# Subinterogari

Sunteţi patronul unei firme. În ultima perioadă unul dintre salariaţii firmei, pe nume Ionescu, s-a remarcat în mod deosebit prin activitatea sa. Aţi decis de aceea să îi măriţi salariul şi pentru a decide cu cât să-l măriţi doriţi să aflaţi care sunt persoanele cu salariu mai mare decât salariul lui Ionescu şi care sunt salariile câştigate de aceştia. Cum faceţi acest lucru? Mai întâi veţi determina salariul angajatului Ionescu:

```
SELECT salariul
FROM angajaţi
WHERE nume='Ionescu'
```

Să notăm cu **S** salariul returnat de această comandă. Acum putem afişa foarte simplu angajaţii cu salariu mai mare decât **S**:

```
SELECT nume, prenume
FROM angajaţi
WHERE salariul>S
```

Întrebarea care se pune acum este dacă nu există posibilitatea de a uni aceste două comenzi în una singură. Răspunsul este afirmativ. Vom înlocui în a doua comandă valoarea **S** cu comanda care a generat această valoare astfel:

```
SELECT nume, prenume
FROM angajaţi
WHERE salariul > ( SELECT salariul
                   FROM angajaţi
                  WHERE nume='Ionescu' )
```

Aşadar am inclus prima interogare în interiorul celei de a doua interogări. O astfel de interogare aflată în interiorul unei alte comenzi SQL se numeşte ***subinterogare***. Subinterogările sunt întotdeauna rulate înaintea comenzii în care sunt incluse, doar pe baza rezultatelor returnate de subinterogări putându-se obţine rezultatele interogării exterioare subinterogării.

Un proces similar cu modul de rulare al subinterogărilor este modul în care calculăm expresiile cu paranteze (figura II.5.1).

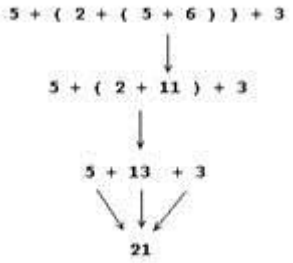



**Figura II.5.1.**

Subinterogările sunt în general folosite atunci când dorim să afişăm informaţii dintr-o tabelă pe baza unor informaţii pe care le preluăm din aceeaşi tabelă sau din alte tabele. De exemplu putem afişa angajaţii care lucrează în acelaşi departament cu angajatul **X** şi sunt mai tineri decât persoana **X.**

Există două tipuri de subinterogări:

- subinterogări simple care returnează o singură linie;
- subinterogări multiple care returnează mai multe linii şi/sau mai multe coloane.

Înainte de a prezenta fiecare din aceste tipuri de subinterogări trebuie să subliniem câteva restricţii de utilizare a subinterogărilor:

- o subinterogare va fi întotdeauna inclusă în paranteză
- subinterogarea nu poate conţine clauza **ORDER BY**.

## Subinterogări simple

Subinterogările simple, aşa cum am precizat, vor returna întotdeauna o singură valoare.

Ele pot să apară în clauza **WHERE** sau în clauza **HAVING** şi sunt folosite împreună cu operatorii **<**, **>**, **<=**, **>=**, **<>**, **=**.

Vom prezenta câteva exemple folosind următoarele tabele:

```
Persoane (Id, IdFirma, Nume, Localitate, DataN)
Firme (Id, Nume, Localitate)
```

Dorim să afişăm toate persoanele care lucrează la aceeaşi firmă la care lucrează şi Ionescu:

```
SELECT Nume FROM persoane
WHERE IdFirma = ( SELECT IdFirma
                  FROM angajati
                  WHERE nume = 'Ionescu')
```

Acelaşi rezultat l-am putea obţine cu ajutorul unui selfjoin astfel:

```
SELECT p.nume
FROM persoane p, persoane i
WHERE p.IdFirma = i.IdFirma AND
      i.nume = 'Ionescu'
```

însă folosirea subinterogărilor este mult mai uşoară şi mai naturală şi în general este mai rapidă.
Iată un exemplu de folosire a operatorului **<>** împreună cu o subinterogare:

```
SELECT nume
FROM persoane
WHERE localitatea <> (SELECT localitatea FROM persoane
                      WHERE nume='Ionescu')
```

Comanda afişează toate persoanele care nu locuiesc în aceeaşi localitate cu Ionescu.
Subinterogările pot folosi funcţii de grup ca în exemplul următor:

```
SELECT nume FROM persoane
WHERE DataN = (SELECT max(DataN) FROM persoane)
```

Această comandă va afişa cea mai tânără persoană din tabela **persoane**, data sa de naştere este cea mai mare, adică este cea mai recentă dată de naştere.
Similar putem utiliza subinterogările simple în clauza **HAVING**. Să vedem de exemplu cum putem afişa codul firmei cu cei mai mulţi angajaţi:

```
SELECT IdFirma FROM persoane
GROUP BY IdFirma
HAVING count(*) = ( SELECT max(count(*))
                    FROM persoane
                    GROUP BY IdFirma )
```

Subinterogarea determină mai întâi numărul maxim de persoane angajate la o firmă, iar apoi afişează Id-ul firmei care are numărul de angajaţi egal cu acest maxim.
***Atenţie!*** Am fi tentaţi să scriem o comandă de forma:

```
SELECT DISTINCT IdFirma
FROM persoane
WHERE count(*) = ( SELECT max(count(*))
                   FROM persoane
                   GROUP BY IdFirma )
```

dar am precizat în capitolul anterior funcţiile de grup **NU** pot să apară în clauza **WHERE.**
Subinterogările pot fi imbricate una în alta pe oricâte nivele. Numărul maxim de nivele de imbricare a interogărilor este teoretic nelimitat. Singura limitare care poate interveni este dată de dimensiunea bufferelor.
În exemplul următor, am construit o interogare care afişează numele firmei care are numărul maxim de angajaţi. Această interogare foloseşte interogarea din exemplul anterior pentru a determina Id-ul firmei cu număr maxim de angajaţi, iar apoi caută în tabela firme numele acestei firme.

```
SELECT nume
FROM firme
WHERE Id = (SELECT IdFirma
            FROM persoane
            GROUP BY IdFirma
            HAVING count(*) = ( SELECT max(count(*))
```

```
                                    FROM personae
                                    GROUP BY IdFirma
                                )
                )
```

Interesant este faptul că în cadrul unei subinterogări se poate face referire la tabelele din clauza **WHERE** a interogării părinte. Astfel dacă dorim să afişăm toate persoanele care lucrează în aceeaşi localitate în care şi locuiesc vom scrie astfel:

```
select nume
from persoane p
where localitate =  ( select localitate
                      from firme f
                      where p.idfirma=f.id
                    )
```

Am folosit subinterogarea pentru a afla localitatea în care se găseşte firma la care lucrează fiecare angajat în parte. Acest tip de subinterogări se numesc ***subinterogări corelate***.

### Subinterogări multiple

Am văzut cum putem utiliza subinterogările simple. Vom studia acum cum utilizăm subinterogările care returnează mai multe linii. Când o subinterogare returnează mai mult de o linie, nu mai este posibil să folosim operatorii de comparaţie **<**, **>**, **<=**, **>=**, **<>**, **=**, deoarece o valoare simplă nu poate fi comparată direct cu un set de valori. Va trebui să comparăm o valoare simplă cu fiecare valoare din setul de valori returnate de subinterogare. Pentru a realiza acest lucru vom folosi cuvintele cheie **ANY** şi **ALL** împreună cu operatorii de comparaţie, pentru a determina dacă o valoare este egală, mai mică sau mai mare decât orice valoare sau decât una din valorile din setul de date returnat de subinterogare. Pentru a exemplifica modul de folosire a subinterogărilor multiple vom utiliza tabela **jucători** cu următorul conţinut: **Tabelul II.5.1.** Tabela **Jucatori**

| ID | NUME | RATING | VARSTA | LOCALITATE |
|---|---|---|---|---|
| 18 | Ion | 3 | 30 | Sibiu |
| 11 | Iulian | 6 | 18 | Brasov |
| 22 | George | 3 | 29 | Bucuresti |
| 38 | Paul | 2 | 20 | Bucuresti |
| 13 | Andrei | 4 | 19 | Sibiu |
| 24 | Marian | 3 | 26 | Cluj-Napoca |
| 48 | Ilie | - | 35 | Sibiu |
| 21 | Alin | 2 | 36 | Brasov |
| 17 | Radu | 1 | 22 | Cluj-Napoca |
| 63 | Vasile | 7 | 41 | Iasi |

## Subinterogări multiple cu operatorul IN

Cum aflăm oare numele şi localitatea jucătorilor a căror rating este egal cu al unui jucător sub **21** de ani? Vom afla mai întâi care sunt ratingurile jucătorilor sub **21** de ani:

```
SELECT rating
FROM jucatori
WHERE varsta<21
```

Vom obţine trei valori ale ratingului şi anume **2**, **4** şi respectiv **6**: **Tabelul II.5.2.**

| RATING |
|---|
| 6 |
| 2 |
| 4 |

apoi vom afişa persoanele a căror rating este **2**, **3** sau **6**:

```
SELECT * FROM jucatori
WHERE rating IN ( 6, 2, 4 )
```

Rezultatul va fi cel din tabelul II.5.3. **Tabelul II.5.3.**

| ID | NUME | RATING | VARSTA | LOCALITATE |
|---|---|---|---|---|
| 11 | Iulian | 6 | 18 | Brasov |
| 21 | Alin | 2 | 36 | Brasov |
| 38 | Paul | 2 | 20 | Bucuresti |
| 13 | Andrei | 4 | 19 | Sibiu |

Aceste două comenzi se pot scrie împreună în una singură prin folosirea unei subinterogări multiple astfel:

```
SELECT * FROM jucatori
WHERE rating IN ( SELECT rating FROM jucatori
                  WHERE varsta<21 )
```

Ce se întâmplă dacă o subinterogare multiplă returnează o valoare nulă iar operatorul folosit este **IN**? De exemplu ce va afişa comanda:

```
SELECT * FROM jucatori
WHERE rating IN ( SELECT rating FROM jucatori
                  WHERE localitate='Sibiu')
```

Mai întâi subinterogarea va afişa ratingurile tuturor persoanelor din Sibiu: **Tabelul II.5.4.**

| RATING |
|---|
| 3 |
| 4 |
| - |

deci interogarea anterioară este echivalentă cu

```
SELECT * FROM jucatori WHERE rating IN ( 3, 4, NULL)
```

Sau

```
SELECT * FROM jucatori
WHERE rating=3 OR rating=4 OR rating=NULL
```

însă din comparaţia cu **NULL** nu rezultă nimic (**NULL** nu poate fi comparat decât cu operatorii **IS NULL** sau **IS NOT NULL** în rest nu vom obţine nici un rezultat), aşadar se vor afişa doar jucatorii cu ratingul egal cu **3**sau **4**:

| ID | NUME | RATING | VARSTA | LOCALITATE |
|---|---|---|---|---|
| 24 | Marian | 3 | 26 | Cluj-Napoca |
| 22 | George | 3 | 29 | Bucuresti |
| 18 | Ion | 3 | 30 | Sibiu |
| 13 | Andrei | 4 | 19 | Sibiu |

**Tabelul II.5.5.**

Dacă însă subinterogarea va returna doar o singură valoare nulă ca de exemplu comanda:

```
SELECT rating FROM jucatori
WHERE nume='Ilie'
```

**Tabelul II.5.6.**

| RATING |
|---|
| - |

atunci interogarea exterioară, neavând cu ce altă valoare să compare, nu va returna nici o linie:

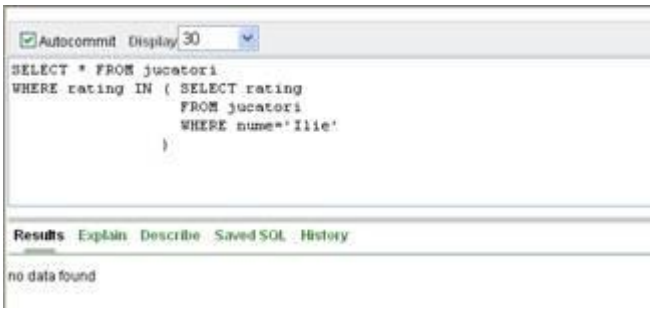



**Figura II.5.2.**

## Subinterogări multiple cu ALL

Fie următoarea comandă:

```
SELECT * FROM jucatori
WHERE rating > ALL ( SELECT rating FROM jucatori
                     WHERE varsta<21 )
```

Interogarea interioară returnează mulţimea valorilor ratingurilor tuturor persoanelor cu vârsta mai mică decât **21**, iar interogarea exterioară va verifica fiecare persoană din tabelă pentru a vedea dacă ratingul său este mai mare decât **fiecare** valoare returnată de către interogarea interioară.

Interogarea interioară va returna valorile **2**, **4**, **6** (tabelul II.5.2), deci comanda anterioară este echivalentă cu

```
SELECT * FROM jucatori
WHERE rating > ALL ( 2, 4, 6 )
```

sau

```
SELECT * FROM jucatori
WHERE rating>2 AND rating>4 AND rating>6
```

În concluzie am afişat toate persoanele al căror rating este mai mare decât ratingul tuturor persoanelor mai mici de 21 de ani.

