6. INDECSI. DESIGN SI OPTIMIZARE DE BAZE DE DATE. JOIN-URI

Cuprins

6.1. INDECSI

6.1.1. Conceptul de index

Un index reprezinta o structura de date aditionala, redundanta, care, atasata unei tabele SQL, poate spori viteza de cautare si ordonare a datelor in acea tabela. Un index se poate defini pentru una sau mai multe coloane ale tabelei; ca efect, informatia din coloanele alese va fi copiata intr-o structura separata, fiecare element din aceasta structura avand o referinta catre inregistrarea de la care a provenit. Indexul poate fi stocat fie sub forma de fisier(e), fie in memoria RAM a statiei pe care ruleaza serverul de baze de date.

Intr-un index, elementele stau ordonate, ceea ce face gasirea unei informatii foarte rapida.

Conceptul de index nu intervine numai in bazele de date. O aplicatie foarte cunoscuta a sa este cuprinsul unei carti: informatia este redundanta (titlurile capitolelor se regasesc si in cadrul cartii) insa ajuta enorm in gasirea unei informatii, fiecare titlu de capitol referind o pagina din cadrul cartii. Fara cuprins, ar fi necesara parcurgerea partiala sau totala a cartii pentru a gasi informatia cautata, in acelasi fel in care intr-o tabela SQL ar fi necesara parcurgerea inregistrarilor tabelei pana la identificarea celei dorite.

Ducand paralela mai departe, unele carti au mai multe tipuri de cuprins. Un bun exemplu este Pagini Aurii: vom gasi acolo atat un cuprins al domeniilor de activitate, ordonat alfabetic pentru o cautare facila, cat si un cuprins al firmelor ordonat dupa numele firmei; ambele contin referinte catre pagina in care se gaseste informatia in cauza. Motivul prezentei mai multor cuprinsuri este ca utilizatorul sa poata efectua o cautare rapida dupa mai multe criterii. In acelasi fel, intr-o tabela SQL se pot defini mai multi indecsi, pentru diferite coloane sau grupuri de coloane, astfel incat sa fie optimizata cautarea informatiei dupa mai multe criterii/coloane.

6.1.2. Explicarea functionarii

Fie o tabela SQL numita Produse care are, printre altele, o coloana Denumire si una Pret. Sa presupunem ca a fost definit un index pentru coloana Denumire. Daca este efectuata urmatoarea interogare:

```
SELECT * FROM Produse WHERE Denumire='Carte'
```

serverul de baze de date se va folosi de index-ul pentru coloana Denumire, va identifica in cadrul lui toate intrarile care au denumirea 'Carte' si apoi se va folosi de referintele pe care index-ul le are catre inregistrarile din tabela pentru a returna inregistrarile solicitate. In acest fel, nu a mai fost necesara parcurgerea intregii tabele SQL.

Daca insa este folosita interogarea:

```
SELECT * FROM Produse WHERE Pret>10
```

neexinstand un index pentru coloana Pret, serverul de baze de date este nevoit sa caute informatia in toata tabela, ceea ce duce la o penalitate de viteza ce creste odata cu sporirea numarului de inregistrari din tabela.

Indexul este util si daca se incearca obtinerea listei de produse ordonate dupa denumire:

```
SELECT * FROM Produse ORDER BY Denumire
```

Informatiile din index fiind deja ordonate, serverul nu mai are altceva de facut decat sa obtina si sa returneze inregistrarea corespunzatoare fiecarei intrari din index, in ordinea din index. O ordonare descrescatoare va insemna o simpla parcurgere in sens invers a intrarilor din index.

6.1.3. Avantaje si dezavantaje ale folosirii indecsilor

Principalul avantaj al crearii de indecsi este sporirea vitezei la cautare si ordonare. De ramarcat insa ca acest lucru se intampla doar in masura in care cautarea/ordonarea se face folosind informatii din coloana sau coloanele pentru care este definit indexul. Acesta este motivul pentru care se recomanda ca indecsii sa fie creati in primul rand pentru coloanele care apar frecvent in clauze WHERE.

Un alt avantaj al indecsilor este acela ca ei pot introduce restrictii asupra coloanei/coloanelor pentru care sunt definiti. Spre exemplu, ei pot impune ca pe coloana respectiva sa existe numai valori unice – lucru de care se asigura serverul MySQL la introducerea fiecarei valori, nefiind necesar ca clientul sa faca aceasta verificare. (detalii suplimentare la tipuri de index).

Dezavantajele sunt doua:

- **penalitatea de viteza** introdusa la operatiile de inserare, stergere si modificare, unde, pe langa modificarea informatiei din tabela, este necesara si actualizarea indexului
- **spatiul suplimentar ocupat** – indexul reproduce parti ale inregistrarilor din tabela. Cu cat se creeaza mai multi indecsi sau cu cat tabela creste in dimensiuni, cu atat mai multa informatie este duplicata, spatiul ocupat de indecsi putand deveni important

6.1.4. Tipuri de index

Din punct de vedere al numarului de coloane cuprinse, un index poate fi:
- **uni-coloana** – indexul este creat pentru o singura coloana
- **multi-coloana** – un index ce cuprinde date din mai multe coloane. Ordinea in care sunt introduse coloanele in index conteaza: MySQL se poate folosi de un index multi-coloana in cazul interogarilor in care clauza WHERE cuprinde valori ale primelor cateva sau ale tuturor coloanelor din index (insa nu si in cazul in care sunt folosite doar valori ale coloanelor 2, 3 etc)

Din punct de vedere al constrangerilor pe care le impun asupra datelor din coloana/coloanele pentru care sunt creati, indecsii pot fi:

- **index simplu** – coloana respectiva este indexata, insa fara a impune restrictii asupra datelor din ea
- **index unic** – valorile acelei coloane trebuie sa fie unice. Acest fapt are doua implicatii:
    - la incercarea de introducere a unei valori duplicat pe o astfel de coloana, MySQL va genera o eroare si nu va efectua operatia. Singura exceptie este in cazul coloanelor ce permit NULL, unde valoarea NULL poate aparea in oricate exemplare pe coloana indexata
    - valorile coloanei fiind unice, serverul MySQL poate folosi algoritmi mult mai eficienti pentru cautarea si ordonarea informatiei. Spre exemplu, daca avem o coloana cu index simplu numita ”DenumireProdus”, o interogare de tipul *SELECT * FROM Produse WHERE DenumireProdus='Crizantema'* ar trebui sa parcurga *toate* valorile indexului pentru a determina setul complet de inregistrari cu valoarea 'Crizantema', deoarece pot exista mai multe inregistrari ce au aceeasi valoare in cadrul coloanei. In schimb, daca indexul pe coloana este de tip unic, MySQL va parcurge indexul doar pana la gasirea primei valori, stiind faptul ca ea nu se poate repeta. Serverul va fi cel care se va asigura de faptul ca valorile coloanei nu se repeta, prin respingerea operatiilor care ar crea duplicate

