5. EXPRESII SQL

Cuprins

5.1. Compoziţia unei expresii

Expresiile sunt omniprezente în instrucţiunile SQL. Ele pot fi întîlnite în locuri precum:
- valorile coloanelor returnate de către instrucţiunile SELECT
- valorile introduse pe coloane de către o instrucţiune UPDATE
- criteriile de ordonare din clauza ORDER BY
- condiţiile din clauza WHERE

O expresie SQL poate fi compusă din:
- valori constante, specificate ca atare în cadrul instrucţiunii SQL. Acestea pot fi numere întregi sau fractionare, şiruri de octeţi sau caractere, date calendaristice etc. Pentru ca serverul SQL sa determine corect tipul de date al unei astfel de valori, este necesar ca ea sa fie reprezentata (semnalizata) de aşa natura încît sa se poată deduce fara echivoc tipul ei de date (sau cel puţin sa se poată converti automat la tipul corect)
- valori ale coloanelor din tabelele care intervin în instrucţiune
- apeluri către funcţii SQL predefinite. MySQL pune la dispoziţia clientului funcţii matematice, funcţii pentru prelucrare de şiruri de caractere, pentru lucrul cu date de tip temporal etc
- NULL

Toate acestea pot fi combinate folosind operatori. Fiecare operator accepta unul sau mai mulţi operanzi şi produce un rezultat ce poate fi la rîndul sau combinat cu alte valori sau rezultate produse de alţi operatori. Exista diferite categorii de operatori ce vor fi prezentate în continuare în cadrul acestui material.

5.2. Reprezentarea valorilor constante în instrucţiunile MySQL

5.2.1. Numere

Valorile constante de tip numeric se reprezinta in cadrul unei instructiuni SQL dupa urmatoarele reguli:
- numere întregi
  - în baza de numeratie 10: o succesiune de cifre, precedata optional de semnul + sau – (ex: 34, -18, +23)
  - în hexazecimal (baza de numeratie 16): x*'valoare'*, X*'valoare'* sau 0x*valoare*, unde *valoare* contine cifre in hexazecimal (0...9 sau A...F). Nu contează daca folosim litere mici sau mari. Valoarea poate fi precedata de semn. Exemplu: x'6ac', 0x21F
- numere fractionare (pot fi si ele precedate de un eventual semn):
  - în notaţie obişnuită: 34.2
  - în notaţie stiintifica: 0.342e+3 (adica 0.342 x 10^3)

```
INSERT INTO Numbers(n1,n2,n3) VALUES(0x21, -11.4e-1, X'DA');

# valorile introduse vor fi 33, -1.14 si 218
```

Felul in care sunt reprezentate valorile numerice constante într-o instrucţiune SQL are implicaţii asupra tipului de date pe care serverul il asignează intern acelei constante:
- notaţia stiintifica, ce foloseşte exponent (ex: 1.745e4) face ca valoarea sa fie reprezentata in memorie ca număr in virgula mobila, cu toate implicaţiile ce decurg de aici (imprecizia stocării anumitor valori etc)
- valorile numerice constante de tip intreg sunt reprezentate in memorie si participa in operaţii ca BIGINT
- valorile de tip numar fractionar in notatie exacta (14.2) sunt reprezentate in memorie si participa in operatii ca DECIMAL

5.2.2. Şiruri de caractere

Pentru a putea fi recunoscute de către server ca şiruri de caractere, valorile de acest fel trebuie incluse intre ghilimele (”...”) sau apostroafe ('...'). Apostroful este de preferat deoarece, dacă sql_mode include opţiunea ANSI_QUOTES, ghilimelele sunt folosite ca delimitatori pentru numele de baze de date, tabele si coloane (echivalente cu `).

Daca dorim ca o valoare de tip sir de caractere reprezentata astfel sa aibă ataşate un anumit character set si collation, vom specifica setul de caractere precedat de un caracter _ (underscore) inaintea valorii, iar collation-ul dupa, precedat de cuvantul cheie COLLATE:

```
INSERT INTO Persoane(Nume,Prenume)
VALUES('Ana', _latin2 'Aman' COLLATE latin2_general_ci);
```

