Capitolul 6

INTEROGAREA BAZELOR DE DATE. LIMBAJUL SQL

Sumar

- Prezentarea generală a limbajului SQL
- Crearea tabelelor şi declararea restricţiilor
- Comenzi de actualizare a tabelelor
- Structura de bază a frazei select
- Joncţiuni
- Funcţii-agregat: COUNT, SUM, AVG, MAX, MIN
- Sub-consultări. operatorul IN
- Reuniune, intersecţie, diferenţă
- Gruparea tuplurilor. Clauzele GROUP BY şi HAVING

6.1 PREZENTAREA GENERALĂ A LIMBAJULUI SQL

După două capitole de luptă surdă cu bazele de date, a devenit aproape evident că folosim bazele de date pentru că avem memoria prea scurtă. Deoarece suntem o lume aşezată pe un morman de hârtii & hârţoage, ne este cu neputinţă să reconstituim ceea ce am făcut adineori, dară-mi-te ieri, săptămâna trecută sau acum un an sau cinci. Necazul e că, de cele mai multe ori, trebuie să ştim nu numai ce-am făcut noi, dar ce-au făcut şi colegii şi partenerii de afaceri. Dacă la oameni mai putem trece cu vederea, în cazul bazelor de date încrederea este elementul cheie (primară, străină...). Aşa încât şi în capitolul 4 şi în capitolul 5 am fost foarte interesaţi să aflăm cum definim cât mai multe restricţii, astfel să diminuăm riscul preluării în baza de date a unor informaţii eronate. Apoi am văzut cum inserăm, modificăm şi ştergem date într-o bază. Tangenţial am abordat şi câteva modalităţi de a obţine informaţii de o manieră asistată, grafică.

Chiar dacă am încercat să păstrăm discuţia la un nivel general, exemplificările au fost făcute pe calapodul ACCESS-ului. Cei care au experienţă în ACCESS şi vor să treacă, forţaţi de împrejurări, pe un SGBD mai performant, gen PostgreSQL, Oracle etc. (trecere care echivalează cu trecerea de la amatori la semi sau chiar profesionişti de-a binelea) vor fi bruscaţi de diferenţele de interfaţă, opţiuni etc.

Pentru atenuarea şocurilor trecerilor de la un SGBD la altul, specialiştii bazelor de date s-au gândit la un limbaj universal dedicat creării tabelelor, definirii restricţiilor, creării utilizatorilor şi grupurilor de utilizatori, definirii drepturilor fiecărui utilizator/grup la obiecte din bază, actualizării înregistrărilor din tabele (inserare, modificare, ştergere) şi, mai ales, extragerii şi prelucrării datelor din bază. Acest limbaj a fost standardizat încă din 1986 (în SUA) şi 1989 (la nivel mondial – ISO) şi se numeşte SQL.

Haideţi să revenim la interogarea FACTURI_DUPA_20IUNIE2005 din figura 5.40 (a cărei execuţie produce rezultatul din figura 5.39). Dacă din modul proiectare (**Design**) alegem opţiunea SQL View, o să ne trezim cu o fereastră în care apare din senin o comandă cel puţin ciudată:

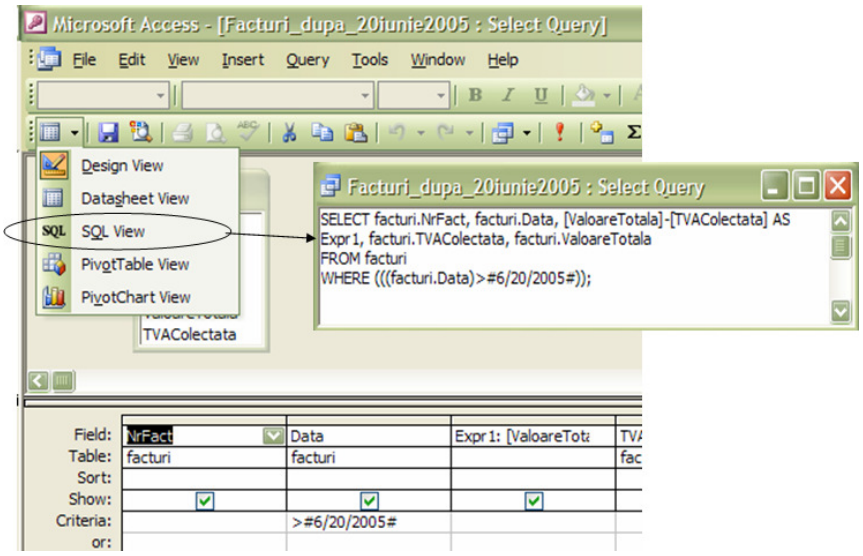

Figura 6. 1. O interogare SQL de-a gata

```
SELECT facturi.NrFact, facturi.Data, [ValoareTotala]-[TVAColectata] AS Expr1,
    facturi.TVAColectata, facturi.ValoareTotala
FROM facturi
WHERE (((facturi.Data)>#6/20/2005#));
```

Ei bine, acest SELECT este chiar o comandă SQL. Dar s-o luăm la pas temeinic. Un mobil determinant al succesului bazelor de date relaţionale l-a constituit, fără nici o îndoială, SQL. De la începuturile bazelor de date, s-a pus problema elaborării unui limbaj universal special dedicat bazelor de date, limbaj care să permită, în egală măsură, definirea relaţiilor (tabelelor), declararea restricţiilor, modificarea datelor din tabele, precum şi extragerea informaţiilor din cele mai diverse din datele existente în bază. În 1970, E.F.Codd sugera "adoptarea unui model relaţional pentru organizarea datelor [...] care să permită punerea la punct a unui sub-limbaj universal pentru gestiunea acestora, sub-limbaj care să fie, în fapt, o formă aplicată de calcul asupra predicatelor".

De atunci până în prezent au fost propuse numeroase limbaje pentru lucrul cu bazele de date, denumite, în general, *limbaje de interogare*. Dintre cele teoretice, cele mai cunoscute sunt algebra relaţională şi calculul relaţional, iar dintre cele practice (comerciale) Quel, QBE şi SQL. De departe, cel mai important este SQL, fundamentat de algebra relaţională.

După mulţi autori, momentul decisiv în naşterea SQL ca limbaj îl constituie lansarea proiectului System /R de către firma IBM, eveniment ce a avut loc în 1974. Tot în 1974 Chamberlin şi Boyce au publicat o lucrare în care este prezentată forma unui limbaj structurat de interogare, "botezat" SEQUEL (Structured English as QUEry Language). În 1975 Chamberlin, Boyce, King şi Hammer publică un articol în care prezintă sub-limbajul SQUARE, asemănător SEQUEL-ului, dar care utiliza expresii matematice şi nu cuvinte din limba engleză. Autorii celor două studii au demonstrat că limbajele SEQUEL şi SQUARE sunt complete din punct de vedere relaţional.

Un colectiv de autori, condus de Chamberlin, elaborează în 1976 o nouă lucrare în care se face referire la SEQUEL 2, acesta fiind "preluat" ca limbaj de interogare al SGBD-ului System /R al firmei IBM. În 1980 Chamberlin schimbă denumirea SEQUEL în SQL - Structured Query Language (Limbaj Structurat de Interogare), dar şi astăzi mulţi specialişti pronunţă SQL ca pe predecesorul său. Anii următori au înregistrat apariţia a o serie întreagă de lucrări care l-au perfecţionat, ultimul deceniu consacrându-l ca pe cel mai răspândit limbaj de interogare a BDR, fiind prezent în numeroase "dialecte" specifice tuturor SGBDR-urilor actuale, de la DB2 la Microsoft SQL Server, de la Oracle la FoxPro şi ACCESS.

American National Standard Institute publică în 1986 standardul SQL ANSI X3.136-1986. Este un standard care se bazează, într-o mare măsură, pe "dialectul" SQL al produsului DB2 de la IBM. În 1989 are loc revizuirea extinderea acestui standard, "născându-se" SQL-89, care mai este denumit şi SQL1. Deşi recunoscut ca bază a multor SGBDR-uri comerciale, SQL1 şi-a atras numeroase critici. În plus, variantele comercializate de diferiţii producători, deşi asemănătoare în esenţă, erau (şi sunt) incompatibile la nivel de detaliu. Pentru a umple golurile SQL1, ANSI a elaborat în 1992 "versiunea" SQL2, specificaţiile fiind prezentate la un nivel mult mai detaliat (dacă SQL1 se întindea pe numai 100 de pagini, SQL2 a fost publicat în aproape 600). IBM a avut un aport incontestabil la apariţia şi maturizarea SQL, fiind un producător cu mare influenţă în "lumea" SGBD-urilor, iar produsul său, DB2, este unul din standardele de facto ale SQL.

Standardul SQL:1999 a fost amânat de câteva ori până la publicarea sa, iar cea mai recentă versiune este SQL:2003. Actualmente, principalele orientări ale SQL vizează transformarea acestuia

într-un limbaj complet, în vederea definirii şi gestionării obiectelor complexe şi persistente. Aceasta include: generalizare şi specializare, moşteniri multiple, polimorfism, încapsulare, tipuri de date definite de utilizator, triggere (declanşatoare) şi proceduri stocate, suport pentru sisteme bazate pe gestiunea cunoştinţelor, expresii privind interogări recursive şi instrumente adecvate de administrare a datelor.

La momentul actual, SQL reprezintă cel mai important limbaj în domeniul bazelor de date, atât prin gama comenzilor şi opţiunilor de care dispune, dar mai ales datorită faptului că s-a reuşit standardizarea sa şi portarea pe toate Sistemele de Gestiune a Bazelor de date semnificative. Cu atât mai mult, cu cât, spre deosebire de majoritatea covârşitoare a altor limbaje, poate fi deprins relativ uşor de neinformaticieni şi utilizat pentru chestiuni de mare fineţe de către profesionişti. Acest capitol se doreşte a fi o prezentare a elementelor esenţiale prin care, dată fiind structura unei baze de date relaţionale, pot fi formulate interogări (fraze SELECT) prin care se obţin răspunsuri la gamă eterogenă de întrebări. În plus, sunt evocate pe scurt comenzile pentru actualizarea unei tabele (INSERT, UPDATE, DELETE), precum şi cele de declarare a structurii bazei de date (CREATE TABLE).

Din punctul de vedere al utilizatorului final, obiectivul principal al SQL constă în a oferi utilizatorului mijloacele necesare formulării unei consultări numai prin descrierea rezultatului dorit, cu ajutorul unei aserţiuni (expresie logică), fără a fi necesară şi explicitarea modului efectiv în care se face căutarea în BD. Altfel spus, utilizatorul califică (specifică) rezultatul iar sistemul se ocupă de procedura de căutare.

Deşi este referit, în primul rând, ca un limbaj de interogare, SQL este mult mai mult decât un instrument de consultare a bazelor de date, deoarece permite, în egală măsură:

- Definirea datelor
- Consultarea BD
- Manipularea datelor din bază
- Controlul accesului
- Partajarea bazei între mai mulţi utilizatori ai acesteia
- Menţinerea integrităţii BD.

Deşi toate clasificările îl încadrează la limbaje de generaţia a IV-a, SQL nu este, totuşi, un limbaj de programare propriu-zis, prin comparaţie cu Basic, Pascal, C, COBOL etc. SQL nu conţine (până la SQL3) instrucţiuni/comenzi pentru codificarea structurilor alternative şi repetitive, cu atât mai puţin facilităţi de lucru cu obiecte vizuale, specifice formularelor de preluare a datelor (căsuţe-text, butoane radio, liste, butoane de comandă etc.). Din acest punct de vedere, poate fi referit ca sub-limbaj orientat pe lucrul cu bazele de date. Comenzile sale pot fi, însă, inserate în programe redactate în limbaje de programare "clasice".

Principalele atuuri ale SQL sunt:

1. Independenţa de producător, nefiind o tehnologie "proprietară".
2. Portabilitate între diferite sisteme de operare.
3. Standardizare.
4. "Filosofia" sa se bazează pe modelul relaţional de organizare a datelor.
5. Este un limbaj de nivel înalt, cu structură ce apropie de limba engleză.
6. Furnizează răspunsuri la numeroase interogări simple, ad-hoc, neprevăzute iniţial.
7. Constituie suportul programatic pentru accesul la BD.
8. Permite multiple imagini asupra datelor bazei.
9. Este un limbaj relaţional complet.

10. Permite definirea dinamică a datelor, în sensul modificării structurii bazei chiar în timp ce o parte din utilizatori sunt conectaţi la BD.
11. Constituie un excelent suport pentru implementarea arhitecturilor client-server.

Principalele comenzi ale SQL, care se regăsesc, într-o formă sau alta, în multe dintre SGBDR-urile actuale sunt prezentate în tabelul 6.1.

Tabel 6.1. Clase de comenzi SQL

Comandă	Scop
Pentru manipularea datelor	
SELECT	Extragerea datelor din BD
INSERT	Adăugarea de noi linii într-o tabelă
DELETE	Ştergerea de linii dintr-o tabelă
UPDATE	Modificarea valorilor unor atribute
Pentru definirea bazei de date	
CREATE TABLE	Adăugarea unei noi tabele în BD
DROP TABLE	Ştergerea unei tabele din bază
ALTER TABLE	Modificarea structurii unei tabele
CREATE VIEW	Crearea unei tabele virtuale
DROP VIEW	Ştergerea unei tabele virtuale
Pentru controlul accesului la BD	
GRANT	Acordarea unor drepturi pentru utilizatori
REVOKE	Revocarea unor drepturi pentru utilizatori
Pentru controlul tranzacţiilor	
COMMIT	Marchează sfârşitul unei tranzacţii
ROLLBACK	Abandonează tranzacţia în curs

6.2 CREAREA TABELELOR ŞI DECLARAREA RESTRICŢIILOR

Limbajul SQL prezintă o serie întreagă de opţiuni care permit crearea tabelelor şi modificarea valorilor unor atribute pentru tuplurile unei relaţii. Mai mult, standardul SQL dispune de opţiuni clare privind specificarea unor restricţii legate de cheile primare, cheile străine etc. Baza de date pe care o vom lua în discuţie pe parcursul acestui capitol este cea prezentată în capitolul 4. Pentru crearea unei tabele comanda SQL este, natural, CREATE TABLE prin care se declară numele tabelei, numele, tipul şi lungimea fiecărui atribut, precum şi restricţiile. Astfel, primele două tabele din baza de date, CODPOST_LOC şi CLIENŢI pot fi create astfel:

```
CREATE TABLE codPost_loc (
        CodPostal CHAR(6) CONSTRAINT pk_cp PRIMARY KEY,
        Localitate CHAR (35) NOT NULL,
        Judet CHAR(25) NOT NULL ) ;
CREATE TABLE clienti  (
        CodClient INTEGER CONSTRAINT pk_cp PRIMARY KEY,
        NumeClient CHAR (30) NOT NULL CONSTRAINT un_numeclient UNIQUE,
        Adresa CHAR(60),
        CodPostal CHAR(6) NOT NULL CONSTRAINT ref_cl_codpost
                REFERENCES codpost_loc (CodPostal)    ) ;
```

Se pot observa cu uşurinţă clauzele PRIMARY KEY folosite pentru declararea cheilor primare, UNIQUE pentru cheile alternative, NOT NULL pentru interzicerea valorilor nule, precum şi

REFERENCES pentru declararea cheii străine. În plus, opţiunea CONSTRAINT ne ajutăm să "botezăm" fiecare restricţie după cum dorim.

Spre deosebire de alte SGBD-uri, în ACCESS comenzile nu pot fi introduse direct, ci incluse în module de program. Iată modulul CREARETABELE():

```
Sub CreareTabele()
   Dim dbs As Database
   'Set dbs = OpenDatabase("Z:\ISA_2006_ZI\BD\vinzari.mdb")
   Set dbs = CurrentDb
   dbs.Execute "CREATE TABLE codPost_loc " _
      & "(CodPostal CHAR(6) CONSTRAINT pk_cp PRIMARY KEY, " _
      & " Localitate CHAR (35) NOT NULL, " _
      & " Judet CHAR(25) NOT NULL) ; "
   dbs.Execute "CREATE TABLE clienti " _
      & "(CodClient INTEGER CONSTRAINT pk_cp PRIMARY KEY, " _
      & " NumeClient CHAR (30) NOT NULL CONSTRAINT un_numeclient UNIQUE, " _
      & " Adresa CHAR(60),  " _
      & " CodPostal CHAR(6) NOT NULL CONSTRAINT ref_cl_codpost REFERENCES codpost_loc (CodPostal) ) ;"
   dbs.Execute "CREATE TABLE facturi " _
      & "(NrFact INTEGER CONSTRAINT pk_facturi PRIMARY KEY, " _
      & " CodClient INTEGER NOT NULL CONSTRAINT ref_fact_cl REFERENCES clienti (CodClient), " _
      & " Data DATETIME NOT NULL, " _
      & " ValoareTotala NUMERIC NOT NULL, " _
      & " TVAColectata NUMERIC NOT NULL) ;"
End Sub
```

Deşi în capitolul 4 spuneam că structura unei baze de date este constantă, există situaţii în care trebuie să:

- adăugăm un atribut;
- eliminăm un atribut;
- schimbăm tipul unui atribut;
- modificăm lungimea unui atribut;
- să declarăm o nouă restricţie
- să anulăm o restricţie în vigoare.

Deoarece baza de date este în uz, nu ne putem permite să ştergem şi apoi să recreăm tabelele, pentru că aceasta echivalează cu pierderea iremediabilă a înregistrărilor existente. Aşa că este necesară folosirea comenzii ALTER TABLE. Dacă în tabela CLIENŢI se doreşte păstrarea şi a codului fiscal al fiecărui furnizor, este necesară adăugarea atributului **CodFiscal**, care este un şir de caractere (un număr precedat de litera R, dacă clientul respectiv este plătitor de TVA) de lungime 10 caractere. Comanda utilizată este:

ALTER TABLE CLIENŢI ADD CodFiscal CHAR(10)

În SQL ştergerea unei tabele din baza de date este realizabilă cu ajutorul comenzii DROP TABLE. Iată cum ar putea lansate comenzile de ştergere ale celor trei tabele pe care abia le-am creat acum jumătate de pagină:

```
Sub StergeTabele()
   Dim dbs As Database
   Set dbs = OpenDatabase("Z:\ ISA_2006_ZI\BD\vinzari.mdb")
   dbs.Execute "DROP TABLE facturi ;"
   dbs.Execute "DROP TABLE clienti ;"
   dbs.Execute "DROP TABLE codpost_loc ;"
End Sub
```

Am trecut în revistă până în acest moment principalele clauze ale comenzii CREATE TABLE (şi ALTER TABLE) pentru declararea cheilor primare, alternative şi străine, şi valorilor nenule.

Nemeritat, a fost omisă clauza CHECK prin care putem defini restricţii utilizator sub forma regulilor de validare la nivel de atribut sau înregistrare. Astfel, în tabela CLIENŢI valorile atributului **CodClient** trebuie să înceapă de la 1001, iar numele clientului se doreşte a fi scris cu majuscule întotdeauna. Aceste două reguli de validare pot fi definite în SQL atât în momentul creării:

```
CREATE TABLE clienti  (
        CodClient INTEGER CONSTRAINT pk_cp PRIMARY KEY
                CONSTRAINT ck_codclient CHECK (CocClient > 1000),
        NumeClient CHAR (30) NOT NULL CONSTRAINT un_numeclient UNIQUE
                CONSTRAINT ck_numeclient CHECK (NumeClient = UPPER(NumeClient)),
        Adresa CHAR(60),
        CodPostal CHAR(6) NOT NULL CONSTRAINT ref_cl_codpost
                REFERENCES codpost_loc (CodPostal)    ) ;
```

cât şi ulterior prin ALTER TABLE. Din păcate, ACCESSul nu este prea îngăduitor în această privinţă, clauza CHECK fiind interzisă. Singura modalitate de declarare a regulilor este cea procedurală:

```
Sub Reguli_Atribute()
   Dim dbs As Database, tdf As TableDef, fld As Field
   Set dbs = CurrentDb

   ' CLIENTI.CodClient > 1000
   Set tdf = dbs.TableDefs("clienti")
   Set fld = tdf.Fields("CodClient")
   fld.ValidationRule = "[CodClient] > 1000"
   fld.ValidationText = "Cel mai mic cod de client acceptat este 1001 !"

   ' CLIENTI.NumeClient se scrie numai cu majuscule
   Set tdf = dbs.TableDefs("clienti")
   Set fld = tdf.Fields("NumeClient")
   fld.ValidationRule = "StrComp(UCase([NumeClient]), [NumeClient], 0) = 0"
   fld.ValidationText = "Literele din numele clientului sunt obligatoriu majuscule !"

   ' Prima litera din CLIENTI.Adresa este majuscula. Restul, la alegere !
   Set tdf = dbs.TableDefs("clienti")
   Set fld = tdf.Fields("Adresa")
   fld.ValidationRule = "StrComp(LEFT(UCase([Adresa]),1), LEFT([Adresa],1), 0) = 0"
   fld.ValidationText = "Prima litera din adresa clientului trebuie sa fie majuscula !"

   'Data facturii
   Set tdf = dbs.TableDefs("facturi")
   Set fld = tdf.Fields("Data")
   fld.ValidationRule = "[Data] BETWEEN #1-1-2005# AND #12-31-2010# "
   fld.ValidationText = "Data facturii trebuie sa se incadreze in intervalul 1 ian.2005 - 31 dec.2010 !"
End Sub
```

Despre reguli la nivel de înregistrare, ce să mai vorbim... Astfel, dacă în tabela FACTURI s-ar dori instituirea regulii după care TVA-ul poate fi cel mult egal cu 0,19/1,19 din valoarea totală a fiecărei facturi (prietenii ştiu de ce !), restricţia ar putea fi definită la creare astfel:

```
CREATE TABLE facturi (
        NrFact INTEGER CONSTRAINT pk_facturi PRIMARY KEY,
        CodClient INTEGER NOT NULL CONSTRAINT ref_fact_cl REFERENCES clienti (CodClient),
        Data DATE NOT NULL,
        ValoareTotala NUMERIC NOT NULL,
        TVAColectata NUMERIC NOT NULL),
        CONSTRAINT ck_tva_valtot CHECK (TVAColectata <= ValoareTotala * 0.19 / 1.19)  );
```

ACCESS-ul este, după cum vă imaginaţi, impasibil la graţiile (de tip CHECK) ale SQL-ului, aşa că singura soluţie e tot cea procedurală:

```
Sub Regula_inregistrare()
```

```
    Dim dbs As Database, tdf As TableDef
    Set dbs = CurrentDb
    Set tdf = dbs.TableDefs("facturi")
    tdf.ValidationRule = "[TVAColectata] <= [ValoareTotala] * 0.19 / 1.19"
    tdf.ValidationText = "TVA poate fi maxim 0,19/1,19 din valoarea totala a facturii !"
End Sub
```

6.3 COMENZI DE ACTUALIZARE A TABELELOR

SQL prezintă comenzi specifice pentru modificarea conţinutului unei tabele, înţelegând prin aceasta trei acţiuni prin care se actualizează baza: a) adăugarea de noi linii la cele existente într-o tabelă; b) ştergerea unor linii, c) modificarea valorii unui atribut.

6.3.1 Adăugare

Să presupunem că, la un moment dat, întreprinderea vinde produse şi firmei RODEX SRL care are sediul pe strada Sapienţei, nr.44 bis, în localitatea Iaşi. Acest nou client trebuie "introdus" în baza de date, operaţiune care în SQL, se realizează prin comanda:

INSERT INTO clienti VALUES (1009, 'RODEX SRL', 'Sapienţei, 44 bis', '706600')

Fraza **INSERT** de mai sus poate fi scrisă şi sub forma:

INSERT INTO clienti (CodClient, NumeClient, Adresa, CodPostal)
VALUES (5009, 'RODEX SRL', 'Sapienţei 44 bis', 706600)

După cum se observă, după numele tabelei (CLIENŢI) au fost enumerate toate atributele pentru care se introduc valori prin clauza **VALUES**. Dacă nu s-ar fi cunoscut adresa clientului RODEX, atunci fraza **INSERT** ar fi avut una din formele:

INSERT INTO clienti (CodClient, NumeClient, Adresa, CodPostal)
VALUES (5009, "RODEX SRL", NULL, '6600') sau

INSERT INTO clienti (CodClient, NumeClient, CodPostal) VALUES (5009, 'RODEX SRL', '6600')

În noua linie a tabelei CLIENŢI valoarea atributului **Adresa** va fi NULL.

Conţinutul celor trei tabele din finalul capitolului 4 a fost realizat în ACCESS prin modulul următor în care comanda INSERT este argumentul unei comenzi *DoCmd.RunSQL*:

```
Sub Inserturi()
    ' CODPOST_LOC
    DoCmd.RunSQL ("INSERT INTO codPost_loc VALUES ('706600', 'Iasi', 'Iasi')  ;")
    DoCmd.RunSQL ("INSERT INTO codPost_loc VALUES ('706610', 'Iasi', 'Iasi')  ;")
    DoCmd.RunSQL ("INSERT INTO codPost_loc VALUES ('705300', 'Focsani', 'Vrancea')  ;")
    DoCmd.RunSQL ("INSERT INTO codPost_loc VALUES ('705725', 'Pascani', 'Iasi')  ;")
    DoCmd.RunSQL ("INSERT INTO codPost_loc VALUES ('706750', 'Tg.Frumos', 'Iasi')  ;")
    ' CLIENTI
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1001, 'TEXTILA SA', 'Bld. Copou, 87', '706600' );")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1002, 'MODERN SRL', 'Bld. Gării, 22', '705300' ) ;")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1003, 'OCCO SRL', NULL,  '706610') ;")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1004, 'FILATURA SA', 'Bld. Unirii, 145', '705300' ) ;")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1005, 'INTEGRATA SA', 'I.V.Viteazu, 115', '705725' ) ;")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1006, 'AMI SRL', 'Galaþiului, 72', '706750' );")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1007, 'AXON SRL', 'Silvestru, 2', '706610' ) ;")
    DoCmd.RunSQL ("INSERT INTO clienti VALUES (1008, 'ALFA SRL', 'Prosperitãþii, 15', '705725' ) ;")
    'FACTURI
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111111, 1003, #6-17-2005#, 17000, 0) ;")
```

```
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111112, 1001, #6-17-2005#,  2850, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111113, 1004, #6-18-2005#,  5850, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111114, 1003, #6-18-2005#,  2850, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111115, 1008, #6-18-2005#,  35700, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111116, 1008, #6-19-2005#,  8700, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111117, 1006, #6-20-2005#,  1100, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111118, 1007, #6-23-2005#,  15000, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111119, 1005, #6-24-2005#,  4720, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111120, 1003, #6-24-2005#,  3000, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111121, 1001, #6-24-2005#,  4250, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111122, 1007, #6-24-2005#,  8750, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111123, 1006, #6-25-2005#,  66000, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111124, 1004, #6-25-2005#,  38600, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111125, 1003, #6-30-2005#,  1280, 0);")
    DoCmd.RunSQL ("INSERT INTO facturi VALUES (111126, 1002, #6-01-2005#,  54250, 0);")
End Sub
```

După cum se observă, deliberat TVA colectată este declarată zero la toate liniile inserate în FACTURI. Aceasta pentru a avea motiv de modificare (UPDATE) – vezi paragraful pe peste linia curentă.

6.3.2 Modificarea valorilor unor atribute

Pentru modificarea valorilor unuia sau multor atribute dintr-o tabelă, comanda utilizată este **UPDATE** care are formatul general: **UPDATE tabelă SET atribut = expresie WHERE predicat**

Ca rezultat, vor fi modificate valorile atributului specificat, noile valori ale acestuia fiind cele care rezultă în urma evaluării expresiei; modificarea se va produce pe toate liniile tabelei care îndeplinesc condiţia specificată în predicat. Astfel comanda:

UPDATE facturi SET TVAColectata = INT(ValoareTotala * 19 / 1.19) / 100

va stabili valoarea TVA colectată pentru toate facturile (lipseşte clauza WHERE, deci vor fi afectate toate liniile din tabela FACTURI) pe 0.19/1.19 din valoarea totală a fiecărei facturi. Funcţia INT extrage doar partea întreagă dintr-un număr real (se elimină, deci, partea fracţionară), iar prin împărţirea rezultatului la 100 ne asigurăm că TVA va avea două poziţii la partea fracţionară.

În ACCESS această actualizare se poate face în două moduri. Grafic, putem crea o interogare de tip pe calapodul clasic al construirii machetelor (prezentat în paragraful 5.4), după "aducerea" tabelei FACTURI în machetă, selectând din meniul **Query Type** a opţiunii *Update Query* – vezi partea stângă a figurii 6.2. Partea dreaptă a figurii prezinta noua formă a machetei, în care apare linia **Update To** şi dispar **Sort** şi **Show**. La rubrica **Update To** se introduce expresia de calcul a valorilor atributului **TVAColectată**. Cea de-a două modalitate este cea procedurală, comanda prin care se lansează UPDATE-rile fiind, ca şi in cazul inserării, *DoCmd.RunSQL*.

```
Sub Updateuri()
   DoCmd.RunSQL ("UPDATE facturi SET TVAColectata = INT(ValoareTotala * 19 / 1.19) / 100  ;")
   DoCmd.RunSQL ("UPDATE facturi SET TVAColectata = 0 WHERE NrFact = 111117  ;")
   DoCmd.RunSQL ("UPDATE facturi SET TVAColectata = INT(ValoareTotala * 9 / 1.09) / 100 " & _
      " WHERE NrFact IN (111118, 111122) ;")
End Sub
```

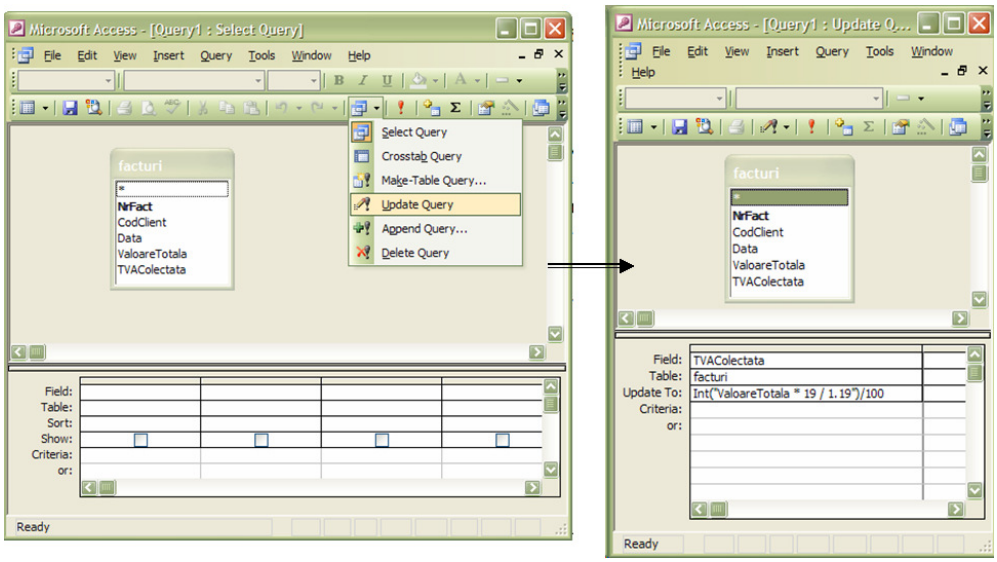

Figura 6.2. Interogare de modificare

Modulul de mai sus, specific ACCESSului conţine, pe lângă comanda UPDATE explicată, alte două, prin care facturii 111117 îi este declarat procentul zero de TVA, iar a doua indică faptul că toate produsele/serviciile din facturile 111118 şi 111122 au procentul de TVA de 9%. Deşi cei mai mulţi utilizatori ar opta pentru varianta grafică, va asigurăm că varianta procedurală este mai productivă atunci când sunt necesare actualizări repetate, aşa cum este cazul de mai sus.

6.3.3 Ştergeri

Operaţiunea de eliminarea a una sau mai multe linii dintr-o tabelă, pe baza unui predicat, se realizează în SQL prin comanda **DELETE** care are sintaxa: **DELETE FROM nume-tabelă WHERE predicat**

Dacă am dori să eliminăm din tabela CLIENŢI linia aferentă clientului MODERN SRL (cod 1002), comanda ar fi:

```
DELETE  FROM clienti WHERE CodClient = 1002
```

În ACCESS, similar interogărilor pentru modificare pot fi create şi interogări pentru ştergere, în care se specifică criteriul pe care trebuie să-l satisfacă liniile pentru a fi şterse din tabela indicată (fiind în criză de spaţiu, nu vom mai prezenta nici o figură în acest scop). Prin program, dacă am dori eliminarea tuturor înregistrărilor din tabelele bazei, apoi re-adăugarea şi re-modificarea lor, ne-am putea servi de blocul următor:

```
Sub Stergeri()
        DoCmd.RunSQL ("DELETE FROM facturi ;")
        DoCmd.RunSQL ("DELETE FROM clienti ;")
        DoCmd.RunSQL ("DELETE FROM codPost_loc ;")
        Call Inserturi
        Call Updateuri
End Sub
```

6.4 STRUCTURA DE BAZĂ A FRAZEI SELECT

Fără îndoială, cea mai gustată parte din SQL este cea legată de interogarea bazei, adică de obţinerea de informaţii din cele mai diverse, prin prelucrări, grupări etc. Ceea ce a fost prezentat în

paragraful 5.4 este doar o părticică din ceea ce poate fi "stors" dintr-o bază de date. Aşadar, în SQL o interogare se formulează printr-o frază **SELECT**. Aceasta prezintă trei clauze principale: **SELECT**, **FROM** şi **WHERE**.

- **SELECT** este utilizată pentru desemnarea listei de atribute (coloanele) din rezultat;
- **FROM** este cea care permite enumerarea relaţiilor din care vor fi extrase informaţiile aferente consultării;
- prin **WHERE** se desemnează condiţia (predicatul), simplă sau complexă, pe care trebuie să le îndeplinească liniile tabelelor (enumerate în clauza FROM) pentru a fi extrase în rezultat.

La modul general (şi simplist) o consultare simplă în SQL poate fi prezentată astfel: **SELECT C1, C2, ..., Cn FROM R1, R2, ..., Rm WHERE P.**

Execuţia unei fraze **SELECT** se concretizează în obţinerea unui rezultat de forma unei tabele (relaţii). Când clauza **WHERE** este omisă, se consideră implicit că predicatul P are valoarea logică "adevărat". Dacă în locul coloanelor **C1**, **C2**, ... **Cn** apare simbolul "*", în tabela-rezultat vor fi incluse toate coloanele (atributele) din toate relaţiile specificate în clauza **FROM**. De asemenea, în tabela-rezultat, nu este obligatoriu ca atributele să prezinte nume identic cu cel din tabela enumerată în clauza **FROM**. Schimbarea numelui se realizează prin opţiunea **AS**.

Uneori, rezultatul interogării "încalcă" poruncile modelului relaţional. Conform restricţiei de entitate, într-o relaţie nu pot exista două linii identice. Or, în SQL, rezultatul unei consultări poate conţine două sau mai multe tupluri identice. Pentru eliminarea liniilor identice este necesară utilizarea opţiunii **DISTINCT**: **SELECT DISTINCT C1, C2, ..., Cn FROM R1, R2, ..., Rm WHERE P**

Am început acest capitol prin a arăta fraza SELECT care se ascunde în spatele unei consultări "grafice" ACCESS. Fraza:

```
SELECT facturi.NrFact, facturi.Data, [ValoareTotala]-[TVAColectata] AS ValFaraTVA,
       facturi.TVAColectata, facturi.ValoareTotala
FROM facturi WHERE (((facturi.Data)>#6/20/2005#));
```

conţine şi un câmp calculat – **ValFărăTVA**, al cărui nume se specifică prin clauza AS, şi ale cărui valori se determină prin relaţia **ValoareTotală** – **TVAColectată**. Predicatul de selecţie (clauza WHERE) asigură extragerea în rezultat doar a facturilor emise după 20 iunie 2005.

Cum se poate introduce o interogare în ACCESS fără a o "desena" cu proiectantul de machete pentru interogări prezentat în paragraful 5.4 ? Mai întâi creăm o interogare foarte simplă – să-i zicem – INTEROG. După modelul indicat în stânga figurii 6.3 îi vizualizăm definiţia folosind opţiunea **SQL View**, iar apoi în fereastra care apare înlocuim fraza SELECT cu cea care ne interesează (frază care nu are, probabil, nici o legătură cu definiţia actuală a interogării), după care se apasă butonul **Run** (semnul mirării).

Iar dacă vrem să facem acelaşi lucru prin program, folosim modulul următor:

```
Sub interogareBETWEEN()
  Dim consultare As QueryDef
  Set consultare = CurrentDb.QueryDefs("interog")
  consultare.SQL = "SELECT * FROM facturi WHERE nrfact BETWEEN 111120 AND 111124 ; "
  DoCmd.OpenQuery ("interog")
End Sub
```

Modulul *interogareBETWEEN()* declară (prin **Dim**) obiectul *consultare* ca fiind definiţia unei interogări (**QueryDef**), preia (prin comanda **Set**) în *consultare* fraza SELECT care constituie definiţia

interogării create anterior – *interog* –, modifică definiţia acestea prin linia *consultare.SQL = "SELECT …"* şi, în final, execută noua variantă a interogării, rezultatul fiind similar variantei din figura 6.3.

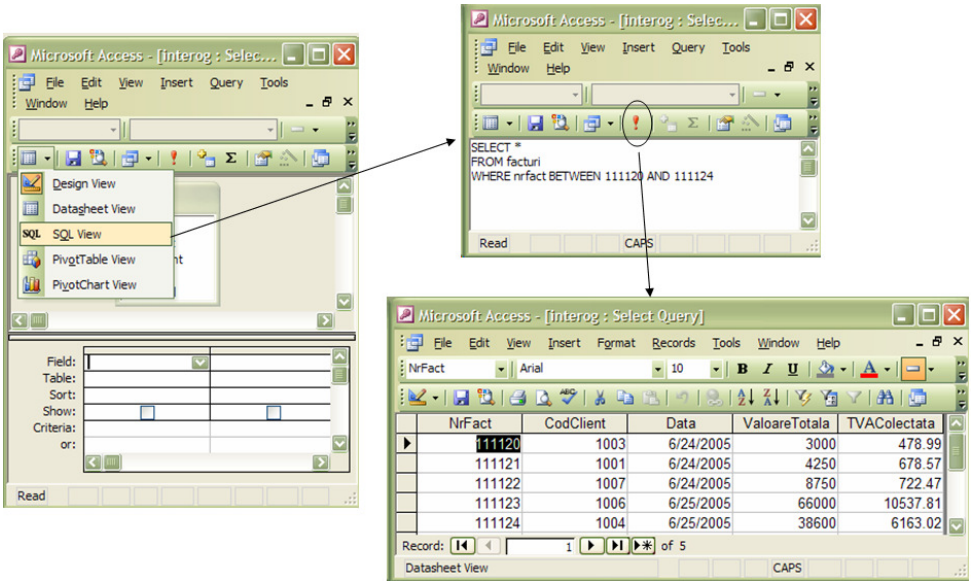

Figura 6.3. Schimbarea definiţiei unei interogări şi re-execuţia sa

Fireşte, noile definiţii pot fi salvate sau se poate renunţa la ele, mai ales atunci lucrăm cu module ce pot apelate ori de câte ori este nevoie.

Operatorul BETWEEN

Fraza SELECT de mai sus conţine operatorul BETWEEN care va extrage din tabelă numai facturile cu numerele cuprinse între 111120 şi 111124. Fără BETWEEN predicatul ar trebuit scris WHERE nrfact >= 111120 AND nrfact <= 111124.

Operatorul LIKE

Operatorul LIKE se foloseşte pentru a compara un atribut de tip şir de caractere (ex. **NumeClient**, **Adresa**, **Localitate**) cu un literal (constantă de tip şir de caractere). Astfel, dacă se doreşte obţinerea unei tabele-rezultat care să conţină numai clienţii ai căror nume începe cu litera M, putem scrie predicatul din clauza **WHERE** sub forma: **NumeClient LIKE "M%".** Deoarece după litera M apare semnul "%" (sau "*"), se vor extrage din tabela CLIENŢI toate tuplurile pentru care valoarea atributului **NumeClient** începe cu litera M, indiferent de lungimea acestuia (ex. MELCRET, MIGAS, MITA, MATSUSHITA etc.). Despre semnul "%" (sau "*") se spune că este un specificator multiplu, joker sau mască. Un alt specificator multiplu utilizat în multe versiuni SQL este liniuţa-de-subliniere ("_") sau semnul de întrebare ("?"). Spre deosebire de "%", "_" substituie un singur caracter. Diferenţa dintre cei doi specificatori multipli este pusă în evidenţă în continuare. Astfel, dacă interesează care sunt clienţii ai căror nume începe cu litera A şi sunt societăţi cu răspundere limitată (SRL-uri), fraza SELECT care furnizează răspunsul este:

SELECT * FROM CLIENŢI WHERE NumeClient LIKE "A__ SRL%"

(vezi partea stângă a figurii 6.4). Dacă s-ar fi utilizat simbolul "%" de maniera următoare:

SELECT * FROM CLIENŢI WHERE NumeClient LIKE "A%SRL%"

rezultatul ar fi fost cel din partea dreaptă a figurii.

CodClient	NumeClient	Adresa	CodPostal
1006	AMI SRL	Galaţiului, 72	706750

CodClient	NumeClient	Adresa	CodPostal
1008	ALFA SRL	Prosperităţii, 15	705725
1006	AMI SRL	Galaţiului, 72	706750
1007	AXON SRL	Silvestru, 2	706610

Figura 6.4. Folosirea specificatorilor multipli

În concluzie, "_" sau "?" înlocuiesc (substituie) un singur caracter, în timp ce "%" sau "*" înlocuiesc un şir de caractere de lungime variabilă (între 0 şi n caractere). Cei doi specificatori multipli pot fi utilizaţi împreună.

Operatorul IN

Un alt operator util este IN cu formatul general: **expresie1 IN (expresie2, expresie3, ...).** Rezultatul evaluării unui predicat ce conţine acest operator va fi "adevărat" dacă valoarea **expresiei1** este egală cu (cel puţin) una din valorile: **expresie2**, **expresie3**, ...

Spre exemplu, pentru a afla care sunt clienţii din localităţile din judeţele Iaşi şi Vaslui, fără utilizarea operatorului **IN** se scrie:

```
SELECT * FROM codpost_loc WHERE Judet = 'Iasi' OR Judet = 'Vaslui'
```

Iar utilizând **IN**:

```
SELECT * FROM codpost_loc WHERE Judet IN  ("Iasi", "Vaslui")
```

Operatorul IS NULL

O valoare nulă este o valoare nedefinită. Este posibil ca la adăugarea unei linii într-o tabelă, valorile unor atribute să fie necunoscute. În aceste cazuri valoarea atributului pentru tuplul respectiv este nulă. Reamintim că, prin definiţie, nu se admit valori nule pentru grupul atributelor care constituie cheia primară a relaţiei. Pentru aflarea clienţilor pentru care nu s-a introdus adresa, se poate scrie:

```
SELECT * FROM clienti WHERE Adresa IS NULL
```

Cum în baza noastră de date, numai clientului OCCO SRL nu-i cunoaştem adresa, rezultatul interogării este cel din figura 6.5 (în ACCESS valorile NULL se afişează ca şi cum ar conţine spaţii).

CodClient	NumeClient	Adresa	CodPostal
1003	OCCO SRL		706610

Figura 6.5. Extragerea valorilor NULLe

Observaţii

- Valoarea NULL nu se confundă cu valoarea zero (pentru atributele numerice) sau cu valoarea "spaţiu" (pentru atributele de tip şir de caractere)
- Operatorul NULL se utilizează cu IS şi nu cu semnul "=". Dacă s-ar utiliza expresia = NULL şi nu expresia IS NULL, rezultatul evaluării va fi întotdeauna fals, chiar dacă expresia nu este nulă !

Opţiunile DISTINCT şi ORDER BY

Dorim să aflăm judeţele în care firma are clienţi. Este necesară parcurgerea relaţiei CODPOST_LOC, singurul atribut care interesează fiind Judeţ:

```
SELECT DISTINCT Judet  FROM codpost_loc
```

După cum se observă în partea stângă a figurii 6.6, SQL nu elimină dublurile automat, iar dacă se doreşte ca în tabela-rezultat o localitate să figureze o singură dată, se utilizează opţiunea DISTINCT (rezultatul în dreapta figurii 6.6):

```
SELECT DISTINCT Judet  FROM codpost_loc
```

Judet
Vrancea
Iasi
Iasi
Iasi
Iasi
Iasi

Judet
Iasi
Vrancea

Figura 6.6. Fără şi cu DISTINCT

În continuare vrem să obţinem denumirea fiecărei localităţi şi judeţul în care se află, dar liniile rezultatului trebuie ordonate în funcţie de judeţ şi, în cadrul aceluiaşi judeţ, în ordinea inversă a localităţii (de la Z la A), fraza **SELECT** se formulează după cum urmează, rezultatul fiind prezentat în figura 6.7.

```
SELECT DISTINCT Localitate, Judet FROM codpost_loc
ORDER BY Judet ASC, Localitate DESC
```

Localitate	Judet
Tg.Frumos	Iasi
Pascani	Iasi
Iasi	Iasi
Focsani	Vrancea

Figura 6.7. Clauza ORDER BY

Opţiunile **ASCENDING** (crescător) şi **DESCENDING** (descrescător) indică deci modul în care se face ordonarea tuplurilor tabelei-rezultat al interogării. Prioritatea de ordonare este stabilită prin ordinea atributelor specificate în **ORDER BY**: ordonarea "principală" se face în funcţie de valorile primului atribut specificat; în cadrul grupelor de tupluri pentru care valoarea primului atribut este identică, ordinea se stabileşte după valoarea celui de-al doilea atribut specificat s.a.m.d. Dacă în **ORDER BY** lipsesc opţiunile **ASCENDING/DESCENDING**, ordonarea se face crescător.

6.5 JONCŢIUNI

După cum afirmam şi în paragraful 5.4 majoritatea informaţiilor obţinute dintr-o bază de date necesită "răsfoirea" simultană a două sau mai multe tabele. Interogarea *Fac_dupa20iunie2005v2* din figura 5.41 foloseşte trei tabele. Folosind opţiunea **SQLView** obţinem o frază SELECT cu totul remarcabilă:

```
SELECT facturi.NrFact, facturi.Data, clienti.NumeClient, codPost_loc.Localitate,
    [ValoareTotala]-[TVAColectata] AS Expr1, facturi.TVAColectata, facturi.ValoareTotala, *
FROM (codPost_loc INNER JOIN clienti ON codPost_loc.CodPostal = clienti.CodPostal)
    INNER JOIN facturi ON clienti.CodClient = facturi.CodClient
WHERE (((facturi.NrFact)>#6/20/2005#) AND ((codPost_loc.Localitate)="Iasi"));
```

Clauza FROM vine acum în centrul atenţiei prin apariţia clauzei INNER JOIN. Iată cum stau lucrurile: deoarece în lista pe care vrem să obţinem se găsesc atribute plasate în cele trei tabele, în clauza FROM trebuie enumerate cele trei **tabele;** în fapt, după cum am văzut în capitolul 4, cele trei tabele sunt legate prin restricţii referenţiale, atributele de legătura fiind cheile străine – cheile primare. Astfel, legătura dintre FACTURI şi CLIENŢI se poate realiza prin intermediul atributului **CodClient** care este cheie primară în CLIENŢI (tabela părinte) şi cheie străină în FACTURI (tabela copil).

Legătura dintre aceste două tabele care prezintă un câmp comun se numeşte joncţiune şi se simbolizează în SQL prin INNER JOIN:

```
SELECT *
FROM facturi INNER JOIN clienti ON facturi.CodClient=clienti.CodClient
```

Fără a intra în prea multe detalii teoretice, reţinem că, ori de câte ori informaţiile necesare şi condiţiile pe care trebuie să le îndeplinească acele informaţii privesc atribute aflate în tabele diferite, trebuie făcută joncţiunea acestor tabele. Când tabele nu pot fi joncţionate direct, trebuie aduse "cu forţa" în clauza FROM şi tabelele care să completeze "lanţul".

Ne interesează, spre exemplu, numărul şi data facturilor emise clienţilor din judeţul Iaşi. Numărul şi data facturilor se găsesc în tabela FACTURI (atributele **NrFact** şi **Data**), însă pentru denumirea judeţului există un atribut (**Judet**) în tabela CODPOST_LOC. Cele două tabele nu pot fi joncţionate direct, aşa încât atragem în interogare şi cea de-a treia tabelă – CLIENŢI:

```
SELECT NrFact, Data
FROM (facturi INNER JOIN clienti ON facturi.codclient=clienti.codclient)
      INNER JOIN codpost_loc ON codpost_loc.codpostal=clienti.codpostal
WHERE judet='Iasi'
ORDER BY NrFact
```

Scrisă sub formă de modul ACCESS – *InterogareJONCTIUNE1()* – şi lansată prin "apăsarea butonului **Run**, fraza SELECT obţine rezultatul din figura 6.8.

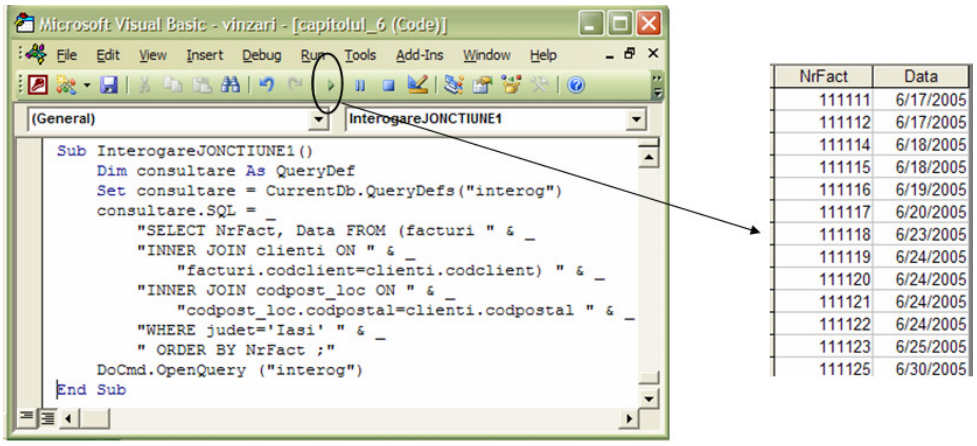

Figura 6.8. Un modul ACCESS cu frază SELECT ce joncţionează cele trei tabele (plus rezultatul)

Lăsam să se înţeleagă, la un moment dat, că în SQL pot fi formulate interogări mult mai complexe decât se poate realiza cu ajutorul machetei din paragraful 5.4. Haideţi să luăm o asemenea problemă, ce-i drept nu atât de complicată precum ameninţam: *Care sunt facturile emise în aceeaşi zi ca şi factura 111113 ?* Dificultatea ţine de faptul că cerinţa este formulată indirect, adică vrem să aflăm facturile emise într-o zi (**Data**), dar noi nu ştim data etalon, ci factura-etalon.

Problema propusă poate fi rezolvată relativ uşor folosind o subconsultare, după cum va fi prezentat într-un paragraf viitor. Până una-alta, soluţia pe care o avem în acest moment la îndemână se bazează pe *autojoncţiune*. Autojoncţiunea înseamnă joncţiunea unei tabele (FACTURI) cu ea-însăşi, practic, joncţiunea a două instanţe ale unei aceleaşi tabele. Pentru a joncţiune cele două instanţe trebuie să aibă pseudonime (aliasuri) diferite, în cazul nostru F1 şi F2. Întrucât ne interesează facturi emise *în aceeaşi zi* cu 111113, autojoncţiunea se face după atributul **Data**:

```
SELECT *
```

```
FROM facturi F1 INNER JOIN facturi F2 ON F1.data=F2.data
WHERE F2.NrFact = 111113
```

Iată rezultatul – vezi figura 6.9. Rezultatul conţine 10 coloane, cinci din prima instanţă a tabelei FACTURI (F1) şi cinci din a doua instanţă (F2). ACCESSul e destul de inspirat să scrie înaintea fiecărui atribut din ce instanţă provine.

F1.NrFact	F1.CodClient	F1.Data	F1.ValoareTotala	F1.TVAColectata	F2.NrFact	F2.CodClient	F2.Data	F2.ValoareTotala	F2.TVAColectata
111113	1004	6/18/2005	5850	934.03	111113	1004	6/18/2005	5850	934.03
111114	1003	6/18/2005	2850	455.04	111113	1004	6/18/2005	5850	934.03
111115	1008	6/18/2005	35700	5700	111113	1004	6/18/2005	5850	934.03

Figura 6.9. Facturile emise în aceeaşi zi ca şi 111113

Toate liniile rezultatului respectă condiţia de joncţiune - F1.data=F2.data. Jumătatea din dreapta a coloanelor se referă strict la factura etalon – 111113, iar cea din stânga la facturile din aceeaşi zi cu 111113, inclusiv factura-etalon. Pentru a răspunde punctual la problemă, precizăm atributele (coloanele) care ne interesează şi eliminăm din rezultat factura 111113:

```
SELECT F1.NrFact, F1.Data
FROM facturi F1 INNER JOIN facturi F2 ON F1.data=F2.data
WHERE F2.NrFact = 111113 AND F1.NrFact <> 111113
```

6.6 FUNCŢII-AGREGAT: COUNT, SUM, AVG, MAX, MIN

Cu funcţiile agregat facem un prim pas spre analiza datelor din bază. Li se spune funcţii agregat deoarece, în absenţa grupării (clauza GROUP BY – vezi ultimul paragraf din acest capitol) rezultatul unei asemenea funcţii are forma unei tabele cu o singură linie.

Funcţia COUNT

Contorizează valorile unei coloane, altfel spus, numără, într-o relaţie, câte valori diferite de NULL are coloana specificată. Dacă în locul unui atribut apare semnul asterisc (*) se numără liniile rezultatului. Astfel, dacă vrem să aflăm câţi clienţi are firma vom folosi interogarea (rezultatul său se află in stânga figurii 6.10).

```
SELECT COUNT (CodClient) AS Nr_Clienti1, COUNT (*) AS Nr_Clienti2 FROM clienti
```

Nr_Clienti1	Nr_Clienti2
8	8

NrClienti1	NrClienti2
16	16

Figura 6.10. Nişte COUNT-uri

Teoretic, în tabela CLIENŢI pot apărea şi clienţi cărora încă nu li s-a trimis încă nici o factură. Dacă vrem să aflăm răspunsul la întrebarea: *La câţi clienţi s-au trimis facturi ?*, am fi tentaţi să folosim interogarea:

```
SELECT COUNT (*) AS NrClienti1, COUNT(CodClient) AS NrClienti2
FROM clienti INNER JOIN facturi ON clienti.CodClient=facturi.CodClient
```

care, însă, ne va furniza un răspuns eronat (vezi partea dreaptă a figurii 6.10). În produsele care respectă recomandările standardelor SQL, rezultatul corect poate fi însă obţinut prin utilizarea clauzei **DISTINCT** astfel:

```
SELECT COUNT (DISTINCT CodClient) FROM facturi
```

În ACCESS această opţiune nu este acceptată, aşa încât mai aşteptăm până la paragraful dedicat subconsultărilor.

Funcţia SUM

Funcţia **SUM** calculează suma valorilor unei coloane. Pentru a afla suma valorilor totale ale facturilor, soluţia este cât se poate de simplă:

```
SELECT SUM (ValoareTotala) AS Total_ValoriFacturi FROM facturi
```

iar totalul valorilor pentru facturile trimise clientului AXON SRL este obţinut astfel:

```
SELECT SUM (ValoareTotala) AS Total_Fact_AXON
FROM facturi INNER JOIN clienti ON  facturi.CodClient = clienti.CodClient
WHERE NumeClient = 'AXON SRL'
```

Funcţiile MAX şi MIN

Determină valorile maxime, respectiv minime ale unei coloane în cadrul unei tabele. Valorile cea mai mică şi cea mai mare ale unei facturi se află astfel:

```
SELECT MIN(ValoareTotala), MAX(ValoareTotala) FROM facturi
```

Atenţie ! La întrebarea *Care este factura emisă cu cea mai mare valoare ?* nu putem răspunde deocamdată. Varianta următoare nu este corectă: **SELECT NrFactura, MAX(ValoareTotala) FROM facturi**

La execuţia acestei interogări se afişează un mesaj de eroare, soluţia problemei fiind posibilă ceva mai pe finalul capitolului.

6.7 SUB-CONSULTĂRI. OPERATORUL IN

O altă facilitate deosebit de importantă a limbajului SQL o constituie posibilitatea includerii (imbricării) a două sau mai multe fraze **SELECT**, astfel încât pot fi formulate interogări cu mare grad de complexitate. Operatorul cel mai des întrebuinţat este **IN**. Astfel, revenind la o problemă anterioară - *Care sunt facturile emise în aceeaşi zi în care a fost întocmită factura 111113 ?* – în locul epuizantei auto-joncţiuni putem recurge la subconsultare:

```
SELECT * FROM facturi WHERE NrFact <> 111113 AND Data IN
      (SELECT Data  FROM facturi WHERE NrFact=111113)
```

Sub-consultarea **SELECT Data FROM facturi WHERE NrFact = 111113** are ca rezultat o tabelă alcătuită dintr-o singură coloană (**Data**) şi o singură linie ce conţine valoarea atributului **Data** pentru factura 111113, ca în stânga figurii 6.11. Clauza **WHERE Data IN** determină căutarea în tabela FACTURI a tuturor tuplurilor (liniilor) care au valoarea atributului **Data** egală cu una din valorile tuplurilor (în cazul nostru, egală cu valoarea tuplului) din tabela obţinută prin "sub-consultare" (în cazul nostru, tabela din stânga figurii). Cu alte cuvinte, în acest caz **WHERE Data IN** va selecta toate facturile pentru care data emiterii este 18/06/2005 – partea dreaptă a figurii 6.11.

Data
6/18/2005

NrFact	CodClient	Data	ValoareTotala	TVAColectata
111114	1003	6/18/2005	2850	455.04
111115	1008	6/18/2005	35700	5700

Figura 6.11 Rezultatul sub-consultării (stânga) şi al interogării

Dacă s-ar schimba condiţia de selecţie, în sensul că ne-ar interesa facturile emise *în alte zile* decât cea în care a fost întocmită factura 111113 operatorul de sub-consultare va fi **NOT IN**:

```
SELECT * FROM facturi WHERE Data NOT IN
        (SELECT Data FROM facturi WHERE NrFact = 111113)
```

Ca să încheiem paragraful cu o interogare mai prezentabilă, ne interesează clienţii cărora li s-au trimis facturi întocmite în aceeaşi zi cu factura 111113:

```
SELECT DISTINCT NumeClient  FROM clienti WHERE CodClient IN
    (SELECT CodClient FROM facturi WHERE Data IN
            (SELECT Data FROM facturi WHERE NrFact = 111113))
```

Am ilustrat modul în care pot fi imbricate (înlănţuite, incluse) trei fraze SELECT.

6.8 REUNIUNE, INTERSECŢIE, DIFERENŢĂ

Operatorul pentru *reuniune* este deci **UNION**. De remarcat că, la reuniune, SQL elimină automat dublurile, deci nu este necesară utilizarea clauzei **DISTINCT**. Operatorul **UNION** este prezent în toate SGBD-urile importante. Dacă două relaţii, R1 şi R2 sunt *uni-compatibile*, adică au acelaşi număr de atribute care corespund sintactic (adică primul atribut din R1 este de acelaşi tip cu primul atribut din R2), se poate scrie **SELECT * FROM R1 UNION SELECT * FROM R2**.

În ceea ce ne priveşte, vrem să aflăm cum se numesc clienţii cărora le-am emis facturi pe 23 sau pe 24 iunie 2005, avem la dispoziţie două variante, una bazată pe operatorul logic OR:

```
SELECT DISTINCT NumeClient  FROM clienti  WHERE CodClient IN
    (SELECT DISTINCT CodClient FROM facturi
    WHERE Data = #6/23/2005# OR  Data = #6/24/2005#)
```

şi o altă bazată pe reuniune:

```
SELECT DISTINCT NumeClient  FROM clienti WHERE CodClient IN
     (SELECT DISTINCT CodClient FROM facturi WHERE Data = #6/23/2005# )
UNION
SELECT DISTINCT NumeClient  FROM clienti WHERE CodClient IN
     (SELECT DISTINCT CodClient FROM facturi WHERE Data = #6/24/2005#)
```

Pentru realizarea *intersecţiei* a două tabele unicompatibile, R1 şi R2, în standardele SQL a fost introdus operatorul **INTERSECT**: **SELECT * FROM R1 INTERSECT SELECT * FROM R2**. Dacă în produsele profesionale, precum DB2 (IBM) sau Oracle operatorul este prezent, în schimb multe din cele din categoria "uşoară", precum Visual Fox Pro şi ACCESS **INTERSECT** rămâne un deziderat, funcţionalitatea sa realizându-se prin subconsultări (operatorul **IN**) sau, uneori, prin joncţiune. Astfel, dacă dorim să aflăm cum se numesc clienţii cărora le-am emis facturi şi pe 23 şi pe 24 iunie 2005, soluţia cea mai la îndemână în ACCESS este:

```
SELECT DISTINCT NumeClient  FROM clienti WHERE CodClient IN
    (SELECT DISTINCT CodClient FROM facturi WHERE Data = #6/23/2005# )
     AND CodClient IN
            (SELECT DISTINCT CodClient FROM facturi WHERE Data = #6/24/2005#)
```

Diferenţa dintre tabelele R1 şi R2 (unicompatibile) se realizează utilizând operatorul **MINUS** sau **EXCEPT**, însă implementările sunt similare operatorului INTERSECT. Astfel, pentru a obţine clienţii cărora le-am emis facturi şi pe 24, *dar nu* şi pe 24 iunie 2005, soluţia ACCESS este:

```
SELECT DISTINCT NumeClient  FROM clienti WHERE CodClient IN
    (SELECT DISTINCT CodClient FROM facturi WHERE Data = #6/24/2005# )
     AND CodClient NOT IN
            (SELECT DISTINCT CodClient FROM facturi WHERE Data = #6/23/2005#)
```

6.9 GRUPAREA TUPLURILOR. CLAUZELE GROUP BY ŞI HAVING

În paragraful 5.4 făceam cunoştinţă cu o primă interogare în care era necesară gruparea liniilor. Ne interesa valoarea zilnică a vânzărilor într-o anumită perioadă, iar cadrul construirii interogării (*VinzariPeZile_interval_la_alegere*) lua forma din figura 5.42. Folosind din nou opţiunea **SQL View** să vedem fraza SELECT ce se ascunde în spatele machetei:

```
SELECT facturi.Data, Sum(facturi.TVAColectata) AS SumOfTVAColectata,
        Sum(facturi.ValoareTotala) AS SumOfValoareTotala,
        Sum([ValoareTotala]-[TVAColectata]) AS ValFaraTVA
FROM facturi
GROUP BY facturi.Data
HAVING (((facturi.Data) Between [Data initiala:] And [Data finala:]))
ORDER BY facturi.Data;
```

SQL permite utilizarea clauzei **GROUP BY** pentru a forma grupe (grupuri) de tupluri ale unei relaţii, pe baza valorilor comune ale unei coloane. În frazele **SELECT** formulate până în acest paragraf, prin intermediul clauzei **WHERE** au fost selectate tupluri din diferite tabele. Prin asocierea unei clauze **HAVING** la o clauză **GROUP BY** este posibilă selectarea anumitor grupe de tupluri ce îndeplinesc un criteriu.

Clauza GROUP BY

Rezultatul unei fraze **SELECT** ce conţine această clauză este o tabelă care va fi obţinută prin regruparea tuturor liniilor din tabelele enumerate în **FROM**, care prezintă o aceeaşi valoare pentru o coloană sau un grup de coloane. Formatul general este:

SELECT coloană 1, coloană 2, ..., coloană m FROM tabelă GROUP BY coloană-de-regrupare

Simplificăm problema, dorind o listă cu totalul zilnic al valorii facturilor emise. Fraza este ceva mai simplă:

SELECT Data, SUM (ValoareTotala) AS Total_Zilnic FROM facturi GROUP BY Data

Tabela-rezultat va avea un număr de linii egal cu numărul de date calendaristice distincte din tabela FACTURI. Pentru toate facturile aferente unei zile se va calcula suma valorilor, datorită utilizării funcţiei **SUM(ValoareTotala).** Succesiunea paşilor este următoarea:

1. Se ordonează liniile tabelei FACTURI în funcţie de valoarea atributului **Data** - figura 6.12.

NrFact	CodClient	Data	ValoareTotala	TVAColectata
111126	1002	6/1/2005	54250.00	8661.76
111112	1001	6/17/2005	2850.00	455.04
111111	1003	6/17/2005	17000.00	2714.28
111113	1004	6/18/2005	5850.00	934.03
111114	1003	6/18/2005	2850.00	455.04
111115	1008	6/18/2005	35700.00	5700.00
111116	1008	6/19/2005	8700.00	1389.07
111117	1006	6/20/2005	1100.00	0.00
111118	1007	6/23/2005	15000.00	1238.53
111120	1003	6/24/2005	3000.00	478.99
111121	1001	6/24/2005	4250.00	678.57
111122	1007	6/24/2005	8750.00	722.47
111119	1005	6/24/2005	4720.00	753.61
111123	1006	6/25/2005	66000.00	10537.81
111124	1004	6/25/2005	38600.00	6163.02
111125	1003	6/30/2005	1280.00	204.36

Figura 6.12. Pasul 1 al grupării

2. Se formează câte un grup pentru fiecare valoare distinctă a atributului **Data** - vezi figura 6.13.

	NrFact	CodClient	Data	ValoareTotala	TVAColectata
grup 1	111126	1002	6/1/2005	54250.00	8661.76
grup 2	111112	1001	6/17/2005	2850.00	455.04
	111111	1003	6/17/2005	17000.00	2714.28
grup 3	111113	1004	6/18/2005	5850.00	934.03
	111114	1003	6/18/2005	2850.00	455.04
	111115	1008	6/18/2005	35700.00	5700.00
grup 4	111116	1008	6/19/2005	8700.00	1389.07
grup 5	111117	1006	6/20/2005	1100.00	0.00
grup 6	111118	1007	6/23/2005	15000.00	1238.53
grup 7	111120	1003	6/24/2005	3000.00	478.99
	111121	1001	6/24/2005	4250.00	678.57
	111122	1007	6/24/2005	8750.00	722.47
	111119	1005	6/24/2005	4720.00	753.61
grup 8	111123	1006	6/25/2005	66000.00	10537.81
	111124	1004	6/25/2005	38600.00	6163.02
grup 9	111125	1003	6/30/2005	1280.00	204.36

Figura 6.13. Al doilea pas al grupării

3. Pentru fiecare din cele nouă grupuri se calculează suma valorilor atributului **ValoareTotala**. Tabela rezultat va avea nouă linii, ca în figura 6.14.

data	VinzariZilnice
6/1/2005	54250
6/17/2005	19850
6/18/2005	44400
6/19/2005	8700
6/20/2005	1100
6/23/2005	15000
6/24/2005	20720
6/25/2005	104600
6/30/2005	1280

Figura 6.14. Rezultatul final al grupării

Dacă interesează numărul facturilor emise pentru fiecare client, răspunsul poate fi obţinut prin interogarea:

```
SELECT NumeClient, COUNT(NrFact) AS NrFacturi_pe_Client
FROM facturi INNER JOIN clienti ON facturi.CodClient = clienti.CodClient
GROUP BY NumeClient
```

Până la standardul SQL:1999 şi publicarea *Amendamentului OLAP* la acest standard, în SQL nu puteau fi calculate, prin **GROUP BY**, subtotaluri pe mai multe niveluri. Pentru aceasta este necesară scrierea de programe în SGBD-ul respectiv.

Clauza HAVING

Clauza **HAVING** permite introducerea unor restricţii care sunt aplicate grupurilor de tupluri, deci nu tuplurilor "individuale", aşa cum "face" clauza **WHERE**. Din tabela rezultat sunt eliminate toate grupurile care nu satisfac condiţia specificată. Clauza **HAVING** "lucrează" împreună cu o clauză **GROUP BY**, fiind practic o clauză **WHERE** aplicată acesteia. Formatul general este:

```
SELECT coloană 1, coloană 2, .... , coloană m FROM tabelă
GROUP BY coloană-de-regrupare HAVING caracteristică-de-grup
```

Pentru facturile emise interesează valoarea zilnică a acestora (în funcţie de data la care au fost întocmite), dar numai dacă aceasta (valoarea zilnică) este de mai mare de 40000 lei noi (RON).

```
SELECT Data, SUM(ValoareTotala) AS Vinzari_Zilnice
FROM facturi GROUP BY Data HAVING SUM(ValoareTotala) > 40000
```

La execuţia acestei fraze, se parcurg cei trei paşi descrişi la începutul acestui paragraf, apoi, dintre cele nouă tupluri obţinute prin grupare, sunt extrase numai cele care îndeplinesc condiţia **SUM(ValoareTotala) > 40000**. Rezultatul final este cel din figura 6.15.

Data	Vinzari_Zilnice
6/1/2005	54250
6/18/2005	44400
6/25/2005	104600

Figura 6.15. Zilele cu vânzări mai mari de 40000 RON

Şi acum, o interogare cu adevărat interesantă: *Să se afişeze ziua în care s-au întocmit cele mai multe facturi !* Iată soluţia:

```
SELECT Data, COUNT(*) AS nr_facturilor FROM facturi
GROUP BY Data HAVING COUNT(*) >= ALL
            (SELECT COUNT(*) FROM facturi GROUP BY Data)
```

Avem de-a face cu o subconsultare al cărei rezultat (stânga figurii 6.16) serveşte drept termen de comparaţie al predicatului (condiţiei) formulat în clauza HAVING.

rezultatul subconsultării

Expr1000
1
2
3
1
1
1
4
2
1

rezultatul consultării dacă nu s-ar fi folosit clauza HAVING

Data	nr_facturilor
6/1/2005	1
6/17/2005	2
6/18/2005	3
6/19/2005	1
6/20/2005	1
6/23/2005	1
6/24/2005	4
6/25/2005	2
6/30/2005	1

rezultatul consultării cu aplicarea clauzei HAVING

Data	nr_facturilor
6/24/2005	4

Figura 6.16. Clauza HAVING cu folosirea unei subconsultări

Cam atât pentru acest capitol, disciplină, semestru (“informatic”) şi an (tot “informatic”) ! Vă mulţumim pentru răbdare, şi vă asigurăm că cine a ajuns cu cititul (şi înţelesul) până în acest punct, are toate şansele unui examen încununat de succes.