- **cheie primara** (*primary key*) – reprezinta un index unic dar care nu permite NULL. Cheia primara are in bazele de date un statut special: ea este gandita sa identifice in mod unic fiecare inregistrare din tabela. Fiecare tabela poate avea un singur index de tip primary key. Valorile coloanei desemnate ca primary key nu pot fi NULL si sunt unice pentru fiecare inregistrare. In cele mai dese cazuri coloana ce joaca rol de cheie primara are un tip de date intreg (pentru ca operatiile ce implica datele ei sa decurga cat mai rapid) si atributul suplimentar AUTO_INCREMENT (pentru asigurarea automata a unicitatii valorilor). Amintim faptul ca intr-o tabela MySQL nu poate exista decat o singura coloana cu atributul AUTO_INCREMENT si aceea trebuie declarata ca primary key.

Cheia primara este un procedeu prin care putem identifica in mod unic fiecare rand al unei tabele. Sa observam faptul ca intotdeauna referirea la o coloana a unei tabele se poate face precis, indicand numele acesteia; in schimb, inregistrarile nu au nume propriu, referirea la una sau mai multe inregistrari facandu-se indirect, prin intermediul valorilor acestora (...WHERE Nume LIKE '%escu' AND Varsta>20). Cheia primara realizeaza pe verticala ceea ce numele de coloane realizau pe orizontala: un fel de „nume" unic pentru fiecare inregistrare. Exista insa o deosebire: daca in cazul coloanelor numele faceau parte din structura tabelei (si deci erau subiect de DDL), cheia primara este un artificiu prin care „numele" randurilor sunt memorate direct in tabela, sub forma de valori (si deci subiect de DML).

***Nota: i**n MySQL exista si alte doua tipuri de indecsi (FULLTEXT si SPATIAL) care sunt insa mai specializati si nu vor fi discutati aici.*

6.1.5. Crearea indecsilor

6.1.5.1. La crearea tabelei

Indecsii pot fi creati odata cu tabela sau ulterior. Oricum ar fi, ei fac parte din definitia tabelei si se manipuleaza cu instructiuni de tip DDL.

La crearea unei tabele, indecsii pot fi specificati in doua moduri:
- dupa lista definitiilor de coloane, precizand tipul de index si apoi, intre paranteze, coloanele pentru care se aplica. Pentru tipul de index se pot folosi cuvintele cheie INDEX, UNIQUE sau PRIMARY KEY
- folosind clauze suplimentare in definitiile coloanelor respective. Clauzele adaugate pot fi, ca si mai sus:
  - INDEX, pentru index simplu
  - UNIQUE, pentru index unic
  - PRIMARY KEY, pentru cheie primara

Exemple:

```
# crearea unei tabele Persoane in care coloana CNP are index de tip unic;index specificat separat
CREATE TABLE Persoane (CNP INT(13), Nume VARCHAR(30) NOT NULL, UNIQUE(CNP))

# acelasi lucru, insa cu specificarea indexului ca parte a definitiei de coloana
CREATE TABLE Persoane (CNP INT(13) UNIQUE, Nume VARCHAR(30) NOT NULL)

# specificare de indecsi multipli pentru tabela Produse (ID si CNP), in 3 variante
CREATE TABLE Produse (id INT NOT NULL PRIMARY KEY, Denumire VARCHAR(100) UNIQUE);
CREATE TABLE Produse (id INT NOT NULL, Denumire VARCHAR(100), PRIMARY KEY(id), UNIQUE(Denumire));
CREATE TABLE Produse (id INT NOT NULL PRIMARY KEY, Denumire VARCHAR(100), UNIQUE(Denumire));

# crearea unui index unic multi-coloana
CREATE TABLE Persoane(Nume VARCHAR(50), Prenume VARCHAR(50), UNIQUE(Nume,Prenume));
```

6.1.5.2. Dupa crearea tabelei

MySQL pune la dispozitie doua posibilitati de definire a unui index odata ce tabela a fost creata si, eventual, populata cu date:

- folosind ALTER TABLE…ADD tip_index(coloana1,coloana2,...), unde tip_index poate fi, din nou, INDEX, UNIQUE sau PRIMARY KEY

```
ALTER TABLE NumeTabela ADD tip_index(col1, col2,...)
```

- folosind CREATE INDEX, care se mapeaza pe ALTER TABLE…ADD INDEX, insa **NU poate fi folosita pentru crearea cheii primare**

```
CREATE [UNIQUE] INDEX nume_index ON NumeTabela (col1, col2, ...)
```

Exemple:

```
# crearea unei tabele Users, initial fara indecsi
CREATE TABLE Users (id INT NOT NULL, Fullname VARCHAR(100), Username VARCHAR(100));

# crearea cheii primare pe coloana id
ALTER TABLE Users ADD PRIMARY KEY(id);

# crearea unui index unic pe coloana Username, in doua variante
ALTER TABLE Users ADD UNIQUE(Username);
CREATE INDEX uname ON Users(Username);
```

Adaugarea unui index dupa crearea/popularea tabelei face subiectul urmatoarelor restrictii:

- daca incercam sa definim un index de tip unic (UNIQUE sau PRIMARY KEY) pe o coloana care contine duplicate, operatia va fi respinsa
- daca incercam sa definim un index de tip PRIMARY pe o colcoloana care permite NULL, operatia va fi respinsa

6.1.6. Vizualizarea indecsilor unei tabele

Vizualizarea indecsilor definiti pentru o tabela se poate realiza in urmatoarele moduri:

- folosind instructiunea SHOW CREATE TABLE. Indecsii se regasesc ca parte a definitiei de tabela
- folosind instructiunea DESCRIBE, prezentata intr-un material anterior si care afiseaza si indecsii tabelei
- folosind instructiunea dedicata SHOW INDEX:

```
SHOW INDEX FROM NumeTabela;
```

O informatie importanta afisata de instructiunile de vizualizare de indecsi este numele indecsilor, ce poate fi necesar in alte operatii (ex: stergere de index).