Deci operatorul **>ALL** se poate interpreta ca **mai mare decât valoarea maximă** din mulţimea de valori returnată de către subinterogare. Similar operatorul **<ALL** se poate interpreta ca **mai mic decât valoarea minimă** din mulţimea valorilor returnate de către subinterogare

Dacă una dintre valorile returnate de către interogarea interioară este nulă atunci interogarea exterioară nu va afişa nici o linie dacă este folosită opţiunea **ALL**. Să vedem un exemplu. Dorim să afişăm toate persoanele cu rating mai mare decât ratingurile tuturor persoanelor din Sibiu:

```
select * from jucatori
where rating >ALL ( select rating
                    from jucatori
                    where localitate='Sibiu' )
```

Interogarea interioară returnează următoarele valorile **3**, **4** şi **NULL** (tabelul II.5.4.) şi interogarea exterioară se poate scrie echivalent:

```
select * from jucatori
where rating>3 AND rating>6 AND rating>NULL
```

Condiţia din clauza **where** are valoarea true doar dacă toate cele trei condiţii sunt adevărate. Însă expresia "**rating>NULL**" are valoarea **NULL**, adică nu este nici adevărată nici falsă. Aşadar condiţia din clauza **WHERE**nu este adevărată niciodată şi comanda nu afişează nici o linie.

### Subinterogări multiple cu ANY

Dacă folosirea opţiunii **ALL** se putea traduce printr-o condiţie compusă cu operatorul **AND**, în cazul opţiunii **ANY** se va putea traduce condiţia în altă condiţie care foloseşte operatorul **OR**.

Fie următoarea comandă:

```
select * from jucatori
where rating >ANY ( SELECT rating FROM jucatori
```

```
                    WHERE vârsta<21 )
```

Am văzut că interogarea interioară returnează valorile **2**, **4** şi **6** (tabelul II.5.2) Comanda exterioară va afişa toţi jucătorii care au un rating mai mare decât a oricărui jucător sub 21 de ani, sau altfel spus se afişează persoanele cu rating mai mare decât a **cel puţin** unei persoane cu vârsta sub **21** de ani.

**Tabelul II.5.7.**

| ID | NUME | RATING | VARSTA | LOCALITATE |
|---|---|---|---|---|
| 63 | Vasile | 7 | 41 | Iasi |
| 11 | Iulian | 6 | 18 | Brasov |
| 13 | Andrei | 4 | 19 | Sibiu |
| 18 | Ion | 3 | 30 | Sibiu |
| 24 | Marian | 3 | 26 | Cluj-Napoca |
| 22 | George | 3 | 29 | Bucuresti |

Putem spune că operatorul **>ANY** poate fi interpretat ca **mai mare decât valoarea minimă** din mulţimea de valori returnată de către subinterogare. Similar operatorul **<ANY** se poate interpreta ca **mai mic decât valoarea maximă** din mulţimea valorilor returnate către subinterogare

Dacă una din valorile returnate de către interogarea interioară este nulă, interogarea exterioară poate afişa totuşi ceva. De exemplu comanda

```
SELECT * FROM jucatori
WHERE rating >ANY ( SELECT rating FROM jucatori
                    WHERE localitate='Sibiu' )
```

va afişa **Tabelul II.5.8.**

| ID | NUME | RATING | VARSTA | LOCALITATE |
|---|---|---|---|---|
| 63 | Vasile | 7 | 41 | Iasi |
| 11 | Iulian | 6 | 18 | Brasov |
| 13 | Andrei | 4 | 19 | Sibiu |

Acest lucru se întâmplă deoarece comanda dată se poate scrie echivalent

```
SELECT * FROM jucatori
WHERE rating >ANY ( 3, 4, NULL )
```

deoarece subinterogarea returnează valorile **3**, **4** şi **NULL** (tabelul II.5.4.), şi această comandă se poate scrie şi

```
SELECT * FROM jucatori
WHERE rating>3 OR rating>4 OR rating>NULL
```

Condiţia din **WHERE** este adevărată dacă cel puţin una din cele trei condiţii este adevărată. Cum ultima condiţie, **rating>NULL**, nu va fi niciodată adevărată, este suficient ca ratingul jucătorului să fie mai mare decât **3**sau mai mare decât **4**, pentru ca el să fie afişat.

Dacă însă subinterogarea va returna o singură valoare nenulă, şi nimic altceva, atunci comanda exterioară nu va afişa nimic:

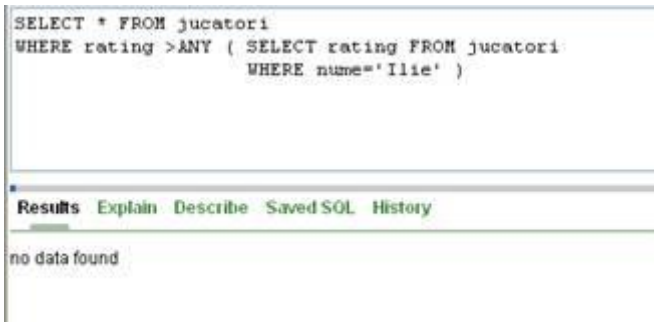



**Figura II.5.3.**

Modul în care se pot folosi opţiunile **ANY**. **IN** şi **ALL** se pot rezuma în figura II.5.4.

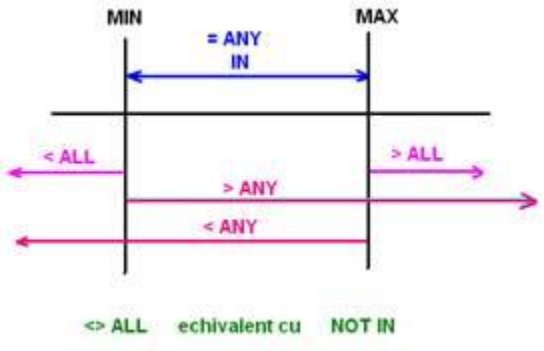



**Figura II.5.4.**

Echivalenţele ce se pot folosi cu aceste opţiuni sunt rezumate în tabelul următor:

**Tabelul II.5.9.**

| IN | =ANY |
|---|---|
| NOT IN | <> ALL |
| < ANY | < maxim |
| > ANY | > minim |
| < ALL | < minim |
| > ALL | > maxim |

## Subinterogări multiple cu EXISTS

Putem folosi operatorul **EXISTS** pentru a verifica dacă o subinterogare returnează vreo linie. De obicei se foloseşte acest operator împreună cu subinterogări corelate. De exemplu comanda următoare afişează toţi angajaţii care sunt managerii altor angajaţi:

```
SELECT employee_id, first_name, last_name
FROM employees a
WHERE EXISTS (SELECT employee_id FROM employees b
              WHERE b.manager_id=a.employee_id)
```

În subinterogare am determinat angajaţii coordonaţi de către un angajat afişat de către interogarea exterioară.
Evident această comandă o putem transcrie cu ajutorul operatorului IN astfel:

```
SELECT employee_id, first_name, last_name
FROM employees a
WHERE employee_id IN
      (SELECT employee_id FROM employees b
       WHERE a.employee_id=b.employee_id)
```

Este destul de uşor de dedus că folosirea operatorului **EXISTS** oferă performanţe mai mari întrucât **IN** compară fiecare valoare returnată de către interogarea exterioară cu fiecare valoare returnată de subinterogare, pe când operatorul **EXISTS** verifică doar existenţa a cel puţin unei linii returnată de subinterogare, fără a face nici o comparaţie.

## Subinterogări multiple în clauza FROM

O subinterogare multiplă poate fi folosită şi în clauza **FROM** a unei interogări ca în exemplul următor:

```
SELECT a.employee_id, first_name, last_name, nrang
FROM employees a, (SELECT manager_id, count(*) nrang
                   FROM employees GROUP BY manager_id
                   HAVING count(*)>0) b
WHERE a.employee_id=b.manager_id
```

care afişează id-ul, numele, prenumele şi numărul de subalterni ai tuturor managerilor (tabelul II.5.10).

**Tabelul II.5.10.**

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | NRANG |
|---|---|---|---|
| 100 | Steven | King | 5 |
| 101 | Neena | Kochhar | 2 |
| 102 | Lex | De Haan | 1 |
| 103 | Alexander | Hunold | 2 |
| 124 | Kevin | Mourgos | 4 |
| 149 | Eleni | Zlotkey | 3 |
| 201 | Michael | Hartstein | 1 |
| 205 | Shelley | Higgins | 1 |

După etapa de modelare a bazelor de date, primul pas în realizarea unei aplicaţii de baze de date constă în crearea obiectelor ce compun baza de date: tabele, indexi, vederi, sinonime etc.
Crearea tabelelor, presupune stabilirea numelor tabelelor şi a coloanelor ce le compun, stabilirea tipurilor de date pe care le au coloanele tabelei, dar şi declararea restricţiilor (constrângerilor) care asigură integritatea şi coerenţa informaţiilor din baza de date.

## Crearea tabelelor

Pentru crearea unei tabele se foloseşte comanda **CREATE TABLE**. Cea mai simplă formă a acestei comenzi, în care pentru moment nu se definesc valori implicite pentru coloane şi nu definim nici o restricţie este:

```
CREATE TABLE numetabel
      (    coloana1 tip1,
           coloana2 tip2,
           …
           coloanan tipn )
```

unde - ***numetabel*** este numele atribuit tabelului nou creat. Acest nume trebuie să respecte restricţiile privind definirea numelor despre care a discutat în capitolul II.1.

- ***coloana1, coloana2, …, coloanan*** sunt numele coloanelor din tabela nou creată
- ***tip1, tip2, …, tipn*** reprezintă tipul datelor ce vor fi reţinute în coloanele tabelei nou create şi dimensiunea (dacă este cazul). Principalele tipurile de date existente în Oracle au fost prezentate în capitolul I.3. Pe lângă numele tipului respectiv se precizează în paranteză lungimea tipului, respectiv numărul de caractere pentru un şir de caractere, sau numărul total de cifre şi numărul de cifre de după virgulă pentru valorile numerice.

De exemplu, pentru crearea tabelei corespunzătoare entităţii **Jucător** despre care am discutat în capitolul I.3 folosim comanda:

```
CREATE TABLE jucatori    (
  nr_legitimatie NUMBER(3),
  nume VARCHAR2(30),  prenume VARCHAR2(30),
  data_nasterii DATE, adresa VARCHAR2(50),
  telefon CHAR(13),   email VARCHAR2(30),
  cod_echipa NUMBER(3)  )
```

Deocamdată nu am definit cheia primară şi cheia străină.
Pentru crearea tabelei **ECHIPE** folosim comanda:

```
CREATE TABLE jucatori (
  cod NUMBER(3),
  nume VARCHAR2(30), localitate VARCHAR2(30),
  adresa_club VARCHAR2(50) )
```

Iată încă un exemplu:

```
CREATE TABLE elevi (
  id NUMBER(5),
       nume VARCHAR2(30), prenume VARCHAR2(30),
       bursier CHAR(1), media NUMBER(4,2) )
```

În acest exemplu, pentru tipul câmpului media s-au precizat două valori. Prima (**4**) reprezintă numărul total de cifre ale numărului, iar al doilea număr reprezintă numărul de cifre zecimale (**2**). Dacă sunt introduse mai mult de două zecimale se va face rotunjire la două zecimale. La partea întreagă pot exista două cifre. Dacă numărul introdus are mai mult de două cifre la partea întreagă se va semnala o eroare. De asemenea, am declarat un câmp **bursier**, care ne va ajuta să memorăm dacă un elev este sau nu bursier. Însă, în Oracle nu există tipul logic (sau boolean), motiv pentru care am optat pentru tipul **CHAR(1)**, pentru un elev bursier vom memora în acest câmp valoarea **'D'**, pentru ceilalţi elevi acest câmp rămânând necompletat.
O altă metodă de creare a unei tabele defineşte structura pe baza structurii unei tabele deja existente şi în acelaşi timp copiază datele din tabela deja existentă. Datele care se copiază din tabela deja existentă (liniile dar şi coloanele ce se copiază) se precizează prin clauza **AS** urmată de o subinterogare. De exemplu comanda următoare creează tabela **bursieri** pe baza tabelei **elevi** deja existentă:

```
CREATE TABLE bursieri
AS   SELECT id, nume, prenume FROM elevi
     WHERE bursier='D'
```

Se observă că nu sunt copiate coloanele **media** şi **bursier** din tabela **elevi**.

## Definirea valorilor implicite pentru coloane

Sintaxa comenzii **CREATE TABLE** prezentată anterior este una mult simplificată. În cadrul acestei comenzi putem utiliza clauza **DEFAULT** pentru a defini o valoare implicită pentru o coloană a tabelei. Această clauză precizează ce valoare va lua un atribut atunci când, la inserarea unei linii în tabelă, nu se specifică în mod explicit valoarea atributului respectiv. Clauza **DEFAULT** apare după precizarea tipului coloanei şi este urmată de constanta care defineşte valoarea implicită:

```
CREATE TABLE angajati
( nume varchar2(30), prenume varchar2(30),
  adresa varchar2(50) DEFAULT 'Necunoscuta',
  localitate varchar2(20) DEFAULT 'Bucuresti',
  data_ang date DEFAULT SYSDATE,
  salar NUMBER(5) DEFAULT 800 )
```

După cum se vede în exemplul anterior valoarea implicită poate fi o constantă dar poate fi de asemenea o expresie, sau o una din funcţiile speciale **SYSDATE** şi **USER** (care returnează numele utilizatorului curent) dar nu poate fi numele altei coloane sau al unei funcţii definite de utilizator.
Pentru o coloană pentru care nu s-a definit o valoare implicită, şi nu face parte din cheia primară sau dintr-o restricţie **NOT NULL** sau **UNIQUE** (despre care povestim mai târziu), sistemul va considera ca valoare implicită valoarea **NULL**.

## II.6.2. Definirea constrângerilor

După cum am precizat în prima parte a manualului, orice bază de date trebuie să stabilească regulile de integritate care să garanteze că datele introduse în baza de date sunt corecte şi valide.
Aceasta înseamnă că dacă există o regulă sau restricţie asupra unei entităţi, atunci datele introduse în baza de date respectă aceste restricţii.
Regulile de integritate se definesc la crearea tabelelor folosind **constrângerile**. Constrângerile pot fi clasificate în:

- constrângeri de domeniu, care definesc valorile pe care le poate lua un atribut (**NOT NULL**, **UNIQUE**, **CHECK**)
- constrângeri de integritate a tabelei, precizând cheia primară a acesteia
- constrângeri de integritate referenţială, care asigură coerenţa între cheile primare (sau unice) şi cheile străine corespunzătoare (**FOREIGN KEY**)

Pe de altă parte constrângerile se pot clasifica după nivelul la care sunt definite în:

- contrângeri la nivel de tabelă care pot acţiona asupra unei combinaţii de coloane
- constrângeri la nivel de coloană.

Constrângerile **NOT NULL** se pot defini doar la nivel de coloană.

Constrângerile **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY** şi **CHECK** pot fi definite atât la nivel de coloană cât şi la nivel de tabelă. Totuşi dacă aceste constrângeri implică mai multe coloane atunci trebuie să fie definite obligatoriu la nivel de tabelă.

Dacă o restricţie se defineşte la nivel de coloană se va folosi sintaxa:

```
nume_coloana tip_data tip_constr
```

sau

```
nume_coloana tip_data CONSTRAINT nume_constr tip_constr
```

La nivel de tabelă folosim sintaxa

```
tip_constr
```

sau

```
CONSTRAINT nume_constr tip_constr
```

Se observă că putem decide să dăm un nume explicit unei constrângeri, ceea ce uşurează referirea ulterioară la acea constrângere, sau putem să nu definim un nume explicit, caz în care sistemul va genera un nume implicit. Dacă se foloseşte cuvântul **CONSTRAINT**, atunci obligatoriu acesta va fi urmat de numele dat explicit constrângerii.

Vom prezenta în continuare modul de definire al fiecăreia dintre aceste constrângeri.

## Restricţia NOT NULL

După cum am văzut în capitolele anterioare, **NULL** este o valoare specială. Necompletarea în tabelă a unei celule conduce la completarea ei cu valoarea **NULL**, semnificând faptul că celula respectivă are de fapt o valoare nedefinită.

Într-un ERD, un atribut poate fi obligatoriu, lucru pe care îl marcam cu o steluţă în faţa atributului respectiv. În baza de date această condiţie se traduce prin faptul că valoarea coloanei respective trebuie obligatoriu completată, adică nu poate conţine valoarea **NULL**. Pentru definirea acestui tip de restricţii folosim restricţia **NOT NULL** pentru coloana respectivă, fie la crearea tabelei fie mai târziu la modificarea structurii acesteia.

La crearea tabelei, restricţia **NOT NULL** se precizează pentru fiecare coloană ce trebuie să respecte această restricţie, după precizarea tipului coloanei respective astfel:

```
CREATE TABLE angajati
( nume varchar2(30) NOT NULL,
  prenume varchar2(30),
  localitate varchar2(20) DEFAULT 'Iasi' NOT NULL
  ... )
```

Se observă că restricţia **NOT NULL** a putut fi folosită în combinaţie cu clauza **DEFAULT**.

## Restricţiile PRIMARY KEY şi UNIQUE

Cheia primară este o coloană sau o combinaţie de coloane care identifică în mod unic liniile unei tabele. Coloanele care fac parte din cheia primară vor fi automat de tip **NOT NULL** fără ca acest lucru să mai trebuiască precizat explicit. Când cheia primară este compusă dintr-o singură coloană, definirea acesteia se poate face la nivel de coloană ca în exemplul următor:

```
CREATE TABLE angajati
( cnp number(13) PRIMARY KEY
  nume varchar2(30),
       ... )
```

sau dacă dorim să atribuim un nume constrângerii putem scrie

```
CREATE TABLE angajati
( cnp number(13) CONSTRAINT angajati_pk PRIMARY KEY
  nume varchar2(30),
       ... )
```

Definirea cheii primare la nivel de tabelă se poate face şi atunci când cheia este compusă dintr-un singur câmp, dar este obligatorie atunci când este compusă din mai multe coloane.

De exemplu tabela **carti** are cheia primară compusă din combinaţia coloanelor **titlu**, **autor**, **data_aparitie**. Comanda de creare a acestei tabele se poate scrie:

```
CREATE TABLE carti
( titlu VARCHAR2(30),
  autor VARHAR2(30),
  data_ap DATE,
  format VARCHAR2(10),
  nr_pag NUMBER(3),
  CONSTRAINT carti_pk
      PRIMARY KEY (titlu, autor, data_ap)
    )
```

sau simplu

```
CREATE TABLE carti
( titlu VARCHAR2(30),
  autor VARCHAR2(30),
  data_ap DATE,
  format VARCHAR2(10),
  nr_pag NUMBER(3),
  PRIMARY KEY (titlu, autor, data_ap)
    )
```

Sintaxa generală de definire a cheii primare este deci

```
PRIMARY KEY (lista_coloane)
```

Similar se poate defini şi restricţia **UNIQUE** care precizează că valoare coloanei definită ca **UNIQUE**, sau combinaţia valorilor coloanelor ce definesc restricţia **UNIQUE** trebuie să fie unică pentru toate liniile din tabelă. Cu alte cuvinte, într-o coloană definită ca **UNIQUE** nu pot exista valori duplicate.

***Atenţie!*** Coloanele definite ca **UNIQUE** pot conţine valori **NULL**, iar acestea pot fi oricâte, adică valoare **NULL** este singura valoare ce poate fi duplicată într-o coloană **UNIQUE**.

Exemple:

```
CREATE TABLE elevi
( nr_matr NUMBER(5) PRIMARY KEY,
  cnp NUMBER(13) UNIQUE,
  nume VARCHAR2(30),
  prenume VARHAR2(30)
 )
```

sau

```
CREATE TABLE elevi
( nr_matr NUMBER(5) PRIMARY KEY,
  cnp NUMBER(13) CONSTRAINT cnp_uk UNIQUE,
  nume VARCHAR2(30),
  prenume VARHAR2(30)
 )
```

sau

```
CREATE TABLE carti
( ISBN varchar2(20) PRIMARY KEY,
  titlu VARCHAR2(30),
  autor VARCHAR2(30),
  data_ap DATE,
  format VARCHAR2(10),
  nr_pag NUMBER(3),
  UNIQUE (titlu, autor, data_ap)
     )
```

sau

```
CREATE TABLE carti
( ISBN varchar2(20) PRIMARY KEY,
  titlu VARCHAR2(30),
autor VARCHAR2(30),
  data_ap DATE,
  format VARCHAR2(10),
  nr_pag NUMBER(3),
  CONSTRAINT carti_uk UNIQUE (titlu, autor, data_ap)
        )
```

## Restricţia FOREIGN KEY

Restricţiile referenţiale sunt categoria de restricţii care creează cele mai mari probleme în gestiunea bazelor de date. Pentru exemplificarea modului de definire a chei străine vom relua un exemplu de ERD din capitolul I.3 şi anume cel din figura II.6.1.

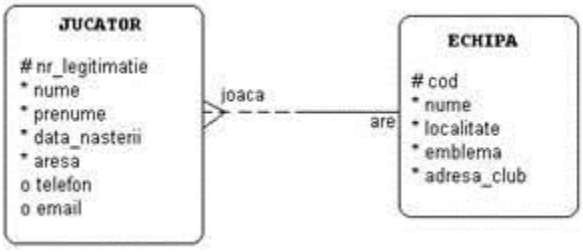



**Figura II.6.1**

Crearea tabelei **jucatori** corespunzătoare entităţii **JUCATOR** din acest ERD se va scrie:

```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3)
        REFERENCES echipe(cod)
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
```

```
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email VARCHAR2(30)
         )
```

sau

```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3) CONSTRAINT ech_fk
        REFERENCES echipe(cod),
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email VARCHAR2(30)
         )
```

sau la nivel de tabelă

```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3),
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email VARCHAR2(30),
  FOREIGN KEY (cod_echipa)
     REFERENCES echipe(cod)
)
```

sau

```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3),
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email VARCHAR2(30),
  CONSTRAINT test_fk FOREIGN KEY (cod_echipa)
        REFERENCES echipe(cod)
)
```

Sintaxa generală este aşadar la nivel de tabelă:

```
[CONSTRAINT nume_const] FOREIGN KEY (lista_coloane)
          REFERENCES tabela_parinte(lista_coloane_referite)
```

iar la nivel de coloană

```
[CONSTRAINT nume_const]
          REFERENCES tabela_parinte(lista_coloane_referite)
```

La definirea unei chei străine se poate utiliza o clauză suplimentară **ON DELETE CASCADE** care precizează că la ştergerea unei linii din tabela părinte se vor şterge automat din tabela copil acele linii care fac referire la linia ce se şterge din tabela părinte. De exemplu, prin folosirea acestei opţiuni, la ştergerea unei echipe se vor şterge automat toţi jucătorii de la acea echipă.

Această clauză se foloseşte astfel:

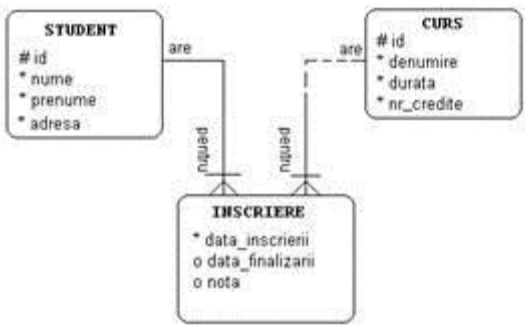
```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3) CONSTRAINT ech_fk
        REFERENCES echipe(cod) ON DELETE CASCADE,
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email VARCHAR2(30)
 )
```

O altă opţiune este **ON DELETE SET NULL** care face ca la ştergerea unui părinte, valorile cheii străine din liniile tabelei copil care fac referire la linia ştearsă vor fi setate pe **NULL**. De exemplu la ştergerea unei echipe, jucătorii acesteia vor deveni liberi de contract, deci codul echipei la care joaca va fi setat pe **NULL**:

```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3) CONSTRAINT ech_fk
        REFERENCES echipe(cod) ON DELETE SET NULL,
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email VARCHAR2(30)
 )
```

Implicit, fără precizarea uneia din aceste două opţiuni, Oracle va interzice ştergerea unei linii din tabela părinte atâta timp cât mai există măcar o linie în tabela copil care face referire la ea.

Să vedem acum cum creăm tabela **inscrieri** corespunzătoare entităţii **inscriere** din figura II.6.2. Observăm că în cheia primară intră şi coloanele ce fac parte din cheia străină.



```
CREATE TABLE inscriere (
   id_student NUMBER(5) NOT NULL
       REFERENCES studenti(id),
   id_curs NUMBER(5) NOT NULL REFERENCES cursuri(id),
   data_inscrierii DATE DEFAULT sysdate NOT NULL,
   data_finalizarii DATE,
   nota NUMBER (4,2),
   PRIMARY KEY (id_student, id_curs, data_inscrierii)
)
```

## Restricţia CHECK

Acest tip de constrângeri specifică o condiţie ce trebuie să fie îndeplinită de datele introduse în coloana (sau coloanele) asupra căreia acţionează. O astfel de constrângere poate limita valorile care pot fi introduse în cadrul unei coloane. Iată câteva exemple de reguli de validare pentru tabela elevi care pot fi implementate cu ajutorul constrângerilor de tip **CHECK**:

- numele şi prenumele unui elev trebuie să înceapă cu o majusculă restul literelor fiind litere mici
- nota unui elev nu poate fi mai mare de **10**
- câmpul bursier poate avea doar valorile '**D**' şi **NULL**
- numărul de absenţe nemotivate va fi cel mult egal cu numărul total de absenţe

Crearea tabelei elevi în această situaţie se poate scrie astfel:

```
CREATE TABLE elevi
( nr_matr NUMBER(5) PRIMARY KEY,
  cnp NUMBER(13) CONSTRAINT cnp_uk UNIQUE,
  nume VARCHAR2(30) NOT NULL
       CHECK nume=LTRIM(INITCAP(nume)),
  prenume VARHAR2(30) NOT NULL
       CHECK nume=LTRIM(INITCAP(nume))
  bursier CHAR(1) CHECK bursier='D',
  nota NUMBER(4,2)
    CONSTRAINT nota_ck CHECK nota<=10
  total_abs NUMBER(3),
  abs_nemotiv NUMBER(3),
  CHECK (abs_nemotiv<=total_abs)
)
```

## Modificarea structurii unei tabele

Modificarea structurii unui tabel se realizează cu ajutorul comenzii ALTER TABLE, permiţând adăugarea sau ştergerea unei coloane, modificarea definiţiei unei coloane, crearea unei noi constrângeri sau ştergerea unor constrângeri existente. Vom prezenta în continuare, pe scurt, fiecare dintre aceste operaţii.

### Adăugarea unei noi coloane

Se realizează folosind clauza **ADD** a comenzii **ALTER TABLE**. Sintaxa este similară cu cea a creării unei coloane în cadrul comenzii **CREATE TABLE**. De exemplu comenda următoare adaugă o colonă**nrgoluri** la tabela **jucatori**:

```
ALTER TABLE jucatori
ADD nrgoluri NUMBER(4)
```

Coloana nou creată va deveni ultima coloană a tabelei. Dacă tabela conţine deja date, coloana adăugată va fi completată cu **NULL** în toate liniile existente. De aceea nu vom putea adăuga o coloană cu restricţia **NOT NULL**la o tabelă ce conţine deja date.

Aşadar o comandă de forma:

```
ALTER TABLE test ADD ex NUMBER(3) NOT NULL
```

sau

```
ALTER TABLE test ADD ex NUMBER(3) PRIMARY KEY
```

Sunt permise doar dacă tabela nu conţine deja date.
Însă comanda

```
ALTER TABLE test ADD ex NUMBER(3) UNIQUE
```

poate fi folosită în orice moment, deoarece după cum am precizat o coloană **UNIQUE** poate conţine oricâte valori **NULL**.

## Ştergerea unei coloane

Se realizează folosind clauza **DROP COLUMN** a comenzii **ALTER TABLE**:

```
ALTER TABLE elevi DROP COLUMN bursier
```

Aşa cum este şi normal, ştergerea unei coloane duce automat şi la ştergerea restricţiilor definite pentru aceasta şi care nu implică şi alte coloane.

De exemplu dacă tabela **elevi** a fost creată cu ajutorul comenzii de la pagina 58, putem şterge fără probleme coloana **nume**:

```
ALTER TABLE elevi DROP COLUMN nume
```

chiar dacă avem definită o restricţie de tip **CHECK** la nivelul acestei coloane. De asemenea putem şterge coloana **nr_matr**, chiar dacă aceasta este cheia primară a tabelei:

```
ALTER TABLE elevi DROP COLUMN nr_matr
```

însă se va genera o eroare dacă încercăm să ştergem coloana **abs_nemotiv**, din cauza restricţiei definită la nivel de tabelă şi care implică coloanele **abs_nemotiv** şi **total_abs**.

O variantă ar fi să ştergem mai întâi toate restricţiile în care apare coloana ce dorim să o ştergem, sau să folosim clauza **CASCADE CONSTRAINTS** astfel:

```
ALTER TABLE elevi DROP COLUMN abs_nemotiv
CASCADE CONSTRAINTS
```

## Modificarea unei coloane

Poate fi făcută cu clauza **MODIFY** ca în exemplul următor:

```
ATLER TABLE elevi MODIFY prenume VARCHAR2(50)
```

Prin care am modificat tipul coloanei **prenume** de le **VARCHAR2(30)** la **VARCHAR2(50)**, deoarece am descoperit la un moment dat că există elevi al căror prenume (compus) are mai mult de 30 de caractere.
Mărirea numărului de caractere pentru o coloană de tip şir de caractere se poate face fără nici o problemă, însă micşorarea acestei dimensiuni se poate face doar dacă tabela este goală, sau coloana respectivă conţine doar valori **NULL**.
Tot cu opţiunea **MODIFY** se poate modifica, sau se poate stabili o valoare implicită, dacă nu exista deja una astfel:

```
ALTER TABLE elevi MODIFY bursier CHAR(1)
DEFAULT 'D'
```

însă această valoare implicită nu va afecta liniile deja existente în tabelă, ci doar liniile ce vor fi introduSe în continuare.

# Adăugarea unei constrângeri

Sintaxa comenzii pentru adăugarea unei constrângeri la nivel de tabelă este:

```
ALTER TABLE nume_tabela
ADD CONSTRAINT nume_constr definitie_constr
```

sau

```
ALTER TABLE nume_tabela
ADD definitie_constr
```

De exemplu comanda următoare defineşte cheia primară pentru o tabelă fictivă

```
ALTER TABLE tabelaexemplu
ADD PRIMARY KEY (coloana1)
```

Această comandă poate fi scrisă echivalent şi

```
ALTER TABLE tabelaexemplu
ADD CONSTRAINT tabelaexemplu_pk PRIMARY KEY (coloana1)
```

Singura constrângere ce nu poate fi adăugată în acest fel este NOT NULL, care poate fi adăugată doar prin modificarea coloanei restective folosind MODIFY:

```
ALTER TABLE tabelaexemplu
MODIFY coloana2 VARCHAR2(20) NOT NULL
```

## Ştergerea unei constrângeri

Ştergerea unei constângeri se face folosind opţiunea DROP CONSTRAINT astfel:

```
ALTER TABLE nume_tabela
DROP CONSTRAINT nume_constrangere
```

sau

```
ALTER TABLE nume_tabela DROP PRIMARY KEY
```

sau

```
ALTER TABLE nume_tabela
DROP UNIQUE(lista_coloane)
```

## Activarea/dezactivarea unei constrângeri

În unele situaţii, este necesară o dezactivare temporară şi apoi reactivarea unei constrângeri. Acest lucru se realizează astfel:

```
ALTER TABLE nume_tabela DISABLE/ENABLE
    CONSTRAINT nume_constrangere [CASCADE]
```

sau

```
ALTER TABLE nume_tabela DISABLE/ENABLE
    PRIMARY KEY [CASCADE]
```

sau

```
ALTER TABLE nume_tabela DISABLE/ENABLE
    UNIQUE (coloana1,coloana2,…) [CASCADE]
```

Clauza **CASCADE** precizează că şi constrângerile dependente sunt deasemenea afectate.

Până în acest moment am exemplificat diverse comenzi pe tabele care am presupus că există deja în baza de date şi sunt deja încărcate cu date. Însă atunci când veţi face propriile voastre aplicaţii va trebui să ştiţi să introduceţi singuri date în tabele, să modificaţi unele dintre aceste date, să ştergeţi la un moment dat o parte dintre ele etc.

### Adăugarea datelor în tabele

Pentru a adăuga linii într-o tabelă se utilizează comanda **INSERT**. Forma generală a acestei comenzi este următoarea:

```
INSERT INTO nume_tabela (lista_coloane)
VALUES (lista_valori);
```

unde ***nume_tabela*** este numele tabelei în care vom insera noua linie,

***lista_coloane*** precizează exact coloanele pe care dorim să le populăm. Această listă este opţională (ea poate lipsi).
***lista_valori*** specifică valorile pe care le va lua, pe rand, coloanele din lista de coloane.

Lista de coloane şi lista de valori trebuie să aibă acelaşi număr de elemente, şi în plus coloanele şi valorile din cele două liste trebuie să corespundă ca ordine şi tip.

Valorile specificate în listă (sau cele implicite) într-o comandă **INSERT**, trebuie să satisfacă toate constrângerile aplicabile coloanelor respective (ca de exemplu **PRIMARY KEY**, **CHECK**, **NOT NULL**).
Dacă la rularea unei comenzi **INSERT** este generată o eroare de sintaxă, sau a fost încălcată o constrângere, linia nu este adăugată la tabelă ci se va genera un mesaj de eroare.
Atunci când din lista de coloane este omisă o coloană, Oracle va completa valoarea acelei coloane cu **NULL**, cu excepţia situaţiei când a fost definită o valoare implicită pentru coloana respectivă. În acest caz, Oracle completează coloana cu valoarea implicită. Dacă omiteţi din lista de coloane o coloană care nu poate avea valoarea **NULL** (s-a definit o restricţie **NOT NULL** sau **PRIMARY KEY**), şi nu este definită o valoare implicită pentru acea coloană, se va genera o eroare.
Pentru a exemplifica modul de funcţionare a comenzii **INSERT** vom crea tabela **jucători**:

```
create table jucatori(
  id NUMBER(5) PRIMARY KEY,
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30),
  rating NUMBER(1) CHECK (rating between 1 and 5),
  varsta NUMBER(2),
  localitatea VARCHAR2(30) DEFAULT 'Timisoara',
  email VARCHAR2(30) UNIQUE
)
```

O comandă completă de inserare a unei linii în această tabelă se poate scrie:

```
insert into jucatori (id, nume, prenume, rating, varsta,
                      localitatea, email)
  values (18, 'Ionescu', NULL, 3, 30,
                      'Sibiu', 'user18@games.ro')
```

Fără a mai specifica coloanele putem scrie următoarea comandă, în care am ţinut cont de ordinea coloanelor în tabelă:

```
insert into jucatori values (11, 'Georgescu',
  'Valeriu', 1, 18, 'Bucuresti', 'user11@games.ro')
```

Comanda următoare are ca efect completarea coloanelor **id, nume, prenume** cu valorile specificate în lista de valori iar coloanele **rating, varsta, localitatea, email** cu valorile implicite pentru aceste coloane, adică **'Timisoara'** pentru **localitate** şi respectiv **NULL** pentru **rating, varsta, email**:

```
insert into jucatori (id, nume, prenume)
  values (22, 'Vasilescu', 'Anca')
```


**Figura II.7.1**

Deşi câmpul email are definită o restricţie **UNIQUE**, putem insera încă o valoare **NULL** în această coloană, doar valorile nenule trebuind să fie unice. Observaţi în comanda următoare cum s-a precizat că dorim setarea valorii implicite şi a valorii NULL pentru câmpurile **localitate, rating** şi **email**.

```
insert into jucatori (id, nume, prenume, rating, varsta,
                      localitatea, email)
  values (37, 'Enescu', 'Monica', NULL, 26,
                      DEFAULT, NULL)
```


**Figura II.7.2**

**Nu** putem însă iniţializa coloanele **`id`** sau **`nume`** cu o valoare implicită, această valoare implicită fiind în acest caz valoare **`NULL`**, care nu este permisă pentru cheia primară sau pentru o coloană având restricţia **`NOT NULL`**:

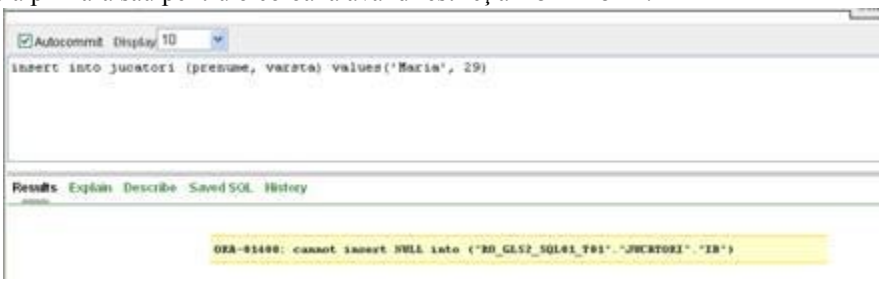



**Figura II.7.3.**

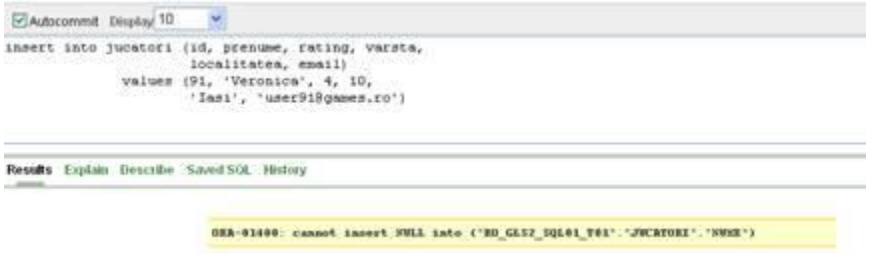



**Figura II.7.4.**

Pentru completarea unui câmp putem folosi o subinterogare ca în următorul exemplu:

```
insert into jucatori (id, nume, prenume)
      values ((select max(id)+1 from jucatori),
              'Plesca','Ovidiu')
```


**Figura II.7.5.**

În Oracle este permisă adăugarea mai multor linii simultan prin preluarea datelor din alte tabele, cu ajutorul unei subinterogări. Comanda următoare, de exemplu, preia toţi angajaţii din tabela **`employees`** care au **`job_id`**-ul egal cu '**`IT_PROG`**' şi îi inserează în tabela **`jucători`**:

```
insert into jucatori (id, nume)
select employee_id, last_name from employees
where job_id='IT_PROG'
```

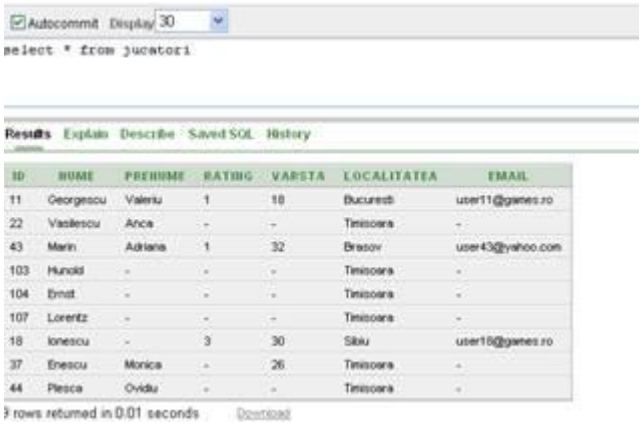



**Figura II.7.6.**

## Ştergerea datelor dintr-o tabelă

Ştergerea uneia sau mai multor linii dintr-o tabelă se face utilizând comanda **DELETE** a cărei sintaxă este:

```
DELETE FROM nume_tabela WHERE conditie
```

Liniile care se vor şterge sunt selectate folosind clauza **WHERE**:

```
DELETE FROM jucatori WHERE id>100
```

Ştergerea liniilor se poate face şi pe baza valorilor returnate de către o subinterogare:

```
DELETE FROM jucatori WHERE id <
(SELECT id FROM jucatori WHERE nume='Ionescu')
```

Dacă este omisă clauza **WHERE**, se vor şterge toate liniile din tabelă, însă structura tabelei rămâne (se şterge doar conţinutul tabelei, nu şi tabela propriu-zisă). Deci comanda:

```
DELETE FROM jucatori
```

şterge toate liniile din tabela **jucatori**. **Atenţie!** Aceste linii nu vor mai putea fi recuperate.

## . Modificarea datelor dintr-o tabelă

Modificarea uneia sau mai multor înregistrări (linii) dintr-o tabelă se realizează cu comanda **UPDATE** care are sintaxa:

```
UPDATE nume_tabela
SET   coloana1 = valoare1,
      coloana2 = valoare2,
      ...
WHERE conditie
```

ca în următorul exemplu:

```
update jucatori
SET prenume='Emilian' WHERE id=18
```

care modifică (completează) prenumele jucătorului cu id-ul **18**.

Modificarea valorilor unei linii se poate face pe baza valorilor returnate de către o subinterogare. Astfel, dacă dorim să îi atribuim jucătorului cu **id**-ul **44** acelaşi rating ca cel al jucătorului cu codul **18**, iar varsta să fie cu **5** mai mare decât vârta jucatorului cu codul **43**, vom scrie:

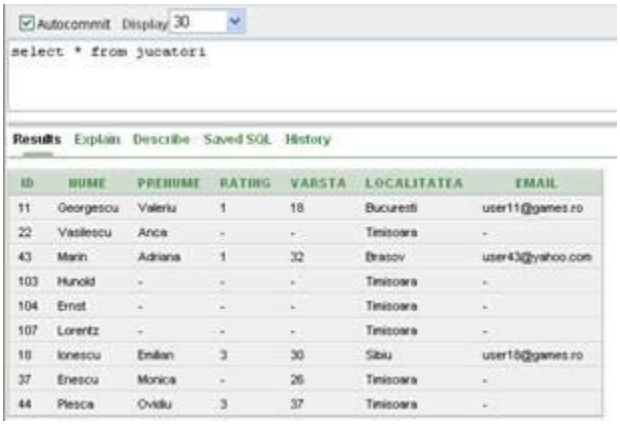
```
UPDATE jucatori
SET rating=(SELECT rating FROM jucatori WHERE id=18),
    varsta=(SELECT varsta+5 FROM jucatori WHERE id=43)
WHERE id=44
```

Dacă o subinterogare utilizată la actualizarea valorilor dintr-o coloană nu returnează nici o valoare, atunci câmpul respectiv va fi iniţializat cu **NULL**:

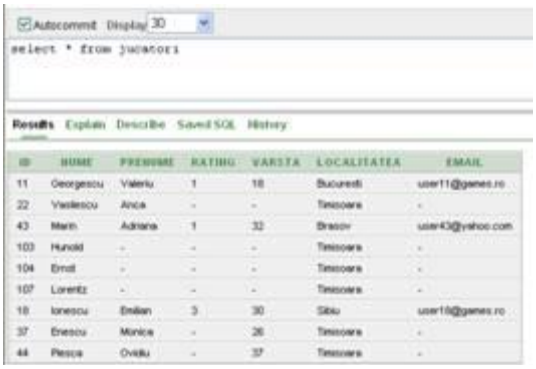
```
UPDATE jucatori
SET rating = (SELECT rating FROM jucatori WHERE id=200)
WHERE id=44
```

Înaintea rulării acestei comenzii conţinutul tabelei **jucatori** era cea din figura II.7.7, iar după rularea sa conţinutul este cel din figura II.7.8. Se observă că iniţial ratingul jucătorului **44** era **3**, iar după rularea comenzii acesta a devenit **NULL**.

**Figura II.7.7.**




| ID | NUME | PRENUME | RATING | VARSTA | LOCALITATEA | EMAIL |
|---|---|---|---|---|---|---|
| 11 | Georgescu | Valeriu | 1 | 18 | Bucuresti | user11@games.ro |
| 22 | Vasilescu | Anca | - | - | Timisoara | - |
| 43 | Marin | Adriana | 1 | 32 | Brasov | user43@yahoo.com |
| 103 | Hunold | - | - | - | Timisoara | - |
| 104 | Ernst | - | - | - | Timisoara | - |
| 107 | Lorentz | - | - | - | Timisoara | - |
| 18 | Ionescu | Emilian | 3 | 30 | Sibiu | user18@games.ro |
| 37 | Enescu | Monica | - | 26 | Timisoara | - |
| 44 | Plesca | Ovidiu | 3 | 37 | Timisoara | - |




| ID | NUME | PRENUME | RATING | VARSTA | LOCALITATEA | EMAIL |
|---|---|---|---|---|---|---|
| 11 | Georgescu | Valeriu | 1 | 18 | Bucuresti | user11@games.ro |
| 22 | Vasilescu | Anca | - | - | Timisoara | - |
| 43 | Marin | Adriana | 1 | 32 | Brasov | user43@yahoo.com |
| 103 | Hunold | - | - | - | Timisoara | - |
| 104 | Ernst | - | - | - | Timisoara | - |
| 107 | Lorentz | - | - | - | Timisoara | - |
| 18 | Ionescu | Emilian | 3 | 30 | Sibiu | user18@games.ro |
| 37 | Enescu | Monica | - | 26 | Timisoara | - |
| 44 | Plesca | Ovidiu | - | 37 | Timisoara | - |

**Figura II.7.7.**

Interesant este că o comandă de forma:

```
UPDATE jucatori
SET rating = (SELECT rating FROM jucatori WHERE id=18),
         varsta = (SELECT varsta+5 FROM jucatori WHERE id=18)
WHERE id=44
```

se poate scrie şi astfel:

```
UPDATE jucatori
SET (rating, varsta) =
          (SELECT rating, varsta FROM jucatori WHERE id=18)
WHERE id=44
```

# View-vederi

Uneori, din motive de securitate, aţi dori să nu permiteţi anumitor utilizatori să aibă acces nelimitat la o tabelă, ci doar la datele ce se găsesc în anumite coloane ale acestei tabele.

De exemplu, într-o firmă, contabila firmei nu va avea acces la coloanele ce se referă la proiectele în care sunt implicaţi la momentul actual fiecare angajat al firmei, însă va avea cu siguranţă acces la date privind salariul, tariful orar cu care este plătit fiecare angajat, numărul de ore lucrate etc. Pe de altă parte, bibliotecara de la biblioteca firmei, nu va avea acces la datele privind salarizarea personalului ci doar la datele personale ale angajaţilor (adresa, telefon, email etc).

Pentru a putea da acces parţial la o tabelă utilizatorilor vom folosi ceea ce numim vederi (sau views). O vedere este o tabelă virtuală, pentru care nu sunt memorate date propriu-zise ci doar definiţia vederii, care are rolul de filtrare a datelor.

Vederile sunt reprezentări logice ale tabelelor existente şi funcţionează ca nişte ferestre prin intermediul cărora pot fi vizualizate şi modificate datele din tabelele fizice (fig. II.8.1).

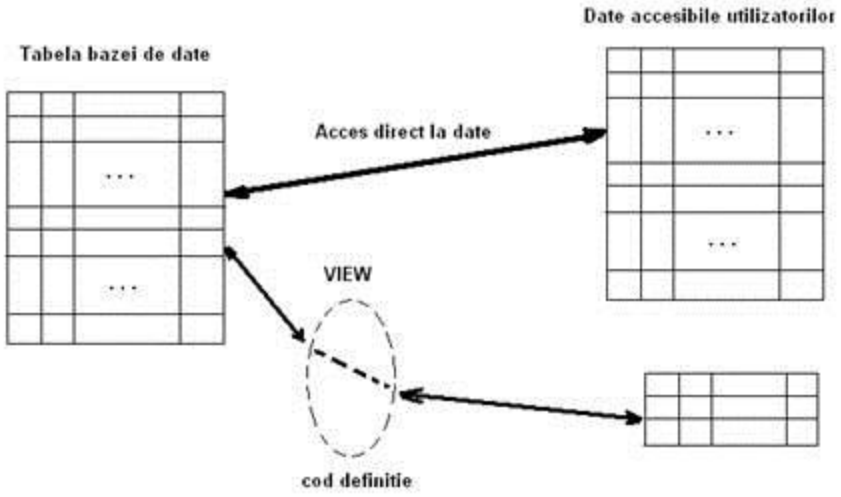



**Figura II.8.1.** Acces direct şi indirect (printr-o vedere) la o tabelă

Pe lângă faptul că oferă protecţie mărită a datelor, vederile mai au un mare avantaj: ele reduc în mod considerabil complexitatea interogărilor pe care utilizatorii trebuie să le scrie. O vedere poate fi construită folosind operaţii complexe de join, care rămân "ascunse" utlizatorului vederii respective, care va folosi interogări simple.

La crearea unei vederi se va folosi o subinterogare, oricât de complexă, însă aceasta **NU** poate folosi clauza **ORDER BY.**

## Crearea şi ştergerea vederilor

Sintaxa generală de a comenzii pentru crearea unei vederi este:

```
CREATE OR REPLACE VIEW nume_nedere
AS subinterogare
```

Opţiunea **OR REPLACE** poate lipsi, aceasta fiind utilă atunci când dorim să modificăm o vedere deja existentă.

De exemplu, următoarea comandă creează o vedere simplă pe baza tabelei **employees**:

```
CREATE OR REPLACE VIEW v1 AS
( SELECT first_name||' '||last_name as Nume,
         salary
  FROM employees WHERE department_id=20)
```

După cum am precizat, o vedere se poate construi folosind mai multe tabele, ca în exemplul următor:

```
CREATE OR REPLACE VIEW v2 AS
( SELECT a.nume ||' '|| a.prenume AS Angajat,
         b.nume ||' '|| b.prenume AS Sef,
              c.nume as Firma, d.nume as Job
  FROM angajat a, angajat b
  WHERE a.id_manager = b.id(+) and
        a.idFirm=c.idFirm(+) and a.idJob=d.idJob(+)
)
```

**Observaţie.** În subinterogarea care defineşte o vedere, toate expresiile (nu şi coloanele simple) trebuie să aibă asociate un alias pentru a putea fi ulterior referite în interogări.

Cum putem interoga aceste vederi? Ele pot fi folosite ca orice tabelă obişnuită, atât în interogări cât şi în operaţiile de actualizare (adăugare, modificare, ştergere), asupra acestora din urmă însă vom reveni în paragrafele următoare. Putem scrie de exemplu:

```
SELECT nume, salary FROM v1
WHERE nume like '%a%'
```

sau

```
SELECT angajat, sef, firma, job
FROM v2
```

O vedere poate fi şterasă cu comanda

```
DROP VIEW nume_vedere
```

**Atenţie!** Ştergerea unei vederi nu afectează în nici un fel datele din tabelele pe baza cărora s-a creat vederea. Toate modificările realizate asupra tabelelor prin intermediul vederii rămân valabile şi după ştergerea acesteia.

## . Actualizarea datelor prin intermediul vederilor

În acest paragraf vom folosi pentru exemplificare tabelele **jucatori** şi **echipe** create cu ajutorul următoarelor comenzi:

```
CREATE TABLE jucatori(
  id NUMBER(5) PRIMARY KEY,
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30),
  rating NUMBER(1) CHECK (rating BETWEEN 1 AND 5),
  varsta NUMBER(2),
  localitatea VARCHAR2(30) DEFAULT 'Timisoara',
  email VARCHAR2(30) UNIQUE
)
```

Să creăm acum următoarele vederi:

```
CREATE OR REPLACE VIEW v1_JucatoriTm AS
( SELECT id, nume, varsta, localitatea FROM jucatori
  WHERE localitatea = 'Timisoara' )
```

şi

```
CREATE OR REPLACE VIEW v2_Jucatori AS
( SELECT nume, prenume FROM jucatori
  WHERE rating IS NOT NULL)
```

Aşadar am creat o vedere pentru toţi jucătorii din Timişoara. Putem interoga simplu această vedere:

```
SELECT * FROM v1_JucatoriTm
```

rezultatul fiind cel din tabelul următor:

**Tabelul II.8.1.**

| ID | NUME | VARSTA | LOCALITATEA |
|---|---|---|---|
| 22 | Vasilescu | - | Timisoara |
| 103 | Hunold | - | Timisoara |
| 104 | Ernst | - | Timisoara |
| 107 | Lorentz | - | Timisoara |
| 37 | Enescu | 26 | Timisoara |
| 44 | Plesca | 37 | Timisoara |

iar comanda

```
SELECT * FROM v2_Jucatori
```

va afişa

**Tabelul II.8.2.**

| NUME | PRENUME |
|---|---|
| Georgescu | Valeriu |
| Marin | Adriana |
| Ionescu | Emilian |

Vom încerca acum, pe rând, să vedem cum funcţionează fiecare operaţie de actualizare a datelor.

O vedere poate fi creată folosind opţiunea **WITH READ OPTION**, prin intermediul unei astfel de vederi neputându-se efectua nici o operaţie de actualizare. Aceste vederi sunt folosite doar pentru vizualizarea datelor:

```
CREATE OR REPLACE VIEW v4_JucatoriTm AS
( SELECT id, nume, varsta, localitatea
  FROM jucatori
  WHERE localitatea = 'Timisoara' )
WITH READ ONLY
```


**Figura II.8.2.**

### Inserarea datelor prin intermediul vederilor

Încercăm să inserăm câte o înregistrare în tabela jucători prin intermediul celor două vederi create anterior:

```
insert into v1_JucatoriTm
values(210, 'Alexandrescu',41,'Iasi')
```

Comanda funcţionează perfect (fig. II.8.3), deşi jucătorul nou inserat nu respectă domeniul vederii **v1_JucatoriTm**, adică deşi putem vizualiza prin intermediul acestei vederi doar jucătorii din Timişoara, am reuşit totuşi să inserăm un jucător din altă localitate. Acest lucru ar putea crea probleme de securitate (am creat vederea tocmai pentru a restricţiona drepturile utilizatorilor).

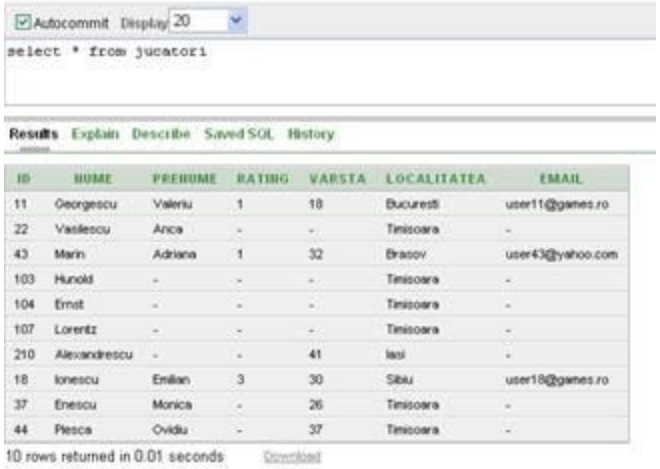



**Figura II.8.3.**

Această problemă poate fi rezolvată prin folosirea opţiunii **WITH CHECK OPTION** la crearea vederii. Vom crea o nouă vedere **v3_jucatoriTm** folosind această opţiune:

```
CREATE OR REPLACE VIEW v3_JucatoriTm AS
( SELECT id, nume, varsta, localitatea FROM jucatori
  WHERE localitatea = 'Timisoara' )
WITH CHECK OPTION
```

De această dată nu mai putem insera valori care sunt în afara domeniului vederii (fig. II.8.4).

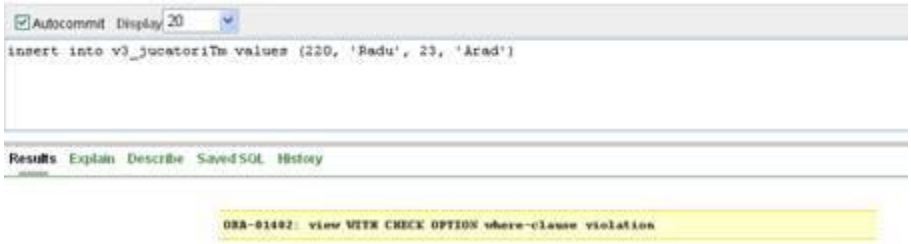



**Figura II.8.4.**

Prin intermediul vederii **v2_jucatori** nu vom putea insera linii în tabela **jucatori**, deoarece prin intermediul vederii nu avem acces la câmpul **id**, care fiind cheie primară nu poate fi iniţializată cu valoarea implicită **NULL** (fig. II.8.5)

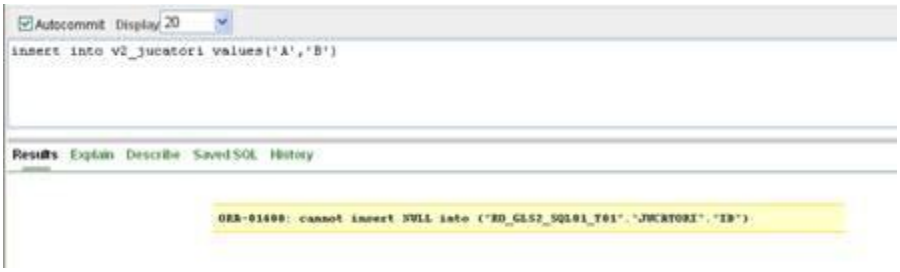



**Figura II.8.5.**

### Ştergerea datelor prin intermediul vederilor

La ştergerea unei înregistrări vom folosi comanda **DELETE** cu formatul deja cunoscut. Evident nu vom putea şterge din tabela decât liniile accesibile prin vederea respectivă. De aceea comanda:

```
DELETE FROM v1_jucatoriTm WHERE id=43
```

nu va genera nici o eroare, însă nu va şterge nici o linie întrucât jucătorul având **id**-ul **43** este din **Brasov**, deci nu avem acces la el prin intermediul vederii **v1_jucatoriTm**.

Similar, nu vom putea folosi în clauza **WHERE** a comenzii **DELETE** coloane care nu sunt vizibile din vederea respectivă. De exemplu comanda

```
DELETE FROM v2_jucatori WHERE id=43
```

va genera o eroare, deoarece câmpul **id** este inaccesibil vederii (fig. II.8.6.)

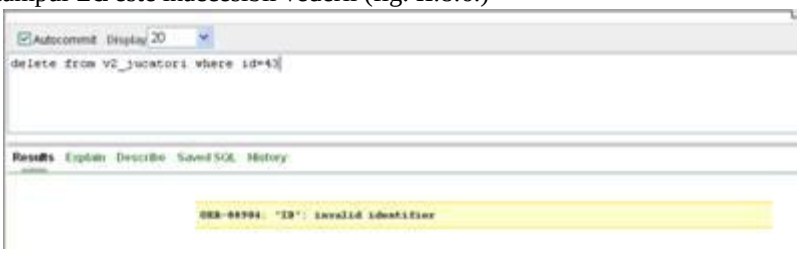



**Figura II.8.6.**

Comenzile

```
DELETE FROM v2_jucatori WHERE prenume='Emilian'
```

şi

```
DELETE FROM jucatori WHERE id=107
```

sunt perfect funcţionale.

### Modificarea datelor prin intermediul vederilor

Ca şi în cazul celorlaltor operaţii de actualizare vom putea modifica doar valorile liniilor şi coloanelor care sunt vizibile din vederea respectivă:

```
UPDATE v1_jucatoriTm
SET varsta=13
WHERE id=103
```

### Restricţii privind utilizarea vederilor

Operaţiile de actualizare a datelor prin intermediul vederilor NU pot fi realizate în următoarele condiţii:

- actualizarea datelor (**ştergere**, **modificare**, **inserare**) nu se poate efectua dacă subinterogarea cu care s-a creat vederea foloseşte:
  - funcţii de grup
  - clauza **GROUP BY**
  - clauza **DISTINCT**
  - pseudocoloanele **ROWNUM** sau **ROWID**
- nu se poate **modifica** un câmp calculat al unei vederi:
  De exemplu, dacă s-a creat vederea

```
CREATE VIEW v5 AS
```

```
( SELECT id, nume, nvl(rating,0) rating
  FROM jucatori)
```

vom putea actualiza câmpurile id şi nume:

```
UPDATE v5
SET nume='Eminescu'
WHERE id=37
```

dar nu putem modifica valoarea din câmpul **rating** (fig. II.8.7.)


**Figura II.8.7.**

- Nu se poate insera o linie într-o tabelă prin intermediul unei vederi decât dacă toate coloanele **NOT NULL** ale tabelei sunt prezente în vedere.

# Secvenţe

Imaginaţi-vă că trebuie să adăugaţi în baza de date a şcolii, datele persoanele ale noilor elevi veniţi în şcoala voastră în clasa a IX-a. Fiecărui elev trebuie să-i asociaţi un **id** unic în întreaga bază de date. Nu ştiţi însă exact care sunt id-urile elevilor deja existenţi în baza de date, pentru a şti care sunt id-urile "libere". Cum rezolvaţi oare această problemă?
O variantă ar fi ca la inserarea unui nou elev să determinaţi cel mai mare id existent în baza de date, şi să-i asociaţi elevului nou inserat un id cu o unitate mai mare decât cel mai mare id. Veţi scrie o comandă de forma:

```
INSERT INTO elevi (id, nume, prenume, …)
VALUES ( SELECT max(id)+1 FROM elevi,
    'Ionescu', 'Ioan', …)
```

O astfel de soluţie poate genera probleme în cazul accesului concurent la baza de date, când este posibil ca doi utilizatori diferiţi să încerce să insereze doi elevi cu acelaşi **id**.
Soluţia este folosirea secvenţelor. Secvenţele sunt obiecte ale bazei de date cate generează automat, în mod secvenţial, liste de numere. Acestea sunt utile când o tabelă foloseşte o cheie primară artificială, ale cărei valori dorim să le generăm automat.

## Crearea şi ştergerea secvenţelor

Sintaxa pentru crearea unei secvenţe este următoarea:

```
CREATE SEQUENCE nume_secventa
      START WITH n1
      INCREMENT BY n2
      MAXVALUE n3 | NOMAXVALUE
      MINVALUE n4 | NOMINVALUE
      CACHE n5 | NOCHACE
      CYCLE | NOCYCLE
```

Să explicăm pe rând care este rolul fiecărei opţiuni din această comandă:

- **START WITH *n1*** – precizează de la ce valoare va începe generarea valorilor. Această opţiune este utilă atunci când câmpul pentru care dorim să generăm valori folosind această secvenţă conţine deja valori. În acest caz, vom preciza în **n1** o valoare mai mare decât toate valorile deja existente în coloana respectivă. Dacă această opţiune nu este prezentă, se va începe implicit de la valoarea **1**.
- **INCREMENT BY n2** – precizează intervalul dintre două numere din secvenţă. Poate fi un număr întreg pozitiv sau negativ, dar nu poate fi zero. Dacă se precizează o valoare negativă, atunci valorile se vor genera în ordine descrescătoare, altfel se vor genera în ordine crescătoare. Dacă omiteţi această opţiune valoarea implicită a incrementului va fi **1**.

- **MAXVALUE *n3*** şi respectiv **MINVALUE *n4*** – aceste clause specifică cea mai mare, respectiv cea mai mică valoare returnată de către secvenţă. **n3** şi respectiv **n4** trebuie să fie numere întrege cu maxim **9** cifre.
- **NOMAXVALUE** – valoarea maximă generată va fi **2147483647** pentru o secvenţă cu increment pozitiv, respectiv **-1** pentru o secvenţă cu increment negativ.
- **NOMINVALUE** – valoarea maximă generată va fi **1** pentru o secvenţă cu increment pozitiv, respectiv **-2147483647** pentru o secvenţă cu increment negativ.
- **CACHE *n5*** – această opţiune este folosită din considerente de eficienţă. Cu această opţiune se vor genera simultan **n5** valori din secvenţă, şi numai atunci când acestea se vor epuiza se vor genera următoarele **n5** valori. În acest fel se vor face mai puţine modificări asupra bazei de date.
- **CYCLE | NOCYCLE** – dacă specificaţi opţiunea **CYCLE** atunci când secvenţa a ajuns la valoarea maximă (respectiv minimă pentru o secvenţă cu increment negativ), secvenţa va reîncepe să genereze valori începând cu**MINVALUE** (respectiv **MAXVALUE** pentru o secvenţă cu increment negativ). Evident, dacă utilizaţi opţiunea **CYCLE** nu există nici o garanţie privind unicitatea valorilor generate.

De exemplu, comanda:

```
CREATE SEQUENCE sec1
START WITH 1 INCREMENT BY 1
```

creează o secvenţă care va genera valori din **1** în **1**, începând cu **1**, adică va genera în ordine valorile **1**, **2**, **3**, etc.
Comanda

```
CREATE SEQUENCE sec2
START WITH 120 INCREMENT BY -3
```

creează o secvenţă care va genera valori descrescătoare din **3** în **3**, începând cu **120**, adică va genera în ordine valorile **120**, **117**, **114**, etc.
Ştergerea unei secvenţe se face simplu cu comanda **DROP SEQUENCE**.

## Utilizarea secvenţelor

Să vedem acum cum generăm efectiv valorile din secvenţă. Vom folosi două pseudocoloane speciale numite **NEXTVAL** şi respectiv **CURRVAL**. **NEXTVAL** generează următoarea valoare din secvenţă, în timp ce **CURVAL** este folosită pentru a afla care a fost valoarea care tocmai a fost generată.
Pentru exemplificare, creăm secvenţa

```
CREATE SEQUENCE sec3
START WITH 5 INCREMENT BY 3
```

şi tabela

```
CREATE TABLE test(nr number(3))
```

şi rulăm de **3** ori comanda:

```
INSERT INTO test values(sec3.NEXTVAL)
```

În acest fel conţinutul tabelei este **5**, **8**, **11** (fig. II.9.1)


**Figura II.9.1.**

Dacă rulăm acum comanda

```
SELECT sec3.currval FROM dual
```

se va afişa valaoarea **11**, adică exact ultima valoare generată de către secvenţă.

**Atenţie!** Pseudocoloanele **NEXTVAL** şi **CURRVAL** nu pot fi folosite în următoarele contexte:

- în clauza **SELECT** a unei vederi
- într-o comandă **SELECT** care foloseşte opţiunea **DISTINCT**
- într-o comandă **SELECT** care foloseşte clauzele **GROUP BY**, **HAVING**, sau **ORDER BY**.
- într-o subinterogare din cadrul unei comenzi **SELECT**, **DELETE** sau **UPDATE**.

- Într-o opţiune **DEFAULT** a comenzii **CREATE TABLE** sau **ALTER TABLE**.

## Modificarea secvenţelor

Comanda **ALTER SEQUENCE** care permite modificarea unei secvenţe are sintaxa similară cu cea a comenzii **CREATE SEQUENCE**:

```
CREATE SEQUENCE nume_secventa
       INCREMENT BY n2
       MAXVALUE n3 | NOMAXVALUE
       MINVALUE n4 | NOMINVALUE
       CACHE n5 | NOCHACE
       CYCLE | NOCYCLE
```

Modificarea unei secvenţe va afecta doar valorile ce se vor genera ulterior. La modificarea unei secvenţe trebuie să se ţină cont de câteva restricţii. De exemplu nu se poate stabili o valoare în clauza **MAXVALUE** care să fie mai mică decât ultima valoare care a fost deja generată de către secvenţă.
Să experimentăm puţin opţiunea de modificare a unei secvenţe. Să rulăm, pe rând, următoarele comenzi:

```
CREATE SEQUENCE sec4;
      CREATE TABLE test1 (n NUMBER(2), v NUMBER(2));
      INSERT INTO test1 values(1, sec4.NEXTVAL);
      INSERT INTO test1 values(2, sec4.NEXTVAL);
      INSERT INTO test1 values(3, sec4.NEXTVAL);
      INSERT INTO test1 values(4, sec4.CURRVAL);
      ALTER SEQUENCE sec4 INCREMENT BY -5 MINVALUE -200;
      INSERT INTO test1 values(5, sec4.NEXTVAL);
      INSERT INTO test1 values(6, sec4.NEXTVAL);
      INSERT INTO test1 values(7, sec4.NEXTVAL);
```

După aceste comenzi, conţinutul tabelei **test** va fi cel din tabelul următor:

**Tabelul II.9.1.**

| N | V |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 3 |
| 5 | -2 |
| 6 | -7 |
| 7 | -12 |

**Atenţie!** În Oracle Database Express Edition este posibil ca referirea la pseudocoloana **CURRVAL** să nu funcţioneze în mod corespunzător.

# II.9.2. Indecşi

Să presupunem că am creat o tabelă cu comanda:

```
CREATE TABLE test (    id integer,  content varchar )
```

şi am inserat o mulţime de linii în această tabelă. La un moment dat avem nevoie să rulăm o interogare de forma:

```
SELECT content FROM test WHERE id = 5;
```

Serverul bazei de date va trebui să parcurgă întreaga tabelă `test`, linie de linie, pentru a căuta toate liniile pentru care **`id`**-ul este 5. Dacă tabela conţine foarte multe linii şi doar puţine linii (poate chiar nici una) for fi returnate de către interogarea anterioară, această metodă este clar ineficientă.

Pentru a un acces direct şi rapid la liniile unei tabele, se vor folosi indecşii.

Indecşii unei tabele funcţionează similar cu indexul unei cărţi de specialitate. Într-un astfel de index, aflat de obicei la sfârşitul unei cărţi se găsesc principalii termeni şi concepte întâlnite în cartea respectivă, sortaţi alfabetic, indicându-se în dreptul fiecărui termen pagina sau paginile la care poate fi întâlnit termenul respectiv în carte. O persoană interesată de un anumit termen, nu va citi întreaga carte, ci va căuta în index pagina sau paginile corespunzătoare.

Există două tipuri de indecşi:

- ***indecşi unici*** – sunt generaţi automat pentru coloanele ce fac parte din cheia primară sau asupra cărora s-a definit o constrângere **`UNIQUE`**.
- ***indecşi non-unici*** – care sunt definiţi de către utilizator.

Crearea unui index se realizează cu comanda:

```
CREATE INDEX nume_index
ON nume_tabela(coloana1, coloana2, ... , coloanan)
```

De exemplu, dacă dorim să creştem viteza operaţiilor de căutare după coloana **`nume`** din tabela **`elevi`** vom crea următorul index:

```
CREATE INDEX elevi_idx1
ON carti(nume)
```

Într-un index putem include mai multe coloane ale unei tabele, ca în următorul exemplu:

```
CREATE INDEX elevi_idx2
ON carti(nume, prenume)
```

De asemenea pot fi incluse în index expresii, nu doar coloane ale unei tabele:

```
CREATE INDEX elevi_idx3
ON carti(UPPER(nume), UPPER(prenume))
```

Pentru a şterge un index folosiţi comanda **`DROP INDEX`**. Indecşii pot fi adăugaţi şi şterşi în orice moment fără a afecta tabela pe care o indexează în nici un fel, ei fiind fizic şi logic independenţi de tabela pe care o indexează. Totuşi, atunci când veţi şterge o tabelă, se vor şterge automat toţi indecşii definiţi pe tabela respectivă.

Odată creat un index, nu mai este necesară nici o intervenţie, acesta fiind actualizat automat după fiecare modificare efectuată asupra tabelei. De asemenea indexul va fi folosit automat în interogări care pot câştiga de pe urma folosirii sale. Un index definit pe o coloană care face parte dintr-o condiţie de join, poate duce la creşterea semnificativă a vitezei de executare a join-ului respectiv.

Aşadar, este indicată crearea unui index atunci când:

- coloana care se indexează conţine o plajă mare de valori
- coloana care se indexează conţine multe valori nule (valorile nule nu sunt incluse în index)
- una sau mai multe coloane sunt frecvent folosite împreună în clauza **`WHERE`** sau în condiţiile de join.
- Tabela este mare şi majoritatea interogărilor returnează un număr mic de linii din această tabelă (**`~5%`** din numărul total de înregistrări)

Când NU este indicat să creaţi un index? Atunci când:

- tabela este mică, în acest caz căutarea secvenţială este acceptabilă
- Coloanele nu sunt foarte des folosite în clauza **`WHERE`** a interogărilor
- majoritatea interogărilor returnează un număr mare de înregistrări (mai mult de **`5%`** din numărul total de înregistrări)
- se efectuează multe operaţii de inserare, ştergere sau modificare asupra tabelei. După fiecare astfel de operaţie sistemul trebuie să actualizeze indexul, operaţie consumatoare de timp
- Coloanele indexate sunt referite cel mai ades ca parte a unor expresii.

## II.9.3. Sinonime

După cum ştiţi sinonimul este un cuvânt cu exact acelaşi înţeles cu un alt cuvânt, adică un cuvânt care poate fi folosit în locul altui cuvânt
Similar în dialectul bazelor de date, administratorul unei baze de date poate defini nume echivalente pentru un obiect al bazei de date.
În principal vom defini un sinonim pentru un obiect al bazei de date pentru a simplifica referirea la acel obiect.
De exemplu pentru a interoga **`tabela1`** din schema unui alt utilizator, fie acesta **`user1`**, atunci vom referi această tabelă prin prefixarea numelui tabelei cu numele utilizatorului în a cărui schemă se găseşte tabela, adică vom scrie **`user1.tabela1`**. Dacă numele utilizatorului este însă **`RO_L2_SQL01_S12`** iar tabela se numeşte **`d_track_listings`**, va trebui să scriem **`RO_L2_SQL01_S12.d_track_listings`** pentru a ne referi la acea tabelă, ceea ce este destul de neplăcut. Pentru aceasta vom defini un sinonim mai scurt pentru tabela respectivă.
Sintaxa comenzii de creare a unui sinonim este:

```
CREATE [PUBLIC] SYNONYM nume_sinonim
FOR obiect
```

De exemplu:

```
CREATE SYNONYM ana_track
FOR RO_L2_SQL01_S12.d_track_listings
```

În continuare, vom putea folosi acest sinonim în locul numelui complet al tabelei.
Se pot defini sinonime pentru tabele, vederi, secvenţe, proceduri sau alte obiecte ale bazei de date.
Opţiunea **`PUBLIC`** este folosită de către administratorul bazei de date pentru a crea un sinonim accesibil tuturor utilizatorilor bazei de date. În mod implicit un sinonim este privat.
Ştergerea unui sinonim se face cu comanda **`DROP SYNONYM`**.

# Drepturi şi roluri

V-aţi întrebat vreodată ce ar însemna ca elevii dintr-o şcoală să aibă acces liber la catalog şi să poată face orice modificare doresc în catalog? Dar dacă orice utilizator conectat la internet ar avea acces nerestricţionat la baza de date a CIA, NASA, a unei bănci şi aşa mai departe?
Evident, în viaţa reală accesul în anumite locuri este restricţionat. Dacă faci parte dintr-un anumit grup restrâns de persoane, ca de exemplu angajaţii băncii, poţi avea acces în anumite zone restricţionate sau la anumite resurse la care alte persoane nu au acces.
Ca şi în lumea reală şi în cazul bazelor de date trebuie să putem defini o serie de drepturi pentru utilizatorii bazei de date, sau să restricţionăm accesul acestora la anumite obiecte ale bazei de date.
Controlul securităţii în Oracle se asigură prin specificarea: utilizatorilor bazei de date, schemelor, privilegiilor (drepturilor) şi rolurilor.

**Utilizatorii bazei de date şi schemele**
Fiecare bază de date are o listă de nume de utilizatori. Pentru a accesa baza de date un utilizator trebuie să folosească o aplicaţie şi să se conecteze cu un nume potrivit. Fiecărui nume de utilizator îi este asociată o parolă. Orice utilizator are un domeniu de securitate care determină privilegiile şi rolurile, cota de spaţiu pe disc alocat şi limitele de resurse ce le poate utiliza (timp CPU etc).
**Privilegiile**
Privilegiul este dreptul unui utilizator de a executa anumite instrucţiuni SQL. Privilegiile pot fi:

- **privilegii de sistem** - permit utilizatorilor să execute o gamă largă de instrucţiuni SQL, ce pot modifica datele sau structura bazei de date. Aceste privilegii se atribuie de obicei numai administratorilor bazei de date.
- **privilegii de obiecte** - permit utilizatorilor să execute anumite instrucţiuni SQL numai în cadrul schemei sale, şi nu asupra întregii baze de date.

Acordarea privilegiilor reprezintă modalitatea prin care acestea pot fi atribuite utilizatorilor. Există două căi de acordare ***explicit*** (privilegiile se atribuie în mod direct utilizatorilor) şi ***implicit*** (prin atribuirea acestora unor roluri, care la rândul lor sunt acordate utilizatorilor).
**Rolurile**
Rolurile sunt grupe de privilegii, care se atribuie utilizatorilor sau altor roluri. Rolurile permit:

- Reducerea activităţilor de atribuire a privilegiilor. Administratorul bazei de date în loc să atribuie fiecare privilegiu tuturor utilizatorilor va atribui aceste privilegii unui rol, care apoi va fi disponibil utilizatorilor;
- Manipularea dinamică a privilegiilor. Dacă se modifică un privilegiu de grup, acesta se va modifica în rolul grupului. Automat modificarea privilegiului se propagă la toţi utilizatorii din grup;
- Selectarea disponibilităţilor privilegiilor. Privilegiile pot fi grupate pe mai multe roluri, care la rândul lor pot fi activate sau dezactivate în mod selectiv;

- Proiectarea unor aplicaţii inteligente. Se pot activa sau dezactiva anumite roluri funcţie de utilizatorii care încearcă să utilizeze aplicaţia.

Un rol poate fi creat cu parolă pentru a preveni accesul neautorizat la o aplicaţie. Această tehnică permite utilizarea parolei la momentul pornirii aplicaţiei, apoi utilizatorii pot folosi aplicaţia fără să mai cunoască parola.
Pentru acordarea unui drept unui anumit utilizator **vasile** se va folosi comanda **GRANT.** De exemplu, pentru a se conecta la baza de date, un utilizator trebuie să aibă permisiunea de a crea o sesiune. Acest drept se alocă de către un utilizator privilegiat (utilizatorul system de exemplu) prin comanda

```
GRANT CREATE SESSION TO vasile
```

Acum utilizatorul **vasile** se poate conecta la baza de date.
Revocarea unui drept unui anumit utilizator se face folosind comanda **REVOKE** ca în exemplul următor:

```
REVOKE CREATE SESSION FROM vasile
```

# Drepturile de sistem

Un drept de system permite unui utilizator să efectueze anumite operaţii asupra bazei de date precumexecutarea comenzilor DDL. Cele mai uzuale drepturi system sunt prezentate în tabelul următor.

**Tabelul II.10.1.** Privilegii sistem

| Drept | Permite… |
|---|---|
| **CREATE SESSION** | conectarea la baza de date |
| **CREATE SEQUENCE** | crearea secvenţelor |
| **CREATE SYNONYM** | crearea sinonimelor |
| **CREATE TABLE** | crearea tabelelor |
| **CREATE ANY TABLE** | crearea unor tabele în orice schemă, nu doar în propria schemă |
| **DROP TABLE** | ştergerea tabelelor |
| **DROP ANY TABLE** | ştergerea unor tabele din orice schemă nu doar din schema proprie |
| **CREATE PROCEDURE** | crearea de proceduri memorate |
| **EXECUTE ANY PROCEDURE** | executarea unei proceduri în orice schemă |
| **CREATE USER** | crearea de utilizatori |
| **DROP USER** | ştergerea utilizatorilor |
| **CREATE VIEW** | crearea vederilor |

## Acordarea drepturilor de sistem

După cum am precizat acordarea drepturilor se face folosind comanda **GRANT**. În exemplul următor se acordă câteva drepturi sistem utilizatorului **ion**:

```
GRANT CREATE SESSION, CREATE USER, CREATE TABLE TO ion;
```

Se poate de asemenea folosi opţiunea **WITH ADMIN OPTION** care permite unui utilizator să aloce şi el drepturile primite cu această opţiune, mai departe, altor utilizatori:

```
GRANT EXECUTE ANY PROCEDURE TO ion WITH ADMIN OPTION;
```

Dreptul acordat utilizatorului **ion**, de a executa orice procedură poate fi acordată de acesta mai departe utilizatorului **george.** Pentru aceasta **ion** se va conecta la baza de date folosind comanda

```
CONNECT ion/test
```

unde **ion** este username-ul iar **test** este parola şi apoi va acorda dreptul lui **george**:

```
GRANT EXECUTE ANY PROCEDURE TO george;
```

Un drept se poate aloca tuturor utilizatorilor bazei de date folosin opţiunea **PUBLIC** ca în următorul exemplu:

```
CONNECT system/manager
GRANT EXECUTE ANY PROCEDURE TO PUBLIC;
```

În acest moment orice utilizator al bazei de date are dreptul de a executa o procedură în orice schemă.

## Drepturile la nivel de obiect

Un drept la nivel de obiect permite unui utilizator să execute anumite acţiuni asupra obiectelor bazei de date, ca de exemplu executarea anumitor comenzi **DML** pe tabelele bazei de date. De exemplu **GRANT INSERT ON adm.elevi** permite unui utilizator

să insereze linii noi în tabela **elevi** din schema **adm**. Cele mai des întâlnite drepturi la nivel de obiect sunt prezentate în tabelul următor:

**Tabelul II.10.2.** Privilegii la nivel de obiect

| Drept | Permite … |
|---|---|
| **SELECT** | Interogarea tabelei |
| **INSERT** | Inserarea de noi linii în tabelă |
| **UPDATE** | Modificarea valorilor din tabelă |
| **DELETE** | Ştergerea datelor din tabelă |
| **EXECUTE** | Executarea unor proceduri memorate |

## Acordarea drepturilor la nivel de obiect

Veţi utiliza de asemenea comanda **GRANT**. Exemplul următor acordă utilizatorului **ion** dreptul de **SELECT**, **INSERT**, şi **UPDATE** pe tabela elevi şi dreptul de **SELECT** asupra tabelei **angajaţi**:

```
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO ion;
GRANT SELECT ON profesori.angajati TO ion;
```

Următoarea comandă permite utilizatorului **ion** să modifice doar valorile din coloanele **prenume** şi **adresa**, din tabela **elevi**, utilizatorului **ion**:

```
GRANT UPDATE (prenume,adresa) ON adm.elevi TO ion;
```

Folosind opţiunea **WITH GRANT OPTION** veţi permite utilizatorului să acorde mai departe dreptul primit şi altor utilizatori:

```
GRANT SELECT ON adm.elevi TO ion WITH GRANT OPTION;
```

Dreptul de a interoga tabela adm.elevi poate fi acum acordat de către **ion** oricărui alt utilizator:

```
CONNECT ion/test
GRANT SELECT ON adm.elevi TO george;
```

**Revocarea drepturilor la nivel de obiect** se va face folosind comanda **REVOKE**. Următoarea comandă revocă dreptul de inserare de noi linii la tabela elevi utilizatorului **ion**:

```
REVOKE INSERT ON elevi FROM ion;
```

Comanda va fi rulată din contul **adm**.

***Observaţie!*** Dacă am acordat un drept unui utilizator **A** folosind opţiunea **WITH GRANT OPTION**, iar acest utilizatorul **A** a acordat şi el la rândul lui dreptul altor utilizatori **B**, **C** şi **D**, atunci când vom revoca dreptul utilizatorului **A**, va fi revocat automat acel drept şi tuturor utilizatorilor cărora utilizatorul A le-a acordat acel drept, respectiv utilizatorilor **B**, **C** şi **D**.

## Gestiunea rolurilor

După cum am precizat la începutul capitolului, putem crea un rol, prin intermediul căruia vom putea acorda drepturi unui grup de utilizatori având rolul respectiv, lucru mult mai uşor decât acordarea drepturilor fiecărui utilizator separat.
De exemplu, în loc să acordăm drepturi de **select**, **insert** şi **update** mai multor utilizatori:

```
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO ion;
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO vasile;
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO gheorghe;
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO maria;
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO alin;
```

E mai comod să creăm un rol, să acordăm drepturi pentru acest rol şi apoi să acordăm rolul respectiv celor cinci utilizatori. Vom scrie aşadar:

```
CREATE ROLE profi;
GRANT SELECT, INSERT, UPDATE ON adm.elevi TO profi;
GRANT profi TO ion, vasile, gheorghe, maria, alin;
```

În orice moment putem şterge un rol folosind comanda **DROP ROLE**. Aceasta va duce la revocarea tuturor drepturilor acordate utilizatorilor prin intermediul acestui rol.
Să dăm un exemplu mai complex de acordare a drepturilor şi privilegiilor. Să presupunem că rulăm pe rând următoarele comenzi:

```
CONNECT hr/test;
CREATE ROLE r1;
CREATE ROLE r2;
GRANT SELECT, INSERT, DELETE ON hr.elevi TO r1
            WITH GRANT OPTION;
GRANT DELETE, UPDATE ON hr.elevi TO r2
```

```
            WITH GRANT OPTION;
GRANT r1 TO user1
GRANT r2 TO user2
GRANT CREATE VIEW TO user3 WITH GRANT OPTION
GRANT DELETE ON hr.elevi TO user3
GRANT UPDATE ON hr.elevi TO user4

CONNECT user2/pas2
GRANT DELETE ON hr.elevi TO user4
GRANT UPDATE ON hr.elevi TO user4
```

În acest moment utilizatorii au următoarele drepturi (figura II.10.1.):

**Tabelul II.10.3.**

| UTILIZATOR | DREPT |
|---|---|
| user1 | SELECT, INSERT, DELETE ON hr.elevi |
| user2 | DELETE, UPDATE ON hr.elevi |
| user3 | DELETE ON hr.elevi<br>CREATE VIEW |
| user4 | DELETE, UPDATE ON hr.elevi |

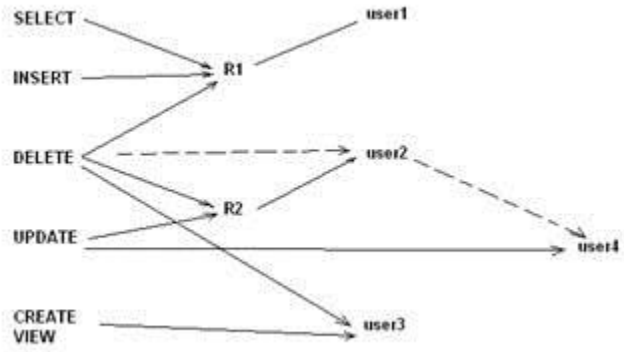



**Figura II.10.1.** Schema de acordare a drepturilor

Dacă acum ştergem rolul **r2**:

```
DROP ROLE r2
```

utilizatorul **user2** va pierde dreptul de **DELETE** şi **UPDATE** asupra tabelei **hr.elevi**, şi prin intermediul său va pierde dreptul de **DELETE** şi utilizatorul **user4**, care a primit acest drept de la **user2**. Deşi user4 a primit de la user2 şi dreptul de**UPDATE**, el nu va pierde acest drept deoarece a primit acest drept şi direct de la utilizatorul **SYSTEM**. Aşadar după ştergerea rolului **r2**, drepturile utilizatorilor sunt următoarele:

**Tabelul II.10.4.**

| UTILIZATOR | DREPT |
|---|---|
| user1 | SELECT, INSERT, DELETE ON hr.elevi |
| user2 | - |
| user3 | DELETE ON hr.elevi<br>CREATE VIEW |
| user4 | UPDATE ON hr.elevi |

## Gestiunea tranzacţiilor

O tranzacţie este un grup de comenzi SQL care sunt văzute ca o singură unitate. Imaginaţi-vă o tranzacţie ca un grup de comenzi SQL care nu pot fi separate, şi al căror efect este în întregime salvat în baza de date, fie este în întregime anulat. Să ne gândim de exemplu la efectuarea unui transfer bancar dintr-un cont în alt cont. O comandă **UPDATE** va efectua operaţia de scădere a sumei de bani

tranzacţionată dintr-un cont, iar o altă comandă **UPDATE** va adăuga suma respectivă la cel de al doilea cont. Dacă ambele operaţii decurg normal fără probleme, atunci ele vor deveni **ambele** permanente. Dacă una dintre aceste două comenzi eşuează (de exemplu nu poate fi contactată banca în care se depun banii) atunci ambele comenzi vor fi anulate. E normal să renunţăm la scăderea sumei de bani dintr-un cont, dacă aceştia nu pot fi depuşi în celălalt cont, în caz contrar ar duce la pierderea banilor respectivi.
În general o tranzacţie poate fi formată din mai multe comenzi **INSERT**, **UPDATE**, şi **DELETE.**
Pentru a face permanentă o tranzacţie folosiţi comanda **COMMIT**. Dacă doriţi să renunţaţi la modificările efectuate în cadrul unei tranzacţii trebuie să rulaţi o comandă **ROLLBACK.**
Comanda **ROLLBACK** fără nici un parametru, încheie tranzacţia curentă şi renunţă la toate modificările făcute în cadrul acestei tranzacţii. Aveţi însă posibilitatea definirii în cadrul unei tranzacţii a unui aşa numit punct de întoarcere, sau punct de salvare. Odată definit un astfel de punct de salvare, veţi putea renunţa doar la o parte din modificările făcute în cadrul tranzacţiei curente.
Definirea unui punct de revenire se face cu comanda **SAVEPOINT** având sintaxa:

```
SAVEPOINT nume_punct_de_revenire
```

Revenirea la un punct de revenire se face cu comanda ROLLBACK astfel:

```
ROLLBACK TO nume_punct_de_revenire
```

Definirea punctelor de revenire este utilă în cazul unor tranzacţii mari, când în cazul în care faceţi o greşeală nu trebuie să renunţaţi la toate operaţiile din cadrul tranzacţiei ci doar la o parte dintre acestea.
O tranzacţie fiind un grup de comenzi SQL tratate ca un întreg, trebuie să stabilim unde începe o tranzacţie şi unde se termină aceasta.
O tranzacţie începe la întâlnirea unuia dintre următoarele evenimente:

- În momentul conectării la baza de date şi la începerea rulării primei comenzi **DML** (**INSERT**, **UPDATE**, **DELETE**).
- La terminarea unei tranzacţii anterioare şi rularea următoarei comenzi **DML.**

O tranzacţie se termină când apare unul dintre următoarele evenimente:

- La executarea unei comenzi **COMMIT** sau **ROLLBACK** (fără nici un parametru, întrucât **ROLLBACK TO ...** nu termină tranzacţia ci doar revine la un punct precizat din cadrul tranzacţiei curente)
- La executarea unei comenzi **DDL** (**CREATE**, **ALTER**, **DROP**, **RENAME**, **TRUNCATE**), caz în care este executată automat comanda **COMMIT.**
- La executarea unei comenzi **DCL** (**GRANT** sau **REVOKE**) caz în care este executată automat comanda **COMMIT.**
- Vă deconectaţi de la baza de date. Dacă ieşiţi normal din **SQL*Plus** cu comanda **Exit**, sau daţi **Logout** din Oracle Database Express Edition atunci are loc un **COMMIT** automat. Dacă ieşirea se face anormal, de exemplu în cazul unei pene de curent, atunci se execută în mod automat o comandă **ROLLBACK.**
- Executaţi o comandă **DML** care eşuează, caz în care are loc un **ROLLBACK** automat pentru acea singură comandă.

Să experimentăm acum modul de folosire a tranzacţiilor.
***Atenţie*!** În Oracle Database Express Edition toate comenzile sunt autocommit, şi nu vor fi recunoscute comenzile **COMMIT**, **ROLLBACK** sau **SAVEPOINT**. Pentru acest exerciţiu puteţi rula comenzile SQL în linia de comandă. Pentru aceasta alegeţi din meniul **Start**, **Programs**, **Oracle Database 10g Express Edition** opţiunea **Run SQL Command Line**. Se va deschide o fereastră în care vă veţi conecta la baza de date folosind comanda

```
CONECT
```

Introduceţi username-ul (**hr**) şi parola şi în acest moment puteţi rula orice comandă SQL.
Pentru a experimenta folosirea tranzacţiilor vom crea următoarea tabelă:

```
create table savepoint_test ( n number )
```

Inserăm acum câteva linii în această tabelă:

```
insert into savepoint_test values (1);
insert into savepoint_test values (2);
insert into savepoint_test values (3);
```

Definim acum un punct de salvare:

```
savepoint sp1;
```

şi mai inserăm câteva linii în tabelă:

```
insert into savepoint_test values (10);
insert into savepoint_test values (20);
insert into savepoint_test values (30);
```

Definim un nou punct de salvare:

```
savepoint sp2;
```

şi inserăm în final încă trei linii:

```
insert into savepoint_test values (100);
insert into savepoint_test values (200);
insert into savepoint_test values (300);
```

Verificăm acum dacă datele au fost inserate în tabelă:

```
select * from savepoint_test;
```

şi vedem că toate datele au fost inserate:


**Figura II.11.1.**

Revenim acum la punctul de revenire sp2

```
ROLLBACK TO sp2
```

şi verificăm conţinutul tabelei:

```
select * from savepoint_test;
```

Observaţi că ultimele linii inserate după definirea punctului de salvare sp2 au fost şterse din tabelă (figura II.11.2.).


**Figura II.11.2.**

Inserăm alte trei linii:

```
insert into savepoint_test values (111);
insert into savepoint_test values (222);
insert into savepoint_test values (333);
```

testăm conţinutul tabelei:

```
select * from savepoint_test;
```


**Figura II.11.3.**

Revenim la punctul de salvare **sp2**:

```
ROLLBACK TO sp2
```

şi verificăm conţinutul tabelei:

```
select * from savepoint_test;
```

Evident ultimele trei linii nu se mai găsesc în tabelă conţinutul tabelei fiind acelaşi cu cel din figura II.11.2. Dacă revenim acum la punctul de salvare sp1, în tabelă nu mai rămân decât trei linii (figura II.11.4.)

```
ROLLBACK TO sp1
select * from savepoint_test;
```

```
SQL> select * from savepoint_test;

         N
----------
         1
         2
         3
        10
        20
        30

6 rows selected.

SQL> rollback to sp1;

Rollback complete.

SQL> select * from savepoint_test;

         N
----------
         1
         2
         3

SQL>
```

**Figura II.11.4.**

Schematic tranzacţia anterioară arată ca în figura II.11.5.

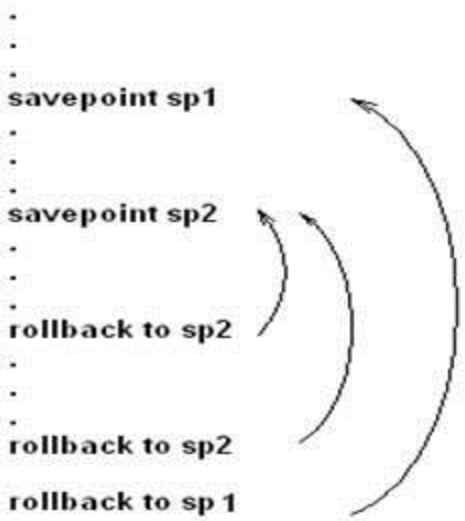



**Figura II.11.5.**