5.2.3. Valori de tip temporal

Valorile de tip temporal sunt reprezentate in cadrul instrucţiunilor SQL după cum urmează:
- pentru tipurile de date DATE, DATETIME, TIME si TIMESTAMP, valorile vor fi încadrate intre apostroafe (la fel ca în cazul şirurilor de caractere)
- pentru tipul de date YEAR valorile vor fi reprezentate ca succesiuni de cifre, la fel ca în cazul numerelor întregi în baza 10
- zilele şi lunile care au o singura cifra nu este obligatoriu sa aibă 0 în fata (ex: '2008-01-08' poate fi reprezentat si ca '2008-1-8')
- separatorul intre an, luna si zi in cadrul datelor nu este obligatoriu – ci poate fi altul (ex: /, ., : etc)
- in cazul anilor reprezentati cu doar doua cifre, valorile 70...99 se referă la anii 1970-1999, iar valorile 00...69 se referă la anii 2000...2069

```
INSERT INTO Dates(Date,Timestamp,Year)
VALUES('2009-07-16', '1989-12-25 18:05:02', 86);
# implicit anul 1986
```

Trebuie spus faptul ca, în cazul de faţă, încadrarea acestor valori intre apostroafe NU face ca ele sa fie percepute de către SQL ca date de tip temporal, ci doar sa poată fi convertite automat la un astfel de tip de date atunci cînd sunt introduse într-o tabela (modul de delimitare fiind identic cu cel al şirurilor de caractere, nici nu ar putea fi percepute ca altceva). Atunci cand, in expresii, avem nevoie sa folosim o valoare de tip temporal scrisă ca atare în cadrul instrucţiunii, putem folosi funcţia predefinită CAST pentru a ne asigura ca valoarea capătă tipul corect de date.

5.3. Operatori

5.3.1. Categorii de operatori

Distingem următoarele categorii de operatori ce pot participa in expresiile MySQL:
- **aritmetici** – se suprapun peste operatorii matematici uzuali (adunare, scadere etc) si produc un rezultat numeric
- **de comparare** – compara valorile expresiilor primite ca operanzi, returnand valori de adevar (adevarat/fals)

- **logici** – folositi pentru compunerea mai multor expresii ce au ca rezultat o valoare de adevar in expresii mai mari (ex: clauza WHERE cu conditii multiple)
- **de conversie** – spre exemplu, conversia pe loc a valorii unui sir de caractere la sir de octeti

5.3.2. Operatori aritmetici

Operatorii aritmetici sunt operatori care produc rezultat numeric si care se mapeaza pe operatorii matematici cunoscuti:

- **+**, **-**, ***** - adunare, scadere, inmultire
- **/** - impartire. Daca impartirea nu este exacta rezulta un numar fractionar
- **DIV** – impartire intreaga. Returneaza catul impartirii primului operand la al doilea
- **%** - modulo. Returneaza restul impartirii primului operand la al doilea

```
SELECT 1+2, 5-1, 2*3, 5%2, 7/2, 7 DIV 2;
+-----+-----+-----+------+--------+---------+
| 1+2 | 5-1 | 2*3 | 5%2  | 7/2    | 7 DIV 2 |
+-----+-----+-----+------+--------+---------+
|   3 |   4 |   6 |    1 | 3.5000 |       3 |
+-----+-----+-----+------+--------+---------+
```

Operatorii aritmetici + si - pot fi aplicati si valorilor de tip temporal, oferind posibilitatea adaugarii sau scăderii unui interval de timp dintr-o dată deja existentă. Pentru specificarea intervalului dorit se foloseste cuvantul cheie INTERVAL urmat de o expresie si de o unitate de masura, aceasta din urma putand fi YEAR, HOUR, MINUTE, SECOND etc:

```
SELECT '2009-08-06 11:34:03' + INTERVAL 3 DAY, '2009-08-06 11:34:03' – INTERVAL 4 MINUTE;
+-----------------------------------------+-----------------------------------------+
| '2009-08-06 11:34:03' + INTERVAL 3 DAY  | '2009-08-06 11:34:03'-INTERVAL 4 MINUTE |
+-----------------------------------------+-----------------------------------------+
| 2009-08-09 11:34:03                     | 2009-08-06 11:30:03                     |
+-----------------------------------------+-----------------------------------------+
```

5.3.3. Operatori de comparare

Operatorii de comparare accepta unul sau mai multi operanzi si returneaza fie 0 sau 1, fie NULL. Valoarea NULL este returnata atunci cand cel putin unul dintre operanzi este NULL. Iata principalii operatori MySQL:

- >, <, >=, <= - operatorii de comparare cunoscuti din matematica. Diferenta este ca in MySQL ei pot fi folositi si pentru a compara siruri de caractere, compararea fiind una alfabetica, in functie de collation. Returneaza 1 daca primul operand se afla in relatia respectiva cu cel de-al doilea si 0 in caz contrar.
- = (egal), **!=** sau <> („diferit de”) - returneaza NULL in cazul in care cel putin unul dintre operanzi este NULL. Operanzii sunt subiect de conversie implicita daca au tipuri de date diferite (vezi mai jos)
- **expresie IS NULL**, **expresie IS NOT NULL**- verifica daca expresia are „valoarea” NULL. In astfel de cazuri nu putem folosi operatorul =, deoarece al doilea operand fiind NULL, el ar returna intotdeauna NULL in loc de rezultatul dorit

- <=> - verificare de egalitate utilizabila si in cazul in care unul dintre operanzi este NULL. Daca doar unul dintre operanzi este NULL, rezultatul produs va fi 0, iar daca ambii sunt NULL returneaza 1
- **operand1 BETWEEN valoare1 AND valoare2** - folosit pentru a pune conditia ca valoarea unei expresii sa se afle intre doua limite (inclusiv capetele!). Poate fi precedat de NOT pentru a pune conditia ca valoarea expresiei sa se afle in afara intervalului (*operand1 NOT BETWEEN valoare1 AND valoare2*)
- **expresie1 LIKE 'expresie2'** – pune conditia ca formatul primului operand sa corespunda celui specificat in expresia ce joaca rolul de operand secund. In expresie2 pot fi folosite caracterele speciale _ (tine loc de un caracter) si % (tine loc de zero sau mai multe caractere). Poate fi precedat de NOT pentru a nega sensul comparatiei (*expresie1 NOT LIKE expresie2*)
- **expresie RLIKE 'regex'** sau **expresie REGEXP 'regex'** – acelasi scop ca si LIKE, insa operandul drept este de aceasta data un regex (regular expression) – o expresie mai complexa si care permite o specificare mult mai flexibila a formatului sirului de caractere dorit. Poate fi de asemenea precedat de NOT
- **expresie1 IN (valoare1, valoare2, ...)** - returneaza 1 daca primul operand se gaseste in lista specificata si 0 in caz contrar. Este o varianta mai scurta care inlocuieste expresia *expresie1 = valoare1 OR expresie2 = valoare2 OR...* Poate fi precedat de NOT.

```
SELECT 2 > 1, 2 > NULL, NULL = NULL, NULL <=> NULL, NULL <> NULL;
+-------+----------+-------------+---------------+--------------+
| 2 > 1 | 2 > NULL | NULL = NULL | NULL <=> NULL | NULL <> NULL |
+-------+----------+-------------+---------------+--------------+
|     1 |     NULL |        NULL |             1 |         NULL |
+-------+----------+-------------+---------------+--------------+

SELECT 5 BETWEEN 1 AND 5, 2 NOT BETWEEN 1 AND 3, 'Test' LIKE 'T%';
+-------------------+----------------------+------------------+
| 5 BETWEEN 1 AND 5 | 2 NOT BETWEEN 1 AND 3 | 'Test' LIKE 'T%' |
+-------------------+----------------------+------------------+
|                 1 |                    0 |                1 |
+-------------------+----------------------+------------------+

SELECT 2 IN (5,8,2,7), 'two' IN ('one', 'two', 'three');
+----------------+--------------------------------------+
| 2 IN (5,8,2,7) | 'two' NOT IN ('one', 'two', 'three') |
+----------------+--------------------------------------+
|              1 |                                    0 |
+----------------+--------------------------------------+
```

5.3.4. Operatori logici

Operatorii logici sunt folositi pentru a agrega expresii care au ca rezultat valori de adevar. In MySQL, valorile de adevar (true si false) sunt implementate ca 1 si 0. Operatorii logici sunt:

- AND sau && - SI logic
- OR sau || - SAU logic

Nota: *daca sql_mode contine optiunea PIPES_AS_CONCAT, operatorul || nu mai este interpretat ca SAU logic, ci ca operator de concatenare pentru stringuri!*

- XOR – SAU exclusiv
- NOT sau ! - negare (inversarea valorii de adevar)

```
SELECT 3>2 AND 5>4, 5<6 OR 7>8, 5>3 XOR 6>1, NOT (7>-1);
+-------------+------------+-------------+------------+
| 3>2 AND 5>4 | 5<6 OR 7>8 | 5>3 XOR 6>1 | NOT (7<-1) |
+-------------+------------+-------------+------------+
|           1 |          1 |           0 |          1 |
+-------------+------------+-------------+------------+
```

Nota: *NOT poate fi inlocuit cu ! numai acolo unde joaca rolul de operator de negare. Spre exemplu, substitutia nu poate fi efectuata in cazul operatorului IS NOT NULL sau pentru definitii de coloana NOT NULL.*

5.3.5. Operatori de evaluare conditionata

Operatorul CASE poate returna diferite valori in functie de valorile unor expresii primite ca operand. El are doua forme:

```
CASE expresie WHEN expresie1 THEN valoare1 WHEN expresie2 THEN valoare2....ELSE valoare_final END
CASE WHEN expresie1 THEN valoare1 WHEN expresie2 THEN valoare2 ....ELSE valoare_final END
```

In prima forma, valoarea lui *expresie* este comparata pe rand cu valorile lui *expresie1*, *expresie2* etc. Daca *expresie* are valoare egala cu *expresie1* operatorul va produce *valoare1*; daca *expresie* este egal cu *expresie2* va produce *valoare2* s.a.m.d. Daca valoarea lui *expresie* nu este egala cu niciuna dintre celelalte valori, se returneaza valoarea din ELSE sau, in caz de absenta a ramurii ELSE, valoarea NULL.

Cea de-a doua forma verifica pe rand valoarea lui *expresie1*, *expresie2* s.a.m.d. si returneaza valoarea primei expresii care se evalueaza ca true (nu are valoarea 0 sau NULL). In cazul in care niciuna dintre expresii nu se evalueaza ca true se va returna valoarea din ELSE sau, in cazul absentei ramurii ELSE, valoarea NULL.

```
-- urmatoarele doua instructiuni sunt echivalente si vor afisa zero, unu sau doi
SELECT CASE FLOOR(RAND()*3) WHEN 0 THEN 'zero' WHEN 1 THEN 'unu' ELSE 'doi';
SELECT CASE FLOOR(RAND()*3) WHEN 1=2 THEN 'zero' WHEN 2=2 THEN 'unu' ELSE 'doi';
-- afiseaza c
SELECT CASE WHEN 2>3 THEN 'a' WHEN 2=3 THEN 'b' ELSE 'c' END;
```

Nota: *a nu se confunda operatorul CASE cu instructiunea decizionala CASE folosita in cadrul rutinelor memorata in baza de date!*

5.3.6. Operatori de conversie

Mentionam aici operatorul BINARY. El este folosit atunci cand dorim ca un sir de caractere sa fie tratat ca sir de octeti in cadrul unei expresii. Operatorul trebuie plasat imediat inaintea sirului caruia i se aplica conversia:

```
CREATE TABLE test(c1 CHAR(1) CHARSET ucs2, c2 CHAR(1) CHARSET ascii);
# fiecare caracter ASCII ocupa un octet, fiecare caracter UCS2 ocupa 2 octeti
INSERT INTO test VALUES ( 'x' , 'x');
SELECT c1=c2, BINARY c1=c2 FROM test;
+-------+--------------+
| c1=c2 | BINARY c1=c2 |
+-------+--------------+
|     1 |            0 |
+-------+--------------+
```

5.3.7. Conversii de tip de date implicite efectuate in expresiile MySQL

Atunci cand operanzii unui operator au tip de date diferit, MySQL efectueaza conversii automate astfel incat operatorul sa se poata in final aplica celor doua valori. Iata cateva exemple:

- numerele sunt automat transformate in siruri de caractere - si invers - acolo unde este nevoie:

```
SELECT 4+5 = '9', 1 IN ('1','2');
# rezultat 1 in ambele cazuri
```

- datele de tip temporal sunt convertite la numere cand sunt folosite intr-un context numeric, si invers:

```
CREATE TABLE date(d DATE, t TIMESTAMP);
INSERT INTO date('2009-08-02', '2009-08-02 20:44:00');
SELECT d+0, t+0;

+----------+----------------+
| d+0      | t+0            |
+----------+----------------+
| 20090802 | 20090802204400 |
+----------+----------------+
```

- valorile coloanelor de tip enumerat sunt automat convertite la string sau la numar in functie de context:
  - atunci cand contextul este unul numeric, fiecare valoare de tip ENUM sau SET va fi convertita la intregul corespunzator ei (se cunoaste faptul ca astfel de valori sunt reprezentate intern in MySQL ca intregi)
  - in contexte string, este folosita valoarea de tip string a coloanei

```
CREATE TABLE Options(opt ENUM('da','nu'));
INSERT INTO Options VALUES('da'),('nu');
SELECT opt+0, concat('Raspuns: ', opt) FROM Options;
+-------+-------------------------+
| opt+0 | concat('Raspuns: ', opt) |
+-------+-------------------------+
|     1 | Raspuns: da              |
|     2 | Raspuns: nu              |
+-------+-------------------------+
```

Nota: *clientul poate efectua conversii de tip de date explicite folosind functiile SQL predefinite CAST() sau CONVERT().*

5.3.8. Cazul valorilor NULL

Dupa cum s-a observat la descrierea operatorilor, exista cateva reguli speciale de care trebuie ţinut cont atunci cand intr-o expresie intervine NULL:

- majoritatea operatorilor vor returna NULL atunci cand unul dintre operanzi este NULL (cu excepţia lui <=>)
- atunci cand dorim sa verificam daca rezultatul unei expresii este sau nu NULL vom folosi IS NULL sau IS NOT NULL, dupa caz
- atunci cand dorim sa comparam doua valori si intentionam ca valorile de tip NULL sa fie considerate egale intre ele dar diferite de orice alta valoare, putem folosi operatorul <=>

5.4. Funcţii SQL predefinite

5.4.1. Sintaxa de apelare

Functiile MySQL predefinite corespund operatiilor mai des intalnite ce se aplica diferitelor tipuri de valori. Fiecare functie poate fi apelata pasandu-i-se zero sau mai multe argumente si produce o valoare de iesire de un anumit tip de date. Atunci cand intr-o expresie intervine un apel către o functie, la evaluarea expresiei el se inlocuieste cu valoarea returnata – aceasta pentru fiecare inregistrare sau pentru grupuri de inregistrari, depinzand de tipul functiei (detalii mai jos).

Pentru apelarea unei functii MySQL se foloseste numele functiei urmat de lista de argumente intre paranteze rotunde:

```
/* afisarea numelui complet al persoanelor prin concatenarea valorilor coloanelor de nume si
prenume, separate prin spatiu */
SELECT CONCAT(Nume, ' ', Prenume);
```

Implicit, intre numele functiei si prima paranteza nu trebuie sa existe spatii. Daca sql_mode cuprinde si optiunea IGNORE_SPACE, va fi permisa includerea de astfel de spatii, insa este posibil ca ele sa introduca ambiguitati in interpretarea anumitor instructiuni particulare (ex: in cazul in care exista coloane ce au acelasi nume cu o functie predefinita).

5.4.2. Tipuri şi categorii

Putem imparti functiile MySQL predefinite in urmatoarele doua tipuri:

- **functii simple** – primesc zero sau mai multe argumente ce pot fi valori constante sau expresii (care pot contine la randul lor alte functii, coloane ale tabelelor s.a.m.d.). O expresie ce contine o astfel de functie aplicata unei coloane de tabela va produce cate o valoare pentru fiecare inregistrare implicata (ex: *SELECT ROUND(Nota) FROM Examene* va returna un set de rezultate in care functia de rotunjire a fost aplicata fiecarei note in parte). Vom aborda urmatoarele categorii mai importante:
  - functii de comparare
  - functii conditionale, care ne permit sa luam decizii in functie de valorile unor expresii
  - functii matematice, pentru prelucrarea de date numerice
  - functii pentru siruri de caractere
  - functii pentru operarea cu date de tip temporal
  - alte functii de interes
- **functii de agregare** – sunt functii care actioneaza asupra mai multor randuri dintr-un set de rezultate (ex: media notelor unei clase, suma preturilor de produse dintr-o tabela etc)

5.4.3. Funcţii simple

5.4.3.1. Funcţii generale de comparare

Aceste functii se aplica atat pentru numere cat si pentru siruri de caractere, in ultimul caz ele realizand comparare alfabetică:

- **LEAST(valoare1, valoare2, ...)** - returnează cea mai mică dintre valorile primite ca argument
- **GREATEST(valoare1, valoare2, ...)** - returnează cel mai mare dintre argumente
- **INTERVAL(N, N1, N2, ...)** - returnează 0 daca N<N1, 1 daca N<N2, 2 daca N<N3 etc (toate valorile sunt tratate ca intregi). Pentru ca funcţia sa opereze corect, este necesar ca N1<N2<N3<...etc

5.4.3.2. Funcţii matematice

Aceste functii realizeaza operatiile aritmetice si matematice uzuale. Iata-le pe cele mai importante:

- valoare absoluