilizat pentru stocarea numerelor întregi cu semn situate în intervalul -2147483647 .. 2147483647. Ca și tipul de dată BINARY_INTEGER, PLS_INTEGER necesită mai puțin spațiu de stocare decât tipul NUMBER. În plus, operațiile cu PLS_INTEGER folosesc aritmetică mașinii, astfel încât sunt mai rapide decât cele cu NUMBER sau BINARY_INTEGER, care folosesc librării aritmetice. Deși tipurile de dată PLS_INTEGER și BINARY_INTEGER au același interval de definire, ele nu sunt complet compatibile. Diferența constă în faptul că atunci când un calcul cu tipul de dată PLS_INTEGER este depășit, se declanșează o excepție, pe când dacă este folosit tipul de dată BINARY_INTEGER acest lucru nu se întâmplă în cazul în care rezultatul este atribuit unei variabile de tip NUMBER. Spre deosebire de operațiile cu tipul NUMBER sau BINARY_INTEGER, operațiile cu tipul PLS_INTEGER sunt efectuate mult mai rapid. Astfel, pentru o mai bună performanță, este preferabil să se utilizeze tipul de dată PLS_INTEGER.

b. Tipuri de date ce stochează caractere

Sunt folosite pentru stocarea datelor alfanumerice, cum ar fi cuvinte sau texte, și manipularea sirurilor de caractere.

- **Tipul de dată VARCHAR2**

Tipul de dată VARCHAR2 se utilizează pentru a stoca siruri de caractere cu lungime variabilă. Datele sunt reprezentate intern în funcție de setul de caractere (care poate fi, de exemplu, ASCII sau EBCDIC). Tipul de dată VARCHAR2 necesită un parametru obligatoriu care reprezintă lungimea maximă ce poate fi până la 32767 bytes și are următoarea sintaxă:

VARCHAR2(lungime_maximă)

Nu se poate utiliza o constantă sau o variabilă pentru a specifica lungimea maximă ci trebuie utilizat un număr întreg din intervalul 1 .. 32767. Trebuie remarcat că lungimea maximă este exprimată în bytes și nu în caractere. Astfel, numărul maxim de caractere va depinde de setul de caractere folosit, acesta fiind specificat la crearea bazei de date. Oracle suportă atât seturi de caractere pe un singur byte (de exemplu ASCII) cât și seturi de caractere pe mai mulți bytes.

În plus, trebuie remarcat că lungimea maximă a unei coloane de tip VARCHAR2 dintr-un tabel este 4000 bytes, deci într-o astfel de coloană nu se pot insera valori de tip VARCHAR2 cu lungimea mai mare de 4000 bytes. În schimb, orice valoare de tip VARCHAR2 se poate insera într-o coloană de tip LONG dintr-un tabel deoarece lungimea maximă a acesteia este 2147483647 bytes sau 2 gigabytes. Invers, nu se pot atribui valori mai mari decât 32767 bytes dintr-o coloană de tip LONG unei variabile de tip VARCHAR2.

Tipul de date VARCHAR2 are următoarele subtipuri:

- STRING;
- VARCHAR.

Acest subtipuri reprezintă de fapt o altă denumire a tipului de dată VARCHAR2 și sunt folosite pentru compatibilitatea cu tipurile de date ANSI/ISO și IBM.

Notă: Deși VARCHAR este sinonim cu VARCHAR2, se recomandă folosirea tipului VARCHAR2 deoarece este posibil ca în implementările PL/SQL viitoare cele două tipuri de date să fie diferite din punct de vedere semantic.

- **Tipul de dată CHAR**

Este utilizat pentru stocarea sirurilor de caractere de lungime fixă și are următoarea sintaxă:

```
CHAR [(lungime_maxima)]
```

Parametrul `lungime_maxima` este optional și este folosit pentru a specifica lungimea maximă ce poate ajunge până la 32767 bytes. Nu se poate utiliza o constantă sau o variabilă pentru a specifica această lungime ci trebuie utilizat un număr întreg din intervalul 1 ... 32767. Dacă nu este specificat acest parametru se consideră a fi egal în mod automat cu 1. La fel ca în cazul tipului de dată VARCHAR2, lungimea maximă este exprimată în bytes și nu în caractere, astfel încât numărul maxim de caractere va depinde de setul de caractere folosit, Oracle suportând atât seturi de caractere pe un singur byte (de exemplu ASCII) cât și seturi de caractere pe mai mulți bytes.

Trebuie remarcat că lungimea maximă a unei coloane de tip CHAR dintr-un tabel este 2000 bytes, deci într-o astfel de coloană nu se pot insera valori de tip CHAR mai mari de 2000 bytes. În schimb, orice valoare de tip CHAR se poate insera într-o coloană de tip LONG dintr-un tabel deoarece lungimea maximă a acesteia este 2147483647 bytes sau 2 gigabytes. Invers, nu se pot atribui valori mai mari decât 32767 bytes dintr-o coloană de tip LONG unei variabile de tip CHAR.

Între tipurile de dată CHAR și VARCHAR2 există mai multe diferențe. Una dintre diferențe este modul de stocare al acestora. În cazul tipului de dată CHAR, dacă datele oferite sunt mai mici decât lungimea definită a câmpului, Oracle completează cu spații câmpul respectiv până la lungimea definită pe când în cazul tipului VARCHAR2 nu sunt niciodată completeate în mod automat. Dacă la momentul atribuirii unei variabile de tip CHAR lungimea sirului este mai mare decât lungimea definită a câmpului, se va declanșa eroare `VALUE_ERROR`. O altă diferență constă în faptul că cele două câmpuri sortează și compară în mod diferit. Câmpurile CHAR sunt sortate și comparate folosind întreaga lor lungime, în timp ce câmpurile cu caractere variabile sunt sortate și comparate numai după caracterele incluse în sir.

Tipul de date CHAR are următorul subtip:

- CHARACTER.

Acest subtip reprezintă de fapt o altă denumire a tipului de dată CHAR și este folosit pentru compatibilitatea cu tipurile de date ANSI/ISO și IBM.

- **Tipul de dată LONG**

Este utilizat pentru stocarea sirurilor de caractere de lungime variabilă ce depășesc 4000 bytes. Tipul de dată LONG este asemănător tipului de dată VARCHAR2, excepție făcând lungimea maximă care, pentru tipul de dată LONG, este de 32760 bytes. Pentru sirurile de caractere cu o lungime sub 4000 bytes este recomandabil să se folosească variabile de tip VARCHAR2.

Se pot insera valori de tip LONG într-o coloană de tip LONG a unei tabele deoarece lungimea maximă a acesteia este de 2147483647 bytes. Invers, nu se pot atribui valori mai mari decât 32760 bytes dintr-o coloană de tip LONG unei variabile de tip LONG. În coloanele de tip LONG se pot stoca texte, siruri de caractere sau chiar documente de dimensiuni reduse. De remarcat este faptul că nu se pot referi coloane de tip LONG în expresii, apelarea funcțiilor SQL sau în anumite clauze ale comenzi SELECT, cum ar fi WHERE, GROUP BY și CONNECT BY.

- **Tipul de dată RAW**

Tipul de dată RAW se folosește pentru a stoca date binare sau siruri de biți de lungime variabilă. De exemplu, o variabilă de tip RAW poate stoca o secvență de caractere grafice și este ca o variabilă de tip VARCHAR2, excepție făcând faptul că PL/SQL nu interpretează datele. Tipul de dată RAW necesită un parametru obligatoriu care reprezintă lungimea maximă, ce poate fi până la 32767 bytes, și are următoarea sintaxă:

```
RAW (lungime_maximă)
```

Nu se poate utiliza o constantă sau o variabilă pentru a specifica lungimea maximă ci trebuie utilizat un număr întreg din intervalul 1 .. 32767. Se pot insera valori de tip RAW într-o coloană de tip RAW a unei tabele numai dacă lungimea maximă este de până la 2000 bytes. În schimb, se poate insera orice valoare de tip RAW într-o coloană de tip LONG RAW a unei tabele deoarece lungimea maximă a acesteia este de 2147483647 bytes. Invers, nu se pot atribui valori mai mari de 32767 bytes dintr-o coloană de tip LONG RAW unei variabile de tip RAW.

- **Tipul de dată LONG RAW**

Acest tip de dată se folosește pentru a stoca date binare sau siruri de biți de lungime variabilă. Tipul de dată LONG RAW este similar tipului de dată LONG, excepție făcând faptul că tipul de dată LONG RAW nu este interpretat de PL/SQL. Lungimea maximă a tipului de dată LONG RAW este de 32760 bytes. Se pot insera orice valori ale unei variabile de tip LONG RAW, într-o coloană de tip LONG RAW a unei tabele deoarece lungimea acesteia este de 2147483647 bytes. Invers, nu se pot atribui valori mai mari de 32760 bytes dintr-o coloană de tip LONG RAW unei variabile de tip LONG RAW.

- **Tipul de dată ROWID**

Pentru fiecare tabel al unei baze de date există o pseudo-coloană ce stochează siruri hexazecimale de tip ROWID, numite identificatori de rând. Fiecare identificator de rând conține adresa unui rând. Cu alte cuvinte, tipul de dată ROWID se folosește pentru a stoca

identificatori de rând. Se poate compara o variabilă de tip ROWID cu o pseudo-colonă în clauza WHERE a unei instrucțiuni UPDATE sau într-o instrucțiune DELETE pentru a identifica ultima înregistrare preluată dintr-un cursor.

Începând cu Oracle8, identificatorii de rând au fost extinși pentru a putea să fie folosiți și de tabele sau indecsii partionați. Astfel, identificatorii de rând conțin în plus față de alte versiuni Oracle și numărul de identificare al obiectului. În consecință, pentru identificarea unică a unui rând se folosește o a patra componentă, numărul de identificare al obiectului, care determină în mod unic segmentul bazei de date în care se află obiectul – obiectele din același segment (de exemplu tabelele dintr-un cluster) au același număr de obiect.

În Oracle8, identificatorul de rând este alcătuit din următoarele patru componente:

- numărul obiectului;
- numărul fișierului în cadrul spațiului tabel (primul fișier este 1);
- numărul blocului din fișier;
- numărul rândului din bloc (primul rând este 0).

Identifierii de rând reprezintă cel mai rapid mod de a accesa rândurile din tabele. În mod normal, un identifier de rând identifică în mod unic un rând. Începând cu Oracle8 însă, rânduri din tabele diferite stocate în același cluster pot avea același identifier de rând. Oracle furnizează pachetul precompilat DBMS_ROWID ce poate fi utilizat în manipularea identifierilor de rând.

• Tipul de date NCHAR și NVARCHAR2

Deoarece seturile de caractere ASCII sau EBCDIC sunt folosite pentru reprezentarea caracterelor alfabetului roman, sistemul Oracle pune la dispoziția programatorului un suport pentru reprezentarea caracterelor limbilor naționale, numit NLS (National Language Support). Acest lucru este necesar pentru reprezentarea alfabetelor ce conțin mii de caractere (de exemplu alfabetul japonez), fapt ce implică utilizarea a 16 biți. Prin urmare, NLS face posibilă procesarea caracterelor pe unul sau mai mulți bytes, conversia dintre seturile de caractere, precum și rularea unei aplicații într-un mediu ce suportă o limbă diferită. Datorită NLS, formatele numerelor și datelor calendaristice se adaptează automat la convențiile de limbă specificate pentru o sesiune a unui utilizator. Prin urmare NLS permite ca utilizatorii din întreaga lume să interacționeze cu sistemul Oracle în limba nativă. În acest sens există două tipuri de date:

- NCHAR;
- NVARCHAR2.

NCHAR

Tipul de date NCHAR este folosit pentru stocarea caracterelor NLS cu lungime fixă și are următoarea sintaxă:

NCHAR[(lungime_maxima)]

Parametrul lungime_maxima este optional și poate fi maximum 32767 bytes. Nu se poate utiliza o constantă sau o variabilă pentru a specifica lungimea maximă ci trebuie utilizat un număr întreg din intervalul 1 ... 32767. Dacă nu se specifică lungimea maximă

atunci este considerată implicit egală cu 1. Modul în care este specificată lungimea maximă depinde de setul național de caractere folosit: pentru un set de caractere cu dimensiune fixă lungimea maximă este specificată în caractere; pentru un set de caractere cu dimensiune variabilă lungimea maximă este specificată în bytes.

Pentru o coloană de tip NCHAR a unui tabel lungimea maximă este de 2000 bytes, prin urmare nu se pot inseră valori de tip NCHAR mai mari de 2000 bytes într-o coloană de tip NCHAR a unui tabel. Dacă în schimb valoarea de tip NCHAR este mai mică decât dimensiunea unei coloane de tip NCHAR, sistemul Oracle adaugă spații valorii respective până ajunge la dimensiunea necesară. O altă observație este faptul că nu se pot inseră valori de tip CHAR într-o coloană de tip NCHAR a unei tabele și nici valori de tip NCHAR într-o coloană de tip CHAR.

NVARCHAR2

Tipul de dată NVARCHAR2 este folosit pentru stocarea caracterelor NLS cu lungime variabilă și are următoarea sintaxă:

NVARCHAR2(lungime_maxima)

Parametrul `lungime_maxima` poate fi maximum 32767 bytes. Nu se poate utiliza o constantă sau o variabilă pentru a specifica lungimea maximă ci trebuie utilizat un număr întreg din intervalul 1 ... 32767. La fel ca și în cazul tipului de dată NCHAR, modul în care este specificată lungimea maximă (în caractere sau bytes) depinde de setul de caractere folosit.

Pentru o coloană de tip NVARCHAR2 a unui tabel lungimea maximă este de 4000 bytes, prin urmare nu se pot inseră valori de tip NVARCHAR2 mai mari de 4000 bytes într-o coloană de tip NVARCHAR2 a unui tabel. De asemenea, nu se pot inseră valori de tip VARCHAR2 într-o coloană de tip NVARCHAR2 a unei tabele și nici valori de tip NVARCHAR2 într-o coloană de tip VARCHAR2.

c. Tipuri de date booleene

- Tipul de dată BOOLEAN**

Poate lua valorile True, False sau Null. Acestei variabile î se pot atribui valori rezultate în urma unei expresii booleene și pot fi testate în instrucțiuni condiționale. Cu aceste tipuri de date sunt permise numai operații logice. Tipul de dată boolean nu permite nici un parametru. Deoarece tipul de dată BOOLEAN este specific PL/SQL, nu se pot inseră valorile True sau False unei coloane și nici nu se pot prelua date dintr-un cursor sau selecta coloane într-o variabilă de tip BOOLEAN.

d. Tipuri de date pentru dată calendaristică și oră.

- Tipul de dată DATE**

Tipul de dată DATE este folosit pentru stocarea datelor calendaristice și include de asemenea informații referitoare la oră. Variabila corespunde tipului de dată calendaristică a

unei coloane dintr-un tabel. Datele calendaristice sunt stocate întotdeauna cu șapte componente: secolul, anul, luna, ziua, ora, minutul și secunda. Dacă ora nu este specificată în mod explicit atunci ea va fi considerată în mod automat egală cu miezul nopții. Același lucru se întâmplă și în cazul zilei care va fi considerată în mod implicit egală cu prima zi din luna curentă. Variabila variază de la data de 1 Ianuarie 4712 î.Cr. până la data de 31 Decembrie 4712 d.Cr. Pentru a returna data și ora curentă a sistemului se folosește funcția SYSDATE. PL/SQL convertește automat șirurile de caractere ce au format implicit de dată în date de tip DATE. Formatul implicit al datei este dat de parametrul de inițializare NLS_DATE_FORMAT (de exemplu acesta poate fi 'DD-MON-YY'). De asemenea, se pot face operații de scădere sau adunare cu date de tip DATE.

- **Tipul de dată MLSLABEL**

Pe lângă aceste tipuri de date scalare mai există și tipul de dată MLSLABEL ce este folosit cu Trusted Oracle. Acest tip de dată stochează etichete generate de sistemul de operare cu securitate pe mai multe nivele. Pentru a realiza compatibilitatea cu Trusted Oracle, sistemul Oracle permite definirea acestui tip de date, singura restricție fiind faptul că acestea nu pot avea decât valoarea Null.

2. Tipuri de date compuse

Sunt acele tipuri de date care au componente interne ce pot fi manipulate în mod individual. În acest sens Oracle oferă programatorului tipurile de date TABLE și VARRAY ce permit definirea tabelelor imbricate respectiv a vectorilor de lungime variabilă, precum și tipul de date RECORD ce permite gruparea mai multor tipuri de date într-o unitate logică.

Tipurile de date TABLE și VARRAY sunt cunoscute și sub denumirea de *colecții*. Colecțiile reprezintă un grup ordonat de elemente, toate de același tip (de exemplu notele studenților dintr-o grupă). Colecțiile sunt uni-dimensionale (nu pot fi multi-dimensionale ca în Pascal) și sunt indexate cu ajutorul numerelor întregi. Colecțiile pot stoca instanțe ale unui tip de obiect, pot fi atribuite ale acestuia și pot fi transferate ca parametrii ai unor proceduri sau funcții. Ele se pot defini și în cadrul unui pachet PL/SQL. Există funcții sau proceduri predefinite ce se numesc *metode ale colecției* și care sunt foarte folositoare în gestionarea colecțiilor. Pentru a apela aceste metode se folosește punctul: `nume_colecție.nume_metodă [(parametrii)]`. Metodele pot fi apelate numai din cadrul instrucțiunilor procedurale. Există mai multe metode: EXISTS, COUNT, LIMIT, FIRST și LAST, PRIOR și NEXT, EXTEND, TRIM, DELETE.

- **Tipul de dată înregistrare RECORD**

O variabilă de tip înregistrare PL/SQL conține o colecție de câmpuri diferite, fiecare câmp putând fi adresat în mod individual. Să presupunem că avem mai multe date referitoare la un angajat: nume, salar, data angajării, vîrstă. Aceste date pot avea tipuri și mărimi diferite, dar din punct de vedere logic ele reprezintă o unitate de sine stătătoare. Variabila de tip înregistrare grupează aceste date, ușurând organizarea și reprezentarea informațiilor. Variabila de tip înregistrare este foarte folositoare atunci când se dorește

extragerea unui rând dintr-un tabel în vederea procesării lui într-un program PL/SQL (în special în cursoare). Pentru a declara o variabilă de tip înregistrare ce se bazează pe o colecție de câmpuri dintr-o tabelă sau vedere se folosește atributul %ROWTYPE. Câmpurile din variabila de tip înregistrare vor avea același nume ca și coloanele tabelului sau vederii.

Variabilele de tip RECORD se definesc în partea de declarații a unui bloc PL/SQL, subprogram sau pachet cu ajutorul următoarei sintaxe:

```
TYPE nume_tip IS RECORD (declarare_câmp[, declarare_câmp]...)
```

unde declarare_câmp înseamnă:

```
nume_câmp tip_câmp [(NOT NULL) (:= | DEFAULT) expresie]
```

De exemplu:

```
DECLARE
  TYPE angajat IS RECORD (
    cod_angaj      INTEGER,
    nume_angaj     VARCHAR2(15),
    nr_angaj       INTEGER(2),
    salariu_angaj REAL(7,2));
  inгинер angajat;      -- declararea unei variabile RECORD
                        -- de tip angajat ce are numele
                        -- inгинер
```

Spre deosebire de variabilele de tip TABLE și VARRAY, variabilele de tip RECORD nu pot fi create și stocate în baza de date. Pentru specificarea tipurilor câmpurilor se pot folosi și atributele %TYPE și %ROWTYPE.

Câmpurile dintr-o variabilă de tip înregistrare se identifică cu ajutorul punctului: `ингинер.salariu_angaj`

- **Tipurile de date TABLE**

În PL/SQL există două feluri de tipuri de date TABLE: *tabele imbricate (nested tables)* și *tabele indexate (index-by tables)*.

- **Tabele imbricate**

Într-o bază de date tabelele imbricate pot fi văzute ca un tabel cu o singură coloană, iar în PL/SQL ca un vector uni-dimensional. Diferența dintre un tabel imbricat și un vector uni-dimensional constă în faptul că vectorii au o lungime dată, pe când, pentru tabelele imbricate lungimea crește dinamic. O altă diferență este faptul că vectorii sunt denși, adică elementele au poziții consecutive. În cazul tabelelor imbricate, elementele se pot șterge astfel încât anumite poziții să lipsească. Este de preferat ca variabilele de tip tabel imbricat să fie folosite pentru colecții de dimensiuni mari. Variabilele de tip tabel imbricat sunt stocate în tabele ale bazei de date generate de sistem și asociate acestora.

Tabelele imbricate se definesc folosind următoarea sintaxă:

```
TYPE nume_tip IS TABLE OF tip_element [NOT NULL]
```

unde `tip_element` poate fi orice tip de variabilă PL/SQL cu excepția următoarelor tipuri: `BINARY_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `PLS_INTEGER`, `POSITIVE`, `POSITIVEN`, `REF CURSOR`, `SIGNTYPE`, `STRING`, `TABLE`, `VARRAY`, tipuri de obiecte ce au atribut `TABLE` sau `VARRAY`.

```
DECLARE
    TYPE tabel_angajat IS TABLE OF angajat;
        -- tipul angajat este definit mai sus
    tabel_angajat;
        -- declararea unei variabile de tip
        -- tabel_angajat
```

Tabelele imbricate sunt o nouătate adusă de versiunea și Oracle8 pot fi definite și manipulate atât în PL/SQL, cât și în SQL, vezi Capitolul 10.

- **Tabele indexate**

Spre deosebire de tabele imbricate, tabelele indexate nu pot fi definite și manipulate în SQL, ci numai în PL/SQL. Ele există și în versiunile anterioare versiunii Oracle 8. La definirea tabelelor indexate trebuie folosită clauza `INDEX BY BINARY_INTEGER`:

```
TYPE nume_tip IS TABLE OF tip_element [NOT NULL]
    INDEX BY BINARY_INTEGER;
```

În cazul tabelelor indexate, `tip_element` poate avea și următoarele tipuri, care nu sunt permise în cazul tabelelor imbricate: `BINARY_INTEGER`, `BOOLEAN`, `LONG`, `LONG RAW`, `NATURAL`, `NATURALN`, `PLS_INTEGER`, `POSITIVE`, `POSITIVEN`, `SIGNTYPE` și `STRING`. Acest lucru se explică prin faptul că tabelele imbricate (ca și tipul `VARRAY`) sunt destinate în special să fie coloane în tabele ale bazei de date. Când sunt declarate local ele ar putea, teoretic, să folosească aceste tipuri, dar acest lucru nu este permis din motive de consistență a definiției. Între tabelele imbricate și tabelele indexate există și alte diferențe importante, pentru detalii se poate consulta documentația Oracle [26].

- **Tipul de dată VARRAY**

Tipul de dată `VARRAY` este folosit pentru a asocia un identificator unei colecții. Această asociere permite manipularea colecției ca un întreg și referirea elementelor individuale. Referirea elementelor se face prin utilizarea sintaxei standard, adică `vector(3)` este al treilea element al variabilei `vector` de tip `VARRAY`. Dimensiunea unei variabile de tipul `VARRAY` trebuie specificată în mod obligatoriu în momentul definirii acesteia. Cu alte cuvinte, dimensiunea nu se poate schimba dinamic ca în cazul tabelelor imbricate. O altă diferență față de tipul `TABLE` este faptul că datele de tip `VARRAY` sunt dense, adică elementele au poziții consecutive. Oracle stochează datele de tip `VARRAY` în același spațiu tabel, pe când tabelele imbricate sunt stocate în tabelele ale bazei de date generate de sistem

și asociate acestora. Dacă se dorește regăsirea unei colecții întregi este de preferat folosirea variabilelor de tip VARRAY, în schimb acest lucru nu mai reprezintă o facilitate dacă colecția are dimensiuni mari. Atunci când colecțiile au dimensiuni mari este de preferat folosirea tabelelor imbricate.

Variabilele de tip VARRAY se definesc cu ajutorul următoarei sintaxe:

```
TYPE nume_tip IS {VARRAY | VARYING ARRAY} (dimensiune)
                  OF tip_element [NOT NULL]
```

Dimensiunea variabilei trebuie declarată în mod obligatoriu. Ca și în cazul tabelelor imbricate, tip_element poate fi orice tip de dată PL/SQL cu excepția următoarelor: BINARY_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, NCHAR, NCLOB, NVARCHAR2, PLS_INTEGER, POSITIVE, POSITIVEN, REF CURSOR, SIGNTYPE, STRING, TABLE, VARRAY, tipuri de obiecte ce au atribut TABLE sau VARRAY.

```
DECLARE
    TYPE calendar IS VARRAY(366) OF DATE;
    an      calendar;
```

3. Tipuri de date referință

Sunt acele tipuri de date ale căror valori, numite *pointeri*, fac referință către un element sau obiect. Pointerii conțin locația de memorie (adresa) a unui element și nu elementul însine. În PL/SQL un pointer are tipul REF X, unde REF este prescurtarea de la REFERENCE iar X este clasa de obiecte.

- **Tipul de dată REF CURSOR**

Tipul de dată REF CURSOR este folosit pentru a face referință la un cursor explicit. Acest tip de dată este utilizat în principal pentru a transfera rezultatele unei interogări care întoarce mai multe rânduri între subprograme stocate PL/SQL și clienții acestuia. Nici PL/SQL și nici unul dintre clienții acestuia nu dețin setul de rezultate ci doar partajează un pointer către aria de lucru în care sunt stocate rezultatele interogării. De exemplu, un client OCI (Oracle Call Interface) precum Oracle Forms și Oracle Server se pot referi altfel la aceeași arie de lucru. Dacă de exemplu se declară o variabilă de tip referință de către un client, se poate ca un anumit cursor să fie deschis și să se preia date din el de către server, iar în continuare să se poată prelua date din nou de către client. O arie de lucru a unei interogări rămâne accesibilă atât timp cât există o variabilă care face referință la aceasta.

Pentru a crea o variabilă de tip referință la un cursor trebuie efectuați doi pași: definirea unui tip REF CURSOR și apoi declararea unei variabile de acest tip. Definirea unui tip REF CURSOR se face utilizând sintaxa:

```
TYPE nume_tip IS REF CURSOR RETURN tip_data_returnat
```

unde tip_data_returnat reprezintă tipul unei înregistrări sau a unui rând dintr-o tabelă. În exemplul următor tipul datelor returnate va reprezenta un rând dintr-o tabelă:

```

DECLARE
    TYPE referinta_angajat_cursor IS REF CURSOR RETURN
        angajati%ROWTYPE;
    p_angajat referinta_angajat_cursor;

```

Se poate declara un tip de variabilă referință către un cursor fără a specifica ce tip de date va returna. Acesta înseamnă că tipul de variabilă referință poate fi asociată oricărui cursor. De exemplu:

```

DECLARE
    TYPE referinta_cursor_generic IS REF CURSOR;

```

Variabilele de tip REF CURSOR nu se pot declara într-un pachet (în afara procedurilor sau funcțiilor). Acest lucru se explică prin faptul că astfel de variabile nu sunt persistente și deci nu pot fi salvate în baza de date. La declararea unei astfel de variabile se creează doar un pointer, nu datele respective.

Pentru mai multe informații despre tipul REF CURSOR se poate consulta documentația Oracle [26].

- **Tipul de dată REF obiect**

Dacă un obiect are dimensiuni mari este inefficient să se transfere copii ale acestuia dintr-un subprogram în alt subprogram. O soluție eficientă este partajarea obiectului respectiv de către mai multe subprograme prin intermediul pointerelor. Partajarea obiectului prezintă două avantaje importante. Primul avantaj îl reprezintă faptul că nu sunt necesare copii ale obiectului iar cel de al doilea avantaj este faptul că dacă un obiect partajat este modificat atunci modificările se vor face într-un singur loc și orice referință către acel obiect va returna valorile modificate. Dacă obiectul respectiv deține mai multe copii, atunci ar fi fost necesare modificări în toate copiile acestuia.

Pointerii conțin adresa unui obiect și nu obiect în sine. Așa cum am mai menționat, în PL/SQL un pointer are tipul REF X, unde REF este prescurtarea de la REFERENCE iar X este clasa de obiecte.

Exemplul următor definește două clase de obiecte numite `adresa_locuinta` și `persoane`. Clasa de obiecte `persoane` conține un pointer către clasa de obiecte `adresa_locuinta` deoarece o locuință poate fi împărțită de mai multe persoane.

```

CREATE TYPE adresa_locuinta AS OBJECT (
    strada          VARCHAR2(35),
    bloc            VARCHAR2(25),
    numar           INTEGER,
    oras            VARCHAR(15),
    fotografie     BLOB);

```

```

CREATE TYPE persoane AS OBJECT (
    nume            VARCHAR2(10),

```

```

prenume      VARCHAR2(15),
data_nastere DATE,
adresa       REF adresa_locuinta,
              --la aceeași adresă pot locui
              --mai multe persoane
telefon      VARCHAR2(15));

```

4. Tipuri de date LOB

Sunt acelă tipuri de date ale căror valori, numite *locatori*, specifică locația unor obiecte de dimensiuni mari (de exemplu texte, imagini grafice, video clip-uri). Cu alte cuvinte, tipul de datea LOB (Large Object) face posibilă stocarea unor blocuri de date nestructurate ce au dimensiunea de până la 4 gigabytes. Există mai multe tipuri de date LOB: BFILE, BLOB, CLOB și NCLOB.

Tipul de date LOB diferă de tipul LONG sau LONG RAW în mai multe feluri. De exemplu, tipurile de datea LOB (cu excepția NCLOB) pot fi atribuite unei variabile de tipLOB este de 4 gigabytes, pe când pentru variabilele de tip LONG RAW lungimea maximă este de 2 gigabytes. Un alt aspect este faptul că tipul de datea LOB suportă acces aleator, pe când tipul LONG RAW suportă numai acces secvențial.

Obiectele ale căror locație este specificată de locator pot fi stocate în fișiere externe, în interiorul rândului sau în exteriorul acestuia. Coloanele de tip BLOB, CLOB, NCLOB ale unui tabel stochează datele în baza de date, adică în interiorul sau în exteriorul unui rând, pe când pentru o coloană de tip BFILE datele sunt stocate în afara bazei de date sub formă de fișiere. PL/SQL operează asupra acestor obiecte de dimensiuni mari prin intermediu locatorilor. De exemplu, atunci când se regăsește o valoare a unei coloane de tip BLOB, se va returna doar locatorul. Locatorul nu poate cuprinde tranzacții sau sesiuni. Pentru a manipula aceste obiecte de dimensiuni mari trebuie utilizat pachetul precompilat DBMS_LOB. Pentru mai multe informații despre tipurile de date LOB și pachetul DBMS_LOB consultați documentația Oracle [26].

- **Tipul de datea BFILE**

Acest tip de date este utilizat pentru a stoca obiecte binare de dimensiuni mari în afara bazei de date, sub formă de fișiere binare. Fiecare variabilă de tip BFILE conține un locator care face referință la un fișier binar ce se află pe server. Locatorul conține un alias ce specifică calea spre fișierul respectiv. Fișierele de tip BFILE doar se pot citi, deci nu pot fi modificate. Numărul maxim de fișiere binare deschise este dat de parametrul de inițializare SESSION_MAX_OPEN_FILES care este dependent de sistem. De asemenea, fișierele binare nu pot participa în tranzacții. Sistemul de operare în care este instalat Oracle gestionează integritatea acestor fișiere. Mărimea unui fișier de tip BFILE este dependență de sistem, dar nu poate depăși 4 gigabytes. Administratorul bazei de date trebuie să se asigure că fișierele de tip BFILE există și că Oracle are permisiunea de citire asupra lor.

- **Tipul de dată BLOB**

Tipul de dată BLOB este folosit pentru a stoca obiecte binare de dimensiuni mari și de lungime variabilă în baza de date în interiorul sau în exteriorul unei înregistrări. Fiecare variabilă BLOB conține un locator ce face referință la un obiect de dimensiuni mari care nu poate depăși însă 4 gigabytes. Spre deosebire de datele de tip BFILE, datele de tip BLOB participă în tranzacții, modificările făcute de pachetul DBMS_LOB sau OCI (Oracle Call Interface) putând fi permanentizate sau derulate înapoi.

- **Tipul de dată CLOB**

Acest tip de dată este utilizat pentru a stoca în baza de date în interiorul sau în exteriorul unei înregistrări blocuri de dimensiuni mari ce conțin caractere reprezentate pe un singur byte. Fiecare variabilă de tip CLOB conține un locator care face referință la un bloc de caractere a cărui mărime maximă nu poate depăși 4 gigabytes. Ca și variabilele BLOB sau NCLOB, variabilele CLOB participă în tranzacții, modificările făcute de pachetul DBMS_LOB sau OCI putând fi permanentizate sau derulate înapoi.

- **Tipul de dată NCLOB**

Tipul de dată NCLOB este folosit pentru a stoca în baza de date în interiorul sau în exteriorul unei înregistrări blocuri de dimensiuni mari ce conțin caractere reprezentate pe un singur byte sau caractere de lungime fixă reprezentate pe mai mulți bytes (nu pot însă conține caractere a căror reprezentare ale dimensiune variabilă). Fiecare variabilă NCLOB conține un locator ce face referință la un bloc de caractere ce nu poate depăși 4 gigabytes. Ca și variabilele de tip CLOB sau BLOB, variabilele NCLOB participă în tranzacții, modificările făcute de pachetul DBMS_LOB sau OCI putând fi permanentizate sau derulate înapoi.

5. Subtipuri de date definite de utilizator

Fiecare tip de bază PL/SQL specifică un set de valori și un set de operații aplicate pe fiecare element de acel tip. Subtipurile specifică același set de operații ca și tipul de bază dar au un subset de valori din setul de valori al tipului de bază. Cu alte cuvinte, un subtip nu reprezintă un nou tip de dată, ci un tip de dată asupra căruia se aplică anumite constrângeri. Pachetul STANDARD conține mai multe subtipuri de date PL/SQL.

Utilizatorul poate defini propriul subtip în partea declarativă a oricărui bloc PL/SQL, subprogram sau pachet utilizând sintaxa:

```
SUBTYPE nume_subtip IS tip_de_bază
```

Următorul exemplu ilustrează definirea unor subtipuri:

```
DECLARE
```

```
    SUBTYPE data_năștere IS DATE;--subtip bazat pe tipul DATE
    SUBTYPE nr_crt IS NATURAL;--subtip bazat pe subtipul NATURAL
```

ANEXA 3 - Tipuri de date PL/SQL 351

```
TYPE timp IS RECORD (minute INTEGER, ore INTEGER);
SUBTYPE ora_exactă IS timp;--subtip bazat pe tipul RECORD
```

Nu se pot specifica constrângeri pe un tip de bază. Prin urmare, următoarele definiții sunt incorecte:

```
DECLARE
    SUBTYPE Accumulator IS NUMBER(7,2); --ilegal, trebuie să
                                         --fie NUMBER
    SUBTYPE Delimiter IS CHAR(1);--ilegal, trebuie să fie CHAR
```

Deși nu se pot defini constrângeri în mod direct pentru un subtip de dată, se poate utiliza un artificiu. Se declară mai întâi o variabilă care conține constrângeri, iar apoi se declară un subtip aferent variabilei folosind atributul %TYPE:

```
DECLARE
    temp VARCHAR2(15);
    SUBTYPE cuvânt IS temp%TYPE; --lungimea maximă a
                                   --subtipului cuvânt este 15
```

După ce a fost definit un subtip se pot declara în continuare variabile care au acest subtip.

Anexa 4A

Descrierea privilegiilor de sistem

Privilegiul de sistem	Operații permise de acordarea privilegiului
Analiză	
ANALYZE ANY	Analizarea unui tabel, cluster sau index în orice schemă
Auditare	
AUDIT ANY	Auditarea unui obiect în orice schemă folosind comenzi AUDIT.
AUDIT SYSTEM	Activează sau dezactivează folosirea comenzi de auditare.
Baza de date	
ALTER DATABASE	Modifică baza de date; adaugă fișiere în sistemul de operare prin intermediu Oracle.
Biblioteca	
CREATE LIBRARY	Creează biblioteci externe în propria schemă.
CREATE ANY LIBRARY	Creează biblioteci externe în orice schemă.
DROP LIBRARY	Sterge biblioteci externe în propria schemă.
DROP ANY LIBRARY	Sterge biblioteci externe în orice schemă.
Cluster	
CREATE CLUSTER	Creează un cluster în propria schemă.
CREATE ANY CLUSTER	Creează un cluster în orice schemă. Comportamentul este asemănător privilegiului CREATE ANY TABLE.
ALTER ANY CLUSTER	Modifică orice cluster în orice schemă.
DROP ANY CLUSTER	Sterge orice cluster în orice schemă.
Index	
CREATE ANY INDEX	Creează un index pe orice tabel, în orice schemă.
ALTER ANY INDEX	Modifică orice index în orice schemă.
DROP ANY INDEX	Sterge orice index în orice schemă.
Instantaneu	
CREATE SNAPSHOT	Creează un instantaneu în propria schemă. (Utilizatorul trebuie să aibă și privilegiul CREATE TABLE.)
CREATE ANY SNAPSHOT	Creează un instantaneu în orice schemă. (Utilizatorul trebuie să aibă și privilegiul CREATE ANY TABLE.)
ALTER ANY SNAPSHOT	Modifică orice instantaneu în orice schemă.
DROP ANY SNAPSHOT	Sterge orice instantaneu în orice schemă.
Legăturile bazei de date	
CREATE DATABASE LINK	Creează o legătură privată a bazei de date în propria schemă.
Legăturile publice ale bazei de date	
CREATE PUBLIC DATABASE LINK	Creează legături publice a bazei de date.
DROP PUBLIC DATABASE LINK	Sterge legături publice a bazei de date.
Privilegiu	
GRANT ANY PRIVILEGE	Acordă orice privilegiu de sistem (nu și privilegiu de obiect).
Procedură	
CREATE PROCEDURE	Creează proceduri, funcții, și pachete în propria schemă.

Privilegiul de sistem	Operații permise de acordarea privilegiului
CREATE ANY PROCEDURE	Creează proceduri, funcții, și pachete în orice schemă. (Necesită ca utilizatorul să aibă și privilegiile ALTER ANY TABLE, BACKUP ANY TABLE, DROP TABLE, SELECT ANY TABLE, INSERT ANY TABLE, UPDATE ANY TABLE, DELETE ANY TABLE, sau GRANT ANY TABLE.)
ALTER ANY PROCEDURE	Compilează orice procedură, funcție sau pachet în orice schemă.
DROP ANY PROCEDURE	Sterge orice procedură, funcție sau pachet în orice schemă.
EXECUTE ANY PROCEDURE	Execută orice procedură sau funcție (de sine stătătoare sau împachetată), sau face referire la orice variabilă publică a unui pachet în orice schemă.
Profil	
CREATE PROFILE	Creează un profil.
ALTER PROFILE	Modifică orice profil din baza de date.
DROP PROFILE	Sterge orice profil din baza de date.
ALTER RESOURCE COST	Setează costurile pentru resursele utilizate în toate sesiunile utilizatorilor.
Rol	
CREATE ROLE	Creează un rol.
ALTER ANY ROLE	Modifică orice rol din baza de date.
DROP ANY ROLE	Sterge orice rol din baza de date.
GRANT ANY ROLE	Acordă orice rol din baza de date.
Secvență	
CREATE SEQUENCE	Creează o secvență în propria schemă.
CREATE ANY SEQUENCE	Creează o secvență în orice schemă.
ALTER ANY SEQUENCE	Modifică orice secvență în orice schemă.
DROP ANY SEQUENCE	Sterge orice secvență în orice schemă.
SELECT ANY SEQUENCE	Referire la orice secvență din orice schemă.
Segment de revenire	
CREATE ROLLBACK SEGMENT	Creează segmente de revenire.
ALTER ROLLBACK SEGMENT	Modifică segmente de revenire.
DROP ROLLBACK SEGMENT	Sterge segmente de revenire.
Sesiune	
CREATE SESSION	Conectează la baza de date.
ALTER SESSION	Permite folosirea comenziilor ALTER SESSION.
RESTRICTED SESSION	Conectează la baza de date atunci când acesta a fost pornită utilizând STARTUP RESTRICT. (Roulurile OSOPER și OSDBA conțin acest privilegiu.)
Sinonim	
CREATE SYNONYM	Creează un sinonim în propria schemă.
CREATE ANY SYNONYM	Creează orice sinonim în orice schemă.
DROP ANY SYNONYM	Sterge orice sinonim în orice schemă.
Sinonim public	
CREATE PUBLIC SYNONYM	Creează un sinonim public.
DROP PUBLIC SYNONYM	Sterge un sinonim public.

Privilegiul de sistem	Operații permise de acordarea privilegiului
Sistem	
ALTER SYSTEM	Permite folosirea comenziilor ALTER SYSTEM.
Spațiu tabel	
CREATE TABLESPACE	Creează un spațiu tabel; adaugă fișiere în sistemul de operare prin intermediu Oracle.
ALTER TABLESPACE	Modifică un spațiu tabel; adaugă fișiere în sistemul de operare prin intermediu Oracle.
MANAGE TABLESPACE	Activează și dezactivează orice spațiu tabel; începe și termină arhivarea spațiilor tabel.
DROP TABLESPACE	Șterge un spațiu tabel.
UNLIMITED TABLESPACE	Utilizează o cantitate nelimitată a oricărui spațiu tabel indiferent de cotele specifice atribuite. Acest privilegiu se poate acorda numai utilizatorilor, nu și rolurilor. În general, în locul acestui privilegiu se atribuie cote specifice din spațiul tabel.
Tabel	
CREATE TABLE	Creează un tabel în propria schemă. De asemenea permite crearea de indecs pe tabelele din propria schemă. (Trebuie însă ca cel ce are acest privilegiu să aibă o cotă în spațiul tabel sau să aibă privilegiul UNLIMITED TABLESPACE.)
CREATE ANY TABLE	Creează un tabel în orice schemă. (Dacă se creează un tabel în schema altui utilizator, proprietarul schemei trebuie să aibă o cotă în spațiul tabel sau să aibă privilegiul UNLIMITED TABLESPACE).
ALTER ANY TABLE	Modifică orice tabel în orice schemă și compilează orice vedere în orice schemă.
BACKUP ANY TABLE	Exportă obiecte din orice schemă.
DROP ANY TABLE	Șterge sau trunchiază orice tabel în orice schemă.
LOCK ANY TABLE	Blochează orice tabel sau vedere în orice schemă.
COMMENT ANY TABLE	Comentează orice tabel, vedere, sau coloană în orice schemă.
SELECT ANY TABLE	Interoghează orice tabel, vedere, sau instantaneu în orice schemă.
INSERT ANY TABLE	Inserează rânduri în orice tabel sau vedere în orice schemă.
UPDATE ANY TABLE	Actualizează rânduri în orice tabel sau vedere în orice schemă.
DELETE ANY TABLE	Șterge rânduri în orice tabel sau vedere în orice schemă.
Tranзaсii	
FORCE TRANSACTION	Forțează comiterea sau derularea înapoi a unei tranзaсii distribuite nesigure în baza de date locală.
FORCE ANY TRANSACTION	Forțează comiterea sau derularea înapoi a oricărei tranзaсii distribuite nesigure în baza de date locală.
Trigger	
CREATE TRIGGER	Creează un declanșator al bazei de date în propria schemă.
CREATE ANY TRIGGER	Creează un declanșator al bazei de date în orice schemă ce este asociat cu orice tabel din orice schemă.
ALTER ANY TRIGGER	Activează, dezactivează sau compilează orice declanșator al bazei de date în orice schemă.
DROP ANY TRIGGER	Șterge orice declanșator din orice schemă.
Utilizator	
CREATE ANY USER	Creează utilizatori; Atribuie cote în orice spațiu tabel, setează spațiu tabel temporare sau default, și atribuie un profil ca parte a comenzi CREATE USER.
BECOME ANY USER	Devine un alt utilizator. (Cerută în efectuarea unui import complet al bazei de date.)

ANEXA 4A - Descrierea privilegiilor de sistem 355

Privilegiul de sistem	Operații permise de acordarea privilegiului
ALTER USER	Modifică caracteristici a altor utilizatori: modifică parola oricărui utilizator, atribuie cote spațiilor tabel, setează spații tabele temporare sau default, atribuie profiluri și roluri, într-o comandă ALTER USER. (Nu este necesar acest privilegiu pentru modificarea propriei parole.)
DROP USER	Sterge un alt utilizator.
Vedere	
CREATE VIEW	Creează o vedere în propria schemă.
CREATE ANY VIEW	Creează o vedere în orice schemă. Pentru a crea o vedere în schema altui utilizator, trebuie acordat și privilegiul CREATE ANY VIEW, iar posesorul acestuia trebuie să aibă privilegiile necesare asupra obiectelor referite de către vedere.
DROP ANY VIEW	Sterge orice vedere în orice schemă.

Alexa 4B

Comenzi Oracle și privilegii necesare

Comanda	Privilegii de sistem necesare
ALTER CLUSTER	ALTER ANY CLUSTER sau ALTER ANY TABLE
ALTER DATABASE	ALTER DATABASE
ALTER FUNCTION	ALTER ANY PROCEDURE
ALTER INDEX	ALTER ANY INDEX
ALTER PACKAGE	ALTER ANY PROCEDURE
ALTER PROCEDURE	ALTER ANY PROCEDURE
ALTER PROFILE	Pentru a modifica resursele unui profil este necesar privilegiul de sistem ALTER PROFILE. Pentru a modifica protecția și parola, sunt necesare privilegiile de sistem ALTER PROFILE și ALTER USER.
ALTER RESOURCE COST	ALTER RESOURCE COST
ALTER ROLE	ALTER ANY ROLE sau rolul respectiv să se fi acordat cu opțiunea WITH ADMIN OPTION.
ALTER ROLLBACK SEGMENT	ALTER ROLLBACK SEGMENT
ALTER SEQUENCE	ALTER ANY SEQUENCE, sau secvența trebuie să fie în propria schemă, sau privilegiul de sistem ALTER pe secvența respectivă.
ALTER SESSION	ALTER SESSION
ALTER SNAPSHOT	ALTER ANY SNAPSHOT, sau instantaneul să fie în propria schemă.
ALTER SNAPSHOT LOG	Doar proprietarul unei tabele master sau un utilizator cu privilegiul de sistem SELECT pentru tabela master poate să utilizeze această comandă.
ALTER SYSTEM	ALTER SYSTEM
ALTER TABLE	ALTER ANY TABLE, sau tabela să fie în propria schemă, sau privilegiul de sistem ALTER pe tabela respectivă.
ALTER TABLESPACE	ALTER TABLESPACE
ALTER TRIGGER	ALTER ANY TRIGGER, sau declanșatorul să fie în propria schemă.
ALTER TYPE	ALTER ANY TYPE, sau obiectul să fie în propria schemă, caz în care este necesar privilegiul de sistem CREATE TYPE sau CREATE ANY TYPE.
ALTER USER	ALTER USER (parola utilizatorului curent se poate schimba fără acest privilegiu).
ALTER VIEW	Este necesar privilegiul de sistem ALTER ANY TABLE, sau vederea trebuie să fie în propria schemă.
ANALYZE	ANALYZE ANY, sau obiectul analizat să fie în propria schemă.
AUDIT (instrucțiuni SQL)	AUDIT SYSTEM
AUDIT (obiecte din schemă)	AUDIT ANY sau obiectul de auditat trebuie să fie în propria schemă.
COMMENT	COMMENT ANY TABLE, sau vederea, instantaneul sau tabela să fie în propria schemă.

Comanda	Privilegii de sistem necesare
COMMIT	Nu este necesar nici un privilegiu.
CREATE CLUSTER	CREATE CLUSTER pentru schema proprie; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY CLUSTER plus, fie o cotă în spațiul tabel ce conține clusterul, fie privilegiul de sistem UNLIMITED TABLESPACE.
CREATE CONTROLFILE	Este necesar ca rolul OSDBA să fie disponibil utilizatorului.
CREATE DATABASE	Este necesar ca rolul OSDBA să fie disponibil utilizatorului.
CREATE DATABASE LINK	Pentru o legătură privată este necesar privilegiul CREATE DATABASE LINK. Pentru o legătură publică este necesar privilegiul CREATE PUBLIC DATABASE LINK împreună cu privilegiul CREATE SESSION pe baza de date de la distanță.
CREATE DIRECTORY	CREATE ANY DIRECTORY
CREATE FUNCTION	CREATE ANY PROCEDURE, sau, dacă funcția este în propria schemă, este necesar privilegiul de sistem CREATE PROCEDURE.
CREATE INDEX	În primul rând, proprietarul schemei care dorește să creeze un index trebuie să dețină fie spațiu suficient în spațiul tabel, fie privilegiul de sistem UNLIMITED TABLESPACE. Pentru propria schemă, trebuie ca tabela sau clusterul respectiv să fie în schemă, plus privilegiul INDEX asupra tabeliei sau privilegiul de sistem CREATE ANY INDEX. Pentru schema altui utilizator, este necesar privilegiul de sistem CREATE ANY INDEX.
CREATE LIBRARY	CREATE ANY LIBRARY. Pentru a utiliza biblioteca este necesar privilegiul de sistem EXECUTE pentru biblioteca respectivă.
CREATE PACKAGE	CREATE ANY PROCEDURE, sau, dacă pachetul este în propria schemă, este necesar privilegiul de sistem CREATE PROCEDURE.
CREATE PACKAGE BODY	CREATE ANY PROCEDURE, sau, dacă pachetul este în propria schemă, este necesar privilegiul de sistem CREATE PROCEDURE.
CREATE PROCEDURE	CREATE ANY PROCEDURE, sau, dacă procedura este în schema proprie, privilegiul de sistem CREATE PROCEDURE. Pentru a crea o procedură este necesar privilegiul de sistem ALTER ANY PROCEDURE.
CREATE PROFILE	CREATE PROFILE
CREATE ROLE	CREATE ROLE
CREATE ROLLBACK SEGMENT	CREATE ROLLBACK SEGMENT, plus, fie privilegiul de sistem UNLIMITED TABLESPACE, fie spațiu suficient în spațiul tabel.
CREATE SCHEMA	Sunt necesare privilegiile pentru instrucțiunile incluse, în caz că există.
CREATE SEQUENCE	Pentru schema proprie este necesar privilegiul de sistem CREATE SEQUENCE. Pentru schema unui alt utilizator este necesar privilegiul de sistem CREATE ANY SEQUENCE.

Comanda	Privilegii de sistem necesare
CREATE SNAPSHOT	CREATE SNAPSHOT, CREATE TABLE, și CREATE VIEW pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY SNAPSHOT, plus, fie spațiu suficient în spațiul tabel, fie privilegiul de sistem UNLIMITED TABLESPACE.
CREATE SNAPSHOT LOG	CREATE TABLE pentru proprietarul tabelei master; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY TABLE împreună cu privilegiile de sistem COMMENT ANY TABLE și SELECT asupra tabelei master.
CREATE SYNONYM	CREATE SYNONYM pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY SYNONYM; Pentru sinonime publice, este necesar privilegiul de sistem CREATE PUBLIC SYNONYM.
CREATE TABLE	CREATE TABLE pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY TABLE, plus, fie spațiu suficient în spațiul tabel, fie privilegiul de sistem UNLIMITED TABLESPACE.
CREATE TABLESPACE	CREATE TABLESPACE, plus spațul tabel SYSTEM trebuie să conțină cel puțin două segmente de rollback, inclusiv segmentul de rollback SYSTEM.
CREATE TRIGGER	CREATE TRIGGER pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY TRIGGER.
CREATE TYPE	CREATE TYPE pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY TYPE.
CREATE TYPE BODY	CREATE TYPE pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY TYPE.
CREATE USER	CREATE USER
CREATE VIEW	CREATE VIEW pentru propria schema; pentru schema altui utilizator este necesar privilegiul de sistem CREATE ANY VIEW.
DELETE	DELETE
DROP CLUSTER	DROP ANY CLUSTER, sau clusterul să fie în propria schema.
DROP DATABASE LINK	Pentru a distruge o legătură la propria bază de date aceasta trebuie să fie în propria schema. Pentru a distruge o legătură publică este necesar privilegiul de sistem DROP PUBLIC DATABASE LINK.
DROP DIRECTORY	DROP ANY DIRECTORY
DROP FUNCTION	DROP ANY PROCEDURE
DROP INDEX	DROP ANY INDEX, sau indexul trebuie să fie în propria schema.
DROP LIBRARY	DROP LIBRARY
DROP PACKAGE	DROP ANY PROCEDURE
DROP PROCEDURE	DROP ANY PROCEDURE
DROP PROFILE	DROP PROFILE
DROP ROLE	DROP ANY ROLE, sau rolul respectiv să se fi acordat cu opțiunea WITH ADMIN OPTION.
DROP ROLLBACK SEGMENT	DROP ROLLBACK SEGMENT

Comanda	Privilegii de sistem necesare
DROP SEQUENCE	DROP ANY SEQUENCE sau secvența să fie în propria schemă.
DROP SNAPSHOT	DROP ANY SNAPSHOT sau instantaneul să fie în propria schemă.
DROP SNAPSHOT LOG	DROP ANY TABLE, sau tabela să fie în propria schemă.
DROP SYNONYM (PRIVATE)	Sinonimul trebuie să fie în propria schemă sau privilegiul de sistem DROP ANY SYNONYM.
DROP SYNONYM (PUBLIC)	Sinonimul trebuie să fie în propria schemă sau privilegiul de sistem DROP ANY PUBLIC SYNONYM.
DROP TABLE	DROP ANY TABLE sau tabela să fie în propria schemă.
DROP TABLESPACE	DROP TABLESPACE.
DROP TRIGGER	DROP ANY TRIGGER sau declanșatorul să fie în propria schemă.
DROP TYPE	DROP ANY TYPE, sau definiția tipului de dată să fie în propria schemă.
DROP TYPE BODY	CREATE TYPE, sau CREATE ANY TYPE, sau DROP ANY TYPE, sau implementarea tipului respectiv de date să fie în propria schemă.
DROP USER	DROP USER
DROP VIEW	DROP ANY VIEW, sau vederea să fie în propria schemă.
GRANT (privilegii de sistem sau roluri)	Pentru a acorda un privilegiu este necesar ca acesta să se fi acordat cu opțiunea WITH ADMIN OPTION. Pentru a acorda un rol este necesar privilegiul de sistem GRANT ANY ROLE, sau rolul respectiv să se fi acordat cu opțiunea WITH ADMIN OPTION, sau rolul să fi fost creat în propria schemă.
INSERT	Tabela trebuie să fie în propria schemă sau este necesar privilegiul de sistem INSERT asupra tabelei. Pentru a putea insera rânduri în orice tabelă este necesar privilegiul de sistem INSERT ANY TABLE.
LOCK TABLE	LOCK ANY TABLE, sau orice privilegiu de obiect asupra tabelei sau vederii, sau tabela sau vederea respectivă să fie în propria schemă.
NOAUDIT (instrucțiuni SQL)	AUDIT SYSTEM
NOAUDIT(obiecte din schemă)	AUDIT ANY, sau obiectul de auditat trebuie să fie în propria schemă.
RENAME	Obiectul respectiv să fie în propria schemă.
REVOKE (privilegii de sistem sau roluri)	Pentru a revoca un privilegiu este necesar ca acesta să se fi acordat cu opțiunea WITH ADMIN OPTION. Pentru a revoca un rol este necesar privilegiul de sistem GRANT ANY ROLE, sau rolul respectiv să se fi acordat cu opțiunea WITH ADMIN OPTION, sau rolul să fi fost creat în propria schemă.
ROLLBACK	Pentru a derula înapoi tranzacția curentă nu este necesar nici un privilegiu.
SAVEPOINT	Nu este necesar nici un privilegiu.
SELECT	Privilegiul de sistem SELECT asupra tabelei sau instantaneului respectiv sau privilegiul de sistem SELECT ANY TABLE asupra oricărei tabele sau instantanei.
SET CONSTRAINT (S)	Privilegiul de sistem SELECT asupra tabelei, sau tabela respectivă să fie în propria schemă.

360 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Comanda	Privilegii de sistem necesare
SET ROLE	Este necesar ca rolurile enumerate în instrucțiunea SET ROLE să fie deja acordate utilizatorului.
SET TRANSACTION	Această comandă trebuie să fie prima instrucțiune dintr-o tranzacție.
TRUNCATE	DELETE TABLE, sau tabela sau clusterul trebuie să fie în propria schemă.
UPDATE	Privilegiul de sistem UPDATE sau tabela sau vederea să fie în propria schemă. Pentru a putea actualiza rândurile din orice tabelă este necesar privilegiul de sistem UPDATE ANY TABLE.

Anexa 4C

Roluri predefinite și privilegiile conținute de acestea

Roluri	Privilegii de sistem
CONNECT	ALTER SESSION; CREATE CLUSTER; CREATE DATABASE LINK; CREATE SEQUENCE; CREATE SESSION; CREATE SYNONYM; CREATE TABLE; CREATE VIEW.
DBA	Toate privilegiile de sistem cu opțiunea WITH ADMIN OPTION.
EXP_FULL_DATABASE	SELECT ANY TABLE; BACKUP ANY TABLE.
IMP FULL DATABASE	BECOME USER
RESOURCE	CREATE CLUSTER; CREATE PROCEDURE; CREATE SEQUENCE;CREATE TABLE CREATE TRIGGER.

•

Anexa 5

Vederile dictionarului de date

Nume vedere	Descriere
DBA_ALL_TABLES ALL_ALL_TABLES USER_ALL_TABLES	Conține descrierea tabelelor (tabele obiect sau tabele relaționale).
ALL_ARGUMENTS USER_ARGUMENTS	Lista tuturor argumentelor dintr-un obiect.
DBA_CATALOG ALL_CATALOG USER_CATALOG	Lista tuturor tabelelor, vederilor, sinonimelor și sevențelor.
DBA_CLUSTER_HASH_EXPRESSIONS ALL_CLUSTER_HASH_EXPRESSIONS USER_CLUSTER_HASH_EXPRESSIONS	Funcțiile hash ale tuturor clusterelor.
DBA_CLUSTERS ALL_CLUSTERS USER_CLUSTERS	Descrierea tuturor clusterelor.
DBA_COL_COMMENTS ALL_COL_COMMENTS USER_COL_COMMENTS	Comentarii asupra coloanelor din tabelele și vederi.
DBA_COL_PRIVS ALL_COL_PRIVS USER_COL_PRIVS	Afișează informații despre coloanele în legătură cu care au fost acordate privilegii.
ALL_COL_PRIVS_MADE USER_COL_PRIVS_MADE	Afișează informații despre coloanele în legătură cu care utilizatorul sau PUBLIC este proprietar de privilegii sau a acordat privilegii.
ALL_COL_PRIVS_RECV USER_COL_PRIVS_RECV	Afișează informații despre coloanele pentru care utilizatorul sau PUBLIC au primit privilegii.
DBA_COLL_TYPES ALL_COLL_TYPES USER_COLL_TYPES	Descrierea tipurilor de colecții.
DBA_CONS_COLUMNS ALL_CONS_COLUMNS USER_CONS_COLUMNS	Conține informații în legătură cu coloanele ce apar în definițiile constrângerilor.
DBA_CONSTRAINTS ALL_CONSTRAINTS USER_CONSTRAINTS	Conține definițiile constrângerilor tabelelor.
DBA_DB_LINKS ALL_DB_LINKS USER_DB_LINKS	Lista legăturilor bazei de date.
ALL_DEF_AUDIT_OPTS	Conține opțiunile implicite de auditare a obiectelor care sunt aplicate la crearea obiectelor.
DBA_DEPENDENCIES ALL_DEPENDENCIES USER_DEPENDENCIES	Lista tuturor dependențelor între obiecte.
DBA_DIRECTORIES ALL_DIRECTORIES	Conține lista tuturor directoarelor.

Nume vedere	Descriere
DBA_ERRORS ALL_ERRORS USER_ERRORS	Lista erorilor referitoare la obiecte.
DBA_EXTENTS USER_EXTENTS	Listează extinderile cuprinse în segmentele bazei de date.
ALL_HISTOGRAMS	Sinonim al vederii ALL_TAB_HISTOGRAMS.
DBA_IND_COLUMNS ALL_IND_COLUMNS USER_IND_COLUMNS	Lista coloanelor incluse în indecsi tabelelor. Este singura vedere care conține informații privind poziția unei coloane într-o cheie primară (coloana COLUMN_POSITION).
DBA_IND_PARTITIONS ALL_IND_PARTITIONS USER_IND_PARTITIONS	Descrie partiiile indecsilor, informații privind nivelul de partiiionare, parametrii de stocare ai partiiiei, precum și diferite statistici referitoare la partiije, determinate de comanda ANALYZE.
DBA_INDEXES ALL_INDEXES USER_INDEXES	Descrierea indecsilor tabelelor accesibile utilizatorului. Pentru a aduna statistici pentru această vedere, se folosește comanda SQL ANALYZE. Această vedere suportă ciuri paralele de indecsi partiiionaj.
DBA_JOBS ALL_JOBS USER_JOBS	Listează toate job-urile. ALL_JOBS este sinonim pentru vedere USER_JOB.
DBA_LIBRARIES ALL_LIBRARIES USER_LIBRARIES	Lista tuturor bibliotecilor.
DBA_LOBS ALL_LOBS USER_LOBS	Afișează datele de tip LOB conținute în tabele.
DBA_METHOD_PARAMS ALL_METHOD_PARAMS USER_METHOD_PARAMS	Descrie parametrii metodelor asociate tipurilor de date.
DBA_METHOD_RESULTS ALL_METHOD_RESULTS USER_METHOD_RESULTS	Descrie rezultatele metodelor asociate tipurilor de date.
DBA_NESTED_TABLES ALL_NESTED_TABLES USER_NESTED_TABLES	Descrie tipurile de dată tabel imbricat existente în tabele.
DBA_OBJECT_TABLES ALL_OBJECT_TABLES USER_OBJECT_TABLES	Descrie tabelele obiect.
DBA_OBJECTS ALL_OBJECTS USER_OBJECTS	Listează toate obiectele conținute în baza de date.
DBA_PART_COL_STATISTICS ALL_PART_COL_STATISTICS USER_PART_COL_STATISTICS	Conține statistici în legătură cu coloanele din partiiile tabelelor.
DBA_PART_HISTOGRAMS ALL_PART_HISTOGRAMS USER_PART_HISTOGRAMS	Conține datele histogramelor pentru partiiile tabelelor.
DBA_PART_INDEXES ALL_PART_INDEXES USER_PART_INDEXES	Conține informații privind partiiile la nivel de obiect pentru toți indecsii partiiionaj.

Nume vedere	Descriere
DBA_PART_KEY_COLUMNS ALL_PART_KEY_COLUMNS USER_PART_KEY_COLUMNS	Descrie coloanele care sunt și chei ale obiectelor partaționate.
DBA_PART_TABLES ALL_PART_TABLES USER_PART_TABLES	Conține informații privind partile la nivel de obiect pentru toate tabelele partaționate.
DBA_REFRESH ALL_REFRESH USER_REFRESH	Listează toate grupurile de reîmprospătare.
DBA_REFRESH_CHILDREN ALL_REFRESH_CHILDREN USER_REFRESH_CHILDREN	Listează toate obiectele din grupurile de reîmprospătare.
DBA_REFS ALL_REFS USER_REFS	Descrie coloanele de tip REF conținute în tabele.
DBA_REGISTERED_SNAPSHOTS ALL_REGISTERED_SNAPSHOTS	Listează toate instantaneile tabelelor locale situate la distanță.
DBA_SEQUENCES ALL_SEQUENCES USER_SEQUENCES	Descrierea secvențelor.
DBA_SNAPSHOT_LOGS ALL_SNAPSHOT_LOGS USER_SNAPSHOT_LOGS	Listează toate jurnalele instantaneelor.
DBA_SNAPSHOT_REFRESH_TIMES ALL_SNAPSHOT_REFRESH_TIMES USER_SNAPSHOT_REFRESH_TIMES	Listează timpul de reîmprospătare al fiecărui instantaneu.
DBA_SNAPSHOTS ALL_SNAPSHOTS USER_SNAPSHOTS	Listează toate instantaneile.
DBA_SOURCE ALL_SOURCE USER_SOURCE	Listează textul sursă al tuturor obiectelor.
DBA_SYNONYMS ALL_SYNONYMS USER_SYNONYMS	Listează informații despre sinonime.
DBA_TAB_COL_STATISTICS ALL_TAB_COL_STATISTICS USER_TAB_COL_STATISTICS	Conține statistici în legătură cu coloanele aflate în vederile DBA_TAB_COLUMNS, ALL_TAB_COLUMNS respectiv USER_TAB_COLUMNS.
DBA_TAB_COLUMNS ALL_TAB_COLUMNS USER_TAB_COLUMNS	Listează coloanele tuturor tabelelor, vederilor și clusterelor.
DBA_TAB_COMMENTS ALL_TAB_COMMENTS USER_TAB_COMMENTS	Conține comentariile create prin comanda COMMENT ON asupra tabelelor și vederilor.
DBA_TAB_HISTOGRAMS ALL_TAB_HISTOGRAMS USER_TAB_HISTOGRAMS	Conține histogramele tabelelor și vederilor.
DBA_TAB_PARTITIONS ALL_TAB_PARTITIONS USER_TAB_PARTITIONS	Descrie tabelele partaționate.
DBA_TAB_PRIVS ALL_TAB_PRIVS USER_TAB_PRIVS	Listează toate privilegiile asupra obiectelor din baza de date.

Nume vedere	Descriere
ALL_TAB_PRIVS_MADE USER_TAB_PRIVS_MADE	Afișează informații despre tabelele în legătură cu care utilizatorul este proprietar de privilegii sau a acordat privilegii.
ALL_TAB_PRIVS_REC'D USER_TAB_PRIVS_REC'D	Afișează informații despre tabelele în legătură cu care utilizatorul sau PUBLIC au primit privilegii.
DBA_TABLES ALL_TABLES USER_TABLES	Conține descrierile de tabele relaționale. Pentru a obține statistici despre această vedere se folosește comanda SQL ANALYZE.
DBA_TRIGGER_COLS ALL_TRIGGER_COLS USER_TRIGGER_COLS	Afișează utilizarea coloanelor în declanșatoare sau în tabelele ce conțin declanșatoare.
DBA_TRIGGERS ALL_TRIGGERS USER_TRIGGERS	Listează informații despre declanșatoare.
DBA_TYPE_ATTRS ALL_TYPE_ATTRS USER_TYPE_ATTRS	Conține descrierile atributelor tipurilor de date.
DBA_TYPE_METHODS ALL_TYPE_METHODS USER_TYPE_METHODS	Conține descrierile metodelor tipurilor de date.
DBA_TYPES ALL_TYPES USER_TYPES	Conține descrierile tipurilor de date.
DBA_UPDATABLE_COLUMNS ALL_UPDATABLE_COLUMNS USER_UPDATABLE_COLUMNS	Conține o descriere a coloanelor ce pot fi modificate dintr-o vedere bazată pe mai multe tabele (join view).
DBA_USERS ALL_USERS USER_USERS	Conține informații despre toți utilizatorii bazei de date. Aceste informații includ numele utilizatorului și data la care a fost creat.
DBA_VIEWS ALL_VIEWS USER_VIEWS	Conține textul vederilor.
AUDIT_ACTIONS	Conține descrieri pentru codurile de înregistrare a auditului.
CAT	Este un sinonim public pentru USER CATALOG.
CLU	Este un sinonim public pentru USER CLUSTERS.
COLS	Este un sinonim public pentru USER TAB COLUMNS.
COLUMN_PRIVILEGES	Afișează informații despre coloanele în legătură cu care utilizatorul este proprietarul unor privilegii, cele în legătură cu care a primit sau acordat privilegii, precum și cele în legătură cu care PUBLIC a primit privilegii.
DBA_AUDIT_OBJECT USER_AUDIT_OBJECT	Toate înregistrările de auditare referitoare la instrucțiuni care privesc toate obiectele din sistem.
DBA_AUDIT_SESSION USER_AUDIT_SESSION	Toate înregistrările care privesc conectările și deconectările efectuate.

Nume vedere	Descriere
DBA_AUDIT_STATEMENT USER_AUDIT_STATEMENT	Toate înregistrările de auditare referitoare la următoarele instrucțiuni: GRANT, REVOKE, AUDIT, NOAUDIT și ALTER SYSTEM.
DBA_AUDIT_TRAIL USER_AUDIT_TRAIL	Conține informații referitoare la operațiile de auditare ale bazei de date.
DBA_BLOCKERS	Listează toate sesiunile care conțin un blocaj.
DBA_CLU_COLUMNS USER_CLU_COLUMNS	Conține localizarea coloanelor tabelelor în coloanile clusterelor.
DBA_DATA_FILES	Conține informații despre fișierele bazei de date.
DBA_DDL_LOCKS	Listează toate blocările DDL din baza de date și toate cererile nerezolvate pentru blocările DDL.
DBA_DML_LOCKS	Listează toate blocările DML din baza de date și toate cererile nerezolvate pentru blocările DML.
DBA_EXP_FILES	Conține descrierea fișierelor de export.
DBA_EXP_OBJECTS	Listează obiecte ce au fost exportate.
DBA_EXP_VERSION	Conține numărul versiunii ultimei sesiuni de export.
DBA_FREE_SPACE USER_FREE_SPACE	Listează extinderile libere din toate spațiile tabel ale bazei de date.
DBA_FREE_SPACE_COALESCED	Conține statistici despre spațiul de memorie din spațiile tabel ale bazei de date.
DBA_LOCKS	Listează toate blocările din baza de date și toate cererile nerezolvate pentru blocări.
DBA_OBJ_AUDIT_OPTS USER_OBJ_AUDIT_OPTS	Listează opțiunile de auditare pentru toate tabelele și vederile.
DBA_OBJECT_SIZE USER_OBJECT_SIZE	Listează dimensiunea în bytes a tuturor obiectelor PL/SQL din baza de date.
DBA_PRIV_AUDIT_OPTS	Descrie opțiunile de auditare a privilegiilor.
DBA_PROFILES	Informații despre limitele resurselor atribuite fiecarui profil.
DBA_REFRESH_CHILDREN ALL_REFRESH_CHILDREN USER_REFRESH_CHILDREN	Listează toate obiectele din grupurile de reimprospătare.
DBA_ROLE_PRIVS USER_ROLE_PRIVS	Descrierea rolurilor acordate utilizatorilor și altor roluri.
DBA_ROLES	Listează toate rolurile existente în baza de date.
DBA_ROLLBACK_SEGS	Conține descrieri ale segmentelor de revenire.
DBA_SEGMENTS USER_SEGMENTS	Conține informații despre spațiul alocat pentru segmentele bazei de date.
DBA_STMT_AUDIT_OPTS	Conține informații ce descriu opțiunile curente ale sistemului de operare de auditare a acțiunilor unui utilizator în întreaga bază de date.
DBA_SYS_PRIVS USER_SYS_PRIVS	Descrierea privilegiilor de sistem acordate utilizatorilor și rolurilor.
DBA_TABLESPACES USER_TABLESPACES	Conține descrieri ale tuturor spațiilor tabel din baza de date.

Nume vedere	Descriere
DBA_TS_QUOTAS USER_TS_QUOTAS	Informații despre cotele spațiilor tabel pentru toți utilizatorii.
DBMS_ALERT_INFO	Conține informații despre mesajele de avertizare.
DBMS_LOCK_ALLOCATED	Listează blocajele alocate utilizatorilor.
DICT	Este un sinonim public pentru DICTIONARY.
DICT_COLUMNS	Conține descrieri ale coloanelor tabelelor și vederilor dicționarului de date.
DICTIONARY	Conține descrieri ale tabelelor și vederilor dicționarului de date.
EXCEPTIONS	Conține informații despre violări ale constrângерilor de integritate. Vederea este creată de scriptul UTLEXCPT.SQL.
GLOBAL_NAME	Conține o singură înregistrare ce conține numele global al bazei de date curente.
IND	Este un sinonim public pentru USER_INDEXES.
NLS_DATABASE_PARAMETERS	Listează parametrii NLS permanenți ai bazei de date.
NLS_INSTANCE_PARAMETERS	Listează parametrii NLS ai instanței.
NLS_SESSION_PARAMETERS	Listează parametrii NLS ai sesiunii utilizatorului.
OBJ	Este un sinonim public pentru USER_OBJECTS.
PRODUCT_COMPONENT_VERSION	Conține informații despre versiune și starea produselor componente.
PUBLIC_DEPENDENCY	Listează dependențe între obiecte folosind numărul obiectului.
PUBLICSYN	Conține informații despre sinonime publice.
RESOURCE_COST	Afișează costul fiecărei resurse.
RESOURCE_MAP	Conține descrieri ale resurselor. Asociază numele resursei cu un număr.
ROLE_ROLE_PRIVS	Conține informații despre rolurile acordate altor roluri.
ROLE_SYS_PRIVS	Conține informații despre privilegiile de sistem acordate rolurilor.
ROLE_TAB_PRIVS	Conține informații despre privilegiile referitoare la tabele acordate rolurilor.
SEQ	Este un sinonim pentru USER_SEQUENCES.
SESSION_PRIVS	Afișează privilegiile care sunt disponibile pentru utilizator în momentul respectiv.
SESSION_ROLES	Listează rolurile permise utilizatorului în momentul respectiv.
STMT_AUDIT_OPTION_MAP	Conține informații despre opțiunile de audit.
SYN	Este un sinonim public pentru USER_SYNONYMS.
SYS_OBJECTS	Asociază ID-urile obiectelor cu tipurile de obiecte și cu adresele segmentelor de date.
SYSTEM_PRIVILEGE_MAP	Conține informații despre codurile privilegiilor de sistem.

Nume vedere	Descriere
TABLE_PRIVILEGE_MAP	Conține informații despre codurile privilegiilor.
TABS	Este un sinonim public pentru USER_TABLES.
USER_RESOURCE_LIMITS	Afișează limitele resurselor pentru utilizatorul curent.

Scopul în care sunt folosite vederile dicționarului de date

Scop	Vederile dicționarului de date
Pentru urmărirea auditului	STMT_AUDIT_OPTION_MAP, AUDIT_ACTIONS, ALL_DEF_AUDIT_OPTS, DBA_STMT_AUDIT_OPTS, DBA_PRIV_AUDIT_OPTS, DBA_AUDIT_SESSION, DBA_AUDIT_STATEMENT, DBA_AUDIT_OBJECT, USER_OBJ_AUDIT_OPTS, USER_AUDIT_TRAIL, USER_AUDIT_SESSION, USER_AUDIT_STATEMENT, USER_AUDIT_OBJECT.
Pachete, proceduri și funcții stocate	ALL_ERRORS, ALL_SOURCE, DBA_ERRORS, DBA_OBJECT_SIZE, DBA_SOURCE, USER_ERRORS, USER_OBJECTS, USER_OBJECT_SIZE, USER_SOURCE.
Privilegii	ALL_COL_PRIVS, USER_COL_PRIVS, DBA_COL_PRIVS, DBA_SYS_PRIVS, ALL_COL_PRIVS_MADE, ALL_COL_PRIVS_RECV, ALL_TAB_PRIVS, ALL_TAB_PRIVS_MADE, ALL_TAB_PRIVS_RECV, USER_ROLE_PRIVS, USER_SYS_PRIVS, COLUMN_PRIVILEGES, SESSION_PRIVS.
Roluri	DBA_ROLES, USER_ROLE_PRIVS, DBA_ROLE_PRIVS, ROLE_ROLE_PRIVS, ROLE_SYS_PRIVS, ROLE_TAB_PRIVS, SESSIONS_ROLES.
Restricții de integritate	ALL_CONSTRAINTS, ALL_CONS_COLUMNS, USER_CONSTRAINTS, USER_CONS_COLUMNS, DBA_CONSTRAINTS, DBA_CONS_COLUMNS.
Utilizatori și resurse	ALL_USERS, USER_USERS, DBA_USERS, USER_TS_QUOTAS, DBA_TS_QUOTAS, USER_RESOURCE_LIMITS, DBA_PROFILES, RESOURCE_COST.
Spații tabel și fișiere de date	DBA_TABLESPACES, USER_TABLESPACES, DBA_DATA_FILES, USER_EXTENTS, USER_SEGMENTS, DBA_EXTENTS.
Indeci	USER_INDEXES, USER_IND_COLUMNS
Tabele	USER_ALL_TABLES, USER_TABLES, USER_TAB_COLUMNS, USER_CONSTRAINT, USER_CONS_COLUMNS, USER_TAB_COMMENTS, USER_COL_COMMENTS, USER_TAB_PARTITIONS
Trigger	USER_TRIGGER
Comentarii	ALL_TAB_COMMENTS, ALL_COL_COMMENTS

Anexa 6

Functii SQL și PL/SQL

Functii caracter

Sintaxă	Descriere
ASCII(caracter)	Returnează codul ASCII al unui caracter specificat ca argument. Această funcție poate primi ca argument și un sir de caractere dar în acest caz numai primul caracter este luat în considerare.
CHR(număr_intreg)	Returnează caracterul asociat codului ASCII specificat ca argument. Această funcție este inversă funcției ASCII iar caracterul returnat depinde de setul de caractere specific sistemului de operare. Se poate folosi fie funcția ASCII, fie funcția CHR pentru a stabili setul de caractere în uz.
CONCAT(șir1, șir2)	Concatenează două siruri de caractere întă-unul singur prin adăugarea celui de-al doilea sir la sfârșitul primului sir. Această funcție îndeplinește aceeași funcție ca și operatorul ' '.
INITCAP(șir)	Returnează un sir în care prima literă din fiecare cuvânt este scrisă cu literă mare, iar celelalte caractere sunt convertite în litere mici. Caracterele nealfabetice, cum ar fi spațiile, numerele și punctuația, rămân neschimbate. Această funcție întoarce același tip de date ca și cel al argumentului.
INSTR(șir, subșir, [poz], [n])	Returnează poziția unde se găsește a <i>n</i> -a apariție a unui subșir întă-un sir. Căutarea începe cu o anumită poziție de plecare specificată. Dacă poziția de plecare este pozitivă, căutarea se face de la stânga la dreapta. În cazul în care subșirul nu va fi găsit, fie că nu există, fie că poziția de plecare sau a <i>n</i> -a apariție ies din limitele sirului, funcția va întoarce valoarea 0. Dacă nu sunt specificate, poziția de plecare și numărul apariției sunt implicit egale cu 1.
INSTRB(șir, subșir, [poz], [n])	Îndeplinește aceeași funcție ca și INSTR, cu deosebirea că poziția de plecare, a <i>n</i> -a apariție și valoarea returnată sunt toate exprimate în octeți. Dacă reprezentarea caracterelor se face pe un singur octet, INSTRB va întoarce aceeași rezultat ca și INSTR.
LENGTH(șir)	Returnează lungimea unui sir, inclusiv spațiile adăugate la tipul de date CHAR. Dacă sirul argument are valoarea NULL, returnează valoarea NULL.
LENGTHB(șir)	Îndeplinește aceeași funcție ca și LENGTH, cu deosebirea că returnează numărul de octeți ocupati. Dacă reprezentarea caracterelor se face pe un singur octet, rezultatele celor două funcții vor fi identice.
LOWER(șir)	Transformă toate caracterele dintr-un sir alfanumeric în litere mici. Caracterele nealfabetice, cum ar fi spațiile, numerele și punctuația, rămân neschimbate. Această funcție întoarce același tip de date ca și cel al argumentului.

Sintaxă	Descriere
LPAD(șir, lungime, [șir])	Completează un sir de caractere, începând din stânga, până la lungimea specificată, cu un sir specificat. Dacă nu se specifică nici un sir, se consideră în mod implicit caracterul spațiu. Dacă lungimea specificată este mai mică decât lungimea sirului ce se dorește a fi completat, acesta va fi trunchiat.
LTRIM(șir, [șir])	Înlătură dintr-un sir specificat pornind de la începutul acestuia, un anumit sir de caractere. Funcția va înălătura caracterele specificate până la primul caracter care diferă de caracterele ce trebuie suprimate. Această funcție este utilă pentru înălăturarea spațiilor, a zerourilor fără semnificație, a numerelor, a cuvintelor, etc. de la începutul unui sir de caractere. Dacă nu se specifică ce caracter trebuie înălăturat, se consideră un caracter spațiu.
REPLACE(șir, sir_căutat, [șir_inlocuire])	Pune căutarea unui sir de caractere într-un anumit sir specificat, iar în cazul în care acesta a fost găsit se înlocuiește cu un alt sir. Dacă nu se specifică sirul de înlocuire, sirul căutat este pur și simplu înălăturat din sirul de intrare.
RPAD(șir, lungime, [șir])	Completează un sir de caractere, începând de la sfârșitul său spre dreapta, până la lungimea specificată, cu un sir specificat. Dacă nu se specifică nici un sir, se consideră în mod implicit caracterul spațiu. Dacă lungimea specificată este mai mică decât lungimea sirului ce se dorește a fi completat, acesta va fi trunchiat.
RTRIM(șir, [șir])	Realizează același lucru ca și funcția LTRIM cu diferența ca eliminarea se face pornind de la sfârșitul sirului spre stânga.
SOUNDEX(șir)	Returnează reprezentarea fonetică a unui sir de caractere. Cu ajutorul acestei funcții se pot căuta sirurile care au aceeași pronunție dar nu aceeași ortografiere.
SUBSTR(șir, poz, [lung])	Returnează un sir de caractere din interiorul altui sir, prin specificarea poziției de început și a lungimii sirului de extras. Dacă lungimea nu este specificată, se va extrage întregul sir. În cazul în care poziția de început este un număr pozitiv, numărătoarea se face de la stânga spre dreapta, iar dacă acesta este un număr negativ, numărătoarea se face de la dreapta spre stânga. Dacă lungimea sau poziția de început sunt trimise funcției ca numere reale, valoarea lor este trunchiată la un întreg, iar dacă lungimea este mai mică decât 1, funcția întoarce valoarea NULL.
SUBSTRB(șir, poz, [lung])	Operează la fel ca SUBSTR, cu deosebirea că poziția de început și lungimea sunt exprimate în octeți. Dacă caracterele sunt reprezentate pe un singur octet, funcția este identică cu SUBSTR.
TRANSLATE(șir, sir1, sir2)	Modifică valorile unui sir prin înlocuirea primului caracter din sir1 cu primul caracter din sir2, s.a.m.d. Dacă nu există un caracter corespondent, atunci acesta va fi șters. Dacă sir1 este mai scurt decât sir2, atunci sir2 se va trunchia. Dacă sir2 este NULL, funcția întoarce valoarea NULL.
UPPER(șir)	Transformă toate caracterele dintr-un sir alfanumeric în litere mari. Caracterele nealfabetice, cum ar fi spațiile, numerele și punctuația, rămân neschimbate. Această funcție întoarce același tip de date ca și cel al argumentului.

Funcții numerice

Sintaxă	Descriere
ABS(nr)	Returnează valoarea absolută a argumentului.
ACOS(nr)	Returnează arccosinusul argumentului, exprimat în radiani. Argumentul poate lua valori între -1 și 1, iar rezultatul este cuprins între 0 și π .
ASIN(nr)	Returnează arcsinusul argumentului, exprimat în radiani. Argumentul poate lua valori între -1 și 1, iar rezultatul este cuprins între $-\frac{\pi}{2}$ și $\frac{\pi}{2}$.
ATAN(nr)	Returnează arctangentă argumentului, exprimată în radiani. Argumentul poate lua orice valoare reală, iar rezultatul este cuprins între $-\frac{\pi}{2}$ și $\frac{\pi}{2}$.
ATAN2(y, x)	Returnează arctangenta, exprimată în radiani, a raportului dintre argumentele y și x. Argumentele pot lua orice valoare reală, mai puțin valoarea 0 pentru al doilea, iar rezultatul este cuprins între $-\pi$ și π .
CEIL(nr)	Returnează o valoare reprezentând cel mai mic număr întreg care este mai mare sau egal cu argumentul.
COS(nr)	Returnează cosinusul unghiului dat ca argument, în radiani.
COSH(nr)	Returnează cosinusul hiperbolic al argumentului.
EXP(n)	Returnează numărul e ridicat la puterea n, unde $e=2,71828183\dots$
FLOOR(nr)	Returnează o valoare reprezentând cel mai mare număr întreg care este mai mic sau egal cu argumentul.
LN(nr)	Returnează logaritmul natural al argumentului. Argumentul trebuie să fie mai mare ca 0.
LOG(bază, nr)	Returnează logaritmul unui număr într-o anumită bază specificată. Baza trebuie să fie o valoare reală mai mare ca 1, iar numărul trebuie să fie mai mare ca 0.
MOD(nr1, nr2)	Returnează restul împărțirii lui nr1 la nr2. Funcția întoarce 0 dacă nu există rest. MOD este utilă pentru a afla dacă un număr este prim sau pentru a determina divizorii unui număr.
POWER(nr, putere)	Returnează rezultatul ridicării unui număr la o anumită putere. Această funcție nu poate ridica un număr negativ la o putere care nu este un număr întreg.
ROUND(nr, [nr_zec])	Rotunjește numărul argument la un număr de zecimale specificat. Dacă numărul de zecimale este pozitiv, rotunjește la dreapta separatorului zecimal; dacă numărul de zecimale este negativ, rotunjește la stânga separatorului zecimal; dacă nu se specifică numărul de zecimale, valoarea implicită este 0, adică se rotunjește la întregul cel mai apropiat.
SIGN(nr)	Determină dacă un număr este negativ, pozitiv sau zero returnând valoarea -1, 0 sau 1.
SIN(nr)	Returnează sinusul unui număr dat ca argument, în radiani.
SINH(nr)	Returnează sinusul hiperbolic al unui număr dat ca argument.
SQRT(nr)	Întoarce rădăcina pătrată a numărului dat ca argument. Acesta trebuie să fie un număr pozitiv sau egal cu 0.
TAN(nr)	Returnează tangentă unui număr dat ca argument, în radiani.
TANH(nr)	Returnează tangentă hiperbolică al unui număr dat ca argument.

Sintaxă	Descriere
TRUNC(nr, [nr_zec])	Trunchiază numărul argument la un număr de zecimale specificat. Dacă numărul de zecimale este pozitiv, trunchiază în dreapta separatorului zecimal la numărul de zecimale specificat; dacă numărul de zecimale este negativ, trunchiază în stânga separatorului zecimal; dacă nu se specifică numărul de zecimale, valoarea implicită este 0, adică se trunchiază la un număr întreg.

Functii pentru data calendaristică

Sintaxă	Descriere
ADD_MONTHS(data, nr) sau ADD_MONTHS(nr, data)	Adună sau scade un număr de luni specificat la o dată calendaristică. Deoarece funcția este suprăincărcabilă, se pot specifica parametrii în orice ordine. Dacă numărul de luni este pozitiv, se va efectua adunarea, dacă este negativ se va efectua scăderea. La adăugarea sau la scăderea de luni calendaristice, Oracle păstrează numărul zilei din calendar, cu excepția cazului în care ultima zi dintr-o lună este mai mare decât ultima zi a lunii rezultate. Dacă luna rezultată are mai puține zile, funcția va returna ultima zi a acelei luni.
LAST_DAY(data)	Returnează ultima zi din lună a datei transmise ca argument. O aplicație utilă a acestei funcții poate fi aflarea numărului de zile rămase până la sfârșitul lunii respective.
MONTHS_BETWEEN (data1, data2)	Returnează numărul de luni dintre două date calendaristice. Dacă numărul zilei este același în ambele luni, rezultatul este un întreg. Dacă zilele diferă, este întors un rezultat fracționar, bazat pe luna de 31 de zile. Dacă a doua dată este anterioră primei date, rezultatul este negativ.
NEW_TIME (data_si_ora, zona1, zona2)	Returnează ora dintr-o anumită zonă geografică, transmitând ca argumente ale funcției data și ora unei zone de referință, numele zonei de referință și cel al zonei unde nu se cunoaște ora.
NEXT_DAY (data, nume_zi)	Returnează o dată ulterioară datei calendaristice specificate ce are numele specificat în al doilea argument al funcției. Întoarce și o oră, care este aceeași cu ora datei argument.
ROUND(data, [mască])	Rotunjeste, pe baza măștii de format, o anumită dată calendaristică. În funcție de masca de format rotunjirea va fi făcută la zi, lună, trimestru, an, secol, etc. Măștile de format sunt prezentate în tabelul 1. Mască implicită este DD. Funcția ROUND utilizează aceeași mască de format ca și funcția TRUNC.
SYSDATE	Returnează data și ora curentă din serverul Oracle. Această funcție nu returnează data și ora clientului Oracle.
TRUNC(data, [mască])	Trunchiază, pe baza măștii de format, o anumită dată calendaristică. În funcție de masca de format rotunjirea va fi făcută la zi, lună, trimestru, an, secol, etc. Măștile de format sunt prezentate în tabelul 1. Mască implicită este DD. Funcția TRUNC utilizează aceeași mască de format ca și funcția ROUND.

Tabelul 1: Măști de format pentru funcția ROUND și TRUNC

Format	Unitatea de rotunjire sau trunchiere
CC SCC	Rotunjește sau trunchiază la secol.
YYYY YY YEAR SYEAR YYY YY Y	Trunchiază la anul curent sau rotunjește la anul următor dacă data calendaristică este după 1 iulie.
IYYY IY IY I	Rotunjește sau trunchiază la anul ISO.
Q	Trunchiază la trimestrul curent sau rotunjește la trimestrul următor dacă data calendaristică este după a 16-a zi a celei de-a doua luni a trimestrului.
MONTH MON MM RM	Trunchiază la luna curentă sau rotunjește la luna următoare dacă data calendaristică este după a 16-a zi a lunii.
WW	Rotunjește sau trunchiază la aceeași zi a săptămânii ca și prima zi a anului.
IW	Rotunjește sau trunchiază la aceeași zi a săptămânii ca și prima zi a anului ISO.
W	Rotunjește sau trunchiază la aceeași zi a săptămânii ca și prima zi a lunii.
DDD DD J	Rotunjește sau trunchiază la zi.
DAY DY D	Rotunjește sau trunchiază la prima zi a săptămânii.
HH HH12 HH24	Trunchiază la ora curentă sau rotunjește la următoarea oră dacă dintr-o oră s-au scurs mai mult de 30 de minute.
MI	Trunchiază la minutul curent sau rotunjește la următorul minut dacă dintr-un minut s-au scurs mai mult de 30 de secunde.

Funcții de conversie

Sintaxă	Descriere
CHARTOROWID(șir)	Convertește tipul de date CHAR sau VARCHAR2 la tipul ROWID.
COVERT(șir, set_dest, [set_sursă])	Este utilizată pentru conversia între diferite seturi de caractere. Setul sursă este opțional și, dacă nu este specificat, se consideră a fi în mod preestabilit setul utilizat de baza de date. Seturile de caractere recunoșcute sunt: <ul style="list-style-type: none"> - US7ASCII (7 biți, ASCII SUA) - WE8DEC (8 biți, vest-european, DEC) - WE8HP (8 biți, vest-european, Hewlett-Packard) - F7DEC (7 biți, francez, DEC) - WEBCDIC500 (IMB, vest-european, EBCDIC) - WEPC850 (vest-european, PC) - WE8ISO8895P1 (vest-european, ISO8895-1)

Sintaxă	Descriere
HEXTORAW(șir, set_dest, [set_sursă])	Convertește șirurile ce conțin valori hexazecimale în valori de tip RAW.
RAWTOHEX(valoare_RAW)	Convertește o valoare de tip RAW într-un șir de caractere în format hexazecimal.
ROWIDTOCHAR(valoare_ROWID)	Convertește o valoare de tip ROWID într-un șir de 18 caractere.
TO_CHAR(dată, mască_format, [NLS])	Convertește valorile de tip DATE într-un șir VARCHAR2, permitând prezentarea datei într-un anumit format. Această funcție va permite formatarea unei valori de tip DATE în diverse moduri. Şabloanele de format sunt prezentate în tabelul 2 - elemente de formatare a datelor calendaristice - de mai jos. Parametrii NLS dă posibilitatea specificării limbii, care va servi la afișarea pe ecran a datei conform sistemului din limba respectivă.
TO_CHAR(nr, mască_format, NLS)	Convertește orice fel de număr într-un șir de caractere VARCHAR2. Această funcție va permite îmbunătățirea afișării valorilor de tip NUMBER. Formatarea se poate face în diverse moduri. Şabloanele de format sunt prezentate în tabelul 3 - elemente de formatare a numerelor, de mai jos.
TO_DATE(șir, mască_format, [NLS])	Convertește un șir de caractere (CHAR sau VARCHAR2), delimitat prin apostrofuri ('), la o valoare DATE. Parametrul opțional NLS va permite precizarea limbii în care se face afișarea. Şabloanele de același format ca și în cazul funcției TO_CHAR cu argument de tip DATE, și sunt prezentate în tabelul 2 - elemente de formatare a datelor calendaristice - de mai jos.
TO_MULTI_BYTE(șir)	Convertește un șir de caractere reprezentate pe câte un singur octet în forma multi-octet. Această funcție operează doar pe sistemele care pot reprezenta caracterele atât pe un octet cât și pe mai mulți.
TO_NUMBER(șir, mască_format, [NLS])	Această funcție este foarte asemănătoare funcției TO_DATE. Funcția convertește un șir de caractere de tip CHAR sau VARCHAR2 într-un număr. Şabloanele au același format ca și în cazul funcției TO_CHAR ce are ca argument un număr, și sunt prezentate în tabelul 3 - elemente de formatare a numerelor - de mai jos.
TO_SINGLE_BYTE(șir)	Convertește toate caracterele dintr-un șir care au reprezentare pe mai mulți octeți în omologele lor cu reprezentare pe un singur octet. Dacă baza de date nu are decât caractere reprezentate pe câte un singur octet, funcția nu are nici un efect, ieșirea este aceeași ca intrarea.

Tabelul 2: Elemente de formatare a datelor calendaristice

Indicator de format	Descriere
BC sau B.C.	Indicator "i. Cr." (Before Christ, înainte de Christos)
AD sau A.D.	Indicator "d. Cr." (Anno Domini, după Christos)
CC, SCC	Cod de secol. Întoarce un număr negativ dacă se utilizează BC cu SCC.

Indicator de format	Descriere
YYYY, YYYY	Anul cu 4 cifre. Întoarce un număr negativ dacă se utilizează BC cu YYYY.
YYYY	An ISO cu 4 cifre.
Y, YY	An de 4 cifre, cu inserarea unei virgule.
YY, YY, Y	Ultimele 3,2 sau ultima cifră a reprezentării anului. Valoarea prestatibilită este secolul curent.
IYY, IY, I	Ultimele 3,2 sau ultima cifră a anului ISO. Valoarea prestatibilită este secolul curent.
YEAR, SYEAR	Întoarce anul, în litere. SYEAR întoarce o valoare negativă dacă se utilizează cu BC.
RR	Ultimele două cifre ale anului, în secolul precedent sau în secolul următor.
Q	Trimestrul (de la 1 la 4).
MM	Numărul de luni, de la 01 la 12.
MONTH	Numele lunii, intotdeauna pe 9 caractere, eventual completate la dreapta cu spații.
MON	Numele lunii, prescurtat la 3 caractere.
RM	Număr roman reprezentând lună, de la I la XII.
WW	Numărul săptămânii în an, de la 1 la 53.
IW	Numărul săptămânii ISO din an, cu valori de la 1 la 52 sau de la 1 la 53.
W	Numărul săptămânii din lună, cu valori de la 1 la 5. Săptămâna 1 începe din prima zi a lunii.
D	Numărul zilei din săptămână, cu valori de la 1 la 7.
DD	Numărul zilei din lună, cu valori de la 1 la 31.
DDD	Numărul zilei din an, cu valori de la 1 la 366.
DAY	Numele zilei din săptămână, în litere, totdeauna reprezentat pe 9 caractere, eventual completat la dreapta cu spații.
DY	Numele zilei din săptămână, prescurtat la două caractere.
J	Ziua din calendarul iulian, numărată începând cu 1 ianuarie 4712 î. Cr.
HH, HH12	Ora din zi, cu valori de la 1 la 12.
HH24	Ora din zi, cu valori de la 0 la 23.
MI	Numărul minutului din oră, de la 0 la 59.
SS	Numărul secundei din minut, de la 0 la 59.
SSSS	Numărul secundelor scurse de la miezul nopții, cu valori de la 0 la 86399 (24 ore * 60 minute * 60 secunde = 86400).
AM sau A . M .	Indicator al orei ante-meridian (dimineață). ·
PM sau P . M .	Indicator al orei post-meridian (după-masă).
Punctuație	Toată punctuația, care este inclusă în limita de 220 de caractere.
Text	Tot textul, care este inclus în limita de 220 de caractere.
TH	Sufix pentru conversia numerelor la formatul ordinal specific limbii engleză: 1 devine 1 st , 2 devine 2 nd etc. Funcționează numai în limba engleză.
SP	Convertește un număr în litere. Funcționează numai în limba engleză. Exemplu: 109 devine one hundred nine.
SPTH	Convertește un număr în forma ordinală ,în litere. Funcționează numai în limba engleză. Exemplu: 1 devine FIRST, 2 devine SECOND etc.
FX	Utilizează corespondență exactă între elementul de date calendaristică și format.
FM	Fill Mode (mod de umplere): Activează și dezactivează suprimarea spațiilor la afișare, după conversie.

Tabelul 3: Elemente de formatare a numerelor

Mască de format	Exemplu	Descriere
9	9999	Fiecare 9 este considerat o cifră semnificativă. Orice zerouri din fața numărului sunt tratate drept spații.
0	09999 sau 99990	Adăugând 0 înainte sau după număr, toate zerourile din stânga sau din dreapta acestuia sunt tratate și afișate ca zerouri, nu ca spații. Vă puteți gândi la acest tip de afișaj ca la unul numeric, de exemplu 00109.
\$	\$9999	Prefix reprezentând simbolul monetar, plasat în fața numărului.
B	B9999	Întoarce orice porțiune a întregului drept spații, dacă întregul este 0. Această mască de format va suprima zerourile din stânga folosind un 0 pentru format.
MI	9999MI	Adaugă automat un spațiu la sfârșitul numărului, fie pentru un semn minus, în cazul în care numărul este negativ, fie pentru un marcat de rezervare, dacă numărul este pozitiv.
S	S9999 sau 9999S	Afișează un semn + în fața sau în urma numărului, dacă este pozitiv, și un semn – dacă valoarea este negativă.
PR	9999PR	Dacă valoarea numărului este negativă, numărul este încadrat de semnele []. Dacă valoarea este pozitivă, se utilizează spații ca marcaje de rezervare.
D	99D99	Indică poziția separatorului zecimal. Cifrele de 9, de o parte și de alta, reprezintă numărul maxim de cifre permis
G	9G999G99 9	Indică un delimitator de grup zecimal (virgula, în reprezentare anglo-saxonă).
C	C99	Întoarce simbolul monetar ISO în poziția specificată.
L	L9999	Indică poziția simbolului monetar local (de ex. \$).
,	9, 999, 99 9	Plasează virgula în pozițiile specificate, indiferent de delimitatorul de grup.
.	9. 99	Indică poziția punctului zecimal, indiferent de cea a separatorului zecimal.
V	999V99	Întoarce numărul din stânga lui V, înmulțit cu 10 ridicat la puterea indicată de numărul din dreapta lui V.
EEEE	9. 99EEEE	Întoarce numărul în reprezentare exponentială.
RM, rm	RM, rm	Întoarce numărul în reprezentare romană, cu majuscule sau minuscule.
FM	FM9, 999. 99	Înlătură spațiile din fața și din urma numărului (Fill mode).

Functii diverse

Sintaxă	Descriere
BFILENAME ('obiect', 'nume_fișier')	Returnează un pointer către calea și locul unde este stocat un fișier binar LOB. Trebuie creat un obiect care memorează calea spre fișierul binar.

Sintaxă	Descriere
DECODE (nume_coloana, valoare1, rezultat1 [,valoare2, , rezultat2,...,] [rezultat])	Aceasta funcție este una dintre cele mai importante funcții SQL. Ea practic facilitează interogările condiționale prin faptul că acționează ca o comandă 'if-then-else' sau 'case' dintr-un limbaj procedural. Primul argument al funcției este numele coloanei care va fi evaluată. Argumentele valoare1, valoare2,...etc. reprezintă valorile care se vor compara cu datele din coloana specificată ca prim argument al funcției, iar în cazul în care acestea vor fi regăsite funcția va returna valoarea rezultat1, rezultat2,...etc. În cazul în care nu vor fi regăsite funcția va returna valoarea rezultat, iar în cazul în care aceasta nu este specificată se va returna valoarea NULL.
DUMP (col, [format], [poz], [lung])	Returnează o valoare de tip VARCHAR2 ce conține informații referitoare la codul tipului de dată, lungimea în bytes, precum și reprezentarea în formatul numeric specificat a unei anumite coloane specificate sau a unei subsecvențe din coloana respectivă prin specificarea poziției de plecare și a lungimii acestora. Formatul numeric poate fi: 8 (octal), 10 (decimal), 16 (hexazecimal), 17 (caractere individuale). Codul tipului de date returnat de către funcție poate fi: 1 (VARCHAR2), 2 (NUMBER), 8 (LONG), 12 (DATE), 23 (RAW), 24 (LONGRAW), 69 (ROWID), 96 (CHAR).
EMPTY_BLOB ()	Inițializează o variabilă sau o coloană de tip BLOB. Această funcție nu există în versiuni anterioare Oracle8.
EMPTY_CLOB ()	Inițializează o variabilă sau o coloană CLOB. Această funcție nu există în versiuni anterioare Oracle8.
GREATEST (val1, [val2], [val3],...)	Returnează cea mai mare valoare dintr-o listă de valori. Toate elementele din lista de valori sunt mai întâi convertite la tipul primului element din listă.
LEAST (val1, [val2], [val3],...)	Returnează cea mai mică valoare dintr-o listă de valori. Toate elementele din lista de valori sunt mai întâi convertite la tipul primului element din listă.
NLS_CHARSET_ID (nume_set_caractere)	Returnează numărul de identificare al setului de caractere NLS asociat cu numele setului transmis ca argument. Această funcție nu există în versiuni anterioare Oracle8.
NLS_CHARSET_NAME (id_set_caractere)	Returnează numele setului de caractere NLS asociat cu identificatorul setului transmis ca argument. Această funcție nu există în versiuni anterioare Oracle8.
NVL (expr1, expr2)	Convertește valorile NULL întâlnite în primul argument al funcției cu valoarea celui de-al doilea argument. Cel de-al doilea argument trebuie să aibă valoare diferită de NULL.
SQLCODE	Returnează codul erorii curente. Este utilizată mai ales în tratarea erorilor. Vezi capitolul PL/SQL.
SQLERRM (cod_eroare)	Returnează mesajul de eroare asociat codului erorii. Este utilizată mai ales în tratarea erorilor. Vezi capitolul PL/SQL.
UID	Returnează numărul unic asociat identificatorului de utilizator pentru utilizatorul curent al bazei de date.
USER	Returnează numele utilizatorului curent al bazei de date, sub formă de sir VARCHAR2.

378 Dezvoltarea aplicațiilor de baze de date în Oracle8 și Forms6

Sintaxă	Descriere
USERENV (opțiune)	Returnează o valoare de tip VARCHAR2 ce conține informații despre mediul de lucru al bazei de date la care s-a conectat utilizatorul. Informația returnată depinde de argumentul funcției care poate fi: OSDBA (returnează TRUE dacă utilizatorul are rolul OSDBA activat), LANGUAGE (returnează limba și setul de caractere folosit de sesiunea curentă), etc.
VSIZE (coloană) Sau VSIZE (valoare)	Returnează numărul de bytes pe care este reprezentată intern o anumită coloană sau valoare; Dacă valoarea este NULL, se întoarce NULL.

Anexa 7

Pachete PL/SQL predefinite utilizate frecvent

DBMS_OUTPUT

Afișează informații (în general, pe ecran) atunci când se execută un bloc sau un subprogram PL/SQL, ajutând la testarea și depanarea acestuia. Pachetul DBMS_OUTPUT lucrează cu un buffer în care poate fi scrisă orice informație utilizând procedurile PUT, PUT_LINE și NEW_LINE. Această informație poate fi regăsită folosind procedurile GET_LINE și GET_LINES.

Functiile și procedurile din pachetul DBMS_OUTPUT

Nume	Tip	Descriere
ENABLE	Procedură	Activează ieșirea mesajului. În SQL*Plus această procedură este echivalentă cu setarea variabilei de mediu SERVEROUTPUT pe valoarea ON. Sintaxa: DBMS_OUTPUT.ENABLE (buffer_size IN INTEGER DEFAULT nr)
DISABLE	Procedură	Deactivează ieșirea mesajului. În SQL*Plus această procedură este echivalentă cu setarea variabilei de mediu SERVEROUTPUT pe valoarea OFF. Sintaxa: DBMS_OUTPUT.DISABLE
PUT	Procedură	Scrie textul, conținut de argumentul funcției, pe linia curentă a buffer-ului. Deoarece procedura PUT plasează informația în buffer, dar nu adaugă sfârșit de linie, efectul acestea nu se va vedea. Pentru a se vedea efectele se utilizează procedura PUT LINE. Procedura poate fi suprîncărcată, adică argumentul poate fi atât de tip NUMBER cât și de tip VARCHAR2 sau DATE. Sintaxa: DBMS_OUTPUT.PUT (item IN NUMBER) DBMS_OUTPUT.PUT (item IN VARCHAR2) DBMS_OUTPUT.PUT (item IN DATE)
PUT_LINE	Procedură	Scrie textul, conținut de argumentul funcției, pe linia curentă a buffer-ului, urmat de delimitatorul sfârșit de linie. Prin urmare pentru următoarea comandă ce folosește buffer-ul, cursorul va fi poziționat la începutul unei noi linii. Procedura poate fi suprîncărcată, adică argumentul poate fi atât de tip NUMBER cât și de tip VARCHAR2 sau DATE. Sintaxa: DBMS_OUTPUT.PUT_LINE (item IN NUMBER) DBMS_OUTPUT.PUT_LINE (item IN VARCHAR2) DBMS_OUTPUT.PUT_LINE (item IN DATE)
NEW_LINE	Procedură	Positionează cursorul buffer-ului de ieșire la începutul unei lini noi prin plasarea în acesta a unui delimitator sfârșit de linie. Sintaxa: DBMS_OUTPUT.NEW_LINE

Nume	Tip	Descriere
GET_LINE	Procedură	<p>Preia linia curentă din buffer-ul de ieșire într-un argument al procedurii.</p> <p>Sintaxă:</p> <pre>DBMS_OUTPUT.GET_LINE(line OUT VARCHAR2, status OUT INTEGER)</pre> <p>Unde:</p> <ul style="list-style-type: none"> - line este un parametru de ieșire al procedurii ce conține linia curentă din buffer-ul de ieșire; - status este un parametru de ieșire al procedurii ce conține starea de execuție a procedurii. Dacă procedura a fost executată cu succes va conține valoarea 0, iar dacă nu a fost returnată nici o linie va conține valoarea 1.
GET_LINES	Procedură	<p>Preia un anumit număr de linii din buffer-ul de ieșire într-un argument al procedurii.</p> <p>Sintaxă:</p> <pre>DBMS_OUTPUT.GET_LINES(lines OUT CHARARR, nrlines IN OUT INTEGER)</pre> <p>Unde:</p> <ul style="list-style-type: none"> - lines este un parametru de ieșire al procedurii ce conține un vector de linii din buffer-ul de ieșire; acest parametru este de tipul CHARARR, definit în specificația pachetului DBMS_OUTPUT ca TABLE OF VARCHAR2 (255). - nrlines este un parametru de intrare/ieșire al procedurii ce conține numărul de linii care se dorește a fi preluat din buffer, iar după execuția procedurii va conține numărul efectiv de linii preluat.

DBMS_DDL

Recompilează proceduri, funcții sau pachete și analizează indecsări, tabele sau clusteri. Acest pachet rescrie comenzi din SQL, astfel încât aceste facilități să fie disponibile și în PL/SQL.

Functiile și procedurile din pachetul DBMS_DDL

Nume	Tip	Descriere
ALTER_COMPILE	Procedură	<p>Este similară cu comenziile ALTER PROCEDURE Pro*C COMPILE, ALTER FUNCTION func COMPILE și ALTER PACKAGE pack COMPILE. Această procedură nu este permisă în cadrul declanșatorilor, procedurilor apelate din Oracle Forms sau tranzacțiilor read-only.</p> <p>Sintaxă:</p> <pre>DBMS_DDL.ALTER_COMPILE (type VARCHAR2, schema VARCHAR2, name VARCHAR2)</pre>
ANALYZE_OBJECT	Procedură	<p>Este similară cu comenziile ANALYZE INDEX, ANALYZE TABLE și ANALYZE CLUSTER.</p> <p>Sintaxă:</p> <pre>DBMS_DDL.ANALYZE_OBJECT (type VARCHAR2, schema VARCHAR2, name VARCHAR2, method VARCHAR2, estimate_rows NUMBER DEFAULT NULL, estimate_percent NUMBER DEFAULT NULL)</pre>

DBMS _UTILITY

Utilități DBA (analizează obiectele dintr-o anumită schemă, verifică dacă serverul rulează în mod paralel și returnează timpul în sutimi de secundă etc.). Acest pachet rescrie comenzi din SQL, astfel încât aceste facilități să fie disponibile și în PL/SQL.

Funcțiile și procedurile din pachetul DBMS _UTILITY

Nume	Tip	Descriere
ANALYZE DATABASE	Procedură	Analizează toate tabelele, clusterele și indecesii din baza de date.
ANALYZE PART_OBJECT	Procedură	Analizează obiectele schemei pentru fiecare parteție, în paralel.
ANALYZE SCHEMA	Procedură	Analizează toate tabelele, clusterele și indecesii dintr-o schemă. Această procedură este echivalentă cu apelarea procedurii DBMS_DDL.ANALYZE_OBJECT pentru toate obiectele unei scheme. <i>Sintaxă:</i> DBMS UTILITY.ANALYZE_SCHEMA(schema VARCHAR2, method VARCHAR2, estimate_rows NUMBER DEFAULT NULL, estimate_percent NUMBER DEFAULT NULL)
COMMA_TO_TABLE	Procedură	Convertește o listă de nume separate prin virgule într-o tabelă PL/SQL de nume.
COMPILE_SCHEMA	Procedură	Compilează toate funcțiile, procedurile și pachetele din schema specificată. Această procedură este echivalentă cu apelarea procedurii DBMS_DDL.ALTER_COMPILE pentru toate procedurile, funcțiile și pachetele accesibile, compilarea fiind făcută în ordinea dependențelor. Această procedură nu este permisă în cadrul declanșatorilor, procedurilor apelate din Oracle Forms sau tranzacțiilor read-only. <i>Sintaxă:</i> DBMS UTILITY.COMPILE_SCHEMA (schema VARCHAR2)
DATA_BLOCK_ADDRESS_BLOCK	Funcție	Preia din adresa blocului de date partea ce conține numărul blocului.
DATA_BLOCK_ADDRESS_FILE	Funcție	Preia din adresa fișierului de date partea ce conține numărul blocului.
DB_VERSION	Procedură	Returnează versiunea bazei de date.
EXEC_DDL_STATEMENT	Procedură	Execută instrucțiunea DDL dată.
FORMAT_CALL_STACK	Funcție	Afișează știva apelului curent <i>Sintaxă:</i> DBMS UTILITY.FORMAT_CALL_STACK RETURN VARCHAR2
FORMAT_ERROR_STACK	Funcție	Afișează știva erorii curente. <i>Sintaxă:</i> DBMS UTILITY.FORMAT_ERROR_STACK RETURN VARCHAR2
GET_HASH_VALUE	Funcție	Calculează valoarea de dispersie a unui sir dat.
GET_PARAMETER_VALUE	Funcție	Preia valoarea parametrului specificat.
GET_TIME	Funcție	Găsește ora curentă în sutimi de secundă.

Nume	Tip	Descriere
IS_PARALLEL SERVER	Funcție	Returnează valoarea True dacă baza de date funcționează în mod paralel și False în caz contrar.
MAKE_DATA_BLOCK_ADDRESS	Funcție	Creează o adresă a unui bloc de date dintr-un număr de fișier și un număr de bloc.
NAME_RESOLVE	Procedură	Aplică RESOLVE numelui obiectului specificat.
NAME_TOKENIZE	Procedură	Apelează analizatorul sintactic pentru a returna un sir împărțit în atomi lexicali.
PORT_STRING	Funcție	Returnează un sir ce conține versiunea serverului Oracle și sistemul de operare.
TABLE_TO_COMMAS	Procedură	Convertește o tabelă PL/SQL de nume într-o listă de nume separate prin virgule.

DBMS_SESSION

Modifică anumite caracteristici ale sesiunii unui utilizator, setează rolul unui utilizator și reinițializează starea unui pachet. Acest pachet descrie comenzi din SQL, astfel încât aceste facilități să fie disponibile și în PL/SQL.

Funcțiile și procedurile din pachetul DBMS_SESSION

Nume	Tip	Descriere
CLOSE_DATABASE_LINK	Procedură	Este similară comenzii SQL ALTER SESSION CLOSE DATABASE dblink <i>Sintaxă:</i> DBMS_SESSION.CLOSE_DATABASE_LINK (dblinc VARCHAR2)
ET_CLOSE_CACHED_OPEN_CURSORS	Procedură	Este similară comenzii ALTER SESSION SET CLOSE_CACHED_OPEN_CURSORS <i>Sintaxă:</i> DBMS_SESSION.ET_CLOSE_CACHED_OPEN_CURSORS (close_cursors BOOLEAN)
FREE_UNUSED_USER_MEMORY	Procedură	Elibereză memoria neutilizată. Nu există nici o comandă SQL echivalentă cu această procedură. Nu are nici un argument.
IS_ROLE_ENABLED	Procedură	Determină dacă un rol este disponibil sau nu. Nu există nici o comandă SQL echivalentă cu această procedură. Nu are nici un argument.
RESET_PACKAGE	Procedură	Reinițializează starea tuturor pachetelor. Nu există nici o comandă SQL echivalentă cu această procedură. Nu are nici un argument.
SET_NLS	Procedură	Este similară comenzii ALTER SESSION SET nls_params=nls_param_values <i>Sintaxă:</i> DBMS_SESSION.SET_NLS (param VARCHAR2, value VARCHAR2)
SET_ROLE	Procedură	Este similară comenzi SET ROLE ... <i>Sintaxă:</i> DBMS_SESSION.SET_ROLE(role cmd VARCHAR2)

Nume	Tip	Descriere
SET_SQL_TRACE	Procedură	Este similară comenzii ALTER SESSION SET SQL_TRACE=[TRUE FALSE] <i>Sintaxă:</i> DBMS_SESSION.SET_SQL_TRACE(sql_trace BOOLEAN)
UNIQUE_SESSION_ID	Procedură	Această funcție returnează un identificator unic al sesiunii. Nu există nici o comandă SQL echivalentă cu această procedură. Nu are nici un argument.

DBMS_SQL

Accesează baza de date folosind SQL dinamic. Noțiunea de SQL dinamic constă din faptul că instrucțiunile sunt create la momentul execuției unui bloc PL/SQL sub forma unor șiruri de caractere, sunt verificate sintactic și apoi sunt executate. Procesarea este diferită pentru comenzi DML, pentru comenzi DDL, pentru comenzi de interogare sau pentru blocuri PL/SQL anonime.

În general procesul constă din următorii pași:

1. Se pune instrucțiunea SQL sau blocul PL/SQL într-un șir de caractere;
2. Se analizează șirul utilizând procedura DBMS_SQL.PARSE;
3. Se stabilește legătura cu variabilele de intrare utilizând procedura DBMS_SQL.BIND_VARIABLE;
4. Dacă comanda nu este o interogare, aceasta se execută utilizând DBMS_SQL.EXECUTE și DBMS_SQL.VARIABLE_VALUE.
5. Dacă comanda este o interogare, se definesc variabilele de ieșire prin procedura DBMS_SQL.DEFINE_COLUMN, se execută interogarea prin procedura DBMS_SQL.EXECUTE, se încarcă rezultatele utilizând DBMS_SQL.VARIABLE_VALUE, DBMS_SQL.COLUMN_VALUE și DBMS_SQL.FETCH_ROWS.

Functiile și procedurile din pachetul DBMS_SQL

Nume	Tip	Descriere
OPEN_CURSOR	Funcție	<p>Deschide un nou cursor și asignează acestuia un identificator.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_SQL.OPEN_CURSOR RETURN INTEGER</pre>
PARSE	Procedură	<p>Analizează sintactic o comandă DML sau DDL și o asociază unui cursor deschis.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_SQL.PARSE(c IN INTEGER, statement IN VARCHAR2, language_flag IN INTEGER)</pre> <p>sau pentru comenzi de dimensiuni mai mari de 32K: <pre>DBMS_SQL.PARSE(c IN INTEGER, statement IN VARCHAR2S, lb IN INTEGER, ub IN INTEGER, lfflg IN BOOLEAN, language_flag IN INTEGER)</pre> <p><i>Unde:</i></p> <ul style="list-style-type: none"> - c este identificatorul cursorului asociat comenzi ce urmează a fi analizată sintactic; - statement este comanda ce urmează a fi analizată; definiția tipului VARCHAR2S este: type varchar2s is table of varchar2(256) index by binary_integer; - lb este limita inferioară pentru elementele din comandă; - ub este limita superioară pentru elementele din comandă; - lfflg (linefeed flag) este setat pe valoarea True atunci când se dorește ca la concatenare să se insereze delimitatorul sfârșit de linie după fiecare element din listă - language_flag determină modul în care Oracle va trata comanda SQL (de exemplu poate avea valoarea V6, adică Versiunea 6) </p>
BIND_VARIABLE	Procedură	<p>Leagă o valoare la o variabilă de intrare.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_SQL.BIND_VARIABLE(c IN INTEGER, name IN VARCHAR2, value IN <tip_data>)</pre> <p><i>unde:</i></p> <ul style="list-style-type: none"> - c este identificatorul unui cursor; - name este numele variabilei; - value este valoarea pe care o ia variabila și al cărei tip_data poate fi NUMBER, DATE, VARCHAR2, BLOB, CLOB, BFILE, etc. deoarece procedura este suprăîncărabilă.
DEFINE_COLUMN	Procedură	<p>Leagă o variabilă la o coloană a cursorului.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_SQL.DEFINE_COLUMN(c IN INTEGER, position IN INTEGER column IN <tip_data>)</pre>

Nume	Tip	Descriere
EXECUTE	Funcție	Execută cursorul. <i>Sintaxă:</i> DBMS_SQL.EXECUTE(c IN INTEGER) RETURN INTEGER
EXECUTE_AND_FETCH	Funcție	Execută cursorul și preia linii. <i>Sintaxă:</i> DBMS_SQL.EXECUTE_AND_FETCH(c IN INTEGER, exact IN BOOLEAN DEFAULT FALSE) RETURN INTEGER
FETCH_ROWS	Funcție	Preia rândurile din cursor. <i>Sintaxă:</i> DBMS_SQL.FETCH_ROWS(c IN INTEGER) RETURN INTEGER
VARIABLE_VALUE	Procedură	Refac valoarea unei variabile de legătură de ieșire. <i>Sintaxă:</i> DBMS_SQL.VARIABLE_VALUE(c IN INTEGER, name IN VARCHAR2, value OUT <tipdata>)
COLUMN_VALUE	Procedură	Mută o valoare de coloană dintr-un cursor într-o variabilă de ieșire. <i>Sintaxă:</i> DBMS_SQL.COLUMN_VALUE(c IN INTEGER, position IN INTEGER, value OUT <tipdata> [, column_error OUT NUMBER] [, actual_length OUT INTEGER])
CLOSE_CURSOR	Procedură	Închide cursorul după procesare. <i>Sintaxă:</i> DBMS_SQL.CLOSE_CURSOR(c IN OUT INTEGER)

DBMS_TRANSACTION

Controlează și îmbunătățește performanțele tranzacțiilor bazei de date. Acest pachet revoie comenzi din SQL, astfel încât aceste facilități să fie disponibile și în PL/SQL.

Funcțiile și procedurile din pachetul DBMS_TRANSACTION

Nume	Tip	Descriere
ADVISE_COMMIT	Procedură	Este similară comenzi SQL ALTER SESSION ADVISE COMMIT <i>Sintaxă:</i> DBMS_TRANSACTION.ADVISE COMMIT
ADVISE_ROLLBACK	Procedură	Este similară comenzi SQL ALTER SESSION ADVISE ROLLBACK <i>Sintaxă:</i> DBMS_TRANSACTION.ADVISE ROLLBACK
ADVISE NOTHING	Procedură	Este similară comenzi SQL ALTER SESSION ADVISE NOTHING <i>Sintaxă:</i> DBMS_TRANSACTION.ADVISE NOTHING

Nume	Tip	Descriere
COMMIT	Procedură	Este similară comenzi SQL COMMIT <i>Sintaxă:</i> DBMS_TRANSACTION.COMMIT
COMMIT_COMMENT	Procedură	Este similară comenzi SQL COMMIT COMMENT text <i>Sintaxă:</i> DBMS_TRANSACTION.COMMIT_COMMENT (cmnt VARCHAR2)
COMMIT_FORCE	Procedură	Este similară comenzi SQL COMMIT FORCE text <i>Sintaxă:</i> DBMS_TRANSACTION.COMMIT_FORCE (xid VARCHAR2, scn VARCHAR2 DEFAULT NULL)
READ_ONLY	Procedură	Este similară comenzi SQL SET TRANSACTION READ ONLY <i>Sintaxă:</i> DBMS_TRANSACTION.READ_ONLY
READ_WRITE	Procedură	Este similară comenzi SQL SET TRANSACTION READ WRITE <i>Sintaxă:</i> DBMS_TRANSACTION.READ_WRITE
ROLLBACK	Procedură	Este similară comenzi SQL ROLLBACK <i>Sintaxă:</i> DBMS_TRANSACTION.ROLLBACK
ROLLBACK_FORCE	Procedură	Este similară comenzi SQL ROLLBACK.. FORCE text.. <i>Sintaxă:</i> DBMS_TRANSACTION.ROLLBACK_FORCE (xid VARCHAR2)
ROLLBACK_SAVEPOINT	Procedură	Este similară comenzi SQL ROLLBACK... TO SAVEPOINT.. <i>Sintaxă:</i> DBMS_TRANSACTION.ROLLBACK_SAVEPOINT (svpt VARCHAR2)
SAVEPOINT	Procedură	Este similară comenzi SQL SAVEPOINT savepoint <i>Sintaxă:</i> DBMS_TRANSACTION.SAVEPOINT (savept VARCHAR2)
USE_ROLLBACK_SEGMENT	Procedură	Este similară comenzi SQL SET TRANSACTION USE ROLLBACK SEGMENT segment <i>Sintaxă:</i> DBMS_TRANSACTION.USE_ROLLBACK_SEGMENT (rb_name VARCHAR2)

DBMS_PIPE

Acest pachet este dedicat operațiilor de comunicare între două sau mai multe sesiuni conectate la aceeași instanță Oracle. Comunicarea între mai multe sesiuni conectate la aceeași instanță se face transmițând și primind mesaje prin canale de comunicație asincronă denumite *pipe*. Când se transmite un mesaj, un canal este denumit "writer" iar atunci când se primește un mesaj canalul este denumit "reader". În funcție de securitatea în

care se dorește a se realiza comunicarea, se pot utiliza canale publice sau canale private. Toate informațiile conținute în aceste canale se volatilizează în momentul în care instanța bazei de date este oprită.

Într-o sesiune pot fi create mai multe mesaje cu ajutorul procedurii `PACK_MESSAGE`. Aceste mesaje vor fi stocate în buffer-ul local până când este apelată procedura `SEND_MESSAGE`, moment în care buffer-ul va fi golit prin transmiterea tuturor mesajelor stocate în acesta. Dacă un proces dorește să primească mesaje, va apela procedura `RECEIVE_MESSAGE` ce va conține numele canalului din care se dorește să se facă citirea. Pentru a accesa fiecare element al mesajului, procesul va trebui să apeleze procedura `UNPACK_MESSAGE`.

În cazul unui canal public, orice utilizator ce are privilegiile necesare poate citi informațiile conținute în acesta. Dacă informația dintr-un canal este citită de un utilizator, buffer-ul este golit și prin urmare informația nu va mai fi disponibilă și pentru ceilalți utilizatori.

Functiile și procedurile din pachetul DBMS_PIPE

Nume	Tip	Descriere
<code>CREATE_PIPE</code>	Funcție	<p>Creează în mod explicit un canal de comunicație public sau privat. Dacă valoarea parametrului <code>private</code> este <code>False</code> va fi creat un canal de comunicație public. Canalele de comunicație publice sunt pot fi create și în mod implicit în momentul în care acesta este referit pentru prima dată și se va distruge în mod automat când nu va mai conține mesaje.</p> <p>Sintaxă:</p> <pre>DBMS_PIPE.CREATE_PIPE (pipename IN VARCHAR2, maxpipesize IN INTEGER DEFAULT 8192, private IN BOOLEAN DEFAULT TRUE) RETURN INTEGER</pre>
<code>PACK_MESSAGE</code>	Procedură	<p>Construiește un mesaj în buffer-ul local. Procedura este suprăîncărcabilă.</p> <p>Sintaxă:</p> <pre>DBMS_PIPE.PACK_MESSAGE(item IN VARCHAR2); DBMS_PIPE.PACK_MESSAGE(item IN NCHAR); DBMS_PIPE.PACK_MESSAGE(item IN NUMBER); DBMS_PIPE.PACK_MESSAGE(item IN DATE); DBMS_PIPE.PACK_MESSAGE_RAW(item IN RAW); DBMS_PIPE.PACK_MESSAGE_ROWID(item IN ROWID);</pre>
<code>SEND_MESSAGE</code>	Funcție	<p>Transmite mesajul din buffer-ul local pe un canal de comunicație specificat. Dacă nu se specifică nici un canal, în mod implicit este creat un canal de comunicație public.</p> <p>Sintaxă:</p> <pre>DBMS_PIPE.SEND_MESSAGE (pipename IN VARCHAR2, timeout IN INTEGER DEFAULT MAXWAIT maxpipesize IN INTEGER DEFAULT 8192) RETURN INTEGER</pre>

Nume	Tip	Descriere
RECEIVE_MESSAGE	Funcție	Copiază mesajul dintr-un anumit canal de comunicație în buffer-ul local. <i>Sintaxă:</i> DBMS_PIPE.RECEIVE_MESSAGE(pipename IN VARCHAR2, timeout IN INTEGER DEFAULT maxwait) RETURN INTEGER
NEXT_ITEM_TYPE	Funcție	Returnează tipul de date al următorului element dintr-un buffer. Valorile posibile returnate de această funcție sunt următoarele: - 0 (buffer-ul este gol); - 6 (NUMBER); - 9 (VARCHAR2); - 12 (DATE). <i>Sintaxă:</i> DBMS_PIPE.NEXT_ITEM_TYPE RETURN INTEGER
UNPACK_MESSAGE	Procedură	Accesează fiecare element în parte al mesajului. Procedura este suprîncărcabilă pentru mai multe tipuri de date. <i>Sintaxă:</i> DBMS_PIPE.UNPACK_MESSAGE(item OUT VARCHAR2); DBMS_PIPE.UNPACK_MESSAGE(item OUT NCHAR); DBMS_PIPE.UNPACK_MESSAGE(item OUT NUMBER); DBMS_PIPE.UNPACK_MESSAGE(item OUT DATE); DBMS_PIPE.UNPACK_MESSAGE_RAW(item OUT RAW); DBMS_PIPE.UNPACK_MESSAGE_ROWID(item OUT ROWID);
REMOVE_PIPE	Funcție	Distrugе un canal de comunicație. <i>Sintaxă:</i> DBMS_PIPE.REMOVE_PIPE(pipename IN VARCHAR2) RETURN INTEGER
PURGE	Procedură	Golește conținutul unui canal de comunicație. <i>Sintaxă:</i> DBMS_PIPE.PURGE(pipename IN VARCHAR2)
RESET_BUFFER	Procedură	Golește conținutul buffer-ului local. <i>Sintaxă:</i> DBMS_PIPE.RESET BUFFER
UNIQUE_SESSION_NAME	Funcție	Returnează numele sesiunii curente conectată la o anumită instanță a bazei de date. <i>Sintaxă:</i> DBMS_PIPE.UNIQUE_SESSION_NAME RETURN VARCHAR2

DBMS_ALERT

Pachetul DBMS_ALERT permite avertizarea unei sesiuni de apariția unui eveniment în baza de date prin așa numitele "avertismente". Sesiunea receptoare se înregistrează pentru a primi înștiințări de la un avertisment, iar avertismentul anunță sesiunea când apare evenimentul.

Funcțiile și procedurile din pachetul DBMS_ALERT

Nume	Tip	Descriere
REGISTER	Procedură	<p>Pernite unei sesiuni să își înregistreze interesul la apariția unui avertisment. Numele avertismentului este transmis prin parametrul name. Într-o sesiune se pot declara interese asupra unui număr nelimitat de avertismente. Dacă o sesiune nu mai este interesată de apariția unui avertisment trebuie apelată procedura REMOVE.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_ALERT.REGISTER(name IN VARCHAR2)</pre>
REMOVE	Procedură	<p>Permite unei sesiuni să nu mai fie interesată de apariția unui avertisment, reducând astfel procesările făcute pentru semnalizarea avertismentului.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_ALERT.REMOVE(name IN VARCHAR2)</pre>
SIGNAL	Procedură	<p>Señalează un avertisment prin transmiterea de mesaje către sesiunile interesate. Această procedură va avea efect numai dacă tranzacția care o include este permanentizată. Dacă procedura SIGNAL a fost apelată înainte de apelarea procedurii WAITONE sau WAITANY, mesajul curent se va transmite sesiunii interesate abia la următorul apel al procedurii SIGNAL.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_ALERT.SIGNAL(name IN VARCHAR2, message IN VARCHAR2)</pre>
WAITANY	Procedură	<p>Această procedură este utilizată pentru a aștepta un semnal de la oricare din avertismente.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_ALERT.WAITANY(name OUT VARCHAR2, message OUT VARCHAR2, status OUT INTEGER, timeout IN NUMBER DEFAULT MAXWAIT)</pre>
WAITONE	Procedură	<p>Această procedură este utilizată pentru a aștepta un semnal de la un anumit avertisment.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_ALERT.WAITONE(name OUT VARCHAR2, message OUT VARCHAR2, status OUT INTEGER, timeout IN NUMBER DEFAULT MAXWAIT)</pre>
SET_DEFAULTS	Procedură	<p>Stabilește intervalul de polling.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_ALERT.SET_DEFAULTS(polling_interval IN NUMBER)</pre>

DBMS_LOCK

În anumite cazuri este necesară blocarea unor resurse într-un anumit mod, identificarea blocării printr-un anumit identificator ce poate fi folosit într-o altă procedură, schimbarea modului blocării și eliberarea resurselor. Pentru a putea realiza acest lucru, pachetul DBMS_LOCK furnizează o serie de funcții și proceduri de gestionare a blocărilor.

Funcțiile și procedurile din pachetul DBMS_LOCK

Nume	Tip	Descriere
ALLOCATE_UNIQUE	Procedură	<p>Alochează un identificator unic unei blocări specificate.</p> <p><i>Sintaxă:</i></p> <pre>DBMS_LOCK.ALLOCATE_UNIQUE(lockname IN VARCHAR2, lockhandle OUT VARCHAR2, expiration_secs IN INTEGER DEFAULT 864000)</pre> <p><i>Unde:</i></p> <ul style="list-style-type: none"> - <i>lockname</i> este un identificator alfanumeric al blocării ales de utilizator; - <i>lockhandle</i> este identificatorul numeric generat de sistem; - <i>expiration_secs</i> reprezintă numărul de secunde de aşteptare pentru ca blocarea specificată să poată fi ştearsă din tabelul DBMS_LOCK ALLOCATED.
REQUEST	Funcție	<p>Face o cerere de blocare a unor resurse într-un anumit mod.</p> <p>Funcția este suprainscrisă, putând fi apelată atât prin identificatorul numeric cât și prin cel alfanumeric al blocării. Valorile returnate au următoarea semnificație:</p> <ul style="list-style-type: none"> - 0 (operatie terminată cu succes); - 1 (tiupul de acțiune al blocării a expirat); - 2 (interblocaj-deadlock); - 3 (eroare referitoare la parametri); - 4 (există deja o blocare a resurselor ce are același identificator); - 5 (identificator alfanumeric ilegal). <p><i>Sintaxă:</i></p> <pre>DBMS_LOCK.REQUEST(id IN INTEGER lockhandle IN VARCHAR2, lockmode IN INTEGER DEFAULT X_MODE, timeout IN INTEGER DEFAULT MAXWAIT, release_on_commit IN BOOLEAN DEFAULT FALSE, RETURN INTEGER)</pre> <p><i>Unde:</i></p> <ul style="list-style-type: none"> - <i>lockmode</i> reprezintă modul de blocare (1 - null, 2 - row share mode, 3 - row exclusive mode, 4 - share mode, 5 - share row exclusive mode, 6 - exclusive mode); - <i>timeout</i> este numărul de secunde de încercări continue pentru obținerea blocării; - <i>release_on_commit</i> este o variabilă booleană a cărui valoare TRUE reprezintă faptul că resursele vor fi deblocate la apariția următoarei comenzi COMMIT sau ROLLBACK.

Nume	Tip	Descriere
CONVERT	Funcție	Schimbă modul de blocare. Funcția este suprainsarcabilă, putând fi apelată atât prin identificatorul numeric cât și prin cel alfanumeric al blocării. Semnificația parametrilor timeout și lockmode precum și valorile returnate sunt similare funcției REQUEST. <i>Sintaxă:</i> DBMS_LOCK.CONVERT(id IN INTEGER lockhandle IN VARCHAR2, lockmode IN INTEGER, timeout IN NUMBER DEFAULT MAXWAIT) RETURN INTEGER;
RELEASE	Funcție	Deblochează în mod explicit resursele blocate cu ajutorul funcției REQUEST. Funcția este suprainsarcabilă, putând fi apelată atât prin identificatorul numeric cât și prin cel alfanumeric al blocării. Valorile returnate au următoarea semnificație: - 0 (operație terminată cu succes); - 3 (eroare referitoare la parametri); - 4 (există deja o blocare a resurselor ce are același identificator); - 5 (identificator alfanumeric ilegal). <i>Sintaxă:</i> DBMS_LOCK.RELEASE(id IN INTEGER) RETURN INTEGER; DBMS_LOCK.RELEASE(lockhandle IN VARCHAR2) RETURN INTEGER;
SLEEP	Procedură	Suspendă o sesiune pentru un anumit interval specificat în secunde. <i>Sintaxă:</i> DBMS_LOCK.SLEEP(seconds IN NUMBER)

UTL_FILE

Pachetul UTL_FILE permite programelor PL/SQL citirea și scrierea fișierelor text ale sistemului de operare. Această facilitate este disponibilă atât pe server cât și pe client, cu observația că în primul caz accesul la fișiere este restricționat la acele directoare explicit menționate într-o listă a directoarelor accesibile, stocată în fișierul de inițializare al parametrilor. Prin urmare, pentru citirea sau scrierea unui fișier, acesta trebuie mai întâi deschis cu ajutorul funcției FOPEN, funcție care va returna un parametru de identificare a fișierului. Dacă se dorește scrierea unui sir de caractere urmat de delimitatorul sfârșit de linie într-un fișier deschis, se va folosi procedura PUT_LINE. Pentru citirea unei linii dintr-un fișier deschis într-un buffer se va folosi procedura GET_LINE.

Pachetul UTL_FILE conține un tip de date privat care este folosit de funcțiile și procedurile conținute în pachet. Acest tip de date are următoarea definiție:

```
TYPE file_type IS RECORD (id BINARY_INTEGER)
```

O altă caracteristică specifică pachetului UTL_FILE, o reprezintă existența a șapte excepții:

- INVALID_PATH (Locația fișierului sau numele acestuia este invalid);
- INVALID_MODE (Modul de deschidere al fișierului este invalid);
- INVALID_FILEHANDLE (Parametrul de identificare a fișierului este invalid);
- INVALID_OPERATION (Fișierul nu poate fi deschis sau nu pot fi efectuate operații asupra lui);
- READ_ERROR (O eroare a sistemului de operare a survenit în timpul citirii fișierului);
- WRITE_ERROR (O eroare a sistemului de operare a survenit în timpul scrierii fișierului);
- INTERNAL_ERROR (Eroare nespecificată).

Functiile și procedurile din pachetul UTL_FILE:

Nume	Tip	Descriere
FOPEN	Funcție	<p>Deschide un fișier pentru scriere sau citire, iar în cazul în care acesta nu există va fi creat un nou fișier cu numele asociat. Funcția returnează un parametru de identificare a fișierului.</p> <p><i>Sintaxă:</i></p> <pre>FUNCTION FOPEN (location IN VARCHAR2, filename IN VARCHAR2, open_mode IN VARCHAR2) RETURN UTL_FILE.FILE_TYPE;</pre> <p><i>Unde:</i></p> <p>open_mode reprezintă modul în care poate fi deschis un fișier 'r'-citire (GET_LINE) , 'w'-scriere (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH) , 'a'-adăugare (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH).</p>
IS_OPEN	Funcție	<p>Testează parametrul de identificare al unui fișier pentru a determina dacă acesta a fost deschis sau nu.</p> <p><i>Sintaxă:</i></p> <pre>FUNCTION IS_OPEN(file_handle IN FILE_TYPE) RETURN BOOLEAN;</pre>
FCLOSE	Procedură	<p>Închide un fișier prin specificarea parametrului de identificare al acestuia.</p> <p><i>Sintaxă:</i></p> <pre>PROCEDURE FCLOSE (file_handle IN OUT FILE_TYPE)</pre>
FCLOSE_ALL	Procedură	<p>Închide toate fișierele deschise într-o sesiune.</p> <p><i>Sintaxă:</i></p> <pre>PROCEDURE FCLOSE_ALL</pre>
GET_LINE	Procedură	<p>Citește o linie de text de maxim 1022 caractere dintr-un fișier și o plasează într-un parametru buffer. Linia de text este citită până la întâlnirea delimitatorului sfârșit de linie sau sfârșit de fișier. Dacă linia nu începe în buffer se declanșează eroarea VALUE_ERROR, iar dacă s-a ajuns la sfârșitul fișierului se declanșează eroarea NO_DATA_FOUND.</p> <p><i>Sintaxă:</i></p> <pre>PROCEDURE GET_LINE(file_handle IN FILE_TYPE, buffer OUT VARCHAR2)</pre>

Nume	Tip	Descriere
PUT	Procedură	Scrie textul din parametrul buffer într-un fișier deschis pentru operația de scriere. Nu este inserat și delimitatorul sfârșit de linie. <i>Sintaxă:</i> PROCEDURE PUT(file_handle IN FILE_TYPE, buffer OUT VARCHAR2)
NEW_LINE	Procedură	Scrie unul sau mai mulți delimitatori sfârșit de linie într-un fișier deschis pentru operația de scriere. <i>Sintaxă:</i> PROCEDURE NEW_LINE(file_handle IN FILE_TYPE, lines IN NATURAL := 1)
PUT_LINE	Procedură	Scrie textul din parametrul buffer într-un fișier deschis pentru operația de scriere și inserează și delimitatorul sfârșit de linie. <i>Sintaxă:</i> PROCEDURE PUT_LINE(file_handle IN FILE_TYPE, buffer OUT VARCHAR2)
PUTF	Procedură	Această procedură este similară procedurii PUT dar conține în plus și elemente de formatare: - '/n' (inserează delimitatorul sfârșit de linie), - '%s' (prima apariție determină inserarea valorii conținute în argumentul arg1, a doua apariție a argumentului arg2 etc.). <i>Sintaxă:</i> PROCEDURE PUTF(file_handle IN FILE_TYPE, format IN VARCHAR2, [arg1 IN VARCHAR2, . . . arg5 IN VARCHAR2])
FFLUSH	Procedură	Scrie într-un fișier deschis toate datele aflate în curs de procesare. În mod normal datele care urmează să fie scrise într-un fișier sunt stocate în buffer. Această procedură forțează practic ca datele aflate în buffer să fie scrise într-un fișier deschis specificat. <i>Sintaxă:</i> PROCEDURE FFLUSH (file_handle IN FILE_TYPE)