6.1.7. Stergerea indecsilor

Stergerea indecsilor se poate realiza in doua moduri:
- folosind ALTER TABLE…DROP tip_index [(nume)]:

```
ALTER TABLE Persoane DROP UNIQUE(CNP);
```

```
ALTER TABLE Persoane DROP PRIMARY KEY;
```

- folosind instructiunea DROP INDEX, care se mapeaza pe cea precedenta dar are sintaxa usor diferita:

```
DROP INDEX nume_index ON NumeTabela
```

Exemple:

```
# stergerea cheii primare pentru tabela Users
ALTER TABLE Users FROP PRIMARY KEY;

# stergerea indexului de pe coloana Username;numele indexului il aflam in prealabil cu SHOW INDEX
DROP INDEX uname ON Users
```

6.1.8. Efecte ale indecsilor unici in operatiile de modificare a datelor

6.1.8.1. Discutie

Odata cu restrictiile introduse de indecsi (mai exact cei unici si primary key), apar si noi cazuri/nuante in operatiile de modificare a datelor din tabele, impreuna cu noi clauze/instructiuni care ofera flexibilitatea necesara in astfel de cazuri. Atunci cand o coloana nu permite duplicate, operatiile de tip INSERT sau UPDATE care ar crea valori duplicat pe coloana in cauza sunt respinse de catre server. Clientul poate preintampina aparitia erorilor indicand de la bun inceput serverului cum sa actioneze in cazul valorilor duplicat, prin folosirea unor clauze speciale in instructiunile INSERT si UPDATE.

Acest lucru este util nu atat in cazul instructiunilor care introduc/modifica o singura inregistrare, ci atunci cand operatia dorita implica mai multe inregistrari. Spre exemplu, daca dorim reunirea email-urilor din doua tabele insa fara a crea duplicate, sau actualizarea numarului de telefon al persoanelor dintr-o tabela folosind date mai noi dintr-o alta tabela etc.

6.1.8.2. Operatii INSERT

La introducerea de date pe o coloana de tip index unic, serverul nu va permite introducerea unei valori care se gaseste deja pe coloana in cauza. Daca clientul nu ia nici un fel de masuri, incercarea de introducere de duplicat se va solda cu o eroare:

```
CREATE TABLE Users(Username VARCHAR(10) UNIQUE);
INSERT INTO Users VALUES('user1');
INSERT INTO Users VALUES('user1');
ERROR 1062 (23000): Duplicate entry 'user1' for key 1
```

Nota: *in cazul indecsilor de tip unic pe coloane care permit NULL se poate introduce valoarea NULL in oricate exemplare fara a fi generata o eroare.*

In cazul in care clientul este constient de posibilitatea aparitiei duplicatelor pe coloane cu index unic, insa doreste ca acestea sa fie tratate intr-un anume mod de catre server fara a genera o eroare (ex: duplicatele sa fie ignorate, suprascrise etc), el are la dispozitie urmatoarele posibilitati:

- atunci cand dorim ca, odata introdusa o valoare, orice duplicat ulterior sa fie ignorat fara a genera o eroare, putem folosi clauza IGNORE in instructiunea INSERT. Ea va determina serverul sa nu introduca valorile duplicat, generand un warning pentru fiecare dintre ele.

```
INSERT IGNORE INTO NumeTabela VALUES(val1,val2,...)
```

Nota: *in raportul prezentat la finalul instructiunii INSERT IGNORE, valoarea „Duplicates" indica numarul de valori duplicat de pe coloane cu index unic si care au fost ignorate.*

- daca dorim ca, la aparitia unei valori duplicat, aceasta sa o suprascrie pe cea existenta in tabela, vom folosi instructiunea REPLACE (extensie non-standard MySQL), care are aceeasi sintaxa ca si INSERT insa trateaza diferit duplicatele. De remarcat insa ca REPLACE inlocuieste **INTREAGA** inregistrare, nu numai valoarea de pe coloana duplicat; ea este echivalenta cu o operatie DELETE ce sterge vechea inregistrare, urmata de o operatie INSERT cu noile valori.

```
REPLACE INTO NumeTabela VALUES(val1,val2,...)
```

Nota: *numarul de inregistrari modificate raportat de instructiunea REPLACE poate fi mai mare decat cel de randuri introduse! Aceasta deoarece, in cazul intalnirii unei valori duplicat pe o coloana cu index unic, va fi efectuata intai stergerea vechii inregistrari si apoi introducerea celei noi, in consecinta rezultand doua randuri modificate.*

- daca dorim ca, la aparitia unei valori duplicat, sa fie modificate numai anumite coloane ale inregistrarii ce contine valoarea deja existenta, putem adauga instructiunii INSERT clauza suplimentara ON DUPLICATE KEY UPDATE, urmata de o serie de atribuiri *numecoloana=expresie*. La fiecare intalnire a unei valori duplicat, serverul va efectua modificarile specificate in clauza suplimentara:

```
INSERT INTO NumeTabela VALUES(val1,val2,...) ON DUPLICATE KEY UPDATE col1=expr1, col2=expr2,...
```

Exemplu: fie o tabela Pacienti care persoanele vizitate de catre un medic de familie. Tabela contine, printre altele, o coloana de tip PRIMARY KEY numita id, una de tip INT numita NumVisits (numarul de vizite) si una de tip DATE numita LastVisit ce memoreaza data ultimei vizite. Dorim sa introducem inregistrari in tabela fara a avea grija duplicatelor:

```
CREATE TABLE Patients(id INT PRIMARY KEY, Name VARCHAR(50), NumVisits INT, LastVisit DATE);

# pacientul John a fost vizitat prima data in 3 iunie 2007
INSERT INTO Patients VALUES(1, 'John', 1, '2007-06-03');

/* a doua vizita a avut loc in 21 august 2007; daca nu luam masuri, introducerea inregistrarii
produce o eroare: */
INSERT INTO Patients VALUES(1, 'John', 2, '2007-08-21');
ERROR 1062 (23000): Duplicate entry '1' for key 1

/* rescriem instructiunea INSERT astfel incat sa actualizeze campurile NumVisits si LastVisit in
caz de pacient duplicat */
```

```
INSERT INTO Patients VALUES(1, 'John', 2, '2007-08-21')
      ON DUPLICATE KEY UPDATE NumVisits=NumVisits+1, LastVisit=CURRENT_DATE;
Query OK, 2 rows affected (0.00 sec)
```

6.1.8.3. Operatii UPDATE

Operatiile de tip UPDATE pot cauza si ele aparitia de valori duplicat pe coloane cu index unic. Clientul poate folosi clauza suplimentara ON DUPLICATE KEY SET urmata de o serie de atribuiri coloana=expresie, la fel ca in cazul lui INSERT:

```
UPDATE NumeTabela SET col1=expr1,... ON DUPLICATE KEY SET colX=exprX, colY=exprY,...
```

6.2. ELEMENTE DE DESIGN AL BAZELOR DE DATE

6.2.1. Structurarea informatiei in tabele

6.2.1.1. Primul impuls

Tehnic, deseori toata informatia dintr-o baza de date ar putea fi pastrata intr-o singura tabela. Sa consideram ca dorim sa memoram in MySQL prezenta studentilor la cursuri, si cream in prima faza o tabela SQL dupa modelul foilor de prezenta distribuite la curs:

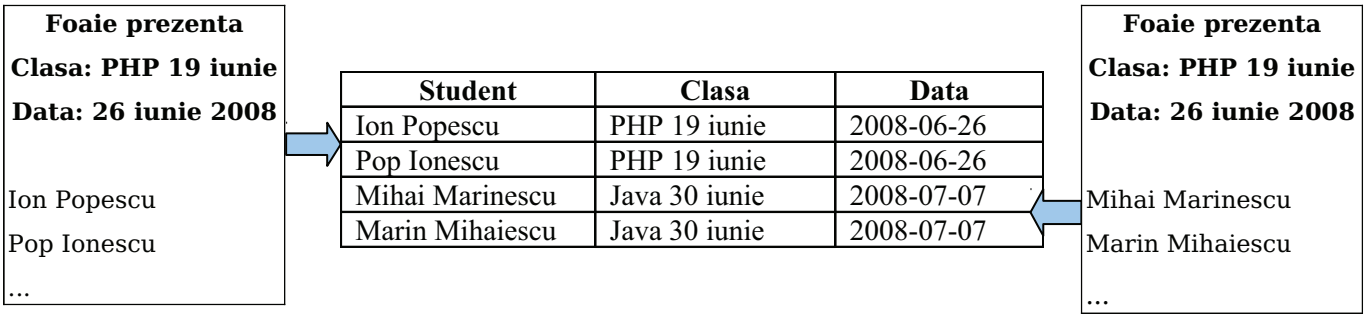


Tabela astfel construita ne permite sa aflam, pentru fiecare student care a venit cel putin o data la curs, in ce date a fost prezent la curs, cate prezente are per total si din ce clase face parte. Constatam insa un mare inconvenient: tabela memoreaza numai prezentele, nu si absentele, de aceea nu putem afla informatii precum:

- in ce date a absentat un student, cate absente are per total la o anumita clasa, ce procent de prezenta are raportat la numarul total de cursuri ale clasei
- care este lista de studenti a unei clase

Student	Clasa	Data	Prezent
Ion Popescu	PHP 19 iunie	2008-06-26	da
Pop Ionescu	PHP 19 iunie	2008-06-26	da
Ulm Ulmeanu	PHP 19 iunie	2008-06-26	nu
Mihai Marinescu	Java 30 iunie	2008-07-07	da
Marin Mihaiescu	Java 30 iunie	2008-07-07	da
Andrei Andreescu	Java 30 iunie	2008-07-07	nu

Luand in considerare aceste aspecte, decidem sa modificam structura tabelei prin introducerea unei coloane care sa memoreze starea studentului (prezent/absent), urmand ca la fiecare curs sa fie trecuti in baza de date toti studentii fiecarei clase, marcand fiecare student prezent sau absent in functie de situatie (vezi tabelul alaturat). Cu aceasta noua structura, putem obtine si informatiile anterior indisponibile – spre exemplu:

```
# lista de studenti a unei clase
SELECT * FROM Prezenta WHERE Clasa='PHP 19 iunie'

# numarul de absente al unui student la o anumita clasa
SELECT COUNT(*) FROM Prezenta
              WHERE Clasa='Java 30 iunie' AND Student='Andrei Andreescu' AND Prezent='nu'
```

Aparent totul este ok si structura tabelei permite realizarea scopurilor propuse. Chiar daca tabela ar putea fi folosita in aceasta forma, exista cateva inconveniente majore – unele sunt deja vizibile in acest stadiu, celelalte se descopera de obicei pe parcursul utilizarii tabelei:

- redundanta. Pentru fiecare data in care are loc o sedinta a unei anumite clase, este memorat in baza de date numele clasei si numele fiecarui student in parte. In plus, daca un student face parte din doua clase, numarul de aparitii ale numelui sau se dubleaza. Concret: pentru o clasa cu 20 de studenti si care are 15 sedinte, numele unui student (acelasi nume de fiecare data!) va aparea de 15 de ori in baza de date, iar numele clasei va aparea de 15x20 = 300 ori. Se remarca un consum de spatiu absolut nejustificat – in fond, la fiecare sedinta este vorba de *acelasi* student si de *aceeasi* clasa.
- posibile greseli la introducerea acelorasi informatii de mai multe ori. Sa presupunem ca exista doua persoane care centralizeaza si introduc in baza de date informatiile legate de prezenta; una dintre ele, mai neatenta, scrie numele lui Marin Mihaiescu fara i ("Marin Mihaescu"). Din acel moment, instructiunile SQL de tip SELECT care vor incerca sa efectueze statistici pentru acest student vor raporta informatii necorespunzatoare cu realitatea: SELECT * FROM Prezenta WHERE Student='Marin Mihaiescu' va intoarce doar rezultatele cu numele scris corect – asadar, numarul de prezente sau de absente raportat va fi mai mic decat cel real (in functie de inregistrarile la care s-au facut greselile). In aceste conditii este posibil ca un factor de decizie, care analizeaza prezenta studentului, sa hotarasca in mod eronat penalizarea acelui student pentru procent prea mare de absente.
- pe masura ce aplicatia se dezvolta, poate aparea necesitatea memorarii unor informatii suplimentare in baza de date. De exemplu, dorim sa includem in baza de date si data de inceput a fiecarei clase, sau data nasterii fiecarui student. A adauga aceste informatii, in forma actuala a tabelei, ar insemna adaugarea a inca doua coloane, care nu ar face decat sa sporeasca teribil redundanta si asa exagerata a tabelei
- sesizam faptul ca in tabela exista de fapt informatii care nu au de a face cu ideea de prezenta la un curs. Si in viata reala, apartenenta studentilor la diferitele clase nu se *defineste* in foaia de prezenta; daca dorim sa aflam din ce clasa face parte un student, ne vom referi probabil la catalogul clasei. In acelasi fel, intuim ca ar trebui sa existe o structura de date separata in care sa se defineasca "catalogul" clasei, si de care sa ne putem folosi mai apoi pentru a indica prezenta fiecarui student din catalog in functie de data

In concluzie, structura de tabele a unei baze de date nu trebuie neaparat (iar uneori este chiar contraindicat!) sa corespunda modului in care este structurata sau interactioneaza informatia in realitate.

6.2.1.2. Corectia

O parte a informatiilor din tabela creata mai sus pot fi refolosite. De exemplu, lista de clase ar putea fi memorata intr-un singur loc (si deci intr-un singur exemplar!) – mai exact o tabela separata –, iar tabela de prezenta sa faca doar referire la inregistrarile din ea de cate ori este nevoie. In plus, desi am fi poate tentati sa includem si numele studentilor in tabela de clase, sesizam la timp ca un student poate face parte din mai multe clase, si deci numele sau ar aparea din nou multiplicat. In consecinta, se impune crearea unei tabele separate cu studenti si a alteia cu clase. Rezulta in final 3 tabele – fie numele lor Studenti, Clase si Prezenta.

Aceasta impartire in mai multe tabele are avantajul ca fiecare tabela contine un singur tip de informatie (studenti, clase, prezenta la curs), ceea ce face extinderea fiecareia foarte usoara: putem oricand adauga in tabela de studenti o coloana cu CNP sau data nasterii, putem adauga in tabela de clase coloane pentru data de

incepere si de incheiere etc., fara a afecta datele existente sau functionalitatea programelor care se bazau pe structura initiala.

Impartirea in tabele trebuie facuta insa de asa natura incat sa existe in continuare conexiunile intre diferitele informatii: sa stim ce student carei clase apartine, la ce student si clasa se refera fiecare inregistrare din tabelul de prezenta etc. Daca consideram exemplul tabelei de prezenta, acest lucru se realizeaza inlocuind numele studentului cu un numar ce refera o inregistrare din tabela de studenti, si numele clasei cu un alt numar ce refera o inregistrare din tabela de clase:

Prezenta			
Student	**Clasa**	**Data**	**Prezent**
~~Ion Popescu~~ 104	~~PHP 19 iunie~~ 23	2008-06-26	da
...	...	...	...

Studenti	
ID	**Nume**
104	Ion Popescu
237	Pop Ionescu
...	...

Clase	
ID	**Nume**
23	PHP 19 iunie
98	Java 30 iunie
...	...

Pentru ca acest sistem sa functioneze, este necesar ca fiecare inregistrare din tabela de studenti si din cea de clase sa fie identificata in mod unic printr-un numar, care sa permita referirea la acea inregistrare din alte tabele. Este momentul sa ne amintim de indecsi de tip PRIMARY KEY – acestia aveau proprietatea ca valorile de pe coloana indexata erau unice si NOT NULL, identificand astfel in mod unic fiecare inregistrare a tabelei pentru care a fost creat indexul. Pentru structura noastra de tabele, avem nevoie asadar de:

- crearea unui index de tip PRIMARY KEY in tabela cu studenti. Tabela contine nume si eventual date de nastere – nici una dintre aceste informatii nu este unica la nivel de student, de aceea suntem nevoiti sa introducem o coloana suplimentara, de tip numeric, care va juca rolul de PRIMARY KEY
- crearea unui index de tip PRIMARY KEY in tabela cu clase. Spre deosebire de tabela de studenti, numele claselor sunt unice, insa din motive de performanta vom prefera tot crearea unei coloane dedicate pentru PRIMARY KEY
- inlocuirea numelor de studenti din tabela de prezenta cu valori ale cheii primare din tabela cu studenti. Coloana ce memora numele de student in schimba tipul de date din sir de caractere in numeric
- inlocuirea numelor de clase din tabela de prezenta cu valori ale cheii primare din tabela cu clase. Idem, coloana respectiva capata un tip de date numeric

Iata cum ar arata comenzile de creare ale celor 3 tabele:

```
CREATE TABLE Studenti (    ID INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
                                Nume VARCHAR(30) NOT NULL,
                                DataNastere DATE);
CREATE TABLE Clase (            ID INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
                                Nume VARCHAR(30));
CREATE TABLE Prezenta (    ID_student INT UNSIGNED NOT NULL,
                                ID_clasa INT UNSIGNED NOT NULL,
                                Data   DATE,
                                Prezent ENUM('da','nu'));
```

Continutul tabelei de prezenta ar putea arata ca in tabelul alaturat. Observam ca exista in continuare redundanta a informatiei pe coloanele ce indica studentul si clasa, insa este una minimala, necesara pentru reconectarea informatiei divizate in tabele.

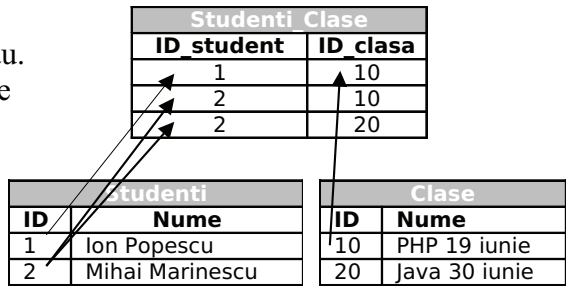
Prezenta			
Student	**Clasa**	**Data**	**Prezent**
1	1	2008-06-19	da
1	1	2008-06-26	da
2	1	2008-06-26	da
3	1	2008-06-26	nu
4	2	2008-07-07	da
5	2	2008-07-07	da
6	2	2008-07-07	nu

Nota: *o coloana a unei tabele care contine valori ale cheii primare dintr-o alta tabela (referind astfel inregistrari din acea tabela) poarta numele de* ***cheie externa****. Acest concept reprezinta baza crearii de relatii intre tabele SQL.*

Observatie: *cheia primara si externa nu este obligatoriu sa fie numerice, dar e mai eficient - atat din punct de vedere al vitezei, cat si al spatiului ocupat*

6.2.2. Tipuri de relatii intre tabele

Relatiile intre informatiile unei baze de date pot fi impartite in 3 categorii:



- **relatii 1-la-1** – de exemplu, relatia intre persoana si CNP-ul sau. In cele mai dese cazuri, informatiile aflate in relatie de 1 la 1 se vor regasi sub forma de coloane ale aceleiasi tabele SQL

- **relatii 1-la-N sau N-la-1** (numite si *one-to-many*) – apar atunci cand unei inregistrari dintr-o tabela ii pot corespunde mai multe inregistrari din alta. Spre exemplu, fiecarei inregistrari din tabela de prezenta ii corespunde o singura inregistrare din tabela de studenti (prezenta este pentru un anume student), insa unei inregistrari din tabela de studenti ii pot corespunde mai multe inregistrari din tabela de prezenta (un acelasi ID de student se va gasi de mai multe ori in tabela de prezenta, corespunzator fiecarei sedinte de curs)

- **relatii M-la-N** (*many-to-many*) – apar atunci cand, in relatia dintre doua tabele, unei inregistrari dintr-o tabela ii pot corespunde mai multe inregistrari din cealalta si reciproc. Spre exemplu, relatia intre studenti si clase: o clasa are mai multi studenti, iar un student poate face parte din mai multe clase. Acest tip de relatie se modeleaza in SQL sub forma unei tabele suplimentare, de legatura, care are drept coloane chei externe ce se refera la inregistrarile din cele doua tabele aflate in relatia M-la-N:

```
CREATE TABLE Studenti_Clase (ID_student INT UNSIGNED NOT NULL, ID_clasa INT UNSIGNED NOT NULL)
```

Nota: *numele coloanei ce joaca rolul de cheie externa nu este obligatoriu sa fie identic cu al cheii primare din tabelul referit.*

Studentul Ion Popescu face parte dintr-o singura clasa (PHP 19 iunie) si deci va fi necesara o singura inregistrare in tabela de legatura, iar Mihai Marinescu face parte din doua; in consecinta, tabela de legatura va avea 3 inregistrari.

Observam faptul ca fiecare inregistrare din tabela de legatura este unica (nu poate aparea de mai multe ori aceeasi combinatie clasa-student), asadar cele doua coloane pot fi folosite pentru a defini o cheie primara multi-coloana.

6.2.3. Specificarea relatiilor ca parte a structurii bazei de date

6.2.3.1. Utilitate

In exemplul de mai devreme, ce continea tabelele Prezenta, Studenti si Clase, MySQL nu este constient de relatiile intre cele 3 tabele, si de aceea nu ne poate opri sa producem date invalide:

- putem introduce in tabela de prezenta o inregistrare al carei ID de student sa nu corespunda nici unei inregistrari din tabela cu studenti
- putem sterge inregistrari din tabela cu studenti, lasand astfel „orfane” inregistrari din tabela Prezenta
- putem modifica ID-ul unei inregistrari din tabela cu studenti fara a modifica si inregistrarile ce-i corespund in tabela Prezenta

Exista insa posibilitatea ca, in anumite conditii (descrise mai jos), relatiile intre tabele sa fie incluse ca parte a structurii bazei de date. Acest fapt ii permite serverului de baze de date sa „supravegheze" operatiile care implica coloanele de legatura dintre aceste tabele, degrevand aplicatia client de verificarile aferente.

6.2.3.2. Sintaxa si conditii

Cheile externe pot fi incluse ca parte a definitiei de tabela, folosind clauza FOREIGN KEY:

```
CREATE TABLE Prezenta(
       IDStudent INT,
       IDClasa INT,
       Data DATE,
       Prezent TINYINT(1),
       FOREIGN KEY(IDStudent) REFERENCES Studenti(id),
       FOREIGN KEY(IDClasa) REFERENCES Clase(id)
)
```

Sintaxa generala a unei definitii de cheie externa este:

```
FOREIGN KEY (NumeColoana1) REFERENCES NumeTabela(NumeColoana2) ON UPDATE actiune ON DELETE actiune
```

unde *actiune* poate fi:

- **CASCADE** – la modificarea inregistrarii din tabela parinte sunt afectate automat toate inregistrarile care depind de aceasta. Spre exemplu, cu ON DELETE CASCADE, daca stergem un student, vor fi sterse automat toate inregistrarile corespunzatoare lui din tabela Prezenta; cu ON UPDATE CASCADE, daca schimbam ID-ul unui student, va fi actualizat id-ul de pe coloana de cheie externa a tuturor inregistrarilor corespunzatoare din tabela Prezenta
- **RESTRICT** – sunt interzise operatiile care ar lasa orfane inregistrari din tabele aflate in relatie cu aceasta
- **NO ACTION** – echivalent cu RESTRICT in MySQL
- **SET NULL** – inregistrarile dependente primesc valoarea NULL pe coloana ce joaca rolul de cheie externa, **in masura in care definitia coloanei nu include modificatorul NOT NULL**

Pentru a putea defini o cheie externa ca parte a structurii unei tabela trebuie indeplinite urmatoarele conditii:

- ambele tabele implicate in relatie trebuie sa foloseasca motorul de stocare InnoDB. Celelalte motoare nu suporta inca aceasta faciiltate
- coloana cheie externa si coloana referita trebuie sa aiba in general acelasi tip de date si aceiasi parametri:
  - pentru numere intregi, ambele coloane trebuie sa aiba aceeasi dimensiune si aceeasi setare SIGNED/UNSIGNED
  - pentru siruri de caractere, coloanele trebuie sa aiba acelasi charset si collation, insa nu neaparat si aceeasi lungime
- ambele coloane implicate (cheia externa dintr-o tabela si cea referita din cealalta) trebuie sa fie indexate

Proprietatea datelor unei coloane de tip cheie externa de a referi numai valori valide din tabela parinte este numita **integritate referentiala**.

6.3. Join-uri

6.3.1. Conceptul de join si tipuri

Un **join** reprezinta o interogare SQL in care sunt manipulate date din mai multe tabele. Cel mai des intalnit exemplu este cel al interogarii de tip SELECT care reuneste coloanele mai multor tabele SQL. Iata pentru inceput o astfel de interogare care nu dispune de o clauza WHERE:

```
SELECT * FROM Studenti,Prezenta
```

Interogarea de mai sus va returna un rezultat obtinut astfel: fiecare inregistrare din tabela Studenti va fi "alipita" pe rand cu fiecare inregistrare din tabela Clase, rezultand de fiecare data o inregistrare care contine coloanele ambelor tabele. Rezultatul interogarii va reprezenta produsul cartezian al inregistrarilor din cele doua tabele; numarul de inregistrari rezultat va fi egal cu produsul numarului de inregistrari al ambelor parti.

Observatie: *faptul ca informatia fost divizata in mai multe tabele, si ca acum fiecare tabela contine informatie care sa permita "conectarea" cu celelalte, nu face baza de date constienta de intentiile noastre. Ceea ce noi gandim ca "chei externe" este pentru motorul de baze de date o informatie ca oricare alta, o succesiune de valori numerice pe o coloana, parte a informatiei utile memorate in baza de date. Pentru ca informatia respectiva sa isi realizeze menirea, este necesara specificarea unor clauze suplimentare in cadrul join-ului.*

Pentru a preciza felul in care se realizeaza corespondenta intre inregistrari in cazul unor tabele intre care exista o relatie – si deci a unifica numai randurile din cele doua tabele care corespund unul altuia – se poate folosi clauza WHERE:

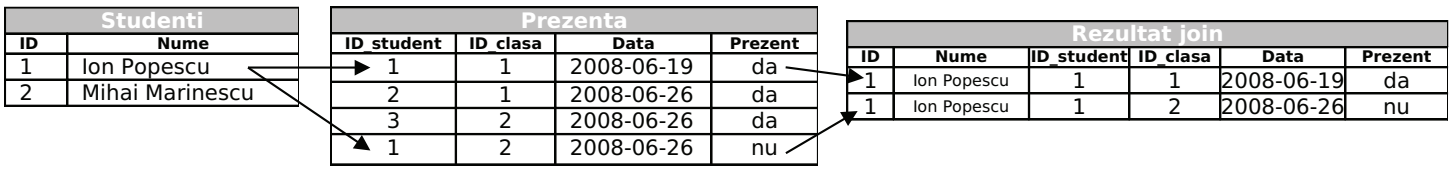
```
SELECT * FROM Studenti,Prezenta WHERE Studenti.ID=Prezenta.ID_student
```

Am precizat in acest fel faptul ca unei inregistrari cu un anumit ID din tabela Studenti ii vor corespunde doar inregistrarile care au aceeasi valoare pe coloana StudentID din tabela Prezenta (urmand, in fond, in ordine inversa logica folosita atunci cand s-a divizat informatia in tabele).

Nota: *clauza WHERE scrisa astfel elimina toate perechile in care coloana comuna are valoarea NULL in oricare dintre tabele, deoarece operatorul = returneaza NULL in cazul in care unul dintre operanzi este NULL.*

La fel ca la orice interogare de tip SELECT, rezultatul furnizat de un join poate contine doar coloanele necesare, si se pot adauga conditii suplimentare pentru restrictionarea inregistrarilor returnate. Spre exemplu, daca dorim sa obtinem lista datelor in care s-a prezentat la cursuri studentul Ion Popescu, vom scrie:

```
SELECT Nume,Data FROM Studenti,Prezenta WHERE ID=ID_student AND Nume='Ion Popescu'
```



Exista doua tipuri de join – inner join si outer join – ce vor fi prezentate in continuare, impreuna cu sintaxele aferente.

6.3.2. Inner join

Rezultatul unui inner join este format numai din randurile care au corespondent in ambele tabele. Daca unui rand dintr-o tabela nu ii corespunde (conform cheii externe) nici unul din cealalta, el nu va fi afisat.

Observatie: *join-urile prezentate in paragraful anterior erau de tip inner join.*

O sintaxa alternativa pentru realizarea unui inner join este:

```
SELECT col1,col2,… FROM tabela1 INNER JOIN tabela2 ON (conditie)
```

Exemplu:

```
SELECT Nume,Data FROM Studenti INNER JOIN Prezenta ON(Student.ID=Prezenta.ID_student)
```

Nota: *pentru a extrage date din doua tabele aflate intr-o relatie de tip many-to-many, este necesara realizarea a doua join-uri. Daca dorim sa obtinem lista de studenti ai clasei PHP 19 iunie, vom scrie:*

```
SELECT Clase.Nume, Studenti.Nume FROM Clase INNER JOIN Studenti_Clase ON ( Clase.ID = ID_clasa )
INNER JOIN Studenti ON ( Studenti.ID = ID_student )
WHERE Clase.Nume='PHP 18 iunie'
```

6.3.3. Outer join

La realizarea unui join, este posibil ca una dintre tabele sa contina inregistrari care nu au corespondent in tabelul al doilea. In cazul unui inner join, toate aceste inregistrari vor fi eliminate din rezultatul final. Un outer join insa returneaza si inregistrarile in cauza, rezultatul produs continand NULL pe coloanele corespunzatoare celui de-al doilea tabel.

Sa presupunem ca am adaugat de curand studentul Victor Manescu in tabela Studenti, si intre timp nu a avut loc nici o sedinta a claselor in care acesta este inscris. Daca dorim sa aflam datele in care a venit la curs fiecare student si efectuam un inner join intre tabela de studenti si cea de prezenta, numele noului student va fi omis din rezultat, deoarece nu exista inregistrari corespunzatoare lui in tabela de prezenta. Un left join ne va afisa insa lista completa de studenti, cu NULL pe coloanele Data si Prezent pentru Victor Manescu:

```
SELECT Nume, Data , Prezent FROM studenti LEFT JOIN Prezenta ON (ID = ID_student)
```

Un outer join poate fi:

- **left join** – sunt pastrate in rezultat toate inregistrarile primului tabel, indiferent daca exista sau nu inregistrari corespondente in cel din dreapta. Este tipul de join prezentat mai sus.
- **right join** – identic ca principiu cu left join, dar pastreaza toate inregistrarile celui de-al doilea tabel

Rezultat outer join		
Nume	**Data**	**Prezent**
Ion Popescu	2008-06-19	da
Ion Popescu	2008-06-26	Nu
Victor Manescu	NULL	NULL
...	...	...

6.4. SUBINTEROGARI

6.4.1. Descriere si utilitate

O subinterogare reprezinta o instructiune SELECT care este plasata in cadrul alteia acolo unde este nevoie de o valoare sau un ansamblu de valori. Spre exemplu, pentru a afla lista tuturor persoanelor care au aceeasi data de nastere ca si Ion Popescu, am putea scrie:

```
SELECT * FROM Persoane WHERE DataNasterii= (
      SELECT DataNasterii FROM Persoane WHERE Nume='Ion Popescu'
)
```

Observam ca in clauza WHERE s-a folosit ca operand drept al operatorului=, in locul unei valori constante, rezultatul unei alte interogari.

O sub-interogare trebuie inclusa intotdeauna intre paranteze rotunde.

Sub-interogarile ofera avantaje precum:
- unele interogari complexe care folosesc multe tabele si join-uri se pot rescrie mai simplu sau mai intuitiv folosind sub-interogari (sub-interogarile se muleaza mai bine pe logica noastra naturala)
- prin sub-interogari se realizeaza o „partitionare” a unei interogari complexe in portiuni ce pot fi gandite si depanate independent

6.4.2. Tipuri de sub-interogari

Privind exemplul de mai sus, intuim faptul ca sub-interogarea trebuie sa indeplineasca unele conditii, astfel incat rezultatele pe care le returneaza sa fie compatibile cu expresia in care participa. Sub-interogarea din exemplu nu ar putea returna mai multe coloane sau mai multe inregistrari, deoarece operandul drept al lui = trebuie sa fie o singura valoare.

Nota: *o alta restrictie importanta aplicata unei sub-interogari este ca ea nu are voie se selecteze dintr-o tabela pe care interogarea parinte o modifica.*

In functie de numarul de randuri si coloane returnate de o sub-interogare, ea poate participa in expresii in mod diferit. Distingem urmatoarele categorii de sub-interogari:
- **sub-interogari de tip scalar**, care returneaza o singura valoare (o singura inregistrare ce are o singura coloana). Rezultatul unei astfel de interogari poate fi folosit in locul valorilor constante din expresiile SQL
- **sub-interogari de tip lista**, care returneaza mai multe randuri dar o singura coloana. Rezultatul lor poate fi folosit in expresii acolo unde este nevoie de o lista de valori (ex: operatorul IN)
- **sub-interogari de tip rand**, ce returneaza un singur rand cu mai multe coloane. Pot fi folosite pentru a compara valorile unui rand cu un set de valori explicit in cadrul instructiunii
- **sub-interogari de tip tabela**, ce produc mai multe randuri si mai multe coloane. Pot fi folosite in locul unei tabele, ca termen al unui join

Dintre cele prezentate, primele doua si ultima sunt mai des intalnite si vor fi dezvoltate in continuare.

6.4.3. Sub-interogari de tip scalar

Sub-interogarile de tip scalar pot fi folosite in expresii SQL oriunde este nevoie de o valoare constanta (ex: operanzi, argumente de functii etc). Ele trebuie sa returneze o singura inregistrare avand o singura coloana, in caz contrar intreaga interogare va esua:

```
# daca in tabela Produse avem mai multe produse din categoria IT, interogarea urmatoare esueaza:
SELECT * FROM Produse WHERE Pret > (SELECT Pret FROM Produse WHERE Categorie='IT');
ERROR 1242 (21000): Subquery returns more than 1 row


# daca sub-interogarea returneaza mai mult de o coloana, intreaga interogare esueaza din nou:
SELECT * FROM Produse WHERE Pret > (SELECT Pret,Categorie FROM Produse LIMIT 1);
ERROR 1241 (21000): Operand should contain 1 column(s)


# o interogare corecta: produsele cu pret mai mare decat media tabelei
SELECT * FROM Produse WHERE Pret> (SELECT AVG(Pret) FROM Produse )
```

6.4.4. Sub-interogari de tip lista

O sub-interogare de tip lista produce mai multe inregistrari cu o singura coloana. Acest set de rezultate poate fi folosit in expresiile SQL pe post de lista de valori – fie in contextele care implica liste (cum este operatorul IN), fie in alte contexte in care interveneau de obicei valori scalare (ex: comparatii). Iata scenarii de utilizare si exemple:

- operatorul IN poate accepta ca lista de valori rezultatul unei sub-interogari de tip lista:

```
# date fiind doua tabele, Produse si Flori, extragem produsele care au denumiri de flori:
SELECT * FROM Produse WHERE Denumire IN (SELECT Nume FROM Flori)
```

- o sub-interogare de tip lista poate fi folosita ca operand al unui operator de comparare, folosind cuvintele cheie ANY, ALL si SOME:
  - *expresie operator ALL(sub-interogare)* - returneaza true daca toate valorile returnate de sub-interogare se afla in relatia respectiva cu valoarea expresiei din stanga. Spre exemplu, *10 < ALL(SELECT * FROM produse WHERE Categorie='IT')* returneaza true daca toate preturile produselor din categoria IT sunt mai mari decat 10. In locul lui 10 se putea afla o expresie mai complexa in care participau valori de coloane (vezi exemple)
  - *expresie operator ANY(sub-interogare)* – returneaza true daca macar una dintre valorile returnate de sub-interogare se afla in relatia respectiva cu valoarea expresiei din stanga. Spre exemplu, *10 < ANY(SELECT * FROM produse WHERE Categorie='IT')* returneaza true daca exista macar un produs din categoria IT care are pretul mai mare decat 10
  - *EXISTS(sub-interogare)* – returneaza true daca sub-interogarea produce cel putin o inregistrare

Observatie: *a scrie* valoare = ANY(sub-interogare) *este echivalent cu a scrie* valoare IN (sub-interogare). *Dupa aceeasi logica,* NOT IN(sub-interogare) *este echivalent cu* != ALL(sub-interogare).

Cuvantul cheie SOME este un sinonim pentru ANY, pentru cazuri in care intelesul in limba engleza al instructiunii SQL ar fi derutant:

```
SELECT c1 FROM t1 WHERE c1 <> ANY  (SELECT c1 FROM t2);
```

Aici, „c1 not equal to any” ar insemna „c1 nu este egal cu nici un” - ori sensul instructiunii este altul: „c1 diferit de macar un”.

Exemple:

```
# studentii care au obtinut note mai mari decat cele ale tuturor studentilor din clasa SQL
SELECT * FROM Catalog WHERE Nota > ALL (SELECT Nota FROM Catalog WHERE Clasa='SQL')

# ce note are studentul Ionescu mai mari decat nota minima a lui Popescu?
SELECT Nota FROM Catalog WHERE Nume='Ionescu' AND Nota > ANY (
       SELECT Nota FROM Catalog WHERE Nume='Popescu' )
```

6.4.5. Sub-interogari de tip tabela

Sub-interogarile de tip tabela returneaza mai multe randuri cu mai multe coloane si pot fi folosite in locul unei tabele in cadrul unui join. In aceste conditii este obligatoriu ca sub-interogarea sa aiba un alias, astfel incat sa se poata face referirea fara echivoc la coloanele sale – atat in conditiile de join, cat si in cadrul clauzei WHERE:

```
# obtinerea listei de inregistrari continute in Lista1 dar nu si in Lista2
SELECT DISTINCT Email FROM Lista1.* LEFT JOIN (SELECT DISTINCT Email FROM Lista2) AS DoNotMail
       ON (Lista1.Email = DoNotMail.Email) WHERE DoNotMail.Email IS NULL
```

6.5. BIBLIOGRAFIE

- Indecsi: MySQL 5 Certification Study Guide, Cap 8.6 – 8.7
- Indecsi unici si operatii INSERT/UPDATE: MySQL 5 Certification Study Guide, Cap. 11.2.2, 11.2.3, 11.3
- Join-uri: MySQL 5 Certification Study Guide, Cap. 12
- Sub-interogari: MySQL 5 Certification Study Guide, Cap. 13
- Detalii join-uri: MySQL Reference Manual: http://dev.mysql.com/doc/refman/5.1/en/join.html
- Detalii sub-interogari: http://dev.mysql.com/doc/refman/5.1/en/subqueries.html