1. Date, informaţii, cunoştinţe

Auzim adesea vorbindu-se despre "Era informaţiilor" sau "societate informaţională" sau "tehnologia informaţiei" însă de multe ori cuvântul "informaţie" este folosit fără a înţelege clar sensul acestui cuvânt, diferenţa dintre date, informaţii, cunoştinţe.

În general, conţinutul gândirii umane operează cu următoarele concepte:

1. **Date** – constau în material brut, fapte, simboluri, numere, cuvinte, poze fără un înţeles de sine stătător, neintegrate într-un context, fără relaţii cu alte date sau obiecte. Ele se pot obţine în urma unor experimente, sondaje etc.
2. **Informaţii** – prin prelucrarea datelor şi găsirea relaţiilor dintre acestea se obţin informaţii care au un înţeles şi sunt integrate într-un context. Datele organizate şi prezentate într-un mod sistematic pentru a sublinia sensul acestor date devin informaţii. Pe scurt informaţiile sunt date prelucrate. Informaţiile se prezintă sub formă de rapoarte, statistici, diagrame etc.
3. **Cunoştinţele** sunt colecţii de date, informaţii, adevăruri şi principii învăţate, acumulate de-a lungul timpului. Informaţiile despre un subiect reţinute şi înţelese şi care pot fi folosite în luarea de decizii, formează judecăţi şi opinii devin cunoştinţe. Cu alte cuvinte, cunoştinţele apar în momentul utilizării informaţiei .

2. Colectarea şi analizarea datelor. Modelul conceptual

Primul pas în realizarea unei aplicaţii de baze de date este analiza datelor şi realizarea unei **scheme conceptuale (model conceptual)** al acestor date.

În această etapă sunt analizate natura şi modul de utilizare a datelor. Sunt identificate datele care vor trebui memorate şi procesate, se împart aceste date în grupuri logice şi se identifică relaţiile care există între aceste grupuri.

Analiza datelor este un proces uneori dificil, care necesită mult timp, însă este o etapă absolut obligatorie. Fără o analiză atentă a datelor şi a modului de utilizare a acestora, vom realiza o bază de date care putem constata în final că nu întruneşte cerinţele beneficiarului. Costurile modificării acestei baze de date este mult mai mare decât costurile pe care le-ar fi implicat etapa de analiză şi realizare a modelului conceptual. Modificarea modelului conceptual este mult mai uşoară decât modificarea unor tabele deja existente, care eventual conţin şi o mulţime de date. Ideea de bază a analizei datelor şi construirii modelului conceptual este "să măsori de două ori şi să tai o singură dată".

Informaţiile necesare realizării modelului conceptual se obţin folosind metode convenţionale precum intervievarea oamenilor din cadrul organizaţiei şi studierea documentelor folosite.

Odată obţinute aceste informaţii ele trebuiesc reprezentate într-o formă convenţională care să poată fi uşor înţeleasă de toată lumea. O astfel de reprezentare este ***diagrama entităţi-relaţii***, numită şi ***harta relaţiilor***, sau ***ERD***-ul (**E**ntity **R**elationship **D**iagram). Aceste scheme sunt un instrument util care uşurează comunicarea dintre specialiştii care proiectează bazele de date şi programatori pe de o parte şi beneficiari, pe de altă parte. Aceştia din urmă pot înţelege cu uşurinţă o astfel de schemă, chiar dacă nu sunt cunoscători în domeniul IT.

În concluzie putem sublinia câteva caracteristici ale ERD-urilor:

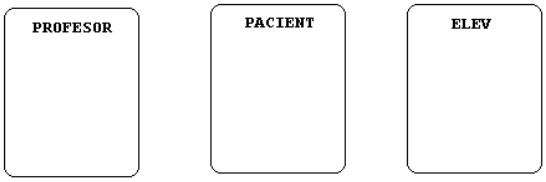
- sunt un instrument de proiectare
- sunt o reprezentare grafică a unui sistem de date
- oferă un model conceptual de înalt nivel al bazelor de date
- sprijină înţelegerea de către utilizatori a datelor şi a relaţiilor dintre acestea
- sunt independente de implementare.

În cele ce urmează vom prezenta principalele elemente care intră în componenţa unui ERD precum şi convenţiile de reprezentare a acestora.

3. Entităţi. Instanţe. Atribute. Identificator unic.

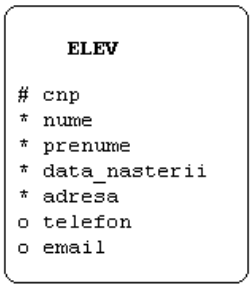
O ***entitate*** este un lucru, obiect, persoană sau eveniment care are semnificaţie pentru afacerea modelată, despre care trebuie să colectăm şi să memorăm date. O entitate poate fi un lucru real, tangibil precum o clădire, o persoană, poate fi o activitate precum o programare sau o operaţie, sau poate fi o noţiune abstractă.

O entitate este reprezentată în ERD printr-un dreptunghi cu colţurile rotunjite. Numele entităţii este întotdeauna un *substantiv la singular* şi se scrie în partea de sus a dreptunghiului cu *majuscule*, ca în figura I.1.1.



O entitate este de fapt o clasă de obiecte şi pentru orice entitate există mai multe ***instanţe*** ale sale. O instanţă a unei entităţi este un obiect, persoană, eveniment, particular din clasa de obiecte care formează entitatea. De exemplu, elevul **X** din clasa a IX-a A de la Liceul de Informatică din localitatea **Y** este o instanţă a entităţii **ELEV**.

După cum se vede pentru a preciza o instanţă a unei entităţi, trebuie să specificăm unele caracteristici ale acestui obiect, să-l descriem (precizăm de exemplu numele, clasa, şcoala etc). Aşadar, după ce am identificat entităţile trebuie să descriem aceste entităţi în termeni reali, adică să le stabilim ***atributele***. Un atribut este orice detaliu care serveşte la identificarea, clasificarea, cuantificarea, sau exprimarea stării unei instanţe a unei entităţi. Atributele sunt informaţii specifice ce trebuie cunoscute şi memorate.



De exemplu atributele entităţii **ELEV** sunt nume, prenume, adresa, număr de telefon, adresa de email, data naşterii etc.
În cadrul unui ERD, atributele se vor scrie imediat sub numele entităţii, cu litere mici. Un atribut este un *substantiv la singular* (vezi figura I.1.2).
Un atribut poate fi ***obligatoriu*** sau ***opţional***. Dacă un atribut este obligatoriu, pentru fiecare instanţă a entităţii respective trebuie să avem o valoare pentru acel atribut, de exemplu este obligatoriu să cunoaştem numele elevilor. Pentru un atribut opţional putem avea instanţe pentru care nu cunoaştem valoarea atributului respectiv. De exemplu atributul **email** al entităţii **ELEV** este opţional, un elev putând să nu aibă adresă de email. Un atribut obligatoriu este precedat în ERD de un asterisc *****, iar un atribut opţional va fi precedat de un cerculeţ **o**.

Atributele care definesc în mod unic instanţele unei entităţi se numesc ***identificator unic*** (**UID**). UID-ul unei entităţi poate fi compus dintr-un singur atribut, de exemplu codul numeric personal poate fi un identificator unic pentru entitatea **ELEV**. În alte situaţii, identificatorul unic este compus dintr-o combinaţie de două sau mai multe atribute. De exemplu combinaţia dintre titlu, numele autorului şi data apariţiei poate forma unicul identificator al entităţii **CARTE**. Oare combinaţia titlu şi nume autor nu era suficientă? Răspunsul este NU, deoarece pot exista de exemplu mai multe volume scrise de Mihai Eminescu având toate titlul Poezii, dar apărute la date diferite.

Atributele care fac parte din identificatorul unic al unei entităţi vor fi precedate de semnul diez **#** (figura I.1.2 şi I.1.3). ***Atributele din UID sunt întotdeauna obligatorii***, însă semnul **#** este suficient, nu mai trebuie pus şi un semn asterisc în faţa acestor atribute.
Valorile unor atribute se pot modifica foarte des, ca de exemplu atributul vârstă. Spunem în acest caz că avem de a face cu un ***atribut volatil***. Dacă valoarea unui atribut însă se modifică foarte rar sau deloc (de exemplu data naşterii) acesta este un atribut ***non-volatil***. Evident este de preferat să folosim atribute non-volatile atunci când acest lucru este posibil.

4. Relaţii între entităţi

În lumea reală, obiectele nu există izolat.Intre ele exista relatii Aşadar, după ce aţi identificat care sunt entităţile şi atributele acestor entităţi este timpul să punem în evidenţă relaţiile care există între aceste entităţi, modul în care acestea comunică între ele. O ***relaţie*** este o asociere, legătură, sau conexiune existentă între entităţi şi care are o semnificaţie pentru afacerea modelată. Orice relaţie este bidirecţională, legând două entităţi sau o entitate cu ea însăşi. De exemplu, elevii studiază mai multe materii, o materie e studiată de către elevi.
Orice relaţie este caracterizată de următoarele elemente:

- 1. numele relaţiei ; 2.opţionalitatea relaţiei; 3. gradul (cardinalitatea) relaţiei.

Să luăm de exemplu relaţia existentă între entităţile **JUCĂTOR** şi **ECHIPĂ**. Vom spune:

Un **JUCĂTOR** joacă într-o **ECHIPĂ**. Si La o **ECHIPĂ** trebuie să joace unul sau mai mulţi **JUCĂTORI**.

- **Numele relaţiei** este: *joacă*.
- Pentru a stabili **opţionalitatea** relaţiei trebuie să răspundem la următoarea întrebare: Un jucător **trebuie** să joace într-o echipă? Se poate ca un jucător să nu joace în nici o echipă? Dacă acceptăm că toţi jucătorii trebuie să joace într-o echipă relaţia este obligatorie sau mandatorie şi vom spune: Un **JUCĂTOR** ***trebuie*** să joace într-o **ECHIPĂ**.

  Dacă însă acceptăm că există jucători care nu joacă în nici o echipă (de exemplu li s-a terminat contractul şi în momentul de faţă nu mai joacă la nici o echipă), atunci relaţia este opţională.

  În acest caz vom spune: Un **JUCĂTOR** ***poate*** juca la o **ECHIPĂ**.
- **Cardinalitatea** relaţiei este dată de numărul de instanţe ale entităţii din partea dreaptă a relaţiei care pot intra în relaţie cu o instanţă a entităţii din partea stângă a relaţiei. Adică va trebui să răspundem la întrebări de genul: La câte echipe poate juca un jucător? Răspunsurile posibile sunt **unul şi numai unul**, sau **unul sau mai mulţi**. Vom spune:

  Un **JUCĂTOR** trebuie/poate să joace la ***o*** **ECHIPĂ** ***şi numai una***.

  sau Un **JUCĂTOR** trebuie/poate să joace la ***una sau mai multe*** **ECHIPE**.

Cea mai realistă varinată a relaţiei este aşadar: Un **JUCĂTOR** poate să joace la o **ECHIPĂ** şi numai una.

4.Convenţii de reprezentare a relaţiilor

În cadrul diagramei entităţi-relaţii, o relaţie va fi reprezentată printr-o linie ce uneşte cele două entităţi.
Deoarece o relaţie este bidirecţională, linia ce uneşte cele două entităţi este compusă din două segmente distincte, câte una pentru fiecare entitate. Tipul segmentului ce pleacă de la o entitate ne va indica **opţionalitatea** relaţiei dintre această entitate şi entitatea aflată în cealaltă parte a relaţiei. Dacă acest segment este continuu este vorba de o relaţie obligatorie, o linie întreruptă indică o relaţie opţională.
De exemplu în figura I.1.4 segmentul ce pleacă de la entitatea JUCĂTOR fiind ***întreruptă*** înseamnă că un jucător ***poate*** juca la o echipă, adică relaţia este opţională. Segmentul ce pleacă dinspre entitatea ECHIPĂ este ***continuă,*** deci la o echipă ***trebuie*** să joace jucători.

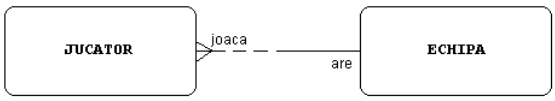


Figura I.1.4. Reprezentarea relaţiilor

Modul în care o linie se termină spre o entitate este important. Dacă se termină printr-o linie simplă, înseamnă că o instanţă şi numai una a acestei entităţi este în relaţie cu o instanţă a celeilalte entităţi. În exemplul anterior, linia de la **JUCATOR** la **ECHIPĂ** se termină în partea dinspre **ECHIPĂ** cu o ***linie simplă***, deci un jucător joacă la ***o*** echipa ***şi numai una***.
Dacă linia se termină cu trei linii (picior de cioară) înseamnă că mai multe instanţe ale entităţii pot corespunde unei instanţe a celeilalte entităţi. În exemplul anterior linia de la **ECHIPĂ** la **JUCĂTOR** se termină cu ***piciorul de cioară,*** înseamnă că unei instanţe a entităţii **ECHIPĂ** îi corespund mai multe instanţe ale entităţii **JUCĂTOR**, adică o echipă are ***unul sau mai mulţi*** jucători.

Caracteristica relaţiei	Valoare	Mod de reprezentare
Numele relaţiei	un verb	se scrie deasupra relaţiei
Opţionalitatea	relaţie obligatorie (TREBUIE)	linie continuă ————
	relaţie opţională (POATE)	linie întreruptă – – – – –
Cardinalitatea	una şi numai una	linie simplă ——
	una sau mai multe	picior de cioară ⋖

Tipuri şi subtipuri

În lumea reală obiectele sunt deobicei clasificate. Astfel vorbim despre animale vertebrate şi nevertebrate, despre licee teoretice, colegii, grupuri şcolare etc. E normal ca în modelarea bazelor de date să putem modela şi astfel de clasificări.
Un **subtip** sau o **subentitate** este o clasificare a unei entităţi care are caracteristici comune cu entitatea generală, precum atribute şi relaţii. Subtipurile se reprezintă în cadrul hărţii relaţiilor ca entităţi în interiorul altei entităţi. Atributele şi relaţiile comune tuturor subtipurilor se vor reprezenta la nivelul **supertipului**, sau **superentităţii**. Atributele şi relaţiile supertipului vor fi moştenite de către subtipuri. Un subtip poate avea la rândul său alte subtipuri incluse.

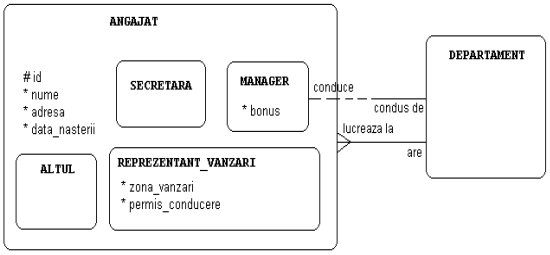


Figura I.4.1. Folosirea subtipurilor şi supertipurilor

Subtipurile trebuie să respecte două reguli importante:

- trebuie să acopere toate cazurile posibile de instanţe ale supertipului, cu alte cuvinte, orice instanţă a supertipului trebuie să aparţină unui subtip. De multe ori ERD-urile includ un subtip "ALTUL" pentru a acoperi toate situaţiile, şi pentru a permite viitoare dezvoltări ale modelului.

subtipurile trebuie să se excludă reciproc. Această regulă se traduce pe exemplul de mai sus în faptul că un angajat nu poate fi, de exemplu, şi manager şi secretară în acelaşi timp.

Documentare Business Rules

Pentru ca modelul conceptual sa fie complet se definesc **reguli structurale** (-indica tipuri de info ce vor fi stocate si cum relationeaza ele) si reguli procedurale (legate de timp , etc, -acestea ne se repr pe ERD, ci trebuie implementate in programare).

Tipuri de relaţii

Variantele de relaţii ce pot exista între două entităţi sunt prezentate mai jos:

- **relaţii one-to-one** – acest tip de relaţie este destul de rar întâlnit. Uneori astfel de relaţii pot fi modelate transformând una dintre entităţi în atribut al celeilalte entităţi.

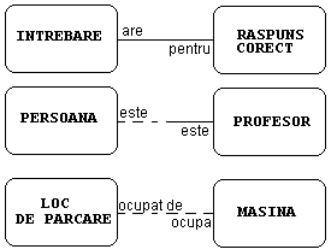


Figura I.1.5. Relaţii one-to-one

- **relaţii one-to-many** – sunt cele mai întâlnite tipuri de relaţii, însă şi aici cazurile c şi d prezentate în figura I.1.6 sunt mai puţin uzuale.

Să facem câteva observaţii pe marginea exemplelor din figura I.1.6. **Cazul a** este foarte des întâlnit. La **cazul b**, am ales o relaţie opţională dinspre **`POEZIE`** spre **`POET`** deoarece poate fi vorba de o poezie populară şi în acest caz nu există un poet cunoscut. La **cazul c**, am considerat că o formaţie nu poate exista fără a avea cel puţin un membru, însă un artist poate avea o carieră solo, deci nu face parte din nici o formaţie. **Varianta d** modelează o colecţie de filme memorate pe CD-uri. Pentru afacerea considerată, un CD conţine obligatoriu un film, dar unul singur, însă un film poate să nu încapă pe un singur CD de aceea el este poate fi memorat pe unul sau mai multe CD-uri.

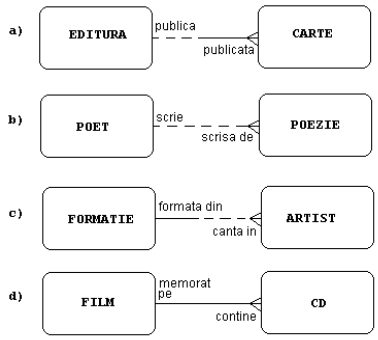


Figura I.1.6. Relaţii one-to-many

- **relaţii many-to-many** – aceste tipuri de relaţii apar în prima fază a proiectării bazei de date, însă ele trebuie să fie ulterior eliminate. Figura I.1.7 prezintă câteva exemple de relaţii many-to-many. La punctul b am considerat că un curs poate apărea pe oferta de cursuri a unei facultăţi, însă poate să nu fie aleasă de nici un student de aceea un curs **poate** fi urmat de unul sau mai mulţi studenţi. Invers, este posibil ca un student să fi terminat studiile şi să se pregătească pentru susţinerea examenului de licenţă şi de aceea el nu mai frecventează nici un curs. La punctul c, un profesor angajat al unei şcoli **trebuie** să predea cel puţin o disciplină. Iar o disciplină din planul de învăţământ trebuie să fie predată de cel puţin un profesor.

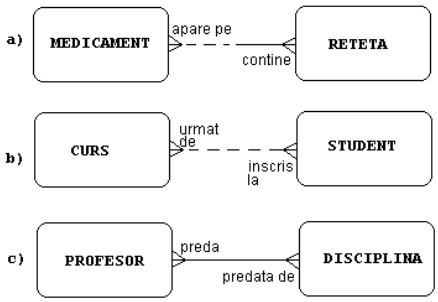


Figura I.1.7. Relaţii many-to-many

Transferabilitate

Spunem că o relaţie este nontransferabilă dacă o asociaţie între două instanţe ale celor două entităţi, odată stabilită, nu mai poate fi modificată. Nontransferabilitatea unei relaţii se reduce la faptul că valorile cheii străine corespunzătoare relaţiei respective nu pot fi modificate.Condiţia de nontransferabilitate a unei relaţii este asigurată prin program. De aceea trebuie să documentăm această restricţie.În ERD o relaţie nontransferabilă se notează cu un romb pe linia corespunzătoare relaţiei, înspre entitatea a cărei cheie străină nu este permis să o modificăm (adică în partea cu many a unei relaţii one-to-many).

În figura I.4.5 este dat un exemplu de relaţie nontransferabilă. Este vorba despre notele date elevilor. Este normal ca o notă dată unui elev să nu poată fi apoi transferată unui alt elev.

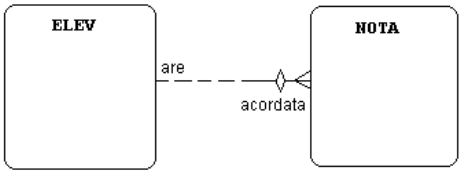


Figura I.4.5. Relaţii nontransferabile

Rezolvarea relaţiilor many-to-many

După cum am precizat mai devreme relaţiile many-to-many pot apărea într-o primă fază a proiectării bazei de date însă ele nu au voie să apară în schema finală. Să considerăm relaţia din figura I.1.14 dintre entităţile STUDENT şi CURS. Se ştie că orice curs se termină în general cu un examen. Unde vom memora nota studentului la fiecare examen?

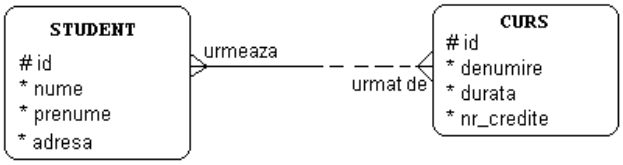


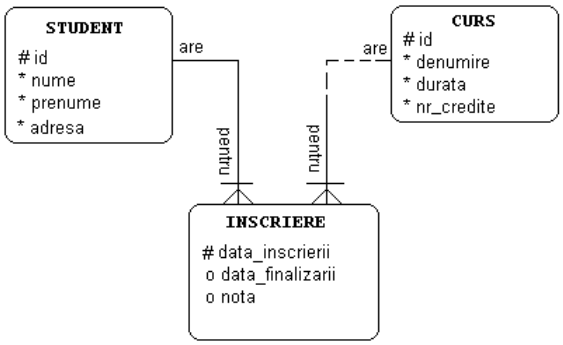
Figura I.1.14

Dacă încercăm să introducem atributul **NOTA** la entitatea **STUDENT**, nu vom şti cărei materii corespunde acea notă, întrucât unei instanţe a entutăţii student îi corespund mai multe instanţe ale entităţii **CURS**. Invers dacă încercăm să memorăm nota în cadrul entităţii **CURS**, nu vom ştii cărui student îi aparţine acea notă.
Rezolvarea unei relaţii many-to-many constă introducerea unei noi entităţi numită ***entitate de intersecţie***, pe care o legăm de entităţile originale prin câte o relaţie one-to-many.
Paşii în rezolvarea unei relaţii many-to-many sunt următorii:

1) se găseşte entitatea de intersecţie, pentru exemplul nostru vom introduce entitate **INSCRIERE**.
2) crearea noilor relaţii
   - opţionalitatea: relaţiile care pleacă **din entitatea de intersecţie sunt întotdeauna obligatorii în această parte**. În partea dinspre entităţile originale, relaţiile vor păstra opţionalitatea relaţiilor iniţiale.
   - cardinalitatea: ambele relaţii sunt de tip one-to-many, iar partea cu many va fi întotdeauna înspre entitatea de intersecţie.
   - numele noilor relaţii
3) adăugarea de atribute în cadrul entităţii de intersecţie, dacă acestea există. În exemplul nostru ne poate interesa de exemplu data la care s-a înscris un student la un curs, data la care a finalizat cursul precum şi nota obţinută la sfârşitul cursului.
4) stabilirea identificatorului unic pentru entitatea de intersecţie: dacă entitatea de intersecţie nu are un identificator unic propriu, atunci acesta se poate forma din identificatorii

unici ai entităţilor iniţiale la care putem adăuga atribute ale entităţii de intersecţie.
În exemplul nostru, identificatorul unic al entităţii de intersecţie este format din id-ul studentului, id-ul cursului şi data înscrierii la curs.

5) Faptul că identificatorul unic al unei entităţi preia identificatorul unic din altă entitate cu care este legată este reprezentat grafic prin bararea relaţiei respective, înspre entitatea care preia UID-ul celeilalte entităţi.



Analiza CRUD-se refera la CREATE, RETRIVE, UPDATE, DELETE-(crea , reface, actualiza, sterge) operatii ce fac din ERD un model complet.Se verifica daca modelul exprima toate operatiile ce se pot face si nu are elem inutile, etc.

UID artificial si compus

UID-(Unique Identifier)-e atributul ce identifica in mod unic entitatea(ex: CNP, cod, id,). Daca e nevoie de o combinatie de mai multe atribute care sa identifice in mod unic entitatea , e vorba de un UID compus. Daca se recurge la o modalitate de identificare printr-un cod artificial oferit in mod automat de program, e vorba de UID artificial.

Ce este normalizarea?

Normalizarea este o tehnică de proiectare a bazelor de date prin care se elimină (sau se evită) anumite anomalii şi inconsistenţe a datelor. O baza de date bine proiectată nu permite astfel ca datele să fie redundante, adică aceeaşi informaţie să se găsească în locuri diferite, sau să memorezi în baza de date, informaţii care se pot deduce pe baza altor informaţii memorate în aceeaşi bază de date. Anomaliile care pot să apară la o bază de date nenormalizată sunt următoarele:

anomalii la actualizarea datelor la o bibliotecă se înregistrează într-o tabelă următoarele date despre cărţi: ISBN, titlu, autor, preţ, subiect, editura, adresa editurii. La un moment dat o editură îşi schimbă adresa. Bibliotecara va trebui să modifice adresa editurii respective, în înregistrările corespunzătoare tuturor cărţilor din bibliotecă apărute la respectiva editură. Dacă această modificare nu se face cu succes, unele dintre înregistrări rămânând cu vechea adresă, apare din nou o inconsistenţă a datelor.

- **anomalii de inserare** – în exemplul anterior, nu vom putea memora adresa unei edituri, lucru inacceptabil dacă dorim să avem informaţii şi despre edituri a căror cărţi nu le avem în bibliotecă, eventual de la care dorim să facem comenzi.
- **anomalii de ştergere** – să presupunem că într-o tabelă memorăm următoarele informaţii: codul studentului, codul cursului, codul profesorului. La un moment dat, nici un student nu mai doreşte să participe la un anume curs. Ştergând toate înregistrările corespunzătoare cursului, nu vom mai putea şti niciodată cine preda acel curs.

Edgar Codd a definit primele trei forme normale 1NF, 2NF şi 3NF. Ulterior s-au mai definit formele normale 4NF, 5NF, 6NF care însă sunt rar folosite în proiectarea bazelor de date.

Prima formă normală

O entitate se găseşte în prima formă normală dacă şi numai dacă:- nu există atribute cu valori multiple;- nu există atribute sau grupuri de atribute care se repetă. Cu alte cuvinte toate atributele trebuie să fie atomice, adică să conţină o singură informaţie.

Dacă un atribut are valori multiple, sau un grup de atribute se repetă, atunci trebuie să creaţi o entitate suplimentară pe care să o legaţi de entitatea originală printr-o relaţie de 1:m. În noua entitate vor fi introduse atributele sau grupurile de atribute care se repetă. Să considerăm entitatea din figura I.2.1, referitoare la notele elevilor unei clase. Câteva observaţii referitoare la această entitate: câte discipline are un elev? Câte perechi (disciplina, nota) va trebui să aibă entitatea Elevi? Să spunem că ştim exact câte discipline maxim poate studia un elev. Ce se întâmplă dacă în anul viitor şcolar acest număr de discipline va fi mai mare? În plus, la o materie un elev poate avea mai multe note. Câte note? Cum memorăm aceste note? Le punem în câmpul corespunzător disciplinei cu virgulă între ele?

Cum rezolvăm această problemă? Vom crea o nouă entitate în care vom introduce disciplina şi nota la disciplina respectivă (vezi figura I.2.2.).

În acest fel fiecărui elev îi pot corespunde oricâte note, iar la o disciplină poate avea oricâte note, singura restricţie conform acestui model fiind că un elev nu va putea primi în aceeaşi zi la aceeaşi materie mai multe note.

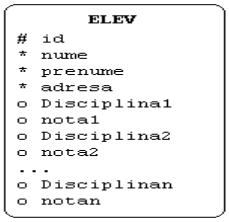


Figura I.2.1.

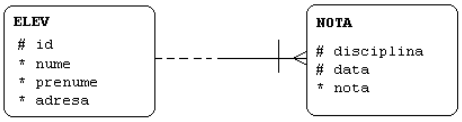


Figura I.2.2

Un alt exemplu de încălcare a regulilor primei formei normale, puţin mai "ascuns", este prezentat în figura I.2.5. De ce? Pentru că adresa este de forma "str. Florilor, bl. 45, sc. A, ap. 28, etaj 3, Braşov, cod 123123", formă care de fapt conţine mai multe informaţii elementare. Aşadar, în mod normal acest atribut ar trebui "spart" în mai multe atribute ca în figura I.2.6.

