Baze de Date-Anul 1
Progres Proiect

Vom lucra in etape setand un deadline pentru fiecare interval in parte.

Pentru prima parte a proiectului trebuie sa proiectati o diagrama cu **minimum 12 tabele** (daca folositi mai putine tabele nu o sa primiti punctaj total – gasiti toate informatiile in Cerinte Proiect si Modalitate Notare)

I. **Cerintele urmatoare se pot implementa pe baza notiunilor invatate pana in acest moment. In cadrul cursului din saptamana 7 – 01.04 – vom discuta un exemplu care o sa cuprinda solutia cerintelor 1-8**

1. Descrierea modelului real, a utilității acestuia și a regulilor de funcționare.
2. Prezentarea constrângerilor (restricții, reguli) impuse asupra modelului.
3. Descrierea entităților, incluzând precizarea cheii primare.
4. Descrierea relațiilor, incluzând precizarea cardinalității acestora.
5. Descrierea atributelor, incluzând tipul de date și eventualele constrângeri, valori implicite, valori posibile ale atributelor.
6. Realizarea diagramei entitate-relație corespunzătoare descrierii de la punctele 3-5.
7. Realizarea diagramei conceptuale corespunzătoare diagramei entitate-relație proiectata la punctul 6. Diagrama conceptuală obținută trebuie să conțină minimum 6 tabele (fără considerarea subentităților), dintre care cel puțin un tabel asociativ.
8. Enumerarea schemelor relaționale corespunzătoare diagramei conceptuale proiectata la punctul 7.

II. Cerintele urmatoare se pot implementa in continuare pe baza notiunilor invatate pana in acest moment

10. Crearea tabelelor în SQL și inserarea de date coerente în fiecare dintre acestea (minimum 5 înregistrări în fiecare tabel neasociativ; minimum 10 înregistrări în tabelele asociative).

11. Formulați în limbaj natural și implementați 5 cereri SQL complexe ce vor utiliza, în ansamblul lor, următoarele elemente:
   a. operație join pe cel puțin 4 tabele
   b. filtrare la nivel de linii
   c. subcereri nesincronizate în care intervin cel puțin 3 tabele
   d. utilizarea a cel puțin 2 funcții pe șiruri de caractere, 2 funcții pe date calendaristice, a funcțiilor NVL și DECODE, a cel puțin unei expresii CASE
   e. ordonări

12. Implementarea a 3 operații de actualizare (UPDATE) sau suprimare (DELETE) a datelor utilizând subcereri.

13. Crearea unei secvențe ce va fi utilizată în inserarea înregistrărilor în tabele (punctul 10).

-- EXEMPLU SECVENTA

1. In cadrul tabelului departamente creati o secventa pentru generarea codurilor de departamente, cu numele SEQ_DEPT. Secvenţa va începe de la 40, va creşte cu 10 de fiecare dată, va avea valoarea maximă 10000 si nu va cicla.

```
SELECT * FROM DEPARTAMENTE;

CREATE SEQUENCE SEQ_DEPT
INCREMENT by 10
START WITH 40
MAXVALUE 10000
NOCYCLE;

--se insereaza date utilizand secventa creata anterior
INSERT INTO departamente
VALUES (SEQ_DEPT.NEXTVAL, 'IT', null);

INSERT INTO departamente
VALUES (SEQ_DEPT.NEXTVAL, 'HR', 100);

SELECT * FROM departamente;
```

```
SELECT seq_dept.CURRVAL
FROM dual;  -- afisam valoarea urmatoare

DESC user_sequences;

SELECT INCREMENT_BY, MAX_VALUE, MIN_VALUE
FROM   USER_SEQUENCES
WHERE  SEQUENCE_NAME = 'SEQ_DEPT';  -- detalii despre secventa creata
```

--Exemplu - referire chei externe -> chei unice

```
CREATE TABLE test1
   ( x number(3),
     y number(3),
     z number(3),
     constraint unic unique(x, y)
   );

CREATE TABLE test2
   ( a number(3),
     b number(3),
     c number(3),
     constraint fk_ab FOREIGN KEY(a, b) REFERENCES test1(x, y)
   );
```

-- Exemplu - cheie primara compusa

```
CREATE TABLE test1
   (id_test1 number(5) constraint pk_test1 primary key,
    titlu varchar2(20)
    );

CREATE TABLE test2
   (id_test2 number(5),
    id_test1 number(5) constraint fk_test2_test1 references test1(id_test1),
    constraint pk_compus primary key(id_test2, id_test1)
    );
```

-- VERIFICARE

INSERT INTO test1
VALUES(10, 'TITLU10');

INSERT INTO test1
VALUES(20, 'TITLU20');

INSERT INTO test1
VALUES(30, 'TITLU30');

commit;

INSERT INTO test2
VALUES(100, 10); -- **corect**

INSERT INTO test2
VALUES(200, 10); -- **corect**

INSERT INTO test2
VALUES(100, 10); -- **unique constraint violated**

INSERT INTO test2
VALUES(300, 50);
-- integrity constraint (SYSTEM.FK_TEST2_TEST1) violated - parent key not found
-- de aceea avem nevoie de constrangerea de cheie externa in tabelul test2, chiar daca avem si
-- constrangerea de cheie primara compusa
-- id_test1 trebuie sa refere valoarea din tabelul de legatura

commit;

-- **STOCARE data si ora**

```
create table test
(data_ang date);

insert into test
values (to_date('10-05-2021 21:30', 'dd-mm-yyyy hh24:mi'));

select * from test;

select to_char(data_ang, 'hh24:mi')
from test;
```

--IMPLEMENTARE SUPERENTITATE SI SUBENTITATI
-- daca avem superentitatea ANGAJAT cu atributele cod_angajat, nume, prenume
-- si subentitatile PROGRAMATOR si SOFER, unde PROGRAMATOR are atributul specific limbaj_prog
-- iar SOFER categorie_permis

```
CREATE TABLE ANGAJAT
   (cod_angajat number(5) constraint pkey_ang1 primary key,
    nume varchar2(20) not null,
    prenume varchar2(20) not null,
    tip_angajat varchar2(25) not null,
    limbaj_prog varchar2(20),
    categorie_permis varchar2(20)
    );

INSERT INTO angajat
VALUES(10, 'KING', 'STEVEN', 'SOFER', NULL, 'B');

INSERT INTO angajat
VALUES(20, 'POP', 'JOHN', 'PROGRAMATOR', 'JAVA', NULL);

SELECT * FROM ANGAJAT;
```

UPDATE

III. Cerintele urmatoare se pot implementa in continuare pe baza notiunilor invatate pana in acest moment

9. Realizarea normalizării până la forma normală 3 (FN1-FN3).

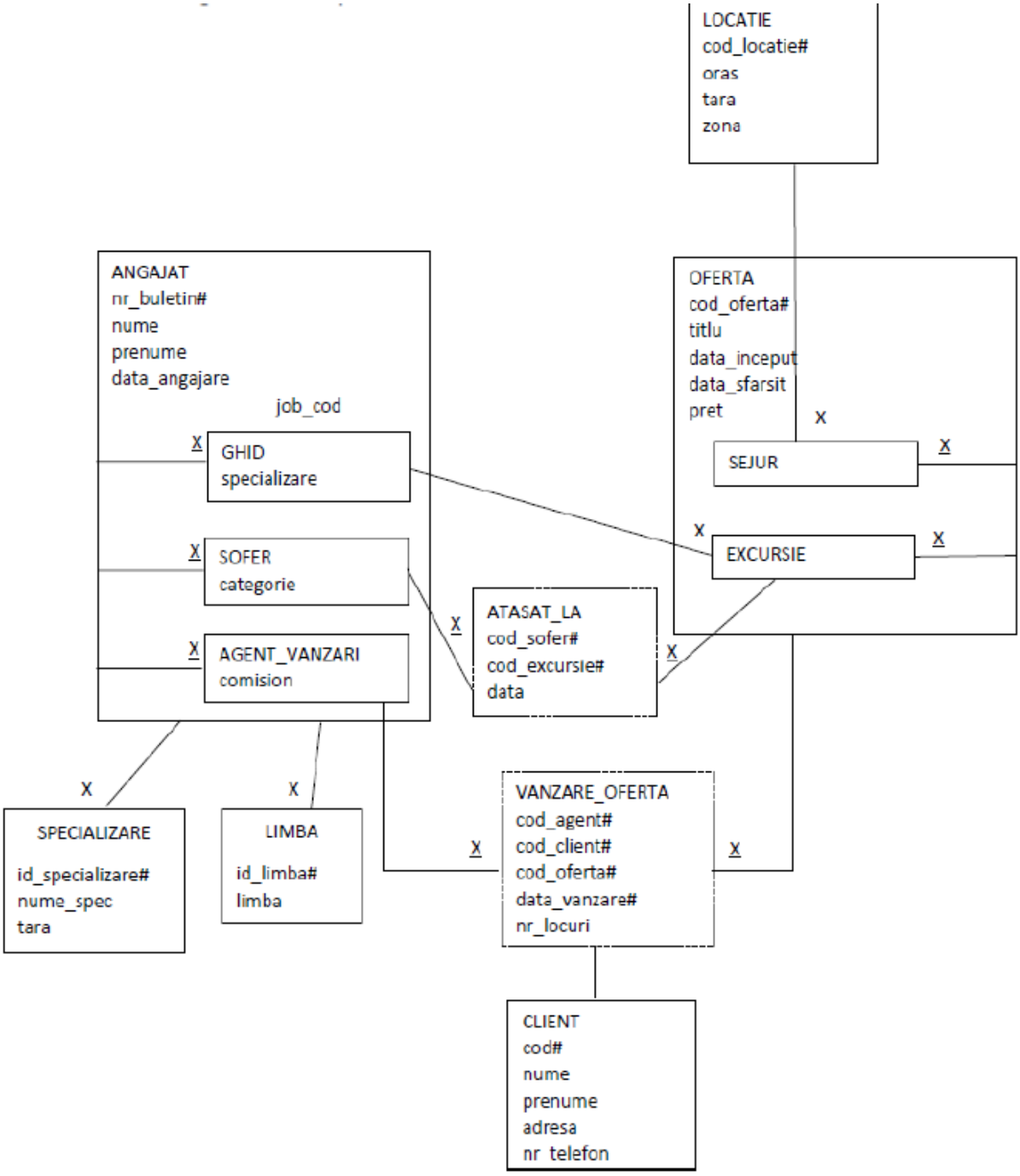
11: subcereri sincronizate în care intervin cel puțin 3 tabele
grupări de date, funcții grup, filtrare la nivel de grupuri

16. Formulați în limbaj natural și implementați în SQL: o cerere ce utilizează operația *outer-join* pe minimum 4 tabele și

17. Optimizarea unei cereri, aplicând regulile de optimizare ce derivă din proprietățile operatorilor algebrei relaționale. Cererea va fi exprimată prin expresie algebrică, arbore algebric și limbaj (SQL), atât anterior cât și ulterior optimizării.

IMPLEMENTARE SUPER SI SUB – ENTITATI:

Pentru implementarea super si sub – entitatilor o sa luam ca reper diagrama pe care o aveti in **Tema4**, cea cu angajatii si ofertele de tip sejur si excursie.



La nivel de design (diagrama conceptuala) se procedeaza exact cum vedeti in diagrama de mai sus, si anume in diagrama conceptuala. Daca **ANGAJATII** de tip **GHID** sunt in relatie cu **OFERTA** de tip **EXCURSIE**, atunci in diagrama conceptuala legam subentitatea **GHID** de subentitatea **EXCURSIE** deoarece aici totul trebuie sa fie proiectat cat mai clar, in special pentru faptul ca diagrama conceptuala ne ghideaza pe noi in logica aplicatiei si implicit in implementare. La fel se observa si in cazul angajatilor de tip **SOFER** care sunt atasati si ei tot la **OFERTA** de tip **EXCURSIE**. Deci in design avem grija sa proiectam totul cat mai clar, luand in considerare toate regulile de functionare.

In implementare lucrurile stau putin altfel pentru ca in SQL nu putem crea un tabel in cadrul altui tabel. In acest caz se implementeaza totul doar intr-un tabel, tipurile de angajati fiind de fapt valori de tip **varchar2**, valori inserate pe coloana **job_cod** (sau ii putem spune **tip_angajat** incat sa fie mai sugestiv numele).

Vom avea asa:

```
CREATE TABLE ANGAJAT
        ( cod number(4) constraint pk_angajat primary key,
          nume ... ,
          prenume ... ,
          data angajare ... ,
          tip_angajat varchar2(20) not null,
          specializare varchar2(20),
          categorie varchar2(20),
          comision varchar2(20)
        );
```

Observam coloana **tip_angajat** ca **nu poate fi NULL** pentru ca un angajat trebuie sa fie de tip **GHID** sau **SOFER** sau **AGENT_VANZARI** (trebuie sa aiba un tip), iar coloanele **specializare**, **categorie** si **comision** pot avea valoarea NULL deoarece daca inseram un angajat de tip **GHID** el o sa aiba o specializare deoarece acest atribut este specific GHIDULUI, dar nu o sa aiba **categorie** si **comision**. La fel se intampla si pentru **SOFER** care are doar **categorie** si pentru **AGENT_VANZARI** care are doar **comision**.

Identic se procedeaza si pentru **OFERTA**. In OFERTA o sa avem cheia externa **cod_angajat** (sau nr_buletin - este acelasi lucru, doar denumirea este diferita). Noi in design observam ca un angajat de tip **GHID** este legat de o oferta de tip **EXCURSIE**, dar facem asta pentru a avea un design cat mai detaliat care sa ne ajute la partea de implementare. In implementare noi avem doar cheia externa **cod_angajat** in **OFERTA**, dar din design noi stim ca un **ANGAJAT** de tip **GHID** conduce **OFERTE** de tip **EXCURSIE**, deci vom implementa asa:

```
SELECT a.cod_angajat, a.nume, a.prenume, a.specializare, o.cod_oferta, o.titlu, o.pret
FROM angajat a join oferta o on (a.cod_angajat = o.cod_angajat)
WHERE tip_angajat = 'ghid' and tip_oferta = 'excursie';
```

Dupa cum observati design-ul ne ajuta sa proiectam cat mai logic diagrama si in acelasi timp ne ajuta la partea de implementare, oferindu-ne o logica a aplicatiei.

EXEMPLU ADAUGARE CONSTRANGERI:

In continuare o sa va prezint un exemplu care o sa va ajute la partea de constrangeri. Am primit intrebarea – Cand stim ca este nevoie doar de o cheie externa sau trebuie o cheie primara compusa? O sa luam exemplul urmator din care o sa intelegeti exact modul de gandire.

Sa luam ca exemplu un tabel in care stocam like-urile pe care userii le lasa la postari. In primul caz ne gandim sa facem in acest tabel o cheie primara compusa din id_like#, id_user#, id_post#. Daca procedam asa stim ca trebuie sa fie unica intreaga intrare formata din cele trei coloanal, deci putem avea astfel de exemple:

like1 user10 post100

like1 user20 post100 - deci userul20 a dat acelasi like la aceeasi postare (este gresit)

like2 user10 post100 - deci userul10 a mai dat inca un like la aceeasi postare (este gresit)

Observam ca nu este corect daca avem cheie primara compusa deoarece in acest caz intreaga intrare (linie) trebuie sa fie unica, dar putem avea variatii ale acelor valori - si anume acelasi like cu alt utilizator, un like diferit cu acelasi utilizator la aceeasi postare, cazuri despre care stim ca nu sunt corecte.

In acest caz noi avem nevoie ca like-ul sa fie unic - deci **cheie primara** si mai stim ca daca un user da un like la o postare, nu mai poate da inca unul la aceeasi postare. Adica nu este correct un exemplu de tipul acesta:

like1 user10 post100

like2 user10 post100

Ce inseamna acest lucru? Inseamna ca o combinatie de tipul **id_user si id_postare trebuie sa fie unica**. Daca am userul10 cu poastarea 100, nu pot sa il mai am inca o data tot cu postare 100. In final vom avea **id_like - primary key**, **user_id - foreign key**, **post_id - foreign key**, iar **user_id si post_id unique**.

PROGRES FINAL

IV. Cerintele urmatoare se pot implementa in etapa finala:

11. utilizarea a cel puțin 1 bloc de cerere (clauza WITH)

16. Formulați în limbaj natural și implementați în SQL: o cerere ce utilizează operația *outer-join* pe minimum 4 tabele și două cereri ce utilizează operația *division*.

18. a. Realizarea normalizării BCNF, FN4, FN5.
b. Aplicarea denormalizării, justificând necesitatea acesteia.