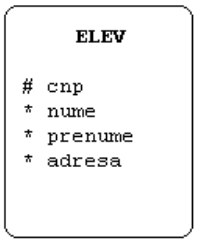


Figura I.2.5

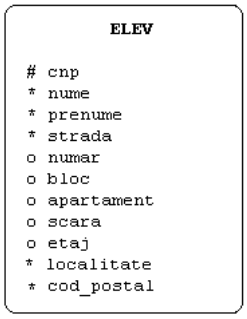


Figura I.2.6.

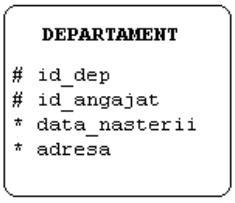
Noile atributele introduse sunt opţionale întrucât dacă elevul locuieşte la casă, probabil atributele bloc, apartament, scara, etaj, nu au sens. Invers dacă elevul locuieşte la bloc, probabil nu poate fi completat numărul.

Pentru acest tip de încălcare a regulilor formei normale 1NF poate fi totuşi ignorată, decizia depinzând de natura fenomenului, sau afacerii modelate. În exemplul anterior, întrucât datele din interiorul unei adrese este puţin probabil să se modifice, modificându-se el mult adresa completă a unui elev, se poate decide să nu operăm modificarea anterioară. Dacă însă aceste informaţii s-ar modifica frecvent, de exemplu denumirile străzilor s-ar modifica mereu, atunci probabil modificarea este de dorit.

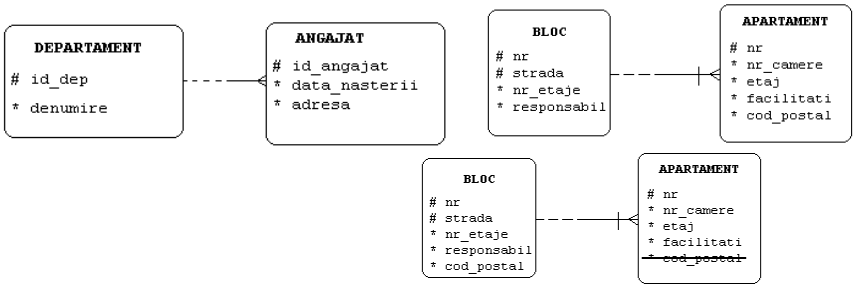
A doua formă normală

O entitate se găseşte în a doua formă normală **dacă şi numai dacă se găseşte în prima formă normală şi în plus orice atribut care nu face parte din UID (unique identifier) va depinde de întregul UID nu doar de o parte a acestuia**.

De exemplu dacă memorăm angajaţii unui departament într-o entitate ca mai jos:



Se observă că **`data_nasterii`** şi adresa sunt două atribute care depind doar de id-ul angajatului nu de întregul UID care este combinaţia dintre atributele **`id_dep`** si **`id_angajat`**. Această situaţie se rezolvă prin crearea unei noi entităţi **`ANGAJAT`**, pe care o legăm de entitatea **`DEPARTAMENT`** printr-o relaţie **`1:m`**.



O situaţie mai specială este în cazul relaţiilor barate, când trebuie ţinut seama că UID-ul unei entităţi este compus din atribute din entitatea respectivă plus un atribut sau mai multe atribute provenite din relaţia barată. Să considerăm următorul exemplu:

Se observă că UID-ul entităţii **`APARTAMENT`** este compus din combinaţia a trei atribute: numărul apartamentului, numărul blocului şi strada. Deci toate atributele din entitatea **`APARTAMENT`** care nu fac parte din UID, trebuie să depindă de întregul UID. Dar se ştie că atributul cod_postal depinde doar de strada si de numărul blocului, nu şi de

numărul apartamentului. Acest lucru ne spune ca acest atribut nu este memorat la locul potrivit. Deoarece depinde doar de combinaţia (strada, nr_bloc), înseamnă că de fapt depinde de UID-ul entităţii **`bloc`**. Aşadar vom muta atributul cod_postal în entitatea **`BLOC`**.

Observaţie. Dacă o entitate se găseşte în prima formă normală şi UID-ul său este format dintr-un singur atribut atunci ea se găseşte automat în a doua formă normală.

A treia formă normală

O entitate se găseşte în a treia formă normală dacă şi numai dacă se găseşte **în a doua formă normală şi în plus nici un atribut care nu este parte a UID-ului nu depinde de un alt atribut non-UID**. Cu alte cuvinte nu se acceptă dependenţe tranzitive, adică un atribut să depindă de UID în mod indirect.

Luăm ca exemplu entitatea **`CARTE`** din figura I.2.10. Atributul **`biografie_autor`** nu depinde de **`ISBN`** ci de atributul **`autor`**. Nerezolvarea acestei situaţii duce la memorarea de date redundante, deoarece biografia unui autor va fi memorată pentru fiecare carte scrisă de autorul respectiv. Rezolvarea acestei situaţii este să creăm o nouă entitate **`AUTOR`**, pe care o legăm de entitatea **`CARTE`** printr-o relaţie **`1:m`** (figura I.2.11.).

Figura I.2.10.

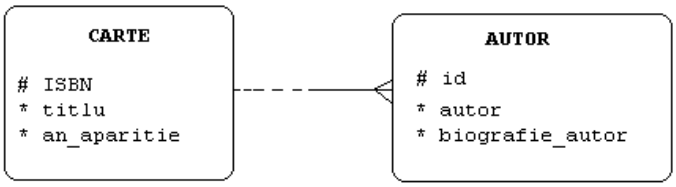


Figura I.2.11.

Atributul nu por avea alte atribute, asa ca el devine entitate.

3. Relaţii exclusive (arce)

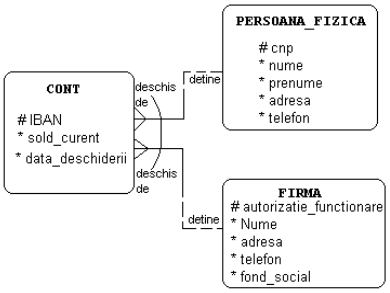
În unele situaţii, relaţiile se pot exclude reciproc, adică dintr-un grup de relaţii, la un moment dat doar una dintre ele poate avea loc. De exemplu, un cont anume la o bancă este deţinut fie de o persoană fizică fie de o firmă dar nu de ambele tipuri de clienţi simultan. Un grup de relaţii exclusive este reprezentat în harta relaţiilor printr-un arc peste relaţiile care fac parte din respectivul grup, ca în figura I.4.2. Toate relaţiile ce fac parte din grupul de relaţii exclusive trebuie să aibă aceeaşi opţionalitate. Un arc aparţine unei singure entităţi, adică va include doar relaţii care pleacă de la o aceeaşi entitate.

O entitate poate avea mai multe arce, dar o anumită relaţie nu poate face parte decât dintr-un singur arc.

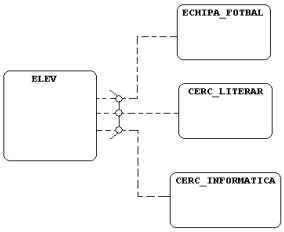
Există două tipuri de relaţii exclusive:

- relaţii exclusive ***obligatorii*** în care toate relaţiile ce fac parte din arcul respectiv sunt obligatorii, ceea ce înseamnă că de fiecare dată, una dintre relaţii are obligatoriu loc. Este şi cazul din figura 1 Evident că un cont trebuie să fie deţinut de o persoană fizică sau de o firmă, o a treia variantă neexistând.
- relaţii exclusive ***opţionale*** caz în care toate relaţiile ce fac parte din arc sunt opţionale. În acest caz de fiecare dată are loc cel mult una dintre relaţii, existând varianta ca pentru o instanţă a entităţii căreia aparţine arcul să nu aibă loc nici una din relaţiile din grupul respectiv. În figura 2, este exemplificată situaţia în care un elev poate opta să facă parte din echipa de fotbal, sau să participe la cercul literar sau la cercul de informatică. Însă regulile şcolii prevăd ca un elev să nu participe la două astfel de activităţi extraşcolare. Relaţiile fiind opţionale, înseamnă că un elev are libertatea de a decide să nu participe la nici o activitate extraşcolară.

. Relaţii exclusive obligatorii



Relaţii exclusive opţionale



Relaţii ierarhice. Relaţii recursive

Haideţi să analizăm care este structura personalului într-o firmă oarecare. În figura I.1.8 este prezentată doar o parte din organigrama unei firme.

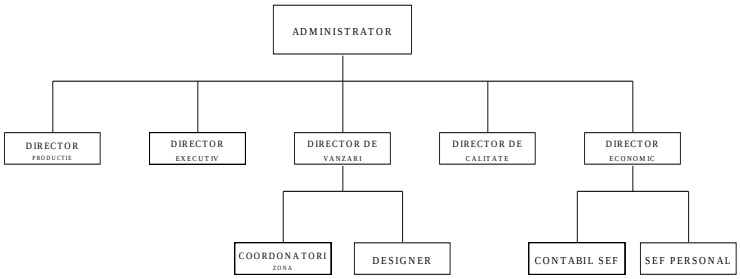


Figura I.1.8. Organigrama unei firme

Un model de proiectare a unei astfel de structuri într-o bază de date ar fi cea din figura următoare:

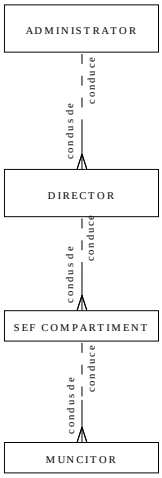


Figura I.1.9. Implementarea unei structuri ierarhice

Problema este că fiecare tip de angajat din figura anterioară este de fapt un angajat şi probabil există foarte multe atribute comune tuturor acestor entităţi ca de exemplu nume, prenume, adresă, telefon, email, data naşterii etc. Vom putea de aceea modela această structură cu ajutorul unei singure entităţi numită **ANGAJAT**. Însă fiecare angajat poate fi condus de către un alt angajat. Aşadar vom avea o relaţie de la entitatea **ANGAJAT** la ea însăşi. O astfel de relaţie se numeşte ***relaţie recursivă***.

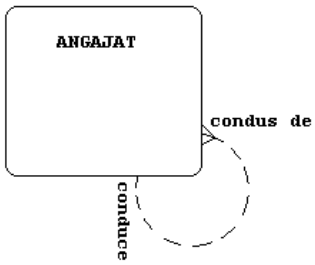
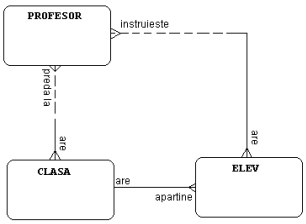
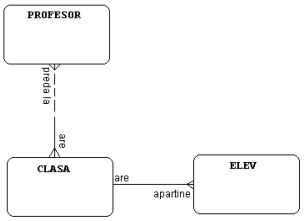


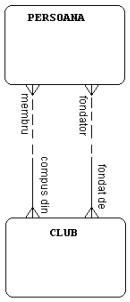
Figura I.1.10. Implementarea unei structuri ierarhice folosind relaţii recursive

Relaţii redundante si multiple

Atunci când o relaţie poate fi dedusă din alte relaţii spunem că acea relaţie este redundantă. Relatia se poate elimina.pot exista si relaţii multiple între entităţi

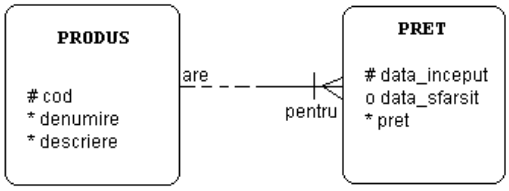




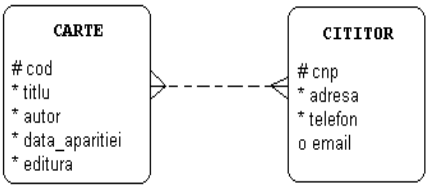


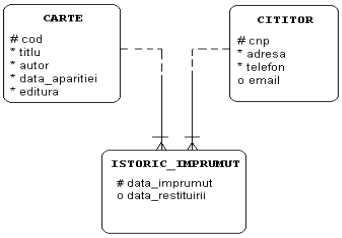
Modelarea datelor istorice

Viaţa înseamnă schimbare, orice lucru se schimbă de-a lungul timpului, şi nu doar obiectele se modifică în timp dar chiar şi relaţiile dintre aceste obiecte se schimbă. Preţul produselor poate suferi modificări destul de des. Factorii care duc la aceste modificări pot fi dintre cei mai diverşi, rata inflaţia, anotimpul etc. Aşadar atributul **`preţ`** din cadrul entităţii **`produs`** se modifică de-a lungul timpului. Dacă nu ne interesează decât preţul actual al fiecărui produs modelul este foarte simplu, ca cel din fig.Dacă însă pentru afacerea modelată este important să reţinem un istoric al preţurilor pentru fiecare produs, atunci atributul preţ se va transforma într-o nouă entitate



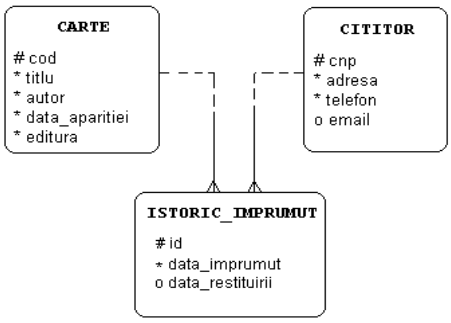
Atributul **`data_sfarsit`** este opţional, deoarece data până la care este valabil preţul curent al unui produs nu este de obicei cunoscut.
Vom considera acum o situaţie puţin mai dificilă. Să presupunem că dorim să modelăm o bază de date pentru o bibliotecă. Evident este important de reţinut un istoric al tuturor împrumuturilor, deoarece pe baza acestora, se pot afla domeniile de interes ale cititorilor, şi astfel vom şti ce achiziţii de carte să facem în viitor, vom putea determina uzura cărţilor astfel încât să le putem înlocui etc.
Într-o primă fază vom obţine o relaţie de many-to-many între entităţile **`CARTE`** şi **`CITITOR`**. Fiecare carte poate fi împrumutată de mai mulţi cititori (evident nu în acelaşi timp), şi fiecare cititor poate împrumuta mai multe cărţi .





Să rezolvăm această relaţie many-to-many. Aplicând ceea ce am învăţat în capitolele anterioare vom obţine schema din fig. a 2 a

Să verificăm că acest caz este cel corect. Cheia primară este acum combinaţia coloanelor **`cod_carte`** şi **`data_imprumut`**. Poate un cititor împrumuta două cărţi în aceeaşi dată? Adică următoarele două înregistrări pot exista simultan în tabela **`ISTORIC_IMPRUMUTURI`**? Răspunsul este DA, combinaţia celor două coloane, pentru cele două înregistrări fiind unică.
Deci bararea automată a celor două relaţii dinspre entitatea de intersecţie nu este întotdeauna o soluţie corectă. Pentru a evita aceste complicaţii putem recurge la introducerea unei chei artificiale în entitatea de intersecţie. În exemplul nostru se poate decide ca pentru fiecare împrumut în parte să se completeze câte o fişa separată care are un număr unic. Obţinem modelul din figura I.4.13, care este de asemenea



unul corect.
Fig. Introducerea unei chei artificiale

Conventii de ridabilitate: Se aplica conventiile Oracle de scriere a ERD-ului:
Entitatea se scrie cu majuscule, singular in interiorul unui dreptunghi cu vf rotunjite.Atributele se scriu cu litere mici , avand in fata unul din semnele #,*,o(UID, obligatoriu, optional).Orientarea liniilor este de la V la E si de sus in jos, evitand intersectia. Se pot folosi subdiagrame de explicare a diagramelor complexe, si explicarea entitatilor cu multe atribute.

Modelarea generica

Modelul generic aduce beneficii daca cerintele afacerii se schimba des. Atunci e nevoie de entitati si atribute noi.Se poate modela o singura entitate **Article type** care sa pastreze oricate tipuri de articole e nevoie, aceasta reduce nr de entitati.

Procesul maparii

Transformarea modelului conceptual, a ERD-ului, în modelul fizic, adică în baza de date propriu zisă, se numeşte mapare. Acest proces implică transformarea fiecărui element al ERD-ului.
Entitati → tabele, (CARTE-carti.dbf)
atribute→ campuri,
coloane, UID→cheie primara,
relatie→cheie straina,
business rules →constrangeri
Se mapeaza procesul de transformare in diagrama tabelei:

Numele coloanei	Tip	Tip cheie	Opţionalitatea
titlu	Varchar2	Pk	*
autor	Varchar2	Pk	*
data_apariţiei	Date		*
Format	Varchar2		*
Nr_pagini	Number		*

Tipuri de date in Oracle:

Tipul de date	Descriere	Dimensiune Maximă
VARCHAR2	Şir de caractere de lungime variabilă	**4000** bytes
CHAR	Şir de caractere de lungime fixă	**2000** bytes
NUMBER(p,s)	Număr având p cifre din care s la partea zecimală. (s negativ reprezintă numărul de cifre semnificative din faţa punctului zecimal)	p (precizia) între 1 şi 38. s (scala) între -84 şi 127.
DATE	Dată calendaristică	De la 1 Ianuarie 4712 BC pana la 31 Decembrie, **9999** AD.
TIMESTAMP	Se memorează data calendaristică, ora, minutul, secunda şi fracţiunea de secundă	Fracţiunea de secundă este memorată cu o precizie de la 0 la 9.
INTERVAL YEAR TO MONTH	perioadă de timp în ani şi luni.	
INTERVAL DAY TO SECOND	memorează un interval de timp în zile, ore, minute şi secunde	
CLOB	Character Large Object	4 Gigabytes
BLOB	Binary Large Object	4 Gigabytes
BFILE	Se memorează adresa unui fişier binar de pe disc	4 Gigabytes

dacă relaţia pe partea many este opţională atunci şi coloanele cheii străine vor fi opţionale. Ce înseamnă acest lucru? Faptul că un jucător poate la un moment dat să nu joace la nici o echipă, atunci câmpul cod_echipă va rămâne necompletat în dreptul lui (va avea valoarea NULL). Dacă însă relaţia este obligatorie pe partea many atunci coloanele ce fac parte din cheia străină vor fi opţionale.
În gereral, la maparea unei relaţii de tip one-to-many, vom introduce în tabela corespunzătoare entităţii de pe partea many a relaţiei cheia primară a entităţii de pe partea one a relaţiei. Câmpurile astfel întroduse se vor numi **cheie străină** (foreign key).
Aşadar:

- cheia străină a unei tabele este cheia primară din tabela referintă
- cheia străină este întotdeauna introdusă în tabela corespunzătoare entităţii din partea many a relaţiei.

Maparea relaţiilor one-to-one

Dându-se două entităţi **A** şi **B** legate între ele printr-o relaţie one-to-one, este evident că putem include cheia primară **A** în cadrulul tabelei **B**, dar putem proceda la fel de bine şi invers, incluzând cheia primară a tabelei **B** în cadrul tabelei **A**, deoarece fiecărei instanţe a entităţii A îi corespunde cel mult o instanţă a entităţii **B**, dar şi invers, oricărei instanţe a entităţii **B** îi corespunde cel mult o instanţă a entităţii **A**.
Pentru relaţia din figura I.3.3 de exemplu putem memora pentru fiecare persoană seria de paşaport, dar şi invers, pentru fiecare paşaport putem memora cnp-ul deţinătorului.

Figura I.3.3.

Decizia depinde de specificul afacerii modelate. Dacă de exemplu ne interesează în primul rând persoanele şi abia apoi datele de pe paşapoarte, atunci vom adopta probabil prima variantă, a memorării seriei de paşaport în cadrul tabelei **PERSOANE**, dacă însă baza de date este destinată evidenţei paşapoartelor, atunci probabil vom adopta varianta a doua.
Uneori este convenabil să memorăm cheia străină în ambele părţi ale relaţiei, în exemplul nostru pentru fiecare paşaport să memorăm cnp-ul persoanei care îl deţine, dar şi pentru fiecare persoană să memorăm seria de paşaport.

Maparea relaţiilor recursive

Dacă vom privi o relaţie recursivă ca pe o relaţie de tipul one-to-many între o entitate şi ea însăşi, atunci acest caz se reduce la ceea ce deja am discutat. Să exemplificăm relaţia din figura I.3.4. Relaţia recursivă din această figură poate fi privită ca o relaţie între două entităţi identice, ca în figura I.3.5.

Figura I.3.4. **Figura I.3.5.**

Aşadar vom introduce în cadrul tabelei **ANGAJAŢI**, marca şefului său. Diagrama de tabela va arăta ca mai jos.

Tabelul I.3.4.

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Marca	Number	Pk	*
Nume	Varchar2		*
Prenume	Varchar2		*
Data_angajarii	Date		*
Adresa	Varchar2		*
Telefon	Varchar2		o
Email	Varchar2		o
Marca_sef	**Number**	**Fk**	**o**

I.3.4. Maparea relaţiilor barate

Relaţiile barate sunt mapate ca cheie străină în tabela aflată în partea many a relaţiei, la fel ca la maparea oricărei relaţii one-to-many. Bara de pe relaţie exprimă faptul că acele coloane ce fac parte din cheia străină vor devenii parte a cheii primare a tabelei din partea many a relaţiei barate.
Pentru exemplul din figura I.3.6, cheia primară a tabelei **`ATRIBUTE`** va fi format din coloanele **`denumire_atribut`** şi **`denumire_entitate`**, aceasta din urmă fiind de fapt cheie străină în tabela **`ATRIBUTE`**.

Figura I.3.6. Maparea relaţiilor barate

Tabelul I.3.5. Tabela **`ENTITĂŢI`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
denumire	Varchar2	Pk	*

Tabelul I.3.5. Tabela **`ATRIBUTE`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
denumire_atribut	Varchar2	Pk	*
denumire_entitate	Varchar2	**Pk, Fk**	*
optionalitate	Varchar2		*

Să considerăm acum un exemplu în care există mai multe relaţii barate, în cascadă.

Figura I.3.7. Relaţii barate în cascadă

Tabelul I.3.6. Tabela **`A`**

Numele coloanei	Tip cheie	Opţionalitate
idA	**Pk**	*
C1		*

Tabelul I.3.7. Tabela **`B`**

Numele coloanei	Tip cheie	Opţionalitate
idB	**Pk**	*
C2		*
idA	**Pk, Fk**	*

Tabelul I.3.8. Tabela **`C`**

Numele coloanei	Tip cheie	Opţionalitate
idC	**Pk**	*
C3		*
idA	**Pk, Fk**	*
idB	**Pk, Fk**	*

Tabelul I.3.9. Tabela **`D`**

Numele coloanei	Tip cheie	Opţionalitatea
idD	**Pk**	*
C4		*
idA	**Fk**	*

. Operaţii specifice prelucrării bazelor de date

Orice sistem de gestiune a bazelor de date (SGBD) trebuie să asigure următoarele **funcţii**:

- definirea structurii bazei de date
- încărcarea datelor în baza de date (adăugarea de noi înregistrări la baza de date)

- accesul la date pentru:
    - interogare (afişarea datelor, sortarea lor, calcule statistice etc.)
    - ştergere
    - modificare
- întreţinerea bazei de date:
    - refacerea bazei de date prin existenţa unor copii de siguranţă
    - repararea în caz de incident
    - colectarea şi refolosirea spaţiilor goale
- posibilitatea de reorganizare a bazei de date prin:
    - restructurarea datelor
    - modificarea accesului la date
- securitatea datelor.

O parte din aceste operaţii pot fi realizate cu ajutorul limbajului SQL, altele cu ajutorul unor programe specializate, care sunt puse la dispoziţia administratorului bazei de date de către sistemul de gestiune al bazelor de date.

. Reguli de integritate

Detalierea caracteristicilor pe care trebuie să le prezinte un SGBD pentru a fi considerat relaţional s-a făcut de E. F. Codd în 1985 sub forma a 13 reguli. Una dintre aceste reguli precizează că restricţiile de integritate trebuie să poată fi definite în limbajul utilizat de SGBD pentru definirea datelor.
Regulile de integritate garantează că datele introduse în baza de date sunt corecte şi valide. Aceasta înseamnă că dacă există orice o regulă sau restricţie asupra unei entităţi, atunci datele introduse în baza de date respectă aceste restricţii. În Oracle, regulile de integritate se definesc la crearea tabelelor folosind **constrângerile.** Dar asupra acestora vom reveni în partea a doua a manualului.
Tipurile de reguli de integritate sunt următoarele:

- **Integritatea entităţilor** – indică faptul că nici o coloană ce face parte din cheia primară nu poate avea valoarea NULL. În plus, pentru fiecare înregistrare, cheia primară trebuie să fie unică.
- **Integritatea de domeniu** – acest tip de reguli permite ca într-o anumită coloană se introducă doar valori dintr-un anumit domeniu. De exemplu putem impune ca salariul unui angajat să fie cuprins între 4500 şi 5000 RON.
- **Integritatea referenţială** – este o protecţie care asigură ca fiecare valoare a cheii străine să corespundă unei valori a cheii primare din tabela referită. De exemplu, referindu-ne la tabelele **JUCĂTORI** şi **ECHIPE**, corespunzătoare ERD-ului din figura I.3.2, **cod** este cheie primară în tabela **ECHIPE**, iar în tabela **JUCĂTORI**, **cod** devine cheie străină. Astfel valoarea câmpului **cod** din cadrul tabelei **JUCĂTORI** corespunzătoare unui anumit jucător trebuie să se regăsească printre valorile câmpului **cod** din tabela **ECHIPE**, altfel ar însemna că jucătorul respectiv joacă la o echipă inexistentă (vezi figura I.3.8).

Figura I.3.8. Exemplu de încălcare a integrităţii referenţiale

Situaţii de încălcare a integrităţii referenţiale pot apărea:

- la **adăugarea** unei noi înregistrări în baza de date, se poate încerca introducerea unor valori invalide pentru câmpurile cheii străine;
- la **actualizarea** bazei de date;
- la **ştergerea** unei înregistrări. De exemplu se şterge înregistrarea corespunzătoare unei anumite echipe (echipa se desfiinţează). Înregistrările jucătorilor care au jucat la acea echipă vor încălca integritatea referenţială, deoarece se vor referi la o echipă care nu mai există. Soluţiile posibile sunt ca la ştergerea unei echipe, toţi jucătorii care au activat la acea echipă să fie şi ei şterşi din baza de date (ştergere în cascadă) sau valoarea câmpului **cod_echipă** pentru acei jucători să fie setată la **NULL**, ceea ce va înseamnă că acei jucători nu activează la nici o echipă.

Programe de validare şi de acţiune

În realizarea modelului conceptual al unei baze de date se ţine cont de modul în care funcţionează afacerea modelată, datele care trebuie să fie memorate, relaţiile dintre acestea etc. Modul de utilizare a diferitelor date, modul în care acestea sunt relaţionate pot diferi de la o afacere la alta.
Regulile afacerii unei organizaţii se referă în esenţă la procesele şi fluxurile tuturor datelor şi activităţilor zilnice din cadrul organizaţiei. Cum funcţionează organizaţia? Care sunt activităţile sale?
Regulile afacerii acoperă următoarele aspecte ale unei organizaţii:

- Orice tip de politici organizaţionale de orice tip şi de la orice nivel al organizaţiei.
- Orice tip de formule de calcule (ca de exemplu modul de calcul al ratelor pentru diverse împrumuturi, modul de calcul al salariilor etc)
- Orice tip de reguli impuse de lege sau reguli interne ale organizaţiei.

Regulile simple ale afacerii pot fi implementate în modelul bazei de date prin intermediul relaţiilor dintre entităţi. Acest tip de reguli se numesc **reguli structurale**.
Alte reguli ale afacerii pot fi implementate folosind regulile de integritate despre care am discutat în paragraful anterior. Există totuşi reguli pentru implementarea cărora va trebui să scriem programe speciale folosind limbaje specializate specifice SGBD-ului utilizat. Acest tip de reguli se numesc numite **reguli procedurale**. În Oracle acest tip de programe se vor scrie folosind limbajul PL/SQL (Procedural Language/Structuded Query Languge) şi se numesc **declanşatoare** (**triggere**).
Există două tipuri de declanşatoare:

- declanşatoare de aplicaţie care se execută când apar anumite evenimente la nivelul anumitor evenimente;
- declanşatoare ale bazei de date care sunt lansate în execuţie când apar diverse evenimente asupra datelor (de exemplu la executarea unor comenzi ca **`INSERT`**, **`UPDATE`**, **`DELETE`**) sau la apariţia unor evenimente system (logarea la baza de date sau delogarea).

Orice declanşator poate avea rol de validare a unei operaţii, poate realiza diferite operaţii suplimentare, ca de exemplu diferite calcule, caz în care vom spune că e vorba de un declanşator de acţiune.

Maparea tipurilor şi subtipurilor

Nici un sistem de gestiune a bazelor de date nu suportă în mod direct supertipurile şi subtipurile. Putem adopta mai multe soluţii ale acestei probleme. Vom exemplifica aceste variante pentru schema din figura I.4.1, în care, pentru simplitate, vom presupune că nu avem nevoie de subentitatea **`ALTUL`**.
Varianta 1. Vom crea o tabelă pentru supertip şi câte o tabelă pentru fiecare subtip. Diagramele de tabelă în acest caz vor fi:

Tabelul I.4.1. Tabela **`ANGAJAŢI`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Id_angajat	Number	Pk	*
Nume	Varchar2		*
Adresa	Varchar2		*
Data_nasterii	Date		*
Id_departament	Number	Fk	*

Tabelul I.4.2. Tabela **`SECRETARE`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Id_angajat	Number	Pk	*

Tabelul I.4.3. Tabela **`MANAGERI`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Id_angajat	Number	Pk	*
Bonus	Number		*
Id_depart_condus	Number	Fk	o

Tabelul I.4.4. Tabela **`REPREZENTANŢI_VÂNZĂRI`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Id_angajat	Number	Pk	*
Zona_vanzari	Varchar2		*
Permis_conducere	Varchar2		*

Am notat cu **`Id_depart_condus`** codul departamentului pe care îl conduce un manager, iar cu **`Id_departament`** codul departamentului în care lucrează un anumit angajat.
Cheia primară a supertipului va fi inclusă în toate tabelele corespunzătoare subtipurilor şi va deveni cheia primară a acelei tabele.
Atributele şi cheile străine provenite din relaţiile de la nivelul supertipului vor fi memorate în tabela corespunzătoare supertipului. Atributele şi relaţiile de la nivel de subtip, se vor memora doar în tabela corespunzătoare subtipului respectiv.
Acest model este cel mai natural dar poate crea multe probleme privind eficienţa întrucât sunt necesare multe operaţii de interogare din tabele multiple, pentru a obţine informaţii suplimentare despre toţi angajaţii.
Varianta 2. Vom crea câte o tabelă pentru fiecare subtip. Atributele şi cheile străine provenite din relaţiile de la nivelul supertipului vor fi introduse în fiecare tabelă astfel obţinută, acestea fiind moştenite de către fiecare subtip.

Tabelul I.4.5. Tabela **`SECRETARE`**

Numele coloanei	Tip	Tip cheie	Opţionalitate
Id_angajat	Number	Pk	*
Nume	Varchar2		*
Adresa	Varchar2		*
Id_departament	Number	Fk	*
Data_nasterii	Date		*

Tabelul I.4.6. Tabela **`MANAGERI`**

Numele coloanei	Tip	Tip cheie	Opţionalitate
Id_angajat	Number	Pk	*
Nume	Varchar2		*
Adresa	Varchar2		*
Data_nasterii	Date		*
Bonus	Number		*
Id_depart_condus	Number	Fk	o
Id_departament	Number	Fk	*

Tabelul I.4.7. Tabela **`REPREZENTANŢI_VÂNZĂRI`**

Numele coloanei	Tip	Tip cheie	Opţionalitate
Id_angajat	Number	Pk	*
Nume	Varchar2		*
Adresa	Varchar2		*
Data_nasterii	Date		*
Id_departament	Number	Fk	*
Zona_vanzari	Varchar2		*
Permis_conducere	Varchar2		*

Varianta 3. Vom crea o singură tabelă pentru supertip. Această tabelă va conţine toate coloanele corespunzătoare atributelor de la nivelul supertipului, dar şi toate coloanele corespunzătoare tuturor atributelor din toate subtipurile.
Atributele de la nivelul supertipului îşi vor păstra opţionalitatea, însă atributele de la nivelul subtipurilor, vor fi toate introduse în tabelă, dar vor fi toate opţionale.
Relaţiile de la nivelul supertipului se transformă normal. Relaţiile de la nivelul subtipurilor se vor implementa cu ajutorul cheilor străine opţionale.

Tabelul I.4.8. Tabela **`ANGAJAŢI`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Id_angajat	Number	Pk	*
Nume	Varchar2		*
Adresa	Varchar2		*
Id_departament	Number	Fk	*
Data_nasterii	Date		*
Bonus	Number		o
Id_depart_condus	Number	Fk	o
Zona_vanzari	Varchar2		o
Permis_conducere	Varchar2		o
Tip_angajat	Numeric		*

Am introdus un atribut suplimentar **`Tip_angajat`**, cu ajutorul căruia vom codifica dacă un angajat este manager, secretară sau reprezentant de vânzări. Deoarece atributele de la nivelul subtipurilor sunt obligatorii pentru subtipul respectiv, va trebui să stabilim o regulă de integritate la nivel de înregistrare, care să verifice că pentru o înregistrare de un tip anume sunt completate câmpurile corespunzătoare. De exemplu, la adăugarea unui nou manager în tabela **`ANGAJAŢI`**, trebuie să verificăm dacă este completat câmpul bonus.

Se observă că vor fi multe câmpuri cu valoarea null, ceea ce înseamnă o risipă de spaţiu de memorie.

Tabelul I.4.9. Tabela **`ANGAJAŢI`**

Id_angajat	Bonus	Id_departament_condus	Zona_vanzari	Permis_conducere	Tip_angajat	…
10	125	5	(null)	(null)	1	
121	(null)	(null)	Transilvania	568147	2	
245	(null)	(null)	(null)	(null)	3	
…						

În acest tabel am codificat managerii cu 1, reprezentanţii de vânzări cu 2, iar secretarele cu 3. Aşadar această variantă de implementare este convenabilă când există puţine atribute şi relaţii la nivelul subtipurilor.

. Maparea arcelor

Pentru a mapa un arc vom crea atâtea chei străine câte relaţii există în arcul respectiv. Pentru modelul din figura I.4.2 vom obţine următoarele tabele:

Tabelul I.4.10. Tabela **`CONTURI`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
IBAN	Number	Pk	*
Sold_curent	Number		*
Data_deschiderii	Date		*
Cnp	Number	Fk1	o
Autorizatie_functionare	Number	Fk2	o

Tabelul I.4.11. Tabela **`PERSOANE_FIZICE`**

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Cnp	Number	Pk	*
Nume	Varchar2		*
Prenume	Varchar2		*
Adresa	Varchar2		*
Telefon	Number		*

Numele coloanei	Tip	Tip cheie	Opţionalitatea
Autorizatie_functionare	Number	Pk	*
Nume	Varchar2		*
Adresa	Varchar2		*
Telefon	Number		*
Fond_social	Number		*

Tabelul I.4.12. Tabela **FIRME** Deşi relaţiile din arc sunt obligatorii, cheile străine corespunzătoare au fost setate ca fiind opţionale, deoarece pentru fiecare înregistrare trebuie să avem completată una din cele două chei străine, iar cealaltă cheie străină trebuie să rămână necompletată (principiul exclusivităţii). Va trebui să implementăm o condiţie de integritate care să verifice această condiţie.

1.1. Noţiuni introductive

SQL (pronunţat fie ca un singur cuvânt "sequel" sau pe litere "S-Q-L") se bazează pe studiile lui E.F. Codd, prima implementare a limbajului SQL fiind dezvoltată de către firma IBM la mijlocul anilor 1970. Mai târziu, compania Relational Software Inc. (cunoscută astăzi sub numele Oracle Corporation) a lansat prima versiune comercială de SQL. În prezent SQL este un limbaj complet standardizat, recunoscut de către Institutul Naţional American de Standarde (ANSI – American National Standards Institute). Puteţi folosi SQL pentru a accesa baze de date Oracle, SQL Server, DB2, sau MySQL.
SQL utilizează o sintaxă simplă, uşor de învăţat şi utilizat. Comenzile SQL pot fi grupate în cinci categori după cum urmează:

- **Limbajul de interogare** Permite regăsirea liniilor memorate în tabelele bazei de date. Vom scrie interogări folosind comanda **SELECT**.
- **Limbajul de manipulare a datelor** (**DML** - Data Manipulation Language) Permite modificarea conţinutului tabelelor. Există următoarele comenzi **DML**:
  INSERT - pentru adăugarea de noi linii într-o tabelă
  UPDATE - pentru modificarea valorilor memorate într-o tabelă
  DELETE - pentru ştergerea liniilor dintr-o tabelă.
- **Limbajul de definire a datelor** (**DDL** - Data Definition Language) Vă permite să definiţi structura tabelelor care compun baza de date. Comenzile din această grupă sunt:
  CREATE - vă permite să creaţi structurile bazei de date. De exemplu, **CREATE TABLE** este utilizată pentru crearea tabelelor, cu **CREATE USER**, puteţi crea utilizatorii bazei de date etc..
  ALTER - permite modificarea structurilor bazei de date. De exemplu, cu comanda **ALTER TABLE** puteţi modifica structura unei tabele.
  DROP - puteţi şterge structuri ale bazei de date. De exemplu pentru a şterge o tabelă folosiţi comanda **DROP TABLE**.
  RENAME - puteţi schimba numele unei tabele.
  TRUNCATE - vă permite să ştergeţi întregul conţinut al unei tabele.
- **Comenzi de control al tranzacţiilor** (**TC** - Transaction Control):
  COMMIT - vă permite să faceţi ca modificările asupra bazei de date să devină permanente.
  ROLLBACK - permite renunţarea la ultimele modificări asupra bazei de date.
  SAVEPOINT – vă permite să definiţi un "punct de salvare" la care să puteţi reveni, renunţând la modificările făcute după acel punct asupra bazei de date.
- **Limbaj de control al datelor** (**DCL - Data Control Language**) Permite definirea şi modificarea drepturilor utilizatorilor asupra bazei de date. Există două comenzi în această categorie:
  GRANT - vă permite să acordaţi drepturi altor utilizatori asupra structurilor bazei voastre de date.
  REVOKE - puteţi să anulaţi anumite drepturi utilizatorilor bazei de date.

Există multe metode prin care puteţi rula comenzile SQL şi a vedea rezultatele rulării acestor comenzi. Pentru scopul acestui manual vă sfătuim să utilizaţi Oracle Database 10g Express Edition, o versiune simplificată a serverului de Oracle, care este ideal pentru utilizarea pe calculatorul personal, fiind de dimensiuni mult reduse faţă de versiunea comercială a programului.
Puteţi descărca gratuit această versiune a serverului Oracle de pe site-ul Oracle de la adresa
http://www.oracle.com/technology/software/products/database/xe/index.html
însă veţi fi solicitat să vă creaţi un cont pe acest site.

Vă prezentăm pe scurt paşii ce trebuie să îi urmaţi pentru a instala şi configura Oracle Database 10g Express Edition.

Pasul 1 Porniţi instalarea dând dublu click pe fişierul executabil descărcat de la adresa menţionată anterior. Urmaţi paşii indicaţi de către programul de instalare. În unul dintre ecranele ce vor apărea vi se solicită introducerea unei parole. Aceasta va fi parola utilizatorului **`SYSTEM`** şi veţi avea nevoie de această parolă ulterior, deci notaţi-o pentru a nu o uita.

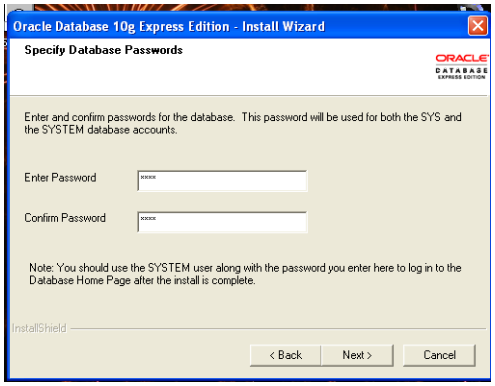


Figura II.1.1 Introduceţi parola utilizatorului **`SYSTEM`**

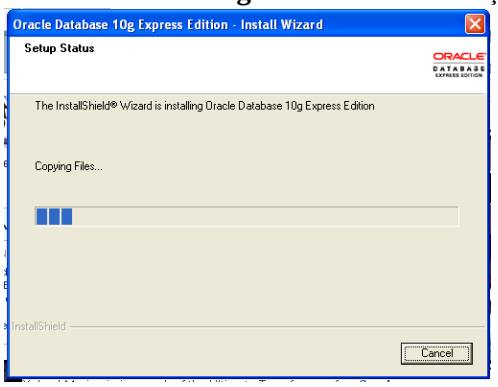


Figura II.1.2. Instalarea aplicaţiei

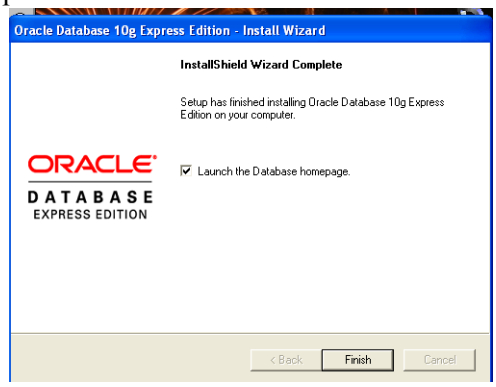


Figura II.1.3. Finalizarea instalării

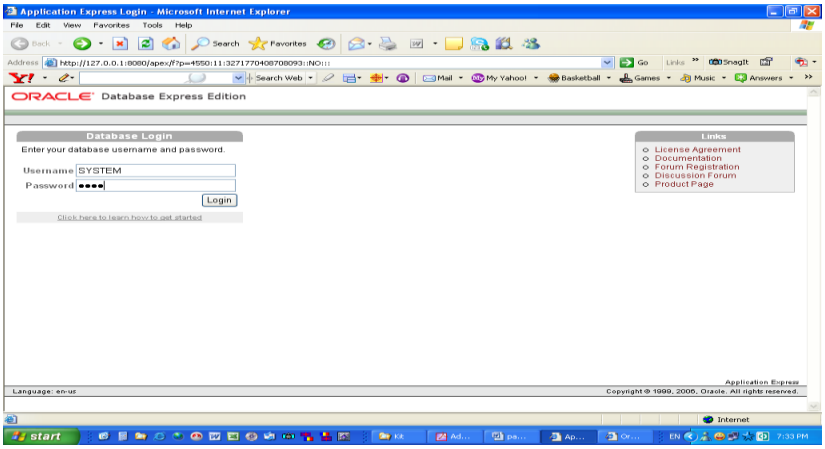


Figura II.1.4 Pagina principală a aplicaţiei Oracle Database 10g Express Edition

Pasul 2 Logaţi-vă cu utilizatorul **`SYSTEM`** şi parola dată la pasul 1.

Pasul 3 După logare alegeţi opţiunea **Administration** şi apoi **Database Users**. În noua fereastră deschisă (figura II.1.5) daţi click pe iconul **HR**.

`HR` va fi numele de utilizator cu care vă veţi putea loga pentru a rula comenzile SQL.

În fereastra Manage Database User (fig. II.1.6), faceţi următoarele setări:

- introduceţi parola pentru contul **HR**
- În caseta **Account Status** selectaţi opţiunea **Unlocked**.
- în zona **Roles** asiguraţi-vă că sunt bifate opţiunile **CONNECT** şi **RESOURCE**.

Apoi daţi click pe butonul **Alter User**.

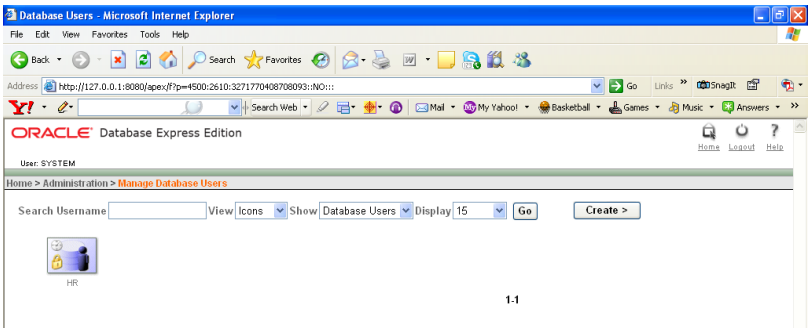


Figura II.1.5. Fereastra Database Users

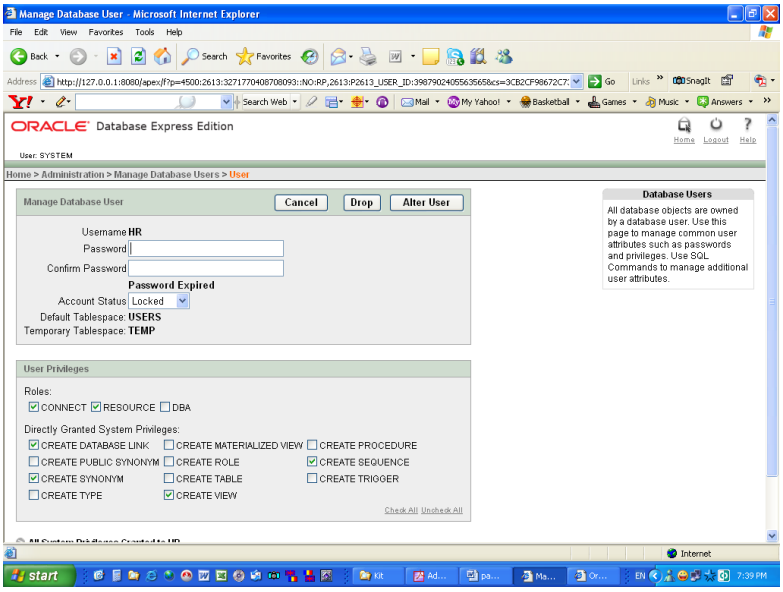


Figura II.1.6. Setarea drepturilor pentru utilizatorul HR

Pasul 4 Apăsaţi butonul **logout** din colţul dreapta sus al paginii şi logaţi-vă cu noul cont creat.

Pasul 5. Pentru rularea comenzilor SQL veţi da click pe butonul SQL (fig. II.1.7) iar apoi pe butonul "SQL Commands" (fig II.1.8)

Figura II.1.7.

Figura II.1.8.

În următoarea fereastră puteţi rula comenzile SQL. Veţi scrie comenzile în caseta text din această fereastră, apoi acţionaţi butonul Run sau apăsaţi tastele Ctrl+Enter. Rezultatele rulării comenzii, sau eventualele erori depistate vor fi afişate sub caseta text în care introduceţi comenzile (fig. II.1.9.).

Dacă rezultatul comenzii va conţine mai multe linii, pentru a le putea vedea pe toate alegeţi din caseta **Display** (aflată deasupra casetei în care introduceţi comenzile SQL) numărul dorit de linii afişate.

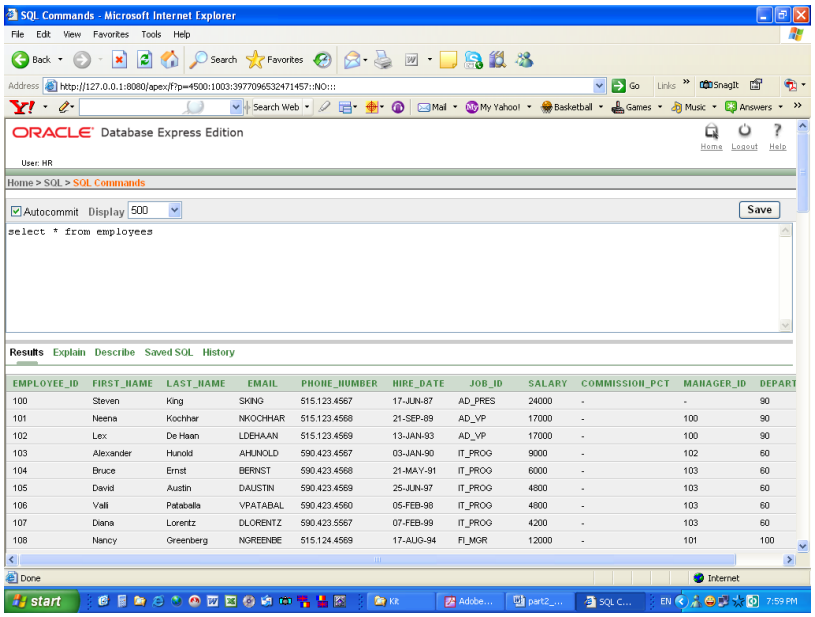


Figura II.1.9. Fereastra SQL Commands

Implicit baza de date conţine câteva tabele populate cu date. Pentru a putea vedea care sunt aceste tabele, care este structura lor, ce date conţin etc., din pagina principală a aplicaţiei alegeţi opţiunea Object Browser. În panoul din stânga daţi click pe numele unei tabele şi în panoul din dreapta aveţi mai multe opţiuni pentru vizualizarea şi modificarea structurii şi conţinutului tabelei respective (fig II.1.10).

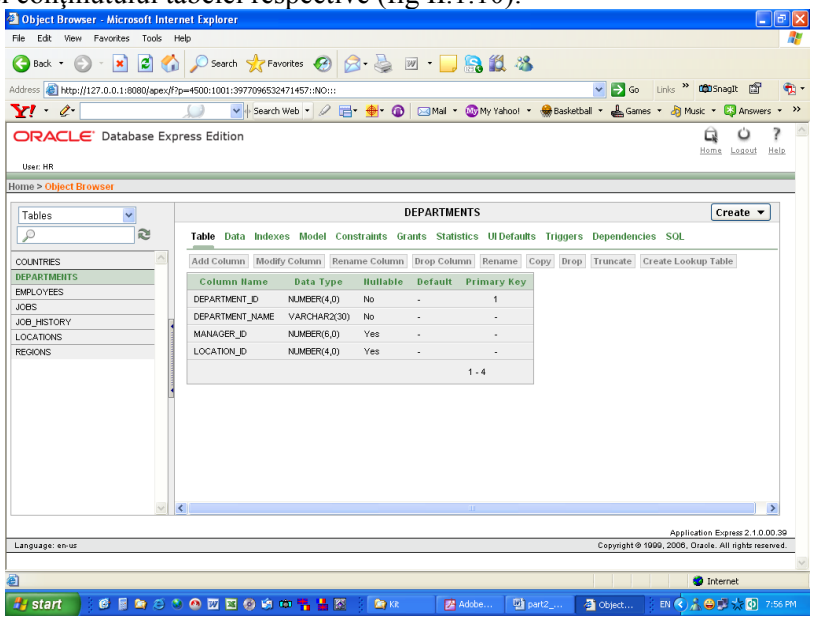


Figura II.1.10. Fereastra Object Browser

1.2. Elemente de bază ale SQL

Vom prezenta foarte pe scurt principalele elemente ce intră în componenţa unei comenzi SQL.

Nume

Toate obiectele dintr-o bază de date, tabele, coloane, vizualizări, indexi, sinonime, etc, au un nume.

Numele poate fi orice şir de maxim **30** de litere, cifre şi caracterele speciale: caracterul de subliniere (underscore **_**), diez (**#**), şi dolar (**$**), primul caracter fiind obligatoriu o literă. Evident numele unui obiect din baza de date trebuie să fie unic.

Cuvinte rezervate
Ca în orice limbaj, şi în SQL există o listă de cuvinte rezervate. Acestea sunt cuvinte pe care nu le puteţi folosi cu alt scop, ca de exemplu pentru denumirea tabelelor voastre.

Constante
O constantă sau literal este o valoare fixă ce nu poate fi modificată. Există:
- constante numerice, de exemplu **2**, **3.5**, **.9** etc. Se observă că dacă un număr real are partea întreagă egală cu zero, ea nu mai trebuie precizată.
- constante alfanumerice (sau şir de caractere). Constantele şir de caractere sunt scrise între apostrofuri şi sunt case-sensitive. Exemple: **'abc'**, **'Numele'**.

Variabile
Variabilele sunt date care pot avea în timp valori diferite. O variabilă are întotdeauna un nume pentru a putea fi referită.
SQL suportă două tipuri de variabile:
- variabilele asociate numelor coloanelor din tabele
- variabile sistem.

Expresii
O expresie este formată din variabile, constante, operatori şi funcţii. Funcţiile vor face obiectul a două dintre următoarele capitole ale manualului. În continuare ne vom ocupa de operatorii ce pot fi folosiţi în expresii.

Operatori aritmetici
Operatorii aritmetici permişi în SQL sunt cei patru operatori din matematică: adunare **+**, scădere **-**, înmulţire *****, împărţire **/**. Ordinea de efectuare a operaţiilor aritmetice este cea din matematică (mai întâi înmulţirea şi împărţirea şi apoi adunarea şi scăderea).

Operatori alfanumerici
Există un singur operator alfanumeric şi anume operatorul de concatenare a două şiruri **||** (două bare verticale fără spaţii între ele).
De exemplu expresia **'abc'||'xyz'** are valoarea **'abcxyz'**.

Operatori de comparaţie
Pe lângă operatorii obişnuiţi de comparaţie: **<**, **>**, **<=**, **>=**, **<>** sau **!=** (pentru diferit), **=**, SQL mai implementează următorii operatori speciali:

- **LIKE** – despre care vom discuta puţin mai târziu în acest capitol
- **BETWEEN** – testează dacă o valoare se găseşte într-un interval definit de două valori. Astfel expresia

  ```
  x BETWEEN a AND b
  ```

  este echivalentă cu expresia

  ```
  (x>=a) AND (x<=b)
  ```
- **IN** – testează dacă o valoare aparţine unei mulţimi de valori specificate. De exemplu expresia:

  ```
  x IN (a,b,c)
  ```

  este echivalentă cu

  ```
  (x=a) OR (x=b) OR (x=c)
  ```
- **IS NULL** şi **IS NOT NULL** – se folosesc pentru a testa dacă o expresie are valoarea **NULL** sau nu. Comparaţia cu **NULL** nu se poate face folosind operatorii obişnuiţi **=** şi respectiv **<>**.

Operatori logici
În ordinea priorităţii lor, aceştia sunt:

- **NOT** – negaţia logică
- **AND** – şi logic, expresia **a AND b** este adevărată dacă şi numai dacă ambii operanzi **a** şi **b** au valoarea adevărat.
- **OR** – sau logic, expresia **a OR b** este adevărată dacă şi numai dacă cel puţin unul dintre operanzii **a** şi **b** au valoarea adevărat.

1.3. Interogarea tabelelor. Comanda SELECT
Comanda **SELECT** este utilizată pentru a extrage date din baza de date. Setul de date returnate prin intermediul unei comenzi **SELECT** este compusă, ca şi tabelele bazei de date, din linii şi coloane, şi vor putea fi simplu afişate,

sau vom putea popula o tabelă cu datele returnate de către comanda **SELECT**, aşa cum vom vedea într-un capitol următor.
Cu ajutorul comenzii **SELECT** putem realiza următoarele tipuri de operaţii:

- **selecţia** – constă în filtrarea liniilor ce vor fi afişate. Vom folosi clauza **WHERE** pentru a defini criteriul sau criteriile pe care trebuie să le îndeplinească o linie pentru a fi returnată de către comanda **SELECT**.
- **proiecţia** – constă în alegerea doar a anumitor coloane pentru a fi afişate.
- **join** – constă în preluarea datelor din două sau mai multe tabele, "legate" conform unor reguli precizate.

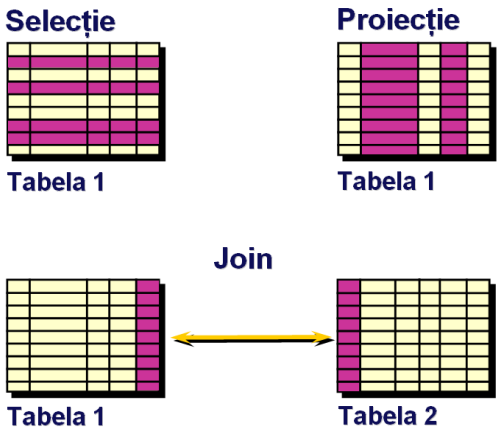


Figura II.1.11. Operaţiile realizate cu ajutorul comenzii **SELECT**

Cea mai simplă formă a comenzii **SELECT** are sintaxa:

```
SELECT Lista_expresii
FROM tabela
```

În clauza **SELECT** se va preciza o listă de coloane sau expresii ce se vor afişa, separate prin câte un spaţiu. În clauza **FROM** precizăm tabela din care se vor extrage coloanele ce vor fi afişate sau pe baza cărora vom realiza diverse calcule.
Vom exemplifica modul de folosire al comenzii **SELECT** pe tabela Persoane, având următoarea structură şi conţinut:

Tabelul II.1.1.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
1	Ionescu	Gheorghe	Brasov	22	5	300
4	Georgescu	Maria	Iasi	30	6	890
5	Marinescu	Angela	Sibiu	-	3	2100
6	Antonescu	Elena	Sibiu	10	1	840
7	Bischin	Paraschiva	Brasov	22	-	500
8	Olaru	Angela	Ploiesti	22	2	1500
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
3	Popescu	Ioan	Bucuresti	10	2	1200

Pentru a afişa toate datele (toate coloanele şi toate liniile) din tabela **persoane** vom scrie simplu:

```
SELECT * FROM persoane
```

Observaţi că în locul listei de coloane am scris un singur asterisc, ceea ce înseamnă că dorim să afişăm toate coloanele tabelei.
Dacă însă dorim să afişăm doar informaţiile din câteva coloane ale tabelei, de exemplu dorim să afişăm numele, prenumele şi localitatea fiecărei persoane vom preciza numele coloanelor în clauza **SELECT**:

```
SELECT nume, prenume, localitate FROM persoane
```

rezultatul fiind cel din tabelul II.1.2.**Tabelul II.1.2**

NUME	PRENUME	LOCALITATE

Ionescu	Gheorghe	Brasov
Georgescu	Maria	Iasi
Marinescu	Angela	Sibiu
Antonescu	Elena	Sibiu
Bischin	Paraschiva	Brasov
Olaru	Angela	Ploiesti
Vasilescu	Vasile	Cluj-Napoca
Popescu	Ioan	Bucuresti

După cum am precizat, putem realiza şi calcule cu coloanele unei tabele. De exemplu pentru a afişa pentru fiecare persoană, salariul mărit cu 10% folosim următoarea comandă:

```
SELECT nume, prenume, salariu, salariu * 1.10
FROM persoane
```

şi obţinem:**Tabelul II.1.3.**

NUME	PRENUME	SALARIU	SALARIU*1.10
Ionescu	Gheorghe	300	330
Georgescu	Maria	890	979
Marinescu	Angela	2100	2310
Antonescu	Elena	840	924
Bischin	Paraschiva	500	550
Olaru	Angela	1500	1650
Vasilescu	Vasile	950	1045
Popescu	Ioan	1200	1320

Aliasul unei coloane

Dacă priviţi tabelul II.1.3. puteţi observa că în capul de tabel afişat sunt trecute numele coloanelor cu majuscule sau expresia care a generat acea coloană, tot cu majuscule. Dacă dorim ca în capul de tabel să apară alt text, sau să nu se folosească doar majuscule va trebui să folosim un ALIAS pentru coloana respectivă. Aliasul este introdus în clauza **SELECT**, imediat după numele coloanei respective astfel:

```
SELECT nume, prenume, salariu AS SalariuVechi,
       salariu * 1.10 AS SalariuNou FROM persoane
```

În această comandă am stabilit două aliase **SalariuVechi** şi respectiv **SalariuNou**. Trebuie subliniat că nu este obligatorie folosirea cuvântului **AS** pentru a defini un alias, însă este de preferat să îl utilizăm pentru o mai mare claritate. Comanda anterioară va afişa:**Tabelul II.1.4.**

NUME	PRENUME	SALARIUVECHI	SALARIUNOU
Ionescu	Gheorghe	300	330
Georgescu	Maria	890	979
Marinescu	Angela	2100	2310
….			
Popescu	Ioan	1200	1320

Puteţi observa că deşi în comanda **SELECT** am scris aliasele folosind atât litere mici cât şi litere mari, la afişare acestea sunt scrise tot cu majuscule. Pentru a evita acest lucru, trebuie să introducem aliasul între ghilimele:

```
SELECT nume, prenume, salariu AS "SalariuVechi",
       salariu * 1.10 AS "SalariuNou" FROM persoane
```

NUME	PRENUME	SalariuVechi	SalariuNou
Ionescu	Gheorghe	300	330
Georgescu	Maria	890	979
Marinescu	Angela	2100	2310
….			
Popescu	Ioan	1200	1320

rezultatul obţinut de aceastăă dată fiind cel din tabelul II.1.5.

De asemenea dacă dorim ca aliasul să conţină mai multe cuvinte de exemplu respectiv **Salariul Nou** **Salariul Vechi**, va trebui să folosim şi de această dată ghilimele, în caz contrar generându-se o eroare. De exemplu comanda următoare va afişa tabelul II.1.6:

```
SELECT nume||' '||prenume "Numele si prenumele",
       salariu AS "Salariu Vechi",
       salariu * 1.10 AS "Salariu Nou" FROM persoane
```

Tabelul II.1.5.

Tabelul II.1.6.

Numele si prenumele	Salariu Vechi	Salariu Nou
Ionescu Gheorghe	300	330
Georgescu Maria	890	979
Marinescu Angela	2100	2310
Popescu Ioan	1200	1320

În cadrul clauzei **SELECT**, se pot folosi orice fel expresii în care se folosesc nume de coloane, constante, operatori, funcţii etc. De exemplu, comanda următoare va afişa tabelul II.1.7.

```
SELECT nume||' '||prenume||' are salariul egal cu '||
       salariu AS "Informatii persoane" FROM persoane
```

Tabelul II.1.7.

Informatii persoane
Ionescu Gheorghe are salariul egal cu 300
Georgescu Maria are salariul egal cu 890
Marinescu Angela are salariul egal cu 2100

Eliminarea liniilor duplicate

Să analizăm rezultatul rulării următoarei comenzi:

```
SELECT localitate, firma
FROM persoane
```

În tabelul II.1.8 se poate observa că în localitatea Braşov există două persoane care lucrează la aceeaşi firma având codul 22.

LOCALITATE	FIRMA
Brasov	**22**
Iasi	30
Sibiu	-
Sibiu	10
Brasov	**22**
Ploiesti	22
Cluj-Napoca	15

Bucuresti	10

Tabelul II.1.8.

Dacă dorim să vedem la ce firme lucrează persoanele din fiecare localitate, însă o firmă să fie afişată o singură dată pentru o localitate anume, deci combinaţia valorilor localitate şi firmă să fie unică, vom folosi clauza **DISTINCT** în cadrul clauzei **SELECT** astfel:

```
SELECT DISTINCT localitate, firma FROM persoane
```

combinaţia (Braşov, 22) fiind afişată acum o singură dată (tabelul II.1.9.).

LOCALITATE	FIRMA
Brasov	22
Bucuresti	10
Cluj-Napoca	15
Iasi	30
Ploiesti	22
Sibiu	10
Sibiu	-

Tabelul II.1.9.

Dar dacă dorim să afişăm doar localităţile ce apar în tabela Persoane, fiecare localitate să fie afişată o singură dată?
Vom scrie:

```
SELECT DISTINCT localitate FROM
persoane
```

rezultatul fiind acum:

Tabelul II.1.10.

LOCALITATE
Brasov
Bucuresti
Cluj-Napoca
Iasi
Ploiesti
Sibiu

Filtrarea liniilor. Clauza WHERE

Imaginaţi-vă că tabela persoane conţine date despre mii de persoane şi că la un moment dat vă interesează doar informaţiile despre persoanele dintr-o anumită localitate. Pentru a putea selecta doar acele linii care ne interesează, trebuie să adăugăm clauza **WHERE** la comanda **SELECT**. În această clauză vom preciza condiţiile pe care trebuie să le îndeplinească o linie pentru a fi afişată. Aşadar clauza **WHERE** permite realizarea operaţiei de selecţie (fig II.1.11).
De exemplu pentru a afişa toate persoanele care provin din **Bucureşti** sau **Braşov** vom scrie:

```
SELECT * FROM persoane
WHERE localitate='Brasov' OR localitate='Bucuresti'
```

care va afişa:

Tabelul II.1.11.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
1	Ionescu	Gheorghe	Brasov	22	5	300
7	Bischin	Paraschiva	Brasov	22	-	500
3	Popescu	Ioan	Bucuresti	10	2	1200

E acum timpul să vedem cum se foloseşte operatorul **LIKE**. Acesta este utilizat pentru a verifica dacă un şir de caractere respectă un anumit "model". Dacă valoarea se potriveşte modelului, operatorul va returna valoarea **true** (adevărat) în caz contrar va returna valoarea **False** (fals).
În model se pot utiliza următoarele caractere speciale:

- caracterul de subliniere (underscore _) ţine locul unui singur caracter, oricare ar fi acesta.
- caracterul procent (%) ţine locul la zero sau mai multe caractere, oricare ar fi acestea.

De exemplu, dacă dorim să afişăm toate persoanele al căror prenume conţine litera **a** pe orice poziţie, vom scrie:

```
SELECT * FROM persoane
```

```
WHERE lower(prenume) LIKE '%a%'
```

Modelul **'%a%'** precizează că în faţa caracterului **a**, în prenume, se pot găsi oricâte caractere, inclusiv zero caractere, iar după caracterul **a** se găsesc de asemenea oricâte caractere, inclusiv zero. Am folosit funcţia **LOWER** pentru a transforma toate caracterele în litere mici, altfel numele care încep cu litera **A**, întrucât acesta e scris cu majuscule, nu ar fi fost afişat.

Rezultatul rulării acestei comenzi arată astfel:

Tabelul II.1.12.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
4	Georgescu	Maria	Iasi	30	6	890
5	Marinescu	Angela	Sibiu	-	3	2100
6	Antonescu	Elena	Sibiu	10	1	840
7	Bischin	Paraschiva	Brasov	22	-	500
8	Olaru	Angela	Ploiesti	22	2	1500
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
3	Popescu	Ioan	Bucuresti	10	2	1200

Dacă însă dorim să afişăm persoanele al căror prenume conţine litera a pe a doua poziţie vom folosi caracterul underscore în model:

```
SELECT * FROM persoane
WHERE prenume LIKE '_a%'
```

Tabelul II.1.13.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
4	Georgescu	Maria	Iasi	30	6	890
7	Bischin	Paraschiva	Brasov	22	-	500
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950

În cazul în care trebuie să verificăm dacă un şir conţine unul dintre caracterele speciale underscore (_), backslash (\), procent (%) vom scrie în model caracterul respectiv precedat de orice caracter special (de exemplu **** sau **&**), iar după model vom preciza cu ajutorul clauzei **ESCAPE** care este caracterul special care introduce secvenţa corespunzătoare caracterelor ****, **_**, **%**.

Pentru a afişa persoanele din tabela **employees** al căror **job_id** conţine caracterul underscore (_) pe a treia poziţie de la sfârşit folosim comanda:

```
SELECT first_name, job_id FROM employees
WHERE job_id LIKE '%&_ _ _' ESCAPE '&'
```

sau

```
SELECT first_name, job_id FROM employees
WHERE job_id LIKE '%\_ _ _' ESCAPE '\'
```

iar dacă dorim să afişăm persoanele al căror job_id conţine un caracter underscore oriunde în şir vom utiliza comanda:

```
SELECT first_name, job_id FROM employees
WHERE job_id LIKE '%&_%' ESCAPE '&'
```

sau

```
SELECT first_name, job_id FROM employees
WHERE job_id LIKE '%\_%' ESCAPE '\'
```

Rezultatele afişate sunt cele din tabelul II.1.14, respectiv II.1.15.

Tabelul II.1.14. **Tabelul II.1.15.**

FIRST_NAME	JOB_ID
Neena	AD_VP
Lex	AD_VP

FIRST_NAME	JOB_ID
Steven	AD_PRES
Neena	AD_VP
Lex	AD_VP
Alexander	IT_PROG
Bruce	IT_PROG
…	…

II.1.4. Sortarea datelor. Clauza ORDER BY

Aţi fost probabil destul de des în situaţia de a trebui să ordonaţi anumite date pe baza unor criterii orarecare. Imaginaţi-vă cam ce ar însemna să căutaţi numărul de telefon al unei persoane într-o carte de telefoane în care persoanele sunt trecute într-o ordine aleatoare, nu ordonate alfabetic aşa cum suntem noi obişnuiţi.
Pentru a preciza criteriile după care se ordonează datele folosim clauza **ORDER BY**. În această clauză se vor preciza coloanele sau expresiile după care se vor ordona liniile unei tabele înainte de a fi afişate.
De exemplu, afişarea datelor din tabela **persoane** în ordine alfabetică (crescătoare) a localităţii se face folosind comanda:

```
SELECT * FROM persoane
ORDER BY localitate
```

Tabelul II.1.16.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
1	**Ionescu**	**Gheorghe**	**Brasov**	**22**	**5**	**300**
7	**Bischin**	**Paraschiva**	**Brasov**	**22**	**-**	**500**
3	Popescu	Ioan	Bucuresti	10	2	1200
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
4	Georgescu	Maria	Iasi	30	6	890
8	Olaru	Angela	Ploiesti	22	2	1500
5	Marinescu	Angela	Sibiu	-	3	2100
6	Antonescu	Elena	Sibiu	10	1	840

Se observă că există mai multe persoane din aceeaşi localitate. Dacă vrem ca persoanele din aceeaşi localitate să fie ordonate descrescător după salariu scriem:

```
SELECT * FROM persoane
ORDER BY localitate, salariu DESC
```

opţiunea **DESC** precizează că sortarea se face descrescător. Pentru a sorta crescător se poate preciza acest lucru cu opţiunea **ASC**, dar aceasta este opţională deoarece implicit datele sunt sortate crescător.
Rezultatul rulării comenzii anterioare este cel din tabelul II.1.17.

Tabelul II.1.17.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
7	**Bischin**	**Paraschiva**	**Brasov**	**22**	**-**	**500**
1	**Ionescu**	**Gheorghe**	**Brasov**	**22**	**5**	**300**
3	Popescu	Ioan	Bucuresti	10	2	1200
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
4	Georgescu	Maria	Iasi	30	6	890
8	Olaru	Angela	Ploiesti	22	2	1500
5	Marinescu	Angela	Sibiu	-	3	2100

6	Antonescu	Elena	Sibiu	10	1	840

Haideţi să sortăm acum tabela **persoane** după codul firmei. Vom scrie:

```
SELECT * FROM persoane ORDER BY firma
```

Rularea acestei comenzi duce la afişarea tabelului II.1.18. Să observăm că Marinescu Angela, deoarece nu are completat codul firmei (valoarea coldlui firmei este **null**) a fost afişată ultima. Aşadar la ordonarea crescătoare (implicită) valorile nule se trec la sfârşit, în timp ce la sortarea descrescătoare valorile nule apar la început.

Tabelul II.1.18.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
6	Antonescu	Elena	Sibiu	10	1	840
3	Popescu	Ioan	Bucuresti	10	2	1200
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
1	Ionescu	Gheorghe	Brasov	22	5	300
7	Bischin	Paraschiva	Brasov	22	-	500
8	Olaru	Angela	Ploiesti	22	2	1500
4	Georgescu	Maria	Iasi	30	6	890
5	Marinescu	Angela	Sibiu	-	3	2100

Comanda

```
SELECT * FROM persoane
ORDER BY firma DESC
```

va face ca Marinescu Angela să fie afişată prima (tabelul II.1.19).

Tabelul II.1.19.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
5	Marinescu	Angela	Sibiu	-	3	2100
4	Georgescu	Maria	Iasi	30	6	890
1	Ionescu	Gheorghe	Brasov	22	5	300
8	Olaru	Angela	Ploiesti	22	2	1500
7	Bischin	Paraschiva	Brasov	22	-	500
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
6	Antonescu	Elena	Sibiu	10	1	840
3	Popescu	Ioan	Bucuresti	10	2	1200

În criteriile de ordonare pot să apară şi expresii nu doar coloane din tabela interogată. Astfel putem scrie:

```
SELECT * FROM persoane ORDER BY prenume || nume
```

reztatul fiind cel din tabelul II.1.20.

De asemenea putem preciza ca sortarea să se facă după o expresie care apare în clauza **SELECT** prin indicarea poziţiei expresiei respective în lista de expresii din clauza **SELECT**. Astfel comanda

```
SELECT nume, prenume, salariu
FROM persoane ORDER BY 3 DESC
```

va sorta descrescător liniile după salariu, deoarece în caluza **SELECT**, salariu este a treia expresie (Atenţie! În tabela persoane salariul este coloana a 7-a):

Tabelul II.1.20.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
6	Antonescu	Elena	Sibiu	10	1	840
7	Bischin	Paraschiva	Brasov	22	-	500
4	Georgescu	Maria	Iasi	30	6	890
1	Ionescu	Gheorghe	Brasov	22	5	300
5	Marinescu	Angela	Sibiu	-	3	2100
8	Olaru	Angela	Ploiesti	22	2	1500
3	Popescu	Ioan	Bucuresti	10	2	1200
2	Vasilescu	Vasile	Cluj-Napoca	15	1	950

Tabelul II.1.21.

NUME	PRENUME	SALARIU
Marinescu	Angela	2100
Olaru	Angela	1500
Popescu	Ioan	1200
Vasilescu	Vasile	950
Georgescu	Maria	890
Antonescu	Elena	840
Ionescu	Gheorghe	300
Bischin	Paraschiva	500

Mai mult în clauza **ORDER BY** putem folosi aliasul unei coloane ca în exemplul următor:

```
SELECT nume||' '||prenume AS "Nume si prenume", salariu
FROM persoane
ORDER BY "Nume si prenume"
```

rezultatul fiind cel din tabelul II.1.22.

Desigur clauzele **WHERE** şi **ORDER BY** pot apărea împreună în aceeaşi comandă, ordinea în care acestea apar fiind **WHERE** şi apoi **ORDER BY**, aceasta fiind şi ordinea în care sunt executate: mai întâi sunt selectate liniile care trebuie să fie afişate şi abia apoi sunt sortate conform criteriului stabilit prin clauza **ORDER BY**. De exemplu, pentru a afişa în ordine descrescătoare a salariilor doar persoanele din Braşov şi Sibiu scriem:

```
SELECT * FROM persoane
WHERE localitate IN ('Sibiu', 'Brasov')
ORDER BY salariu DESC
```

rezultatul rulării acestei comenzi fiind cel din tabelul II.1.23.

Tabelul II.1.22.

Nume si prenume	SALARIU
Antonescu Elena	840
Bischin Paraschiva	500
Georgescu Maria	890
Ionescu Gheorghe	300
Marinescu Angela	2100
Olaru Angela	1500
Popescu Ioan	1200

Vasilescu Vasile	950

Tabelul II.1.23.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
5	Marinescu	Angela	Sibiu	-	3	2100
6	Antonescu	Elena	Sibiu	10	1	840
1	Ionescu	Gheorghe	Brasov	22	5	300
7	Bischin	Paraschiva	Brasov	22	-	500

.1.5. Afişarea primelor `n` linii

La sfârşitul anului şcolar, dirigintele clasei vă roagă să-l ajutaţi să afle care sunt primii trei elevi din clasă, în ordinea descrescătoare a mediei generale, pentru a şti cui să dea premiile. Aşadar se pune problema ca la afişarea datelor dintr-o tabelă să afişaţi doar primele **n** linii.
Pentru aceasta veţi avea nevoie de pseudocoloana **ROWNUM** care returnează numărul de ordine al unei linii într-o tabelă. De exemplu comanda următoare va afişa codul, numele şi prenumele persoanelor împreună cu numărul de ordine al acestora în tabela **persoane**:

```
SELECT cod, nume, prenume, rownum
FROM persoane
```

rezultatul este cel din tabelul următor:

Tabelul II.1.24.

COD	NUME	PRENUME	ROWNUM
1	Ionescu	Gheorghe	1
4	Georgescu	Maria	2
5	Marinescu	Angela	3
6	Antonescu	Elena	4
7	Bischin	Paraschiva	5
8	Olaru	Angela	6
2	Vasilescu	Vasile	7
3	Popescu	Ioan	8

Deşi ne-am aştepta ca într-o comandă **SELECT** care foloseşte clauza **ORDER BY**, **ROWNUM** să ne afişeze numărul de ordine al înregistrărilor în ordinea dată de **ORDER BY**, acest lucru nu se întâmplă, numărul de ordine fiind cel din tabela iniţială. Observaţi în acest sens tabelul II.1.25 afişat la rularea comenzii următoare

```
select rownum, cod, nume, prenume,
       localitate, firma, job, salariu
from persoane
order by salariu desc
```

Tabelul II.1.25.

ROWNUM	COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
3	5	Marinescu	Angela	Sibiu	-	3	2100
6	8	Olaru	Angela	Ploiesti	22	2	1500
8	3	Popescu	Ioan	Bucuresti	10	2	1200

7	2	Vasilescu	Vasile	Cluj-Napoca	15	1	950
2	4	Georgescu	Maria	Iasi	30	6	890
4	6	Antonescu	Elena	Sibiu	10	1	840
5	7	Bischin	Paraschiva	Brasov	22	-	500
1	1	Ionescu	Gheorghe	Brasov	22	5	300

Aşadar dacă dorim să afişăm primele 3 înregistrări din tabela iniţială vom putea scrie simplu:

```
SELECT cod, nume, prenume, rownum
FROM persoane
WHERE ROWNUM<=3
```

afişându-se rezultatul dorit (tabelul II.1.26.)

Tabelul II.1.26.

COD	NUME	PRENUME	ROWNUM
1	Ionescu	Gheorghe	1
4	Georgescu	Maria	2
5	Marinescu	Angela	3

însă, pentru a afişa persoanele cu cele mai mici trei salarii, comanda următoare nu afişează ceea ce am dori, deaorece Oracle prima dată va returna primele trei înregistrări din tabela persoane şi abia apoi le va sorta:

```
select rownum, cod, nume, prenume,
       localitate, firma, job, salariu
from persoane
where rownum<=3
order by salariu desc
```

comanda aceasta afişând: **Tabelul II.1.27.**

ROWNUM	COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
3	5	Marinescu	Angela	Sibiu	-	3	2100
2	4	Georgescu	Maria	Iasi	30	6	890
1	1	Ionescu	Gheorghe	Brasov	22	5	300

Pentru a obţine rezultatul dorit de noi vom folosi o subinterogare astfel:

```
select *
from (select * from persoane
       order by salariu)
where rownum<=3
```

În acest fel am forţat Oracle să sorteze mai întâi liniile şi apoi să afişeze primele trei linii din tabela obţinută.

Tabelul II.1.28.

COD	NUME	PRENUME	LOCALITATE	FIRMA	JOB	SALARIU
1	Ionescu	Gheorghe	Brasov	22	5	300
7	Bischin	Paraschiva	Brasov	22	-	500

6	Antonescu	Elena	Sibiu	10	1	840

2.1. Tipuri de funcţii

Funcţiile Oracle sunt împărţite astfel:

- **Funcţii singulare** – acestea operează la un moment dat asupra unei singure înregistrări. Aceste funcţii vor fi discutate în acest capitol
- **Funcţiile de grup** – operează asupra unui grup de înregistrări şi returnează o singură singură valoare pentru întregul grup.

Funcţiile singulare pot fi folosite în:

- clauza **SELECT**, pentru a modifica modul de afişare a datelor, pentru a realiza diferite calcule etc.
- clauza **WHERE**, pentru a preciza mai exact care sunt înregistrările ce se afişează
- clauza **ORDER BY**

Funcţiile singulare (single-row functions) pot fi la rândul lor împărţite în:

- Funcţii care operează asupra şirurilor de caractere
- Funcţii numerice
- Funcţii pentru manipularea datelor calendaristice
- Funcţii de conversie – care convertesc datele dintr-un tip în altul
- Funcţii de uz general.

Unele funcţii, precum **TRUNC** şi **ROUND** pot acţiona asupra asupra mai multor tipuri de date, dar cu semnificaţii diferite.

II.2.2. Tabela DUAL

În cele ce urmează vom folosi tabela **DUAL** pentru a testa modul de operare a funcţiilor singulare.
Această tabela este una specială, care conţine o singură coloană numită "**DUMMY**" şi o singură linie (vezi figura II.2.1).
Tabela **DUAL** se foloseşte atunci când realizăm calcule, sau evaluăm expresii care nu derivă din nici o tabelă anume.

Fie de exemplu comanda

```
SELECT (5*7-3)/2 FROM DUAL;
```

Expresia evaluată în această comandă nu are în componenţă nici o coloană a vreunei tabele, motiv pentru care este nevoie să apelăm la tabela **DUAL**.
Putem privi tabela **DUAL** ca pe o variabilă în care memorăm rezultatele calculelor noastre.
Tabela **DUAL** este o facilitate specifică Oracle. Este echivalentul tabelei **SYSDUMMY1** din **DB2**, tabelă aflată în shema sistem **SYSIBM**. În **Microsoft SQL Server 2000** este permisă scrierea de interogări fără clauza **FROM**.

II.2.3. Funcţii asupra şirurilor de caractere

Şirurile de caractere pot conţine orice combinaţie de litere, numere, spaţii, şi alte simboluri, precum semne de punctuaţie, sau caractere speciale. În Oracle există două tipuri de date pentru memorarea şirurilor de caractere:

- **CHAR** – pentru memorarea şirurilor de caractere de lungime fixă
- **VARCHAR2** – pentru memorarea şirurilor de caractere având lungime variabilă.

- **LOWER(sir)** – converteşte caracterele alfanumerice din şir în litere mari.
- **UPPER(sir)** – converteşte caracterele alfanumerice din şir în litere mici.

- **INITCAP(sir)** – converteşte la majusculă prima literă din fiecare cuvânt al şirului. Cuvintele sunt şiruri de litere separate prin orice caracter diferit de literă. Literele din interiorul cuvântului care erau scrise cu majuscule vor fi transformate în litere mici.

Exemplu	Rezultatul afişat
`SELECT LOWER(first_name) FROM employees;`	afişează prenumele persoanelor din tabela **employees** scrise cu litere mici
`SELECT LOWER('abc123ABC') FROM DUAL;`	`abc123abc`
`SELECT UPPER('abc123ABC') FROM DUAL;`	`ABC123ABC`
`SELECT INITCAP('aBc def*ghi') FROM dual;`	`Abc Def*Ghi` ***Explicaţie*** şirul conţine **3** cuvinte **aBc def** şi **ghi**

- **CONCAT(sir1, sir2)** – concatenează două şiruri de caractere

Exemplu	Rezultatul afişat

`SELECT CONCAT('abc','def') FROM dual;`	`abcdef`
	Explicaţie comanda poate fi transcrisă folosind operatorul de concatenare astfel:
	`SELECT 'abc'||'def' FROM dual;`

- **SUBSTR(sir,poz,nr)** – extrage din **sir** cel mult **nr** caractere începând din poziţia **poz**.

Observaţii

- dacă din poziţia **poz** până la sfârşitul şirului sunt mai puţin de **nr** caractere, se vor extrage toate caracterele de la poziţia **poz** până la sfârşitul şirului.
- parametrul **poz** poate fi şi o valoare negativă, ceea ce înseamnă că poziţia de unde se va începe extragerea caracterelor din şir se va determina numărând caracterele din şir de la dreapta spre stânga (vezi ultimele 3 exemple de mai jos)
- dacă **nr** nu este specificat, se va returna subşirul începând cu caracterul de pe poziţia **poz** din şir până la sfârşitul şirului.

Exemplu	Rezultatul afişat
`select substr('abcdef',3,2) from dual`	`cd`
`select substr('abcdef',3,7) from dual`	`cdef` ***Explicaţie.*** Chiar dacă din poziţia 3 până la sfârşitul şirului nu mai sunt 7 caractere se returnează caracterele rămase
`select substr('abcdef',3) from dual`	`cdef` ***Explicaţie.*** Acelaşi rezultat ca mai sus dacă nu se specifică numărul de caractere ce se extrag
`select substr('abcdef',7,3) from dual`	nu se va afişa nimic deoarece nu există poziţia **7** în şir, acesta având doar **5** caractere.
`select substr('abcdef',-4,2) from dual`	`cd` ***Explicaţie.*** Se extrag două caractere începând cu al patrulea caracter din dreapta.
`select substr('abcdef',-4,7) from dual`	`cdef`
`select substr('abcdef',-10,5) from dual`	nu se va afişa nimic deoarece şirul conţine mai puţin de 10 caractere

- **INSTR(sir,subsir,poz,k)** – returnează poziţia de început a celei de a **k**-a apariţii a subşirului **subsir** în şirul **sir**, căutarea făcându-se începând cu poziţia **poz** .

  Dacă parametrii **poz** şi **k** lipsesc, atunci se va returna poziţia primei apariţii a subşirului **subsir** în întregul şir **sir**.

  Poziţia de unde începe căutarea poate fi precizată şi relativ la sfârşitul şirului, ca şi în cazul funcţiei **substr**, dacă parametrul **poz** are o valoare negativă.

Exemplu	Rezultatul afişat
`select instr('abcdabcdabc','cd') from dual`	3
`select instr('abcd','ef') from dual`	0
`select instr('abcd','bce') from dual`	0
`select instr('abababababab','ab',4,2) from dual`	7 ***Explicaţie.*** Se începe căutarea din poziţia a patra, adică în zona subliniată cu o linie, şi se afişează poziţia de start a celei de a doua apariţii, (subşirul subliniat cu linie dublă)
`select instr('abababababab','ab',-4,1) from dual`	9

- **LENGTH(sir)** – returnează numărul de caractere din şirul **sir**.

Exemplu	Rezultatul afişat
`select length('abcd') from dual`	4

- **LPAD(sir1,nr,sir2)** – completează şirul **sir1** la stânga cu caracterele din şirul **sir2** până ce şirul obţinut va avea lungimea **nr**.
  Dacă lungimea şirului **sir1** este mai mare decât **nr**, atunci funcţia va realiza trunchierea şirului **sir1**, ştergându-se caracterele de la sfârşitul şirului.

Exemplu	Rezultatul afişat
`select lpad('abcd',3,'*') from dual`	abc
`select lpad('abcd',10,'*.') from dual`	*.*.*.abcd
`select lpad('abc',10,'*.') from dual`	*.*.*.*abc
`select lpad('abc',5,'xyzw') from dual`	xyabc

- **RPAD(sir,nr,subsir)** – similară cu funcţia **LPAD**, completarea făcându-se la dreapta.

Exemplu	Rezultatul afişat
`select rpad('abcd',3,'*') from dual`	abc
`select rpad('abcd',10,'*.') from dual`	abcd*.*.*.
`select rpad('abc',10,'*.') from dual`	abc*.*.*.*

`select rpad('abc',5,'xyzw') from dual`	`abcxy`

- **`TRIM(LEADING ch FROM sir)`**
  `TRIM(TRAILING ch FROM sir)`
  `TRIM(BOTH ch FROM sir)`
  `TRIM(sir)`
  `TRIM(ch FROM sir)`
  - funcţia **`TRIM`** şterge caracterele **`ch`** de la începutul, sfârşitul sau din ambele părţi ale şirului **`sir`**.
  - în ultimele două formate ale funcţiei este subînţeleasă opţiunea **`BOTH`**.
  - dacă **`ch`** nu este specificat se vor elimina spaţiile inutile de la începutul, sfârşitul sau din ambele părţi ale şirului **`sir`**.

Exemplu	Rezultatul afişat
`select trim(leading 'a' from 'aaxaxaa') from dual`	`xaxaa`
`select trim(trailing 'a' from 'aaxaxaa') from dual`	`aaxax`
`select trim(both 'a' from 'aaxaxaa') from dual`	`xax`
`select trim('a' from 'aaxaxaa') from dual`	`xax`
`select '*'||trim('  abc ')||'*' from dual`	`*abc*`

- **`REPLACE(sir,subsir,sirnou)`** - înlocuieşte toate apariţiile subşirului **`subsir`** din şirul **`sir`** cu şirul **`sirnou`**. Dacă nu este specificat noul şir, toate apariţiile subşirului **`subsir`** se vor elimina.

Exemplu	Rezultatul afişat
`select replace('abracadabra','ab','xy') from dual`	`xyracadxyra`
`select replace('abracadabra','ab','xyz') from dual`	`xyzracadxyzra`
`select replace('abracadabra','a') from dual`	`brcdbr`

Combinarea funcţiilor asupra şirurilor de caractere

Evident într-o expresie pot fi folosite două sau mai multe astfel de funcţii, imbricate ca în următorul exemplu.

```
SELECT substr('abcabcabc',1,instr('abcabcabc','bc')-1)||
```

```
    'xyz' ||
    substr('abcabcabc',instr('abcabcabc','bc')+length('bc'))
  FROM dual
```

Să analizăm pe această comandă

```
instr('abcabcabc','bc')
```

retunează poziţia primei apariţii a şirului '**bc**' în şirul '**abcabcabc** ', adică **2**. Primul apel al funcţiei **substr** este deci echivalent cu apelul

```
substr('abcabcabc',1,1)
```

adică extrage doar prima litera '**a**'. Al doilea apel al funcţiei substr este echivalent cu
```
substr('abcabcabc',4)
```
adică extrage toate caracterele de la poziţia 4 până la sfârşitul şirului, deci '**abcabc**'. Aşadar cele două apeluri extrag subşirul de dinaintea primei apariţii a lui '**bc**' în şirul '**abcabcabc**', şi respectiv de după această apariţie. Cele două secvenţe se concatenează apoi între ele incluzându-se şirul '**xyz**'. În concluzie comanda înlocuieşte prima apariţie a şirului '**bc**' din şirul '**abcabcabc**' cu şirul '**xyz**'.

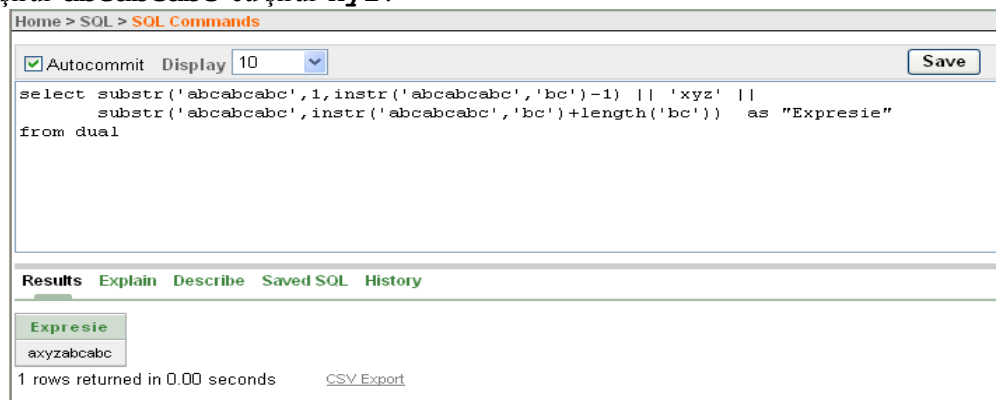


Figura II.2.2 Combinarea funcţiilor caracter

II.2.4. Funcţii numerice

Aceste funcţii operează asupra valorilor numerice şi returnează un rezultat numeric. Funcţiile numerice oferite de Oracle sunt destul de puternice.

- **ABS(n)** – returnează valoarea absolută a argumentului.

Exemplu	Rezultatul afişat
`select abs(-5.23) from dual`	5.23
`select abs(5) from dual`	5

- **ACOS(n), ASIN(n), ATAN(n)** – sunt funcţiile trigonometrice inverse, cu semnificaţia din matematică. Valoarea returnată de aceste funcţii este exprimată în radiani.
- **SIN(n), COS(n), TAN(n)** – sunt funcţiile trigonometrice cu aceeaşi semnificaţie ca şi la matematică. Argumentul acestor funcţii trebuie precizat în radiani.

Exemplu	Rezultatul afişat
`select sin(3.1415/2) from dual`	.999999998926914037495206086034346145374
`select cos(3.1415/2) from dual`	.000046326794880048353556705900494195 94

- **POWER(m,n)** – calculează valoarea m^n.

Exemplu	Rezultatul afişat
`select power(2,5) from dual`	32

`select power(2,0.5) from dual`	`1.41421356237309504880168872420969807855`
`select power(2,-1) from dual`	`.5`
`select power(2,-0.75) from dual`	`.594603557501360533358749985280237957651`

- **SQRT(x)** – calculează rădăcina pătrată a argumentului. Apelul **SQRT(x)** returnează aceeaşi valoare ca şi **POWER(x,0.5)**.

Exemplu	Rezultatul afişat
`select sqrt(3) from dual`	`1.732050807568877293527446341505872366 94`

- **REMAINDER(x,y)** – în cazul în care ambii parametrii **x** şi **y** sunt numere întregi, funcţia calculează restul împărţirii lui **x** la **y**. Dacă cel puţin unul dintre parametrii este număr real, funcţia determină mai întâi acel multiplu a lui **y** care este cel mai apropiat de **x**, şi returnează apoi diferenţa dintre **x** şi acel multiplu.

Exemplu	Rezultatul afişat
`select remainder(10,3) from dual`	`1` ***Explicaţie.*** Cel mai apropiat de `10` multiplu a lui `3` este `9`. `10-9=1`.
`select remainder(5,3) from dual`	`-1` ***Explicaţie.*** Cel mai apropiat de `5` multiplu a lui `3` este `6`, iar `5-6=-1`.
`select remainder(10,3.5) from dual`	`-0.5` ***Explicaţie.*** Cel mai apropiat de `10` multiplu a lui `3.5` este `10.5`, iar `10-10.5=-0.5`.
`select remainder(-10,3.5) from dual`	`0.5` ***Explicaţie.*** Cel mai apropiat de `-10` multiplu a lui `3.5` este `-10.5`, iar `-10-(-10.5)=0.5`.

- **MOD(x,y)** – dacă cei doi parametrii sunt numere întregi, atunci funcţia returnează acelaşi rezultat ca şi funcţia **REMAINDER**, adică restul împărţirii lui **x** la **y**. Teorema împărţirii cu rest este extinsă de această funcţie şi pentru numerele reale. Adică se ţine cont de relaţia

x=y * cât + rest

unde restul trebuie să fie în modul strict mai mic decât **y**.

Exemplu	Rezultatul afişat
`select mod(10,3) from dual`	`1` ***Explicaţie.*** `10=3*3+1`.
`select mod(5,3) from dual`	`2` ***Explicaţie.*** `5=3*1+2`
`select mod(10,3.5) from dual`	`3` ***Explicaţie.*** `10=3.5*2+3`.
`select mod(-10,3.5) from dual`	`-3` ***Explicaţie.*** `-10=3.5*(-2)-3`.
`select mod(-10,-3.5) from dual`	`-3` ***Explicaţie.*** `-10=-3.5*2-3`.
`select mod(10,-3.5) from dual`	`3` ***Explicaţie.*** `10=-3.5*(-2)+3`.

Se observă din exemplele anterioare că restul are întotdeauna acelaşi semn cu primul parametru.

- **SIGN(x)** – returnează semnul lui **x**, adică `1` dacă **x** este număr pozitiv, respectiv `-1` dacă **x** este număr negativ.
- **CEIL(x)** – returnează cel mai mic număr întreg care este mai mare sau egal decât parametrul transmis.

- **FLOOR(x)** – returnează cel mai mare număr întreg care este mai mic sau egal decât parametrul transmis.

Exemplu	Rezultatul afişat
`select ceil(3) from dual`	`3`
`select ceil(-3) from dual`	`-3`
`select ceil(-3.7) from dual`	`-3`
`select ceil(3.7) from dual`	`4`
`select floor(3) from dual`	`3`
`select floor(-3) from dual`	`-3`
`select floor(-3.7) from dual`	`-4`
`select floor(3.7) from dual`	`3`

- **ROUND(x,y)** – rotunjeşte valoarea lui **x** la un număr de cifre precizat prin parametrul **y**.
  Dacă al doilea parametru este un număr pozitiv, atunci se vor păstra din **x** primele **y** zecimale, ultima dintre aceste cifre fiind rotunjită, în funcţie de de următoarea zecimală.
  Al doilea argument poate fi o valoare negativă, rotunjirea făcându-se la stânga punctului zecimal. Cifra a **|y|+1** din faţa punctului zecimal (numărând de la punctul zecimal spre stânga începând cu **1**) va fi rotunjită în funcţie cifra aflată imediat la dreapta ei. Primele **|y|** cifre din stânga punctului zecimal vor deveni **0**.
  Cel de al doilea argument este opţional, în cazul în care nu se precizează, este considerată implicit valoarea **0**.

Exemplu	Rezultatul afişat
`select round(745.123,2) from dual`	`745.12`
`select round(745.126,2) from dual`	`745.13`
`select round(745.126,-1) from dual`	`750`
`select round(745.126,-2) from dual`	`700`
`select round(745.126,-3) from dual`	`1000`
`select round(745.126,-4) from dual`	`0`
`select round(745.126,0) from dual`	`745`
`select round(745.826,0) from dual`	`746`
`select round(745.826) from dual`	`746`

- **TRUNC(x)** – este asemănătoare cu funcţia **ROUND**, fără a rotunji ultima cifră.

Exemplu	Rezultatul afişat
`select trunc(745.123,2) from dual`	`745.12`
`select trunc(745.126,2) from dual`	`745.12`
`select trunc(745.126,-1) from dual`	`740`
`select trunc(745.126,-2) from dual`	`700`
`select trunc(745.126,-3) from dual`	`0`
`select trunc(745.126,-4) from dual`	`0`
`select trunc(745.126,0) from dual`	`745`

`select trunc(745.826,0) from dual`	`745`
`select trunc(745.826) from dual`	`745`

II.2.5. Funcţii asupra datelor calendaristice

Una dintre caracteristicile importante ale Oracle este abilitatea de a memora şi opera cu date calendaristice. Tipurile de date calendaristice recunoscute de Oracle sunt:

- **`DATE`** - valorile având acest tip sunt memorate într-un format intern specific, care include pe lângă ziua, luna şi anul, de asemenea ora, minutul, şi secunda.
- **`TIMESTAMP`** – valorile având acest tip memorează data calendaristică, ora, minutul şi secunda dar şi fracţiunea de secundă.
- **`TIMESTAMP WITH [LOCAL] TIME ZONE`** – este similar cu **`TIMESTAMP`**, însă se va memora şi diferenţa de fus orar faţă de ora universală, a orei de pe serverul bazei de date, sau a aplicaţiei client, în cazul în care se include opţiunea **`LOCAL`**.
- **`INTERVAL YEAR TO MONTH`** – memorează o perioadă de timp în ani şi luni.
- **`INTERVAL DAY TO SECOND`** – memorează un interval de timp în zile, ore, minute şi secunde.

Să exemplificăm aceste tipuri de date creând o tabelă de test cu comanda:

```
create table test3
  (data1 DATE, data2 TIMESTAMP(5),
   data3 TIMESTAMP(5) WITH TIME ZONE,
   data4 TIMESTAMP(5) WITH LOCAL TIME ZONE)
```

Vom insera acum o linie nouă în această tabelă:

```
insert into test3
  values(sysdate,systimestamp,systimestamp,systimestamp)
```

şi la afişarea tabelei

```
select * from test3
```

vom obţine rezultatul din figura II.2.3.

DATA1	DATA2	DATA3	DATA4
27-FEB-07	27-FEB-07 05.49.35.02886 AM	27-FEB-07 05.49.35.02886 AM -06:00	27-FEB-07 11.49.35.02886 AM

Figura II.2.3

Aritmetica datelor calendaristice

Oracle ştie să realizeze operaţii aritmetice asupra datelor calendaristice, astfel adăugarea valorii **`1`** la o dată calendaristică, va duce la obţinerea următoarei date calendaristice:

```
SELECT sysdate, sysdate+5, sysdate-70 from dual
```

SYSDATE	SYSDATE+5	SYSDATE-70
21-APR-07	26-APR-07	10-FEB-07

Figura II.2.4. Adunarea unui număr întreg la o dată calendaristică

De asemenea se poate face diferenţa dintre două date calendaristice, obţinându-se numărul de zile dintre cele două date:

```
SELECT first_name, last_name,
          hire_date, sysdate-hire_date
FROM employees
```

FIRST_NAME	LAST_NAME	HIRE_DATE	SYSDATE-HIRE_DATE
Steven	King	17-JUN-87	7248.18565972222222222222222222222222

Neena	Kochhar	21-SEP-89	6421.18565972222222222222222222222222
Lex	De Haan	13-JAN-93	5211.18565972222222222222222222222222
Alexander	Hunold	03-JAN-90	6317.18565972222222222222222222222222
…	…	…	…

Figura II.2.5. Diferenţa dintre două date calendaristice

Deşi implicit o dată calendaristică de tip **DATE** nu este afişată în format complet (nu se afişează ora, minutul, secunda), în tabelă se memorează complet. De aceea poate fi uneori derutant rezultatul unor operaţii aritmetice cu date calendaristice, după cum se vede în figura II.2.6. în care diferenţa dintre ziua de astăzi şi cea de ieri este de **1.187997**….

```
SELECT sysdate-TO_DATE('20-APR-07','dd-MON-yy') FROM dual
```

SYSDATE-TO_DATE('20-APR-07','DD-MON-YY')
1.187997685185185185185185185185185519

Figura II.2.6.

De ce se obţine acest lucru? Simplu, data de **20 aprilie** a fost precizată fără oră, aşadar a fost considerată implicit ora **00:00**. Iar **sysdate** ne-a furnizat data curentă incluzând şi ora. Aşadar de ieri de la ora **00:00** până astăzi la ora **12:32** a trecut mai mult de o zi.

Funcţii cu date calendaristice

Oracle oferă un număr foarte mare de funcţii care operează asupra datelor calendaristice, dar în cele ce urmează ne vom opri asupra celor mai importante dintre acestea.

- **SYSDATE** – returnează data şi ora curentă a serverului bazei de date.
- **CURRENT_DATE** – returnează data şi ora curentă a aplicaţiei client. Aceasta poate să difere de data bazei de date.
- **SYSTIMESTAMP** – returnează data în formatul **TIMESTAMP**.

```
select CURRENT_DATE, sysdate, systimestamp
from dual
```

CURRENT_DATE	SYSDATE	SYSTIMESTAMP
21-APR-07	21-APR-07	21-APR-07 04.33.32.445081 AM -05:00

Figura II.2.7. Funcţiile **SYSDATE**, **CURRENT_DATE** şi **SYSTIMESTAMP**

- **ADD_MONTHS(data,nrluni)** – adaugă un număr de luni la data curentă. Dacă al doilea parametru este un număr negativ, se realizează de fapt scăderea unui număr de luni din data precizată.

Exemplu	Rezultatul afişat
`select sysdate, ADD_MONTHS(sysdate,2) from dual`	`27-FEB-07 27-APR-07`
`select sysdate, ADD_MONTHS(sysdate,-2) from dual`	`27-FEB-07 27-DEC-07`

- **MONTHS_BETWEEN(data1,data2)** – determină numărul de luni dintre două date calendaristice precizate. Rezultatul returnat poate fi un număr real (vezi figura II.2.8). Dacă prima dată este mai mică (o dată mai veche) atunci rezultatul va un număr negativ.

```
select sysdate, hire_date,
    MONTHS_BETWEEN(sysdate, hire_date),
    MONTHS_BETWEEN(hire_date, sysdate)
```

```
from employees
```

SYSDATE	HIRE_DATE	MONTHS_BETWEEN(SYSDATE, HIRE_DATE)	MONTHS_BETWEEN(HIRE_DATE,SYSDATE)
21-APR-07	17-JUN-87	238.135216173835125448028673835125448029	-238.135216173835125448028673835125448029
21-APR-07	21-SEP-89	211	-211
21-APR-07	13-JAN-93	171.264248431899641577060931899641577061	-171.264248431899641577060931899641577061
21-APR-07	03-JAN-90	207.586829077060931899641577060931899642	-207.586829077060931899641577060931899642
21-APR-07	21-MAY-91	191	-191
…	…	…	…

Figura II.2.8. Funcţia **MONTHS_BETWEEN**

- **LEAST(data1,data2,…)** – determină cea mai veche (cea mai mică) dată dintre cele transmise ca parametru.
- **GREATEST(data1,data2,…)** – determină cea mai recentă (cea mai mare) dată dintre cele transmise ca parametru.

```
select hire_date,sysdate,
least(hire_date,sysdate),greatest(hire_date,sysdate)
from employees
```

HIRE_DATE	SYSDATE	LEAST(HIRE_DATE,SYSDATE)	GREATEST(HIRE_DATE,SYSDATE)
17-JUN-87	21-APR-07	17-JUN-87	21-APR-07
21-SEP-89	21-APR-07	21-SEP-89	21-APR-07
13-JAN-93	21-APR-07	13-JAN-93	21-APR-07
03-JAN-90	21-APR-07	03-JAN-90	21-APR-07
21-MAY-91	21-APR-07	21-MAY-91	21-APR-07
…	…	…	…

Figura II.2.9. Funcţiile **LEAST** şi **GEATEST**

- **NEXT_DAY(data, 'ziua')** – returnează următoarea dată de **'ziua'** de după data transmisă ca parametru, unde **'ziua'** poate fi **'Monday'**, **'Tuesday'** etc. În exemplele care urmează data curentă este considerată ziua de marţi, 27 februarie 2007.
- **LAST_DAY(data)** – returnează ultima zi din luna din care face parte data transmisă ca parametru.

Exemplu	Rezultatul afişat
`select next_day(sysdate,'Friday') from dual`	`02-MAR-07`
`select next_day(sysdate,'TUESDAY') from dual`	`06-MAR-07` ***Explicaţie.*** Chiar dacă ziua curentă este o zi de marţi, funcţia va returna următoarea zi de marţi.
`select last_day(sysdate) from dual`	`28-FEB-07`

select last_day(sysdate+20) from dual	**31-MAR-07**
select last_day(ADD_MONTHS(sysdate,12)) from dual	**29-FEB-07** ***Explicaţie.*** Ziua returnată de **sysdate** este **27-FEB-07**, la care adăugăm **12** luni, deci obţinem data de **27-FEB-08**, iar anul **2008** este un an bisect de aceea ultima zi din lună este **29-FEB-08**.

- **ROUND(data,'format')** – dacă nu se precizează formatul, funcţia rotunjeşte data transmisă ca parametru la cea mai apropiată oră **12 AM**, adică dacă ora memorată în **data** este înainte de miezul zilei atunci se va returna ora **12 AM** a datei transmise. Dacă ora memorată în **data** este după miezul zilei se va returna ora **12 AM** a zilei următoare.

```
select to_char(sysdate,'dd-MON-YY hh:mi AM'),
       round(sysdate) from dual
```

TO_CHAR(SYSDATE,'DD-MON-YYHH:MIAM')	ROUND(SYSDATE)
21-APR-07 04:41 AM	21-APR-07

Figura II.2.10. Funcţia **ROUND**

În cazul în care este specificat formatul, data va fi rotunjită conform formatului indicat. Câteva dintre formatele cele mai uzuale sunt:

- **y**, **yy**, **yyyy**, **year** – se rotunjeşte data la cea mai apropiată dată de 1 Ianuarie. Dacă data este înainte de 1 iulie, se va returna data de 1 ianuarie a aceluiaşi an. Dacă data este după data de 1 iulie se va returna data de 1 ianuarie a anului următor.
- **mm**, **month** – rotunjeşte data la cel mai apropiat început de lună. Orice dată calendaristică aflată după data de **16**, inclusiv, este rotunjită la prima zi a lunii următoare.
- **ww**, **week** – se rotunjeşte data la cel mai apropiat început de săptămână. Prima zi a săptămânii este considerată lunea. Pentru datele aflate după ziua de joi, inclusiv, se va returna ziua de luni a săptămânii următoare.

Exemplu	Rezultatul afişat
select sysdate, round(sysdate,'year'), round(ADD_MONTHS(sysdate,5),'year') from dual	27-FEB-07 01-JAN-07 01-JAN-08
select sysdate, round(sysdate,'mm'), round(sysdate+16,'mm'), round(sysdate+17,'mm') from dual	27-FEB-07 01-MAR-07 01-MAR-07 01-APR-07
select sysdate, round(sysdate,'ww'), round(sysdate+1,'ww'), round(sysdate+2,'ww') from dual	27-FEB-07 26-FEB-07 26-FEB-07 05-FEB-07

- **TRUNC(data,'format')** – trunchează data specificată conform formatului specificat. Se pot folosi aceleaşi formate ca şi în cazul funcţiei **ROUND**.

Exemplu	Rezultatul afişat
select sysdate, trunc(sysdate,'year'), trunc(ADD_MONTHS(sysdate,5),'year')	27-FEB-07 01-JAN-07 01-JAN-07

from dual	
select sysdate, trunc(sysdate,'month'), trunc(sysdate+16,'month'), trunc(sysdate+17,'month') from dual	27-FEB-07 01-FEB-07 01-MAR-07 01-MAR-07
select sysdate, trunc(sysdate,'ww'), trunc(sysdate+1,'ww'), trunc(sysdate+2,'ww') from dual	27-FEB-07 26-FEB-07 26-FEB-07 26-FEB-07

II.2.6. Funcţii de conversie

Oracle oferă un set bogat de funcţii care vă permit să transformaţi o valoare dintr-un tip de dată în altul.

Transformarea din dată calendaristică în şir de caractere

Transformarea unei date calendaristice în şir de caractere se poate realiza cu ajutorul funcţiei **TO_CHAR**. Această operaţie se poate dovedi utilă atunci când dorim obţinerea unor rapoarte cu un format precis. Sintaxa acestei funcţii este:

TO_CHAR (dt, format)

dt poate avea unul din tipurile pentru date calendatistice (**DATE**, **TIMESTAMP**, **TIMESTAMP WITH TIME ZONE**, **TIMESTAMP WITH LOCAL TIME ZONE**, **INTERVAL MONTH TO YEAR**, or **INTERVAL DAY TO SECOND**). Formatul poate conţine mai mulţi parametrii care pot afecta modul în care va arăta şirul returnat. Câţiva din aceşti parametrii sunt prezentaţi în continuare.

Aspect	Parametru	Descriere	Examplu
Secolul	**CC**	Secolul cu două cifre	**21**
Trimestrul	**Q**	Trimestrul din an în care se găseşte data	**3**
Anul	**YYYY**, **RRRR**	Anul cu patru cifre.	**2006**
	YY, **RR**	Ultimele două cifre din an.	**06**
	Y	Ultima cifră din an	**6**
	YEAR, **Year**	Numele anului	**TWO THOUSAND-SIX, Two Thousand-Six**
Luna	**MM**	Luna cu două cifre	**02**
	MONTH, **Month**	Numele complet al lunii.	**JANUARY**, **January**
	MON, **Mon**	Primele trei litere ale denumirii lunii.	**JAN, Jan**
	RM	Luna scrisă cu cifre romane.	**IV**
Săptămâna	**WW**	Numărul săptămânii din an.	**35**
	W	Ultima cifră a numărului săptămânii din an.	**2**
Ziua	**DDD**	Numărul zilei din cadrul anului.	**103**
	DD	Numărul zilei în cadrul lunii	**31**
	D	Numărul zilei în cadrul săptămânii.	**5**
	DAY, **Day**	Numele complet al zilei din săptămână	**SATURDAY, Saturday**
	DY, **Dy**	Prescurtarea denumirii zilei din săptămână.	**SAT**, **Sat**
Ora	**HH24**	Ora în formatul cu **24** de ore.	**23**
	HH	Ora în formatul cu **12** ore.	**11**
Minutele	**MI**	Minutele cu două cifre	**57**
Secundele	**SS**	Secundele cu două cifre	**45**
Sufixe	**AM** sau **PM**	**AM** sau **PM** după cum e cazul.	**AM**
	A.M. sau **P.M.**	**A.M.** sau **P.M.** după cum e cazul.	**P.M.**
	TH	Sufix pentru numerale (**th** sau **nd** sau **st**)	
	SP	Numerele sunt scrise în cuvinte.	

În cadrul formatului se pot folosi oricare dintre următorii separatori

- / , . ; :

Dacă în şirul returnat dorim să includem şi anumite texte acestea se vor include între ghilimele.
Iată în continuare şi câteva exemple de folosire a acestei funcţii.

Exemplu	Rezultatul afişat
select sysdate, to_char(sysdate,'MONTH DD, YYYY') to_char(sysdate,'Month DD, YYYY') to_char(sysdate,'Mon DD, YYYY') from dual	**28-FEB-07 FEBRUARY 28, 2007 February 28, 2007 Feb 28, 2007**
select to_char(sysdate,'"Trimestrul "Q "al anului " Year') from dual	**Trimestrul 1 al anului Two Thousand Seven**
select to_char(sysdate,'"Secolul "CC')	**Secolul 21**

Exemplu	Rezultatul afişat
`from dual`	
`select to_char(sysdate,'Day, dd.RM.YYYY') from dual`	`Wednesday, 28.II.2007`
`select to_char(sysdate,'Dy, D, DD, DDD') from dual`	`Wed, 4, 28, 059`
`select to_char(sysdate,'HH24:MI/HH:MI AM') from dual`	`21:53/09:53 PM`
`select to_char(sysdate+1,'ddth') from dual`	`01st`
`select to_char(sysdate+1,'ddspth') from dual`	`first`
`select to_char(sysdate+2,'Ddspth') from dual`	`Second`
`select to_char(sysdate+10,'DDspth') from dual`	`TENTH`
`select to_char(sysdate,'mmsp') from dual`	`two`

Transformarea din şir de caractere în dată calendaristică

Folosind funcţia **TO_DATE** se poate transforma un şir de caractere precum **'May 26, 2006'** într-o dată calendaristică. Sintaxa funcţiei este:

TO_DATE(sir,format)

Formatul nu este obligatoriu, însă dacă nu este precizat, şirul trebuie să respecte formatul implicit al datei calendaristice **DD-MON-YYYY** sau **DD-MON-YY**. Formatul poate folosi aceiaşi parametrii de format ca şi funcţia **TO_CHAR**.

Exemplu	Rezultatul afişat
`select to_date('7.4.07', 'MM.DD.YY') from dual;`	`04-JUL-07`
`select to_date('010101','ddmmyy') from dual`	`01-JAN-01`

Formatul RR şi formatul YY

Aşa cum s-a precizat anterior în formatarea unei date calendaristice se pot folosi pentru an atât **YY** (respectiv **YYYY**) cât şi **RR** (respectiv **RRR**). Diferenţa dintre aceste două formate este modul în care ele interpretează anii aparţinând de secole diferite. Oracle memorează toate cele patru cifre ale unui an, dar dacă sunt transmise doar două din aceste cifre, Oracle va interpreta secolul diferit în cazul celor două formate.
Vom începe printr-un exemplu:

```
select to_char(to_date('05-FEB-95','DD-MON-YY'),
                'DD-MON-YYYY') as "YY Format",
```

```
        to_char(to_date('05-FEB-95','DD-MON-RR'),
                'DD-MON-RRRR') as "RR Format"
from dual
```

YY Format	RR Format
05-FEB-2095	05-FEB-1995

Figura II.2.11. Formatele **YY** şi **RR**

Se observă modul diferit de interpretare a anului.
Dacă utilizaţi formatul **YY** şi anul este specificat doar prin două cifre, se presupune că anul respectiv face parte din acelaşi secol cu anul curent. De exemplu, dacă anul transmis este **15** iar anul curent este **2007**, atunci anul transmis este interpretat cu **2015**. De asemenea **75** interpretat ca **2075**.

```
select to_char(to_date('15','yy'),'yyyy'),
       to_char(to_date('75','yy'),'yyyy')
from dual
```

TO_CHAR(TO_DATE('15','YY'),'YYYY')	TO_CHAR(TO_DATE('75','YY'),'YYYY')
2015	2075

Figura II.2.12. Formatul **YY**

Dacă folosiţi formatul **RR** şi anul transmis este de două cifre, primele două cifre ale anului transmis este determinat în funcţie de cele două cifre transmise şi de ultimele două cifre ale anului curent. Regulile după care se determină secolul datei transmise sunt următoarele:
Regula 1: Dacă anul transmis este între **00** şi **49**, şi ultimele două cifre ale anului curent sunt între **00** şi **49** atunci secolul este acelaşi cu secolul anului curent. De exemplu dacă anul transmis este **15** iar anul curent este **2007**, anul transmis este interpretat ca fiind **2015**.
Regula 2: Dacă anul transmis este între **50** şi **99** iar anul curent este între **00** şi **49** atunci secolul este secolul prezent minus **1**. De exemplu dacă transmiteţi **75** iar anul curent este **2007**, anul transmis este interpretat ca fiind **1975**.
Regula 3: Dacă anul transmis este între **00** and **49** iar anul prezent este între **50** şi **99**, secolul este considerat secolul prezent plus 1. De exemplu dacă aţi transmis anul **15** iar anul curent este **1987**, anul transmis este considerat ca fiind anul **2015**.
Regula 4: Dacă anul transmis este între **50** şi **99**, iar anul curent este între **50** şi **99**, secolul este acelaşi cu a anului curent. De exemplu, dacă transmiteţi anul **55** iar anul prezent ar fi **1987**, atunci anul transmis este considerat ca fiind anul **1955**.

```
select to_char(to_date('04-JUL-15','DD-MON-RR'),
                'DD-MON-YYYY') as dt1,
       to_char(to_date('04-JUL-75','DD-MON-RR'),
                'DD-MON-YYYY') as dt2
from dual
```

DT1	DT2
04-JUL-2015	04-JUL-1975

Figura II.2.13. Formatul **RR**

Transformarea din număr în şir de caractere

Pentru a transforma un număr într-un şir de caractere, se foloseşte funcţia **TO_CHAR**, cu următoarea sintaxă:

```
TO_CHAR(numar,format)
```

Formatul poate conţine unul sau mai mulţi parametrii de formatare dintre cei prezentaţi în tabelul următor.

Parametru	Exemplu de format	Descriere

Parametru	Exemplu de format	Descriere
9	**999**	Returnează cifrele numărului din poziţiile specificate, precedat de semnul minus dacă numărul este negativ
0	**0999**	Completează cifrele numărului cu zerouri în faţă
.	**999.99**	Specifică poziţia punctului zecimal
,	**9,999**	Specifică poziţia separatorului virgulă
$	**$999**	Afişează semnul dolar
EEEE	**9.99EEEE**	Returnează scrierea ştiinţifică a numărului.
L	**L999**	Afişează simbolul monetar.
MI	**999MI**	Afişează semnul minus după număr dacă acesta este negativ.
PR	**999PR**	Numerele negative sunt închise între paranteze unghiulare.
RN **rn**	**RN** **rn**	Afişează numărul în cifre romane.
V	**99V99**	Afişează numărul înmulţit cu **10** la puterea **x**, şi rotunjit la ultima cifră, unde **x** este numărul de cifre **9** de după **V**.
X	**XXXX**	Afişează numărul în baza **16**..

Vom exemplifica în continuare câteva dintre aceste formate.

Exemplu	Rezultatul afişat
`select to_char(123.45,'9999.99') from dual`	`123.45`
`select to_char(123.45,'0000.000') from dual`	`0123.450`
`select to_char(123.45,'9.99EEEE') from dual`	`1.23E+02`
`select to_char(-123.45,'999.999PR') from dual`	`<123.450>`
`select to_char(1.2373,'99999V99') from dual`	`124`
`select to_char(1.2373,'L0000.000') from dual`	`$0001.237`
`select to_char(4987,'XXXXXX') from dual`	`137B`
`select to_char(498,'RN') from dual`	`CDXCVIII`

Transformarea şir de caractere în număr

Transformarea inversă din şir de caractere într-o valoare numerică se realizează cu ajutorul funcţiei **TO_NUMBER**:

TO_NUMBER(sir,format)

Parametrii de formatare ce se pot folosi sunt aceeaşi ca în cazul funcţiei **TO_CHAR**. Iată câteva exemple.

Exemplu	Rezultatul afişat
`select to_number('970.13') + 25.5 FROM dual`	`995.63`
`select`	`-12345.67`

Exemplu	Rezultatul afişat
`to_number('-$12,345.67','$99,999.99') from dual;`	

II.2.7. Funcţii de uz general

Pe lângă funcţiile care controlează modul de formatare sau conversie al datelor, Oracle oferă câteva funcţii de uz general, care specifică modul în care sunt tratate valorile **NULL**.

- **NVL(val1,val2)** – funcţia returnează valoarea **val1**, dacă aceasta este nenulă, iar dacă **val1** este **NULL** atunci va returna valoarea **val2**. Funcţia **NVL** poate lucra cu date de tip caracter, numeric sau dată calendaristică, însă este obligatoriu ca cele două valori să aibă acelaşi tip.

```
select first_name, commission_pct, NVL(commission_pct,0.8)
from employees
where employee_id between 140 and 150
```

rezultatul returnat de această comandă este cel din figura II.2.14.

FIRST_NAME	COMMISSION_PCT	NVL(COMMISSION_PCT,0.8)
Trenna	-	.8
Curtis	-	.8
Randall	-	.8
Peter	-	.8
Eleni	.2	.2

Figura II.2.14. Funcţia **NVL**

- **NVL2(val1,val2,val3)** – dacă valoarea **val1** nu este nulă atunci funcţia va returna valoarea **val2**, iar dacă **val1** are valoarea **NULL** atunci funcţia va returna valoarea **val3** (vezi figura II.2.15.).

```
select first_name, commission_pct,
       NVL2(commission_pct,'ARE','NU ARE')
from employees where employee_id between 140 and 150
```

FIRST_NAME	COMMISSION_PCT	NVL2(COMMISSION_PCT,'ARE','NUARE')
Trenna	-	NU ARE
Curtis	-	NU ARE
Randall	-	NU ARE
Peter	-	NU ARE
Eleni	.2	ARE

Figura II.2.15 Funcţia **NVL2**

- **NULLIF(expr1,expr2)** – dacă cele două expresii sunt egale, funcţia returnează **NULL**. Dacă valorile celor două expresii sunt diferite atunci funcţia va returna valoarea primei expresii (vezi figura II.2.16.).

```
select employee_id, first_name, last_name,
       NULLIF(length(first_name),length(last_name))
from employees where employee_id between 103 and 142
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	NULLIF(LENGTH(FIRST_NAME),LENGTH(LAST_NAME))
103	Alexander	Hunold	9

104	Bruce	Ernst	-
107	Diana	Lorentz	5
124	Kevin	Mourgos	5
141	Trenna	Rajs	6
142	Curtis	Davies	-
…	…	…	…

Figura II.2.16 Funcţia **NULLIF**

- **COALESCE(expr1, expr2, ..., exprn)** – funcţia returnează valoarea primei expresii nenule (vezi figura II.2.17).

```
select coalesce(null, null, '33', 'test') from dual
```

COALESCE(NULL,NULL,'33','TEST')
33

Figura II.2.17 Funcţia **COALESCE**

II.2.8 Funcţii şi expresii condiţionale

Oracle SQL oferă posibilitatea de a construi expresii alternative asemănătoare structurilor **IF-THEN-ELSE** prezente în alte limbaje.

- **DECODE(expresie, val11, val12, val21, val22, ..., valn1, valn2, val) –** această compară valoarea expresiei cu valorile **val11**, **val21**, ..., **valn1**. Dacă valoarea expresiei este egală cu valoarea **vali1**, atunci funcţia va returna valoarea **vali2**. Dacă funcţia nu este egală cu nici una din valorile **vali1**, atunci funcţia va returna valoarea **val**.

```
select DECODE('Maria' ,'Dana', 'Ea este Ana' ,
                       'Maria','Ea este Maria' ,
                       'Nu e nici Ana nici Maria')
from dual
```

  această comandă va afişa mesajul "**Ea este Maria**" însă următoarea comandă va afişa "**Nu e nici Ana nici Maria**".

```
select DECODE('Valeria' ,'Dana', 'Ea este Ana' ,
                       'Maria','Ea este Maria' ,
                       'Nu e nici Ana nici Maria')
from dual
```

- În locul funcţiei **DECODE** se poate folosi expresia condiţională **CASE**. Funcţia **CASE** utilizează cuvintele cheia **when**, **then**, **else**, şi **end** pentru a indica ramura selectată. În general orice apel al funcţiei **DECODE** poate fi transcris folosind funcţia **CASE**. Chiar dacă o expresie folosind **CASE** este mai lungă decât expresia echivalentă care foloseşte funcţia **DECODE**, varianta cu **CASE** este mult mai uşor de citit şi greşelile sunt depistate mai uşor. În plus varianta **CASE** este compatibilă **ANSI-SQL**.

  Cele două comenzi de mai sus por fi transcrise cu ajutorul funcţiei **CASE** astfel:

```
select CASE 'Maria'
         WHEN 'Dana' THEN 'Ea este Ana'
         WHEN 'Maria' THEN 'Ea este Maria'
         ELSE 'Nu e nici Ana nici Maria'
       END
from dual
```

```
select CASE 'Valeria'
        WHEN 'Dana' THEN 'Ea este Ana'
        WHEN 'Maria' THEN 'Ea este Maria'
        ELSE 'Nu e nici Ana nici Maria'
                                END
from dual
```

3.Interogari multiple

În capitolele anterioare am aflat cum putem afişa informaţii din baza de date, însă la fiecare rulare a unei comenzi **SELECT** am afişat date dintr-o singură tabelă.
Unul dintre rezultatele procesului de normalizare este acela că datele sunt memorate, de cele mai multe ori, în tabele diferite. De aceea, la afişarea diferitelor rapoarte va trebui să puteţi prelua date din mai multe tabele printr-o singură comandă SQL.
Din fericire SQL oferă facilităţi pentru combinarea datelor din mai multe tabele şi afişarea lor într-un singur raport. O astfel de operaţie se numeşte **join**, sau **interogare multiplă**.
Pe parcursul acestui capitol vom folosi ca exemple tabela **Persoane** a cărei cheie primară este atributul **IdPersoana**, tabela **Firme** a cărei cheie primară este atributul **IdFirm**, şi tabela **Joburi** cu cheia primară **IdJob**. Presupunem că aceste tabele conţin următoarele înregistrări:

Tabelul II.3.1. Tabela **Persoane**

IDPERSOANA	NUME	PRENUME	LOCALITATE	IDFIRM	IDJOB
1	Ionescu	Gheorghe	Brasov	22	5
2	Vasilescu	Vasile	Cluj-Napoca	15	1
3	Popescu	Ioan	Bucuresti	10	2
4	Georgescu	Maria	Iasi	30	6
5	Marinescu	Angela	Sibiu	-	3
6	Antonescu	Elena	Sibiu	10	1
7	Bischin	Paraschin	Brasov	15	-
8	Olaru	Angela	Ploiesti	22	2

Tabelul II.3.2. Tabela **Firme**

IdFirm	Nume	Localitate
10	SC Crisib SA	Sibiu
15	SC SoftCom	Alba Iulia
20	SC TimTip	Timisoara
22	Brasoveanca	Brasov

Tabelul II.3.3. Tabela **Joburi**

IdJob	Nume
1	Reprezentant Vanzari
2	Manager
6	Operator IT
3	Programator
4	Administrator
5	Administrator retea

În Oracle există două moduri diferite de a scrie joinurile:

- Prima metodă foloseşte sintaxa specifică Oracle. În acest caz condiţiile de join sunt incluse în clauza **WHERE**. Această metodă este mai uşor de înţeles, însă are dezavantajul că în aceeaşi clauză **WHERE** se includ atât condiţiile de filtrare a înregistrărilor afişate cât şi condiţiile de join.
- A doua variantă foloseşte sintaxa ANSI/ISO, care este puţin mai greoaie, însă comenzile scrise folosind această sintaxă sunt portabile şi în alte SGBD-uri care folosesc limbajul SQL.

Indiferent de sintaxa folosită există mai multe moduri de legare a tabelelor şi anume:

- **Produsul cartezian** – leagă fiecare înregistrare dintr-o tabelă cu toate înregistrările din cealaltă tabelă.

- **Equijoin** – sunt legate două tabele cu ajutorul unei condiţii de egalitate
- **NonEquijoin** - în acest caz condiţia de join foloseşte alt operator decât operatorul de egalitatea
- **SelfJoin** – este legată o tabelă cu ea însăşi, e folosită de obicei în conjuncţie cu relaţiile recursive.
- **OuterJoin** – sunt o extensie a equijoinului, când pentru unele înregistrări dintr-o tabelă nu există corespondent în cealaltă tabelă, şi dorim ca aceste înregistrări fără corespondent să fie totuşi afişate.

II.3.1. Produsul cartezian

a) Sintaxa Oracle

După cum am precizat, acest tip de legătură între două tabele, va lega fiecare rând din prima tabelă cu fiecare rând din cea de a doua tabelă. De exemplu comanda:

```
    SELECT p.nume, p.prenume, f.nume
FROM persoane p, firme f
```

Va afişa următoarele informaţii

Tabelul II.3.4. Produsul cartezian între tabelele **`Persoane`** şi **`Firme`**

Nume	Prenume	Nume
Ionescu	Gheorghe	SC Crisib SA
Vasilescu	Vasile	SC Crisib SA
Popescu	Ioan	SC Crisib SA
Georgescu	Maria	SC Crisib SA
Marinescu	Angela	SC Crisib SA
Antonescu	Elena	SC Crisib SA
Bischin	Paraschin	SC Crisib SA
Olaru	Angela	SC Crisib SA
Ionescu	Gheorghe	SC SoftCom
Vasilescu	Vasile	SC SoftCom
Popescu	Ioan	SC SoftCom
Georgescu	Maria	SC SoftCom
Marinescu	Angela	SC SoftCom
Antonescu	Elena	SC SoftCom
Bischin	Paraschin	SC SoftCom
Olaru	Angela	SC SoftCom
Ionescu	Gheorghe	SC TimTip
Vasilescu	Vasile	SC TimTip
Popescu	Ioan	SC TimTip
Georgescu	Maria	SC TimTip
Marinescu	Angela	SC TimTip
Antonescu	Elena	SC TimTip
Bischin	Paraschin	SC TimTip
Olaru	Angela	SC TimTip
Ionescu	Gheorghe	Brasoveanca
Vasilescu	Vasile	Brasoveanca
Popescu	Ioan	Brasoveanca
Georgescu	Maria	Brasoveanca
Marinescu	Angela	Brasoveanca
Antonescu	Elena	Brasoveanca
Bischin	Paraschin	Brasoveanca
Olaru	Angela	Brasoveanca

adică se obţin 8x4 = 32 înregistrări (tabela persoane conţine 8 înregistrări, tabela firme 4 înregistrări)

De remarcat că notaţia **`p.nume`**, **`p.prenume`**, **`f.nume`**, precum şi literele **`p`** şi **`f`** care urmează după numele tabelelor din clauza **`FROM`**. Spunem că am definit un alias al fiecărei tabele. Am fost nevoiţi să folosim acest alias, deoarece în ambele tabele există o coloană cu numele **`nume`** şi dacă nu prefaţăm numele acestei coloane cu aliasul tabelei se va genera o ambiguitate pe care serverul bazei de date nu va şti să o rezolve. Aliasul tabelei este obligatoriu să-l folosim când două tabele conţin coloane cu acelaşi nume. În exemplul anterior coloana prenume nu este obligatoriu să o prefaţăm cu aliasul coloanei, astfel comanda anterioară poate fi scrisă şi astfel:

```
    SELECT p.nume, prenume, f.nume
FROM persoane p, firme f
```

Aşadar, produsul cartezian apare atunci când nu este precizată nici o condiţie privind modul de legare al celor două tabele.

b) Sintaxa ANSI

Pentru a obţine produsul cartezian, în sintaxa ANSI vom folosi clauza **CROSS JOIN** în cadrul clauzei **FROM** ca în exemplul următor.

```
    SELECT p.nume, p.prenume, f.nume
FROM persoane p CROSS JOIN firme f
```

Rezultatul obţinut va coincide cu cel obţinut anterior.

II.3.2. Equijoin

Oare cum procedăm dacă dorim să afişăm pentru fiecare persoană, numele firmei la care lucrează? Să vedem de exemplu cum aflăm numele firmei la care lucrează Ionescu Gheorghe. Ne uităm în tabela **persoane**, la valoarea din coloana **IdFirm**. Această valoare este **22.** Apoi, în tabela **firme** căutăm firma având codul **22**, şi preluăm numele acestei firme din coloana **nume**. Acest nume este Brasoveanca. Aşadar Ionescu Gheorghe lucrează la firma Brasoveanca. Deci a trebuit ca valoarea din coloana **IdFirm** din tabela **Persoane** să coincidă cu valoarea coloanei **IdFirm** din tabela Firme.

a) Sintaxa Oracle

Cum realizăm acest lucru folosind SQL? Simplu. Vom preciza condiţia de egalitate dintre coloanele **IdFirm** din cele două tabele în clauza **WHERE** ca mai jos:

```
    SELECT p.nume, prenume, f.nume
    FROM persoane p, firme f
WHERE p.idfirm = f.idfirm
```

Tabelul II.3.5. Equijoin între tabelele **Persoane** şi **Firme**

Nume	Prenume	Nume
Ionescu	Gheorghe	Brasoveanca
Vasilescu	Vasile	SC SoftCom
Popescu	Ioan	SC Crisib SA
Antonescu	Elena	SC Crisib SA
Bischin	Paraschin	SC SoftCom
Olaru	Angela	Brasoveanca

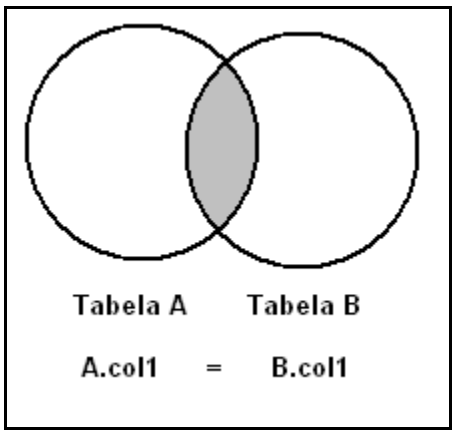


Figura II.3.1. Equijoin

Bineînţeles că în condiţia de equijoin pot fi precizate mai multe condiţii. Dacă de exemplu tabelele **elevi** şi **note** ar conţine următoarele coloane:

```
Elevi (#nume, #prenume, *adresa)

Note(#nume, #prenume, #disciplina, #data, *nota)
```

atunci pentru a afişa toate notele unui elev vom folosi comanda:

```
    SELECT a.nume, a.prenume,
           b.disciplina, b.data, b.nota
FROM elevi a, firme b
WHERE a.nume=b.nume AND a.prenume=b.prenume
```

b) Sintaxa ANSI

În cazul sintaxei ANSI lucrurile se complică uşor. În principal equijoinul se realizează folosind opţiunea **NATURAL JOIN** în cadrul clauzei **from** astfel:

```
SELECT nume, prenume, nume
FROM persoane NATURAL JOIN firme
```

Însă dacă rulăm această comandă vom fi surprinşi că ea nu afişează nici o linie. De ce? Pentru că **NATURAL JOIN**-ul leagă cele două tabele pe toate coloanele cu nume comun din cele două tabele. Adică, comanda anterioară este echivalentă cu următoarea comandă scrisă folosind sintaxa Oracle:

```
    SELECT p.nume, prenume, f.nume
    FROM persoane p, firme f
WHERE p.idfirm = f.idfirm AND p.nume=f.nume
```

ori nu are nici un sens să punem condiţia ca numele firmei (**f.nume**) să coincidă cu numele persoanei (**p.nume**).

Reguli de folosire a opţiunii **NATURAL JOIN**:

- tabelele sunt legate pe toate coloanele cu nume comun
- coloanele cu nume comun trebuie să aibă acelaşi tip
- în clauza **SELECT** coloanele comune celor două tabele NU vor fi prefaţate de aliasul tabelei.

Pentru a lega două tabele folosind sintaxa ANSI dar condiţia de egalitate să fie pusă doar pe anumite coloane (nu pe toate coloanele cu nume comun ci doar pe o parte din acestea) se va folosi în loc de **NATURAL JOIN** clauza **JOIN**, iar coloanele pe care se face joinul se precizează în opţiunea **USING**. Astfel comanda pentru afişarea firmelor la care lucrează fiecare angajat se scrie astfel:

```
    SELECT p.nume, prenume, f.nume
    FROM personae p JOIN firme f
USING (IdFirm)
```

Restricţii la folosirea clauzei **JOIN** cu clauza **USING**:

- în clauza **USING** se trec în paranteză, separate prin virgulă, numele coloanelor pe care se va face joinul
- coloanele din clauza **USING** trebuie să aibă acelaşi tip în cele două tabele

Dacă în cele două tabele există nu există coloane cu acelaşi nume, sau coloanele cu nume comun au tipuri diferite în cele două tabele, se va folosi clauza **JOIN** în conjuncţie cu **ON.** În clauza **ON** pe poate trece orice condiţie de join între cele două tabele.

```
SELECT p.nume, prenume,
       f.nume
FROM persoane p JOIN firme f
ON (p.IdFirm=f.IdFirm)
```

Rezultatul obţinut este acelaşi cu cel din tabelul II.3.5.

II.3.3. Nonequijoin

a) Sintaxa Oracle

Să presupunem că în tabela **Note** avem trecute mai multe note ale elevilor unei şcoli. Structura tabelei este

```
Note(#nume, #prenume, #disciplina, #data, *nota)
```

Dorim să înlocuim notele cu calificative, şi ştim de exemplu că notele de 9 şi 10 sunt transformate în calificativul **FOARTE BINE**, notele de **7** şi **8** în **BINE** etc. Aceste echivalenţe sunt memorate în tabela **CALIFICATIVE** cu structura următoare

```
CALIFICATIVE(#id, *nota1, *nota2, *calificativ)
```

cu semnificaţia că notele cuprinse între notele **nota1** şi **nota2**, inclusiv, se vor transforma în **calificativ**.

Pentru a scrie calificativele corespunzătoare fiecărei note din tabela **note**, vom scrie următoarea comandă:

```
SELECT nume, prenume, disciplina, data, calificativ
FROM note, calificative
```

```
WHERE nota BETWEEN nota1 AND nota2
```

b) Sintaxa ANSI

Echivalent vom scrie:

```
SELECT nume, prenume, disciplina, data, calificativ
FROM note JOIN calificative
ON (nota BETWEEN nota1 AND nota2)
```

II.3.4. Self Join

Ţinând cont de faptul că SelfJoin-ul este de fapt un equijoin dintre o tabela şi ea însăşi, lucrurile sunt mult mai simple. Considerăm de exemplu tabela **angajaţi** cu următoarea structură:

Angajaţi (#id, *nume, *prenume, *id_manager)

în câmpul id_manager memorându-se codul şefului fiecărui angajat.

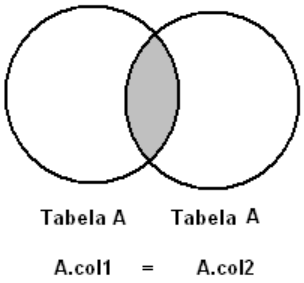


Figura II.3.2. SelfJoin

Dorim să afişăm numele fiecărui angajat şi numele şefului acestuia. Vom folosi următoarele comenzi:

a) Sintaxa Oracle

```
SELECT a.nume ||' '|| a.prenume AS "Angajat",
       b.nume ||' '|| b.prenume AS "Sef"
FROM angajat a, angajat b
WHERE a.id_manager = b.id
```

adică vom privi tabela **angajaţi** o dată ca tabelă de angajaţi (a) şi apoi ca tabelă de manageri.

b) Sintaxa ANSI

```
SELECT a.nume ||' '|| a.prenume AS "Angajat",
       b.nume ||' '|| b.prenume AS "Sef"
FROM angajat a JOIN angajat b
ON (a.id_manager = b.id)
```

II.3.5. OuterJoin

Să privim pentru început la tabelul II.3.5, rezultatul rulării unei comenzi de equijoin. Se poate observa că lipsesc din acest tabel două persoane: **Georgescu** şi **Marinescu**. De ce oare? Se poate vedea în tabelele II.3.1 şi II.3.2 că **Georgescu** nu lucrează încă la nici o firmă, iar Marinescu este asignat unui firme care nu există (poate încă nu există sau a fost desfiinţată). Deci pentru aceşti doi angajaţi nu se poate găsi nici o înregistrare în tabela **Firme** pentru care condiţia de equijoin să fie îndeplinită, şi de aceea nu sunt afişaţi.

Dacă dorim totuşi să afişăm toţi angajaţii din tabela persoane, indiferent dacă lucrează sau nu la o firmă, va trebui să putem suplini cumva această lipsă de informaţii.

Pentru a indica lipsa de informaţii dintr-o tabelă, vom folosi secvenţa (+) imediat după numele coloanei din tabela respectivă din condiţia de join din clauza **WHERE**.

De exemplu următoarea comandă va afişa toate persoanele cu sau fără firmă corespunzătoare vom scrie în sintaxa Oracle:

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a, firme b
WHERE a.IdFirm = b.IdFirm (+)
```

Rezultatul rulării acestei comenzi este cel din tabelul II.3.6.

Tabelul II.3.6. Outer Join

Nume	Prenume	NumeFirma
Antonescu	Elena	SC Crisib SA
Popescu	Ioan	SC Crisib SA
Bischin	Paraschin	SC SoftCom
Vasilescu	Vasile	SC SoftCom
Olaru	Angela	Brasoveanca
Ionescu	Gheorghe	Brasoveanca
Marinescu	Angela	-
Georgescu	Maria	-

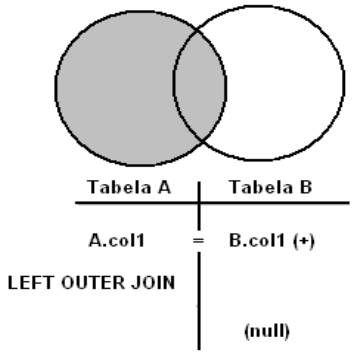


Figura II.3.3. Left Outer Join

Se observă că semnul **(+)** se găseşte după coloana **IdFirm** din tabela **firme** (**b**). Această tabelă fiind a doua tabelă din clauza **FROM**, vom spune că este vorba de un **LEFT OUTER JOIN**, adică sunt afişate toate înregistrările din tabela din stânga din clauza **FROM** *cu sau fără înregistrări corespunzătoare în tabela a doua*. Sintaxa ANSI foloseşte clauza **LEFT OUTER JOIN** împreună cu **ON**. Comanda anterioară este echivalentă cu următoarea comandă în sintaxa ANSI:

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a LEFT OUTER JOIN firme b
ON (a.IdFirm = b.IdFirm)
```

Dacă vom pune semnul **(+)** în dreptul celeilalte tabele, adică vom scrie:

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a, firme b
WHERE a.IdFirm (+) = b.IdFirm
```

se vor afişa toate firmele, cu sau fără angajaţi, adică toate înregistrările din tabela aflată în dreapta în clauza **FROM** (firme), cu sau fără înregistrări corespunzătoare în cealaltă tabelă, adică cu sau fără angajaţi. Este aşadar vorba despre un **RIGHT OUTER JOIN**. Astfel în sintaxa ANSI vom scrie:

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a RIGHT OUTER JOIN firme b
ON (a.IdFirm = b.IdFirm)
```

Rezultatul obţinut va fi cel din tabelul II.3.7.

Tabelul II.3.7. Right Outer Join

Nume	Prenume	NumeFirma
Ionescu	Gheorghe	Brasoveanca
Vasilescu	Vasile	SC SoftCom
Popescu	Ioan	SC Crisib SA
Antonescu	Elena	SC Crisib SA
Bischin	Paraschin	SC SoftCom
Olaru	Angela	Brasoveanca
-	-	SC TimTip

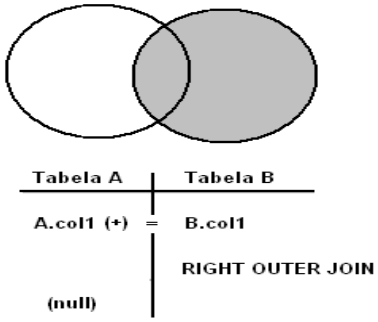


Figura II.3.4. Right Outer Join

ATENŢIE este importantă ordinea tabelelor în clauza **FROM** nu ordinea în care sunt scrise cele două părţi ale egalităţii din clauza **WHERE** respectiv **ON**. Astfel comenile:

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a, firme b
WHERE a.IdFirm = b.IdFirm (+)
```

şi

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a, firme b
WHERE b.IdFirm (+) = a.IdFirm
```

sunt echivalente şi reprezintă un **LEFT OUTER JOIN**, chiar dacă semnul **(+)** apare o dată în stânga semnului de egalitate şi o dată în dreapta semnului de egalitate.

De asemenea, deşi următoarele două comenzi sunt echivalente:

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a, firme b
WHERE a.IdFirm = b.IdFirm (+)
```

şi

```
SELECT a.nume, a.prenume, b.nume
FROM firme b, persoane a
WHERE a.IdFirm = b.IdFirm (+)
```

prima este un **LEFT OUTER JOIN** iar a doua este un **RIGHT OUTER JOIN**, pentru că se afişează toate înregistrările din tabela a (cea care nu are + în dreptul ei), tabelă care în prima comandă se găseşte în stânga în clauza **FROM**, iar în a doua comandă se găseşte în dreapta în clauza **FROM**.

V-aţi putea întreba acum cum am putea să afişăm toate înregistrările din ambele tabele, indiferent dacă ele au sau nu corespondent în cealaltă tabelă. Am dori deci să obţinem tabelul următor:

Tabelul II.3.8. Full Outer Join

Nume	Prenume	NumeFirma
Antonescu	Elena	SC Crisib SA
Popescu	Ioan	SC Crisib SA
Bischin	Paraschin	SC SoftCom
Vasilescu	Vasile	SC SoftCom
Olaru	Angela	Brasoveanca
Ionescu	Gheorghe	Brasoveanca
Marinescu	Angela	-
Georgescu	Maria	-
-	-	SC TimTip

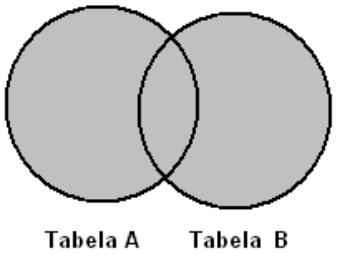


Figura II.3.5. Full Outer Join

Apar atât persoanele care nu sunt încă angajate, sau a căror firmă nu mai există în baza de date, dar şi firmele pentru care nu avem nici un angajat memorat în baza de date. Am fi tentaţi să scriem:

```
SELECT a.nume, a.prenume, b.nume
FROM firme b, persoane a
WHERE a.IdFirm (+) = b.IdFirm (+)
```

adică să punem (+) în ambele părţi ale semnului de egalitate pentru că avem de suplinit lipsa de informaţii din ambele tabele. Însă ***sintaxa Oracle nu permite acest lucru! Singura modalitate de a obţine un FULL OUTER JOIN este de a folosi sintaxa ANSI:***

```
SELECT a.nume, a.prenume, b.nume
FROM persoane a FULL OUTER JOIN firme b
ON (a.IdFirm = b.IdFirm)
```

Tabelul următor face o sinteză a comenzilor **JOIN** din acest capitol, punând faţă în faţă comenzile echivalente folosind cele două sintaxe.**Tabelul II.3.9.** Comparaţie între sintaxa Oracle şi sintaxa ANSI

Sintaxa Oracle	Sintaxa ANSI/ISO
Produsul Cartezian	
SELECT p.nume, p.prenume, f.nume FROM persoane p, firme f	SELECT p.nume, p.prenume, f.nume FROM persoane p **CROSS JOIN** firme f
Equijoin	
SELECT p.nume, prenume, f.nume FROM persoane p, firme f WHERE p.idfirm = f.idfirm	SELECT p.nume, prenume, f.nume FROM personae p **JOIN** firme f **USING** (IdFirm)
SELECT p.nume, prenume, f.nume FROM persoane p, firme f WHERE p.idfirm = f.idfirm AND p.nume=f.nume	SELECT nume, prenume, FROM persoane p **NATURAL JOIN** firme f NU AFIŞEAZĂ NIMIC !!!
SELECT a.nume, a.prenume, b.disciplina, b.data, b.nota FROM elevi a, firme b WHERE a.nume=b.nume AND a.prenume=b.prenume	SELECT nume, prenume, disciplina, data, nota FROM elevi **NATURAL JOIN** note
SELECT p.nume, prenume, f.nume FROM persoane p, firme f WHERE p.IdFirm=f.IdFirm	SELECT p.nume, prenume, f.nume FROM persoane p **JOIN** firme f **USING** (IdFirm)
Nonequijoin	
SELECT nume, prenume, disciplina, data, calificativ FROM note, calificative WHERE nota BETWEEN nota1 AND nota2	SELECT nume, prenume, disciplina, data, calificativ FROM note **JOIN** calificative **ON** (nota BETWEEN nota1 AND nota2)
Selfjoin	
SELECT a.nume ||' '|| a.prenume AS "Angajat", b.nume ||' '|| b.prenume AS "Sef" FROM angajat a, angajat b WHERE a.id_manager = b.id	SELECT a.nume ||' '|| a.prenume AS "Angajat", b.nume ||' '|| b.prenume AS "Sef" FROM angajat a **JOIN** angajat b **ON** (a.id_manager = b.id)
Outer Join	
SELECT a.nume, a.prenume, b.nume FROM persoane a, firme b WHERE a.IdFirm = b.IdFirm **(+)**	SELECT a.nume, a.prenume, b.nume FROM persoane a **LEFT OUTER JOIN** firme b on(a.IdFirm = b.IdFirm)
SELECT a.nume, a.prenume, b.nume FROM persoane a, firme b WHERE a.IdFirm **(+)** = b.IdFirm	SELECT a.nume, a.prenume, b.nume FROM persoane a **RIGHT OUTER JOIN** firme b **ON** (a.IdFirm = b.IdFirm)
NU EXSITA ECHIVALENT !	SELECT a.nume, a.prenume, b.nume

Sintaxa Oracle	Sintaxa ANSI/ISO
	`FROM persoane a` **`FULL OUTER JOIN`** `firme b` **`ON`** `(a.IdFirm = b.IdFirm)`

II.3.6. Operatorii `UNION`, `INTERSECT`, `MINUS`

Un caz mai special de interogare a mai multor tabele este acela în care combinăm rezultatele a două sau mai multe interogări independente una de cealaltă.
Operatorii folosiţi în acest scop sunt:
`UNION ALL` – returnează toate liniile returnate de de interogările pe care le leagă, inclusiv duplicatele (dacă cele două subinterogări returnează amânduua o aceeaşi linie, acest operator le va include pe ambele în rezultat)
`UNION` – asemănător cu operatorul anterior însă sunt eliminate duplicatele
`INTERSECT` – afişează liniile returnate de ambele interogări
`MINUS` – returnează liniile care sunt returnate de prima interogare dar nu sunt returnate şi de a doua interogare.
Atenţie! Numărul de coloane şi tipul coloanelor returnate de cele două nterogări trebuie să fie acelaţi, chiar dacă au alt nume.
Sintaxa folosirii acestor operatori este

```
interogare operator interogare
```

Vom exemplifica utilizarea acestor operatori pe două tabele formale

Tabelul II.3.10. Tabela **`A`**

ColA	ColB
A	10
A	15
B	7
C	20
C	30
D	40

Tabelul II.3.11. Tabela **`B`**

ColC	ColD
A	8
B	6
B	7
C	15
C	30
C	60
D	8

Tabelul II.3.12. Tabela **`C`**

ColE	ColF
A	10
B	6
C	20
D	8
E	10

Interogarea

```
SELECT ColA, ColB FROM A
UNION ALL
SELECT ColC, ColD FROM B
```

va afişa tabela II.3.13. Comanda următoare va elimina duplicatele, rezultatul fiind cel din tabela II.3.14.

```
SELECT ColA, ColB FROM A
UNION
SELECT ColC, ColD FROM B
```

Tabelul II.3.13. Utilizarea operatorului **`UNION ALL`**

COLA	COLB
A	10
A	15
B	7
C	20
C	30

Tabelul II.3.14. Utilizarea operatorului **`UNION`**

COLA	COLB
A	8
A	10
A	15
B	6
B	7

D	40
A	8
B	6
B	7
C	30
C	15
C	60
D	8

C	15
C	20
C	30
C	60
D	8
D	40

Similar comenzile următoare vor afişa tabelul II.3.14 şi respectiv II.3.15:

```
SELECT ColA, ColB FROM A
INTERSECT
SELECT ColC, ColD FROM B
```

şi

```
SELECT ColA, ColB FROM A
MINUS
SELECT ColC, ColD FROM B
```

Tabelul II.3.14. Utilizarea operatorului **INTERSECT**

COLA	COLB
B	7
C	30

Tabelul II.3.15. Utilizarea operatorului **MINUS**

COLA	COLB
A	10
A	15
C	20
D	40

Un exemplu practic de folosire a acestor operatori poate fi dat dacă ne imaginăm că pentru cercul de informatică de la liceul vostru, profesorul coordonator de cerc a întocmit un tabel **info** conţinând **numele**, **prenumele** şi **clasa** elevilor înscrişi la acest cerc. Similar, profesorul de la cercul de matematică a realizat un tabel **mate** cu aceeleaşi coloane, memorând elevii de la cercul de matematică.
Directorul şcolii doreşte, de exemplu, o listă cu elevii înscrişi la ambele cercuri. Nu aveţi altceva de făcut decât să scrieţi următoarea comandă:

```
SELECT nume, prenume, clasa FROM info
INTERSECT
SELECT nume, prenume, clasa FROM mate
```

Desigur puteţi combina mai mult de două interogări folosind operatorii **UNION**, **INTERSECT** şi **MINUS**. Implicit operatorii sunt evaluaţi de jos în sus, însă puteţi indica ordinea de efectuare a acestor operaţii prin folosirea parantezelor. De exemplu comanda

```
SELECT colA, colB FROM A
UNION
SELECT colC, colD FROM B
INTERSECT
SELECT colE, colF FROM C
```

va returna tabelul II.3.16 în timp ce comanda

```
SELECT colA, colB FROM A
UNION
(SELECT colC, colD FROM B
```

```
INTERSECT
SELECT colE, colF FROM C)
```
va returna tabelul II.3.17.

Tabelul II.3.16.

COLA	COLB
A	10
B	6
C	20
D	8

Tabelul II.3.17.

COLA	COLB
A	10
A	15
B	6
B	7
C	20
C	30
D	8
D	40

4.2. Funcţii de grup

Într-un capitol anterior am discutat despre funcţiile singulare, adică despre funcţiile care operează la un moment dat asupra unei singure înregistrări.
Este acum momentul să discutăm despre funcţiile de grup, care returnează o singură valoare pentru un **grup sau set** de linii dintr-un tabel. Puteţi calcula cea mai mare valoare dintr-un set de valori, puteţi determina numărul de înregistrări ce respectă o anumită condiţie etc.
Pentru exemplificarea acestor funcţii vom folosi tabela **VOTURI** şi tabela **JUDEŢE** care conţin următoarele date[1[1]].

Tabelul II.4.1. Tabela **VOTURI**

Judet	Candidat	Număr_voturi

B	1	347016
B	2	1552
B	3	1374
IS	1	196508
IS	2	1038
IS	3	1267
SB	1	65084
SB	2	561
SB	3	533
B	4	96744
B	5	25656
B	6	13361
IS	4	35784
IS	5	5558
IS	6	4094
SB	4	19937
SB	5	4323
SB	6	2366
B	7	25937
B	8	4619
B	9	4323
IS	7	3682
IS	8	1291
IS	9	327
SB	7	4225
SB	8	765
SB	9	3797
B	10	2037
B	11	22687
B	12	514366
IS	10	1312
IS	11	3781
IS	12	12184
SB	10	660
SB	11	3768
SB	12	105993
SB	13	100
B	13	(null)
IS	13	(null)

Tabelul II.4.2. Tabela **JUDETE**

Cod_judeţ	Judet	Număr_alegători
B	Bucureşti	1750192
IS	Iaşi	650029
SB	Sibiu	363380

Vom prezenta în continuare principalele funcţii de grup.

- **COUNT(x)** – determină numărul de valori ale lui **x**. Funcţia, ca de altfel toate funcţiile de grup ignoră câmpurile completate cu **NULL**, adică va număra doar valorile nenule ale lui x.
  De exemplu, comanda

```
SELECT COUNT(JUDET), COUNT(numar_voturi)
FROM voturi
```

  va afişa numărul total de înregistrări din tabelă, **39** (câmpul **JUDET** nu are nici o valoare **NULL**) precum şi numărul de linii pentru care câmpul **numar_voturi** este nenul, adică **37**, ultimele două linii din tabel având valoare **null** în câmpul **numar_voturi**.

Tabelul II.4.3.

COUNT(JUDET)	COUNT(NUMAR_VOTURI)
39	37

Funcţia **COUNT** poate fi folosită în combinaţie cu clauza **DISTINCT**, pentru a număra doar valorile distincte dintr-un domeniu. De exemplu dacă dorim să ştim pentru câte judeţe avem rezultatele votării în tabela noastră, vom folosi comanda:

```
SELECT count(distinct judet)
FROM voturi
```

Se va obţine valoarea **3**, întrucât avem doar **3** judeţe înregistrate (Bucureşti, Iaşi, Sibiu).

Tabelul II.4.4.

COUNT(DISTINCTJUDET)

3

Să vedem încă un exemplu:

```
SELECT count(distinct candidat), count(candidat)
FROM voturi
```

Evident primul apel de funcţie afişează valoarea **13** , deoarece există **13** candidaţi pentru care au fost exprimate voturi, iar a doua comandă afişează valoarea **39**, adică exact numărul de linii din tabel deoarece toate liniile au completat câmpul **candidat**.

Tabelul II.4.5.

COUNT(DISTINCTCANDIDAT)	COUNT(CANDIDAT)
13	39

- **MAX(x)** – determină valoarea maximă a valorilor expresiei **x**.

Să vedem de exemplu cum putem afla care este cel mai număr de voturi exprimate pentru un candidat într-un judeţ.

```
SELECT MAX(numar_voturi)
FROM voturi
```

Tabelul II.4.6.

MAX(NUMAR_VOTURI)
514366

Se poate observa pe tabelul cu datele din tabela voturi că acest maxim a fost obţinut în Bucureşti de către candidatul având codul **12**.

Totuşi această informaţie nu este foarte relevantă pentru că şi populaţia din Bucureşti este mult mai mare decât în celelalte judeţe. Ar trebui să putem determina numărul de voturi primite de către un candidat raportat la numărul de alegători (persoane cu drept de vot). SQL ne permite să aplicăm funcţiile de grup nu doar pe câmpuri din baza de date ci şi pe expresii, ca în exemplul următor:

```
SELECT max(100*numar_voturi/numar_alegatori)
FROM voturi v, judete j
WHERE v.judet=j.cod_judet
```

Tabelul II.4.7.

MAX(100*NUMAR_VOTURI/NUMAR_ALEGATORI)
30.230651247867402838950262219070226097 6

Prin această comandă am obţinut cel mai mare procent de voturi obţinut de către un candidat într-un judeţ. Acest procent a fost obţinut raportat la totalul persoanelor cu drept de vot şi a fost obţinut de către candidatul cu codul 1 în judeţul Iaşi:

```
SELECT 100*numar_voturi/numar_alegatori,
       j.judet, v.candidat
FROM voturi v, judete j
WHERE v.judet=j.cod_judet
```

Tabelul II.4.8.

100*NUMAR_VOTURI/NUMAR_ALEGATORI	JUDET	CANDIDAT
19.827310375090275809739731412325047766 2	Bucuresti	1
.088675985263331108815489957673215281523	Bucuresti	2

.07850567252050060793330103211533363196 7	Bucuresti	3
30.230651247867402838950262219070226097 6	Iasi	1
…	…	…

În acest moment nu ştim încă să scriem o comandă pentru a afişa judeţul şi candidatul pentru care s-a obţinut valoarea maximă, dar vom afla cum realizăm acest lucru în capitolul următor.

- **MIN(x)** – determină valoarea minimă a valorilor expresiei **x**.
- **SUM(x)** – determină suma valorilor expresiei **x**.

Cum aflăm oare numărul total de voturi valabil exprimate în judeţul Sibiu? Foarte simplu:

```
SELECT sum(numar_voturi)
FROM voturi
WHERE judet='SB'
```

Tabelul II.4.9.

SUM(NUMAR_VOTURI)
212112

- **AVG(x)** – determină media valorilor expresiei **x**. De exemplu, putem afla procentul mediu obţinut un candidat în toate judeţele:

```
SELECT avg(100*numar_voturi/numar_alegatori)
FROM voturi v, judete j
WHERE (candidat=12) and
      (v.judet=j.cod_judet)
```

Comanda afişează media procentelor obţinute în fiecare judeţ de către candidatul cu codul **12**:

Tabelul II.4.10.

AVG(100*NUMAR_VOTURI/NUMAR_ALEGATORI)
20.144045084597346892608799213577190 6663

Am dori să afişăm un tabel cu procentele obţinute de toţi candidaţii, însă vom vedea cum realizăm acest lucru într-un paragraf următor.

După cum am precizat la funcţia **COUNT**, funcţiile de grup, deci şi **AVG** ignoră valorile **NULL**. Aşadar dacă vom rula comanda:

```
SELECT avg(numar_voturi)
FROM voturi
WHERE candidat=13
```

vom obţine valoarea **100**, deşi în baza de date există 3 linii pentru candidatul **13**, şi doar o linie are completat câmpul **numar_voturi** cu valoarea **100**. Dacă dorim să obţinem valoarea **33.333**, adică **100/3**, vom scrie:

```
SELECT AVG(NVL(numar_voturi,0))
FROM voturi
WHERE candidat=13
```

adică înlocuim valorile **null** cu valoarea **0**, pentru ca acestea să intre în calculul mediei.

- **STDEV(x)** – funcţie statistică definită ca fiind abaterea pătratică a expresiei date. Cu cât valoarea funcţiei este mai mică cu atât valorile expresiei **x** sunt mai apropiate de medie.
- **VARIANCE(x)** – este o funcţie statistică care calculează dispersia expresiei **x**. Se defineşte ca pătratul abaterii medii pătratice.

Observaţie. Funcţiile **COUNT, MIN, MAX** pot fi aplicate şi datelor de tip şir de caractere sau dată calendaristice, celelalte funcţii fiind aplicabile doar valorilor numerice.

De exemplu comanda următoare va afişa data celei mai vechi angajări, data celei mai recente angajări, numărul de date de angajare, şi numărul de date distincte de angajare din tabela **employees**:

```
 select min(hire_date), max(hire_date),
        count(distinct hire_date), count(hire_date)
from employees
```

Tabelul II.4.11.

MIN(HIRE_DATE)	MAX(HIRE_DATE)	COUNT(DISTINCTHIRE_DATE)	COUNT(HIRE_DATE)
17-JUN-87	29-JAN-00	19	20

II.4.3. Gruparea datelor. Clauza GROUP BY

Uneori am putea dori să grupăm liniile dintr-o tabelă şi să obţinem anumite informaţii despre grupurile respective. De exemplu am dori să calculăm numărul total de voturi obţinut de fiecare candidat în toată ţara. Cu ceea ce am învăţat până acum, am putea rula o comandă de forma celei de mai jos pentru fiecare candidat în parte:

```
SELECT sum(numar_voturi)
FROM voturi
WHERE candidat=1
```

Tabelul II.4.12.

SUM(NUMAR_VOTURI)
608608

însă această metodă nu este convenabilă, întrucât am dori să obţinem un tabel cu toate aceste date, ca în tabelul II.4.13.

O astfel de grupare a datelor se poate face folosind clauza **GROUP BY**. Comanda care a fost rulată pentru a obţine rezultatul din tabelul II.4.13, este:

```
 SELECT candidat, sum(numar_voturi) AS "TOTAL VOTURI"
 FROM voturi
GROUP BY candidat
```

Tabelul II.4.13.

CANDIDAT	TOTAL VOTURI
1	608608
2	3151
3	3174
4	152465
5	35537
6	19821
7	33844
8	6675
9	8447
10	4009
11	30236
12	632543
13	100

Tabelul II.4.14.

CANDIDAT	NUMAR_VOTURI
1	65084
1	196508
1	347016
2	561
2	1038
2	1552
3	533
3	1267
3	1374
…	…

Se observă că pentru fiecare grup de înregistrări s-a obţinut câte o singură valoare, adică pentru fiecare candidat am obţinut o sumă a tuturor voturilor primite. De exemplu candidatul cu codul 1 a obţinut în Bucureşti **347016** voturi, la Iaşi **196508** voturi iar la Sibiu **65084** voturi, în total **608608** voturi adică exact valoarea din tabelele II.4.12 şi II.1.3.

Clauza **GROUP BY** poate fi folosită şi fără funcţii de grup, doar pentru a afişa liniile grupate după anumit criteriu, ca în exemplul următor:

```
SELECT candidat,numar_voturi FROM voturi
                    GROUP BY candidat, numar_voturi
```

Să vedem acum de exemplu cum aflăm procentul mediu obţinut de către fiecare candidat.

```
SELECT candidat,AVG(100*numar_voturi/numar_alegatori)
FROM voturi v, judete j WHERE v.judet=j.cod_judet
GROUP BY candidat
```

Tabelul II.4.15.

CANDIDAT	AVG(100*NUMAR_VOTURI/NUMAR_ALEGATORI)
1	22.656229561845598975692085315442433647 6
2	.13424833638246597421520567348011821838
3	.14003282051378316208662701165544554467
4	5.506383429113772239432943207791936674 12
5	1.17019960040031154685700409684022462069
6	.68144301477468349708451524754117789898
7	1.07036088696521333741348008106908880327
8	.22434794824565058705465479428445048096 1
9	.44740619415717432386337983296486539869 3
10	.16661747574531093409946880514800875407 6
11	.97161839160269742583080767121400220691
12	20.14404508459734689260879921357719066 63
13	.02751940117783037041113985359678573394 2

Reguli de folosire a clauzei GROUP BY

În clauza **GROUP BY** nu se acceptă aliasele coloanelor, comanda următoare va genera o eroare

```
SELECT department_id As Departament,
    job_id, MAX(salary)
   FROM employees GROUP BY Departament, job_id
```

-toate câmpurile care apar în select în afara funcţiilor de grup trebuie să apară în clauza **BROUP BY** ca în exemplele de mai jos:

```
SELECT department_id, job_id, MAX(salary)
   FROM employees GROUP BY department_id, job_id
```

sau

```
SELECT department_id, department_name, max(salary)
FROM employees NATURAL JOIN departments
GROUP BY department_id, department_name
```

sau

```
SELECT upper(last_name), sum(salary)
FROM employees GROUP BY last_name
```

Observaţi în acest ultim exemplu că deşi în clauza **SELECT** câmpului **last_name** îi este aplicată o funcţie (simplă nu de grup!) , în clauza **GROUP BY**, **last_name** poate să apară fără funcţia respectivă. Aveţi grijă să nu confundaţi funcţiile singulare cu cele de grup!

-Nu se pot folosi funcţii de grup în clauza **WHERE**. De aceea următoarea comandă nu va putea fi rulată ea generând o eroare:

```
SELECT * FROM voturi
WHERE numar_voturi=max(numar_voturi)
```

Pentru a putea afla ce candidat/candidaţi au obţinut cele mai multe voturi vom folosi o subinterogare (asupra acestui subiect vom reveni în capitolul următor) astfel:

```
SELECT * FROM voturi
WHERE numar_voturi =
      (SELECT max(numar_voturi) from voturi)
```

-în clauza **GRUP BY** pot să apară şi alte coloane care nu apar în **SELECT**

```
SELECT MAX(salary) FROM employees
GROUP BY departments
```

-funcţiile de grup pot fi imbricatre ca în exemplul următor, în care am determinat cel mai mare număr total de voturi obţinut de către un candidat.

```
SELECT max(sum(numar_voturi)) FROM voturi
GROUP BY candidat
```

Tabelul II.4.16.

MAX(SUM(NUMAR_VOTURI))
632543

II.4.4. Selectarea grupurilor. Clauza HAVING

De multe ori nu ne interesează să afişăm toate grupurile de obţinute prin folosirea clauzei **GROUP BY**. Pentru a filtra grupurile folosim clauza **HAVING**. Aşa cum am văzut în exemplele anterioare putem folosi clauza **GROUP BY** fără clauza **HAVING** însă clauza **HAVING** poate fi folosită doar atunci când este prezentă clauza **GROUP BY**.
Haideţi să analizăm un exemplu. Să presupunem că dorim să afişăm toţi candidaţii care au obţinut un procent în alegeri mai mare de 5% din numărul total de persoane cu drept de vot. Pentru aceasta procedăm astfel:

- folosim clauza **GROUP BY** pentru a grupa liniile după candidaţi şi calculăm pentru fiecare candidat procentul obţinut:

```
SELECT candidat,
       100*sum(numar_voturi)/sum(numar_alegatori)
FROM voturi v JOIN judete j
ON v.judet=j.cod_judet
GROUP BY candidat
```

Tabelul II.4.17.

CANDIDAT	100*SUM(NUMAR_VOTURI)/SUM(NUMAR_ALEGATORI)
1	22.022281798276958215024527780964039309 6
2	.114017906347551618341432066351112190219
3	.114850153839139586358522811360974322994
4	5.516896252389545379380019040375220590 82
5	1.285894743850505192319730670237852714 63
6	.717216414381091915945898123499014510416
7	1.224634091534921285670398874511913984 69
8	.241532695928247239742640127862162446 75

9	.30565193745406808001589230862197545883 1
10	.14506435625113755567464333671901262157 6
11	1.09407978937625221585894635296484550411
12	22.88836195963165449715787481622708922 16
13	.00361846735473029572648150004287883815 4

- Folosim clauza **HAVING** pentru a filtra grupurile care se vor afişa

```
SELECT candidat,
       100*sum(numar_voturi)/sum(numar_alegatori)
FROM voturi v JOIN judete j
ON (v.judet=j.cod_judet)
GROUP BY candidat
HAVING 100*sum(numar_voturi)/sum(numar_alegatori)>5
```

Tabelul II.4.18.

CANDIDAT	100*SUM(NUMAR_VOTURI)/SUM(NUMAR_ALEGATORI)
1	22.022281798276958215024527780964039309 6
4	5.516896252389545379380019040375220590 82
12	22.88836195963165449715787481622708922 16

Bineînţeles că putem folosi clauzele **WHERE**, **GROUP BY** şi **HAVING** împreună. În acest caz, clauza **WHERE** va filtra mai întâi liniile din tabelă, liniile rămase vor fi grupate apoi conform criteriului dat de clauza **GROUP BY** şi în final sunt afişate doar acele grupuri care respectă condiţia dată de clauza **HAVING**. (figura II.4.2.)
Atenţie! Trebuie făcută distincţia clară dintre clauzele **WHERE** şi **HAVING**. Clauza **WHERE** acţionează asupra liniilor în timp ce **HAVING** acţionează la nivel de grup.

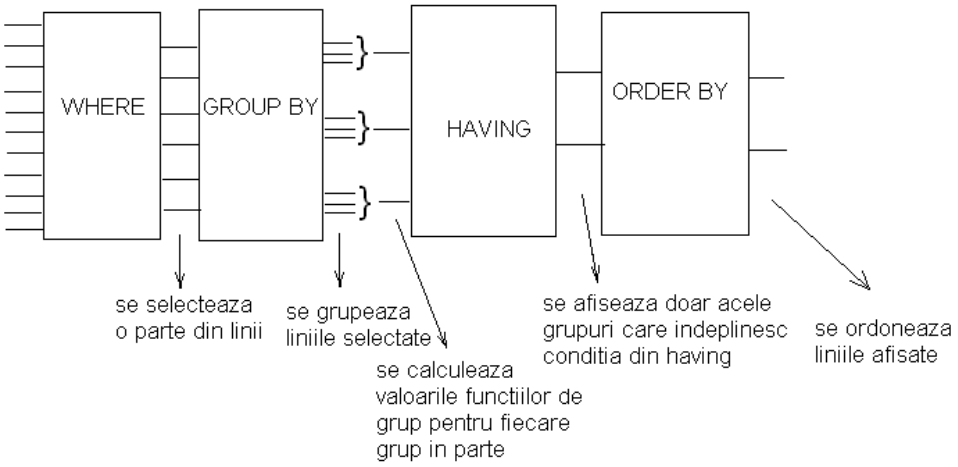


Figura II.4.2. Ordinea de executare a clauzelor comenzii **SELECT**

Să vedem de exemplu cum se evaluează comanda următoare

```
SELECT candidat,
       100*sum(numar_voturi)/sum(numar_alegatori)
FROM voturi v JOIN judete j
ON (v.judet=j.cod_judet)
WHERE numar_voturi>15000
GROUP BY candidat
HAVING 100*sum(numar_voturi)/sum(numar_alegatori)>5
```

Tabelul II.4.19.

CANDIDAT	100*SUM(NUMAR_VOTURI)/SUM(NUMAR_ALEGATORI)
1	22.022281798276958215024527780964039309 6
4	5.516896252389545379380019040375220590 82
12	29.351212071318128741296724218526740513 2

Observaţi însă mai întâi că prin adăugarea clauzei **WHERE**, rezultatele obţinute diferă puţin de cele din tabelul II.4.18, aceasta pentru că la calculul procentului obţinut de către candidatul 12 de exemplu nu mai este inclusă următoarea linie din tabelă

Tabelul II.4.20.

JUDET	CANDIDAT	NUMAR_VOTURI
IS	12	12184

Aşadar comanda se evaluează astfel:

- Mai întâi sunt filtrate liniile din tabelă

```
SELECT candidat, numar_voturi, numar_alegatori
FROM voturi v JOIN judete j
ON (v.judet=j.cod_judet)
WHERE numar_voturi>15000
```

Tabelul II.4.21.

CANDIDAT	NUMAR_VOTURI	NUMAR_ALEGATORI
1	347016	1750192
1	196508	650029
1	65084	363380
4	96744	1750192
5	25656	1750192
4	35784	650029
4	19937	363380
7	25937	1750192
11	22687	1750192
12	514366	1750192
12	105993	363380

Observaţi că au fost afişate doar 11 linii din totalul de 39 câte are tabela.

- Liniile obţinute la pasul anterior sunt grupate pe candidaţi şi se aplică funcţiile de grup

```
SELECT candidat,
       100*sum(numar_voturi)/sum(numar_alegatori)
FROM voturi v JOIN judete j
ON (v.judet=j.cod_judet)
WHERE numar_voturi>15000
GROUP BY candidat
```

Tabelul II.4.22.

CANDIDAT	100*SUM(NUMAR_VOTURI)/SUM(NUMAR_ALEGATORI)
1	22.022281798276958215024527780964039309 6
4	5.516896252389545379380019040375220590 82
5	1.465896313090220958614826259061862927 04

7	1.481951694442666861692888551656046879 43
11	1.296257782003345918619214349054275188 09
12	29.351212071318128741296724218526740513 2

- În final sunt afişate doar acele linii obţinute la pasul anterior care îndeplinesc condiţia din clauza **HAVING**.

```
SELECT candidat,
       100*sum(numar_voturi)/sum(numar_alegatori)
FROM voturi v JOIN judete j
ON (v.judet=j.cod_judet)
WHERE numar_voturi>15000
GROUP BY candidat
HAVING 100*sum(numar_voturi)/sum(numar_alegatori)>5
```

Tabelul II.4.23.

CANDIDAT	100*SUM(NUMAR_VOTURI)/SUM(NUMAR_ALEGATORI)
1	22.022281798276958215024527780964039 3096
4	5.516896252389545379380019040375220590 82
12	29.351212071318128741296724218526740513 2

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).

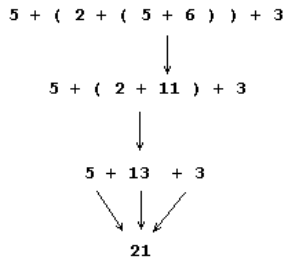


Figura II.5.1.

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**:

Tabelul II.5.5.

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

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:

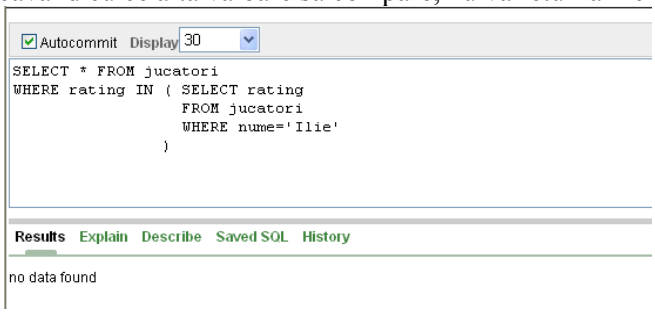


Figura II.5.2.

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:

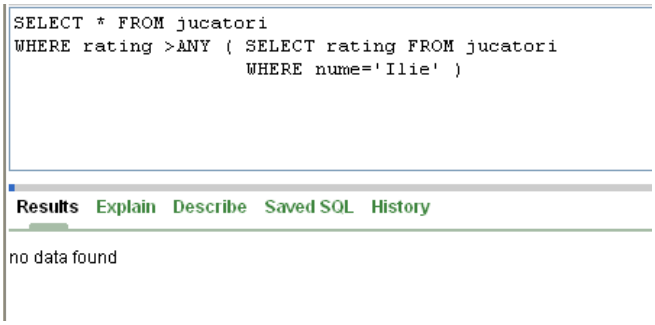


Figura II.5.3.

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

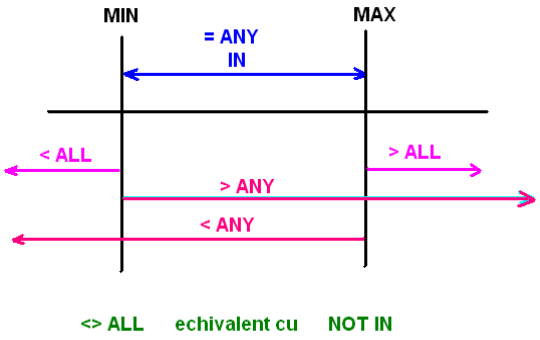


Figura II.5.4.

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)
     )
```

```
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)
         )
```

sau

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.

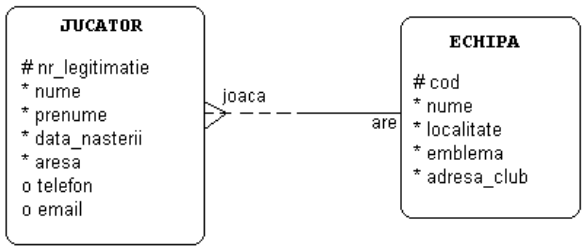


Figura II.6.1

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)
        )
```

```
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

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:

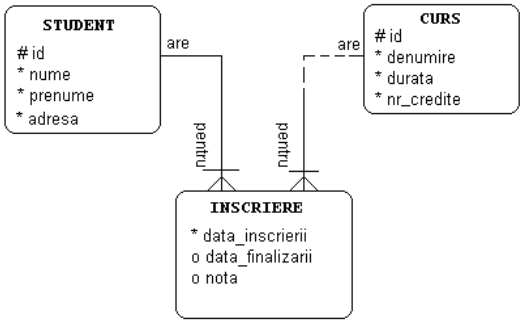
```
CREATE TABLE jucatori
( nr_legitimatie NUMBER(5) PRIMARY KEY,
  cod_echipa NUMBER(3) CONSTRAINT ech_fk
        REFERENCES echipe(cod) ON DELETE CASCADE,
  nume VARCHAR2(30) NOT NULL,
  prenume VARCHAR2(30) NOT NULL,
  datan DATE NOT NULL,
  adresa VARCHAR2(60) NOT NULL,
  telefon NUMBER(3),
  email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

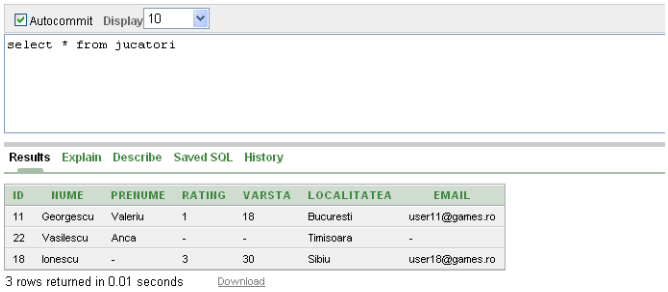


Figura II.7.1

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)
```

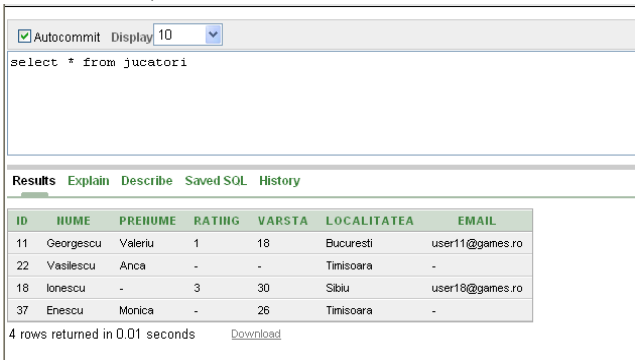


Figura II.7.2

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**:

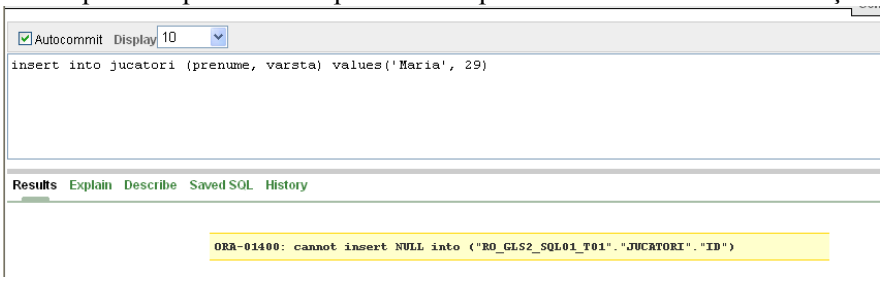


Figura II.7.3.

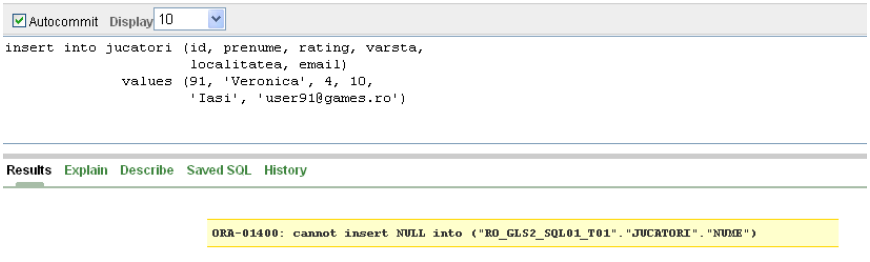


Figura II.7.4.

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')
```

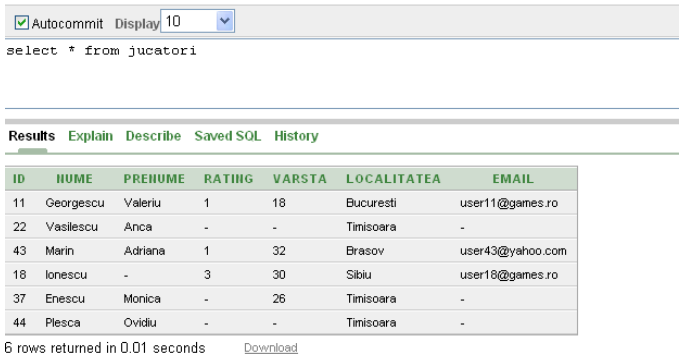


Figura II.7.5.

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'
```

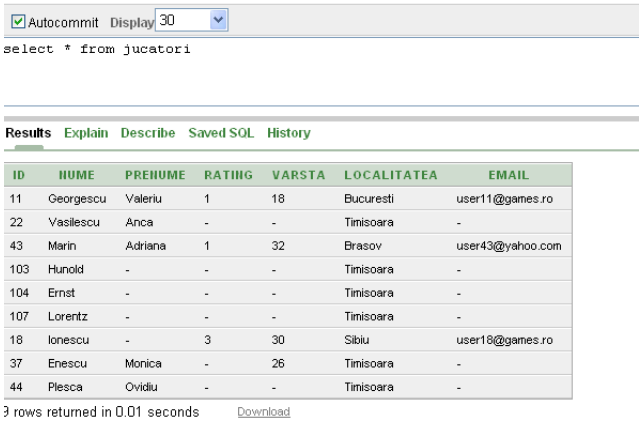


Figura II.7.6.

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:

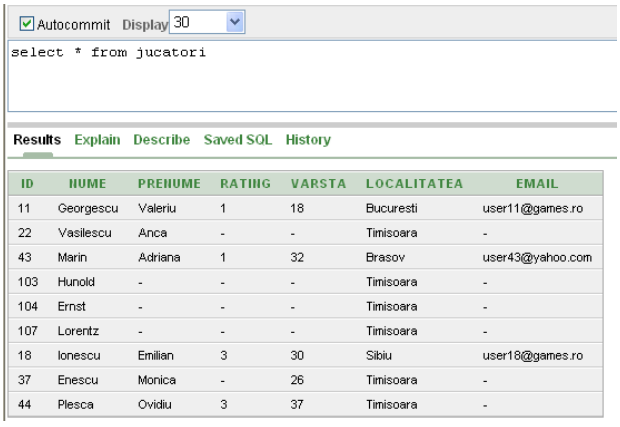
```
UPDATE jucatori
SET rating=(SELECT rating FROM jucatori WHERE id=18),
    varsta=(SELECT varsta+5 FROM jucatori WHERE id=43)
WHERE id=44
```

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

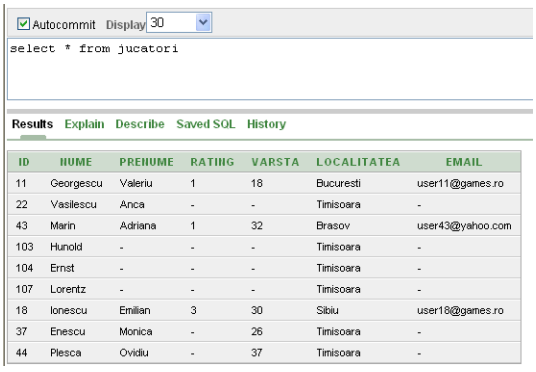
```
UPDATE jucatori
SET rating = (SELECT rating FROM jucatori WHERE id=200)
WHERE id=44
```

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).

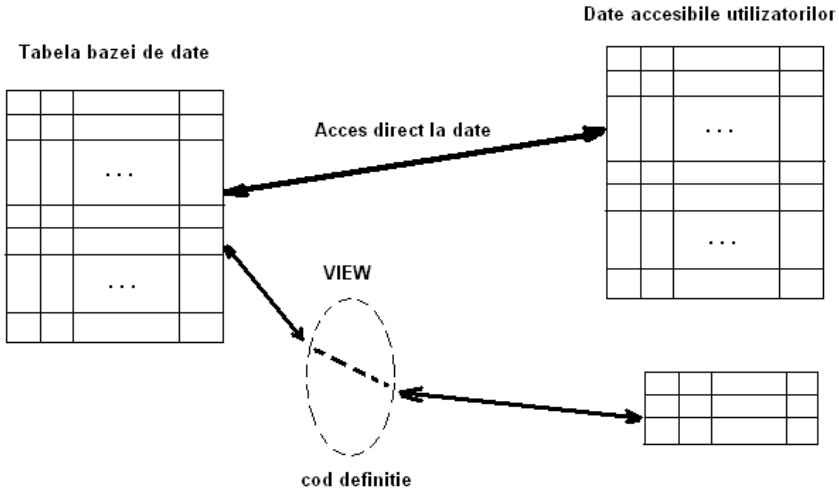


Figura II.8.1. Acces direct şi indirect (printr-o vedere) la o tabelă

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
```

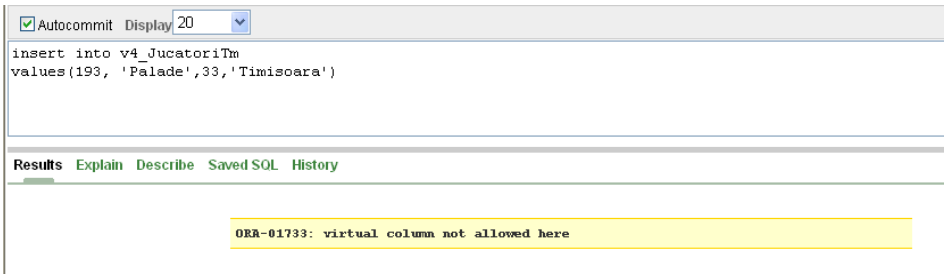


Figura II.8.2.

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).

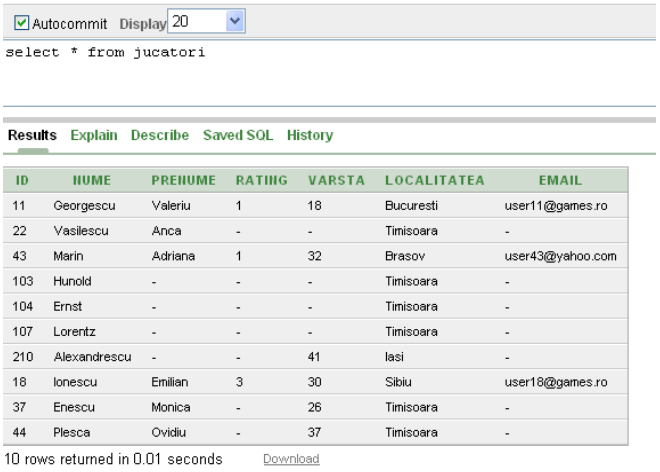


Figura II.8.3.

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).

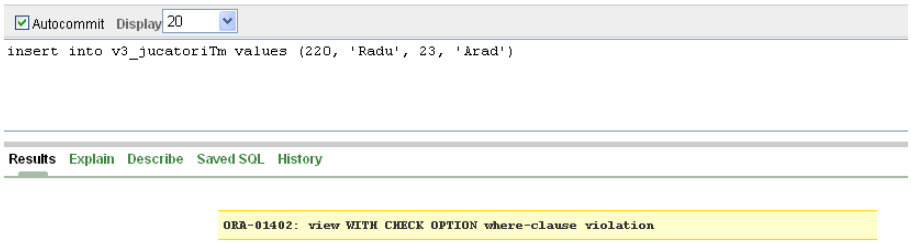


Figura II.8.4.

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)

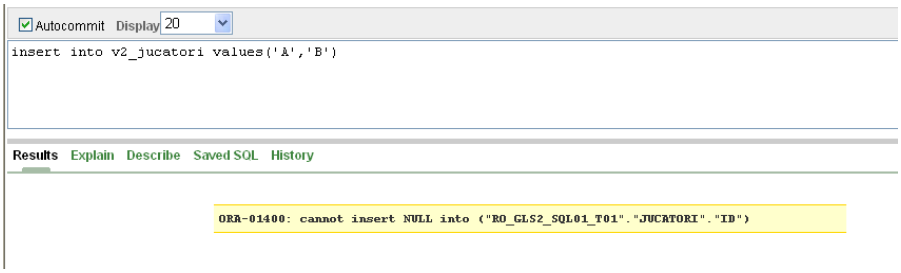


Figura II.8.5.

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.)

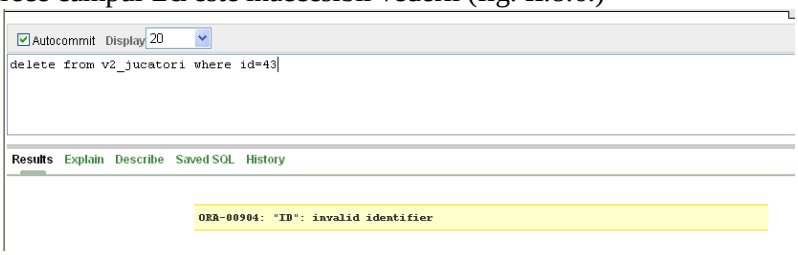


Figura II.8.6.

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.)

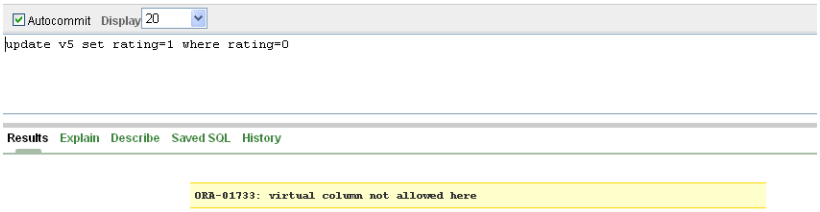


Figura II.8.7.

- 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 întregi 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)

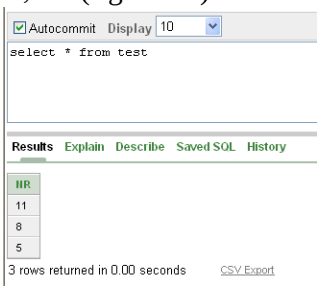


Figura II.9.1.

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 CREATE VIEW
user4	DELETE, UPDATE ON hr.elevi

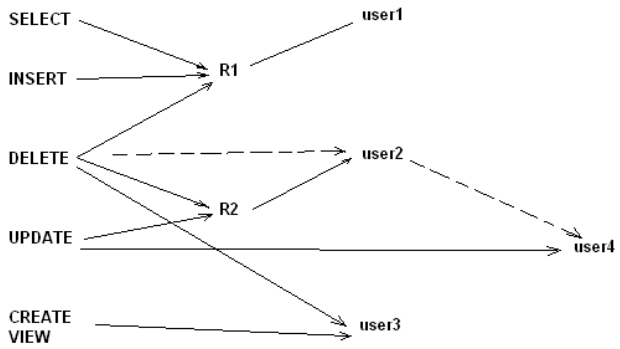


Figura II.10.1. Schema de acordare a drepturilor

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 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;
```

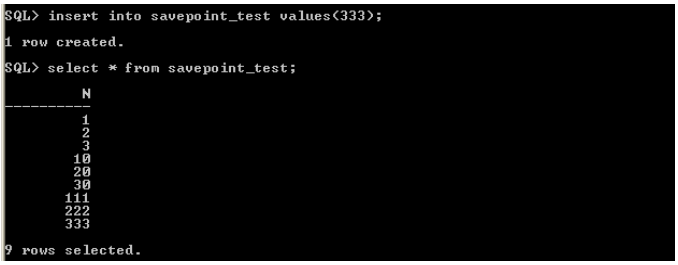


Figura II.11.3.

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;
```

Figura II.11.4.

Schematic tranzacţia anterioară arată ca în figura II.11.5.

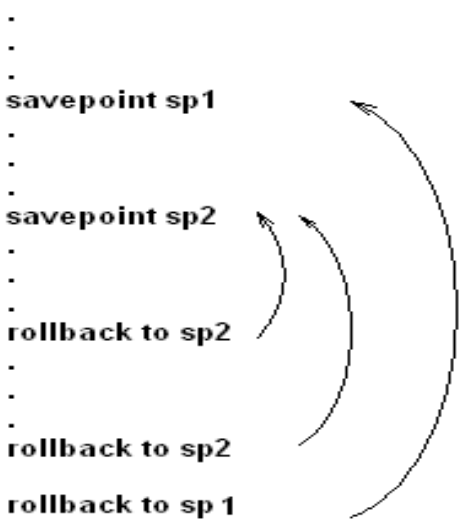


Figura II.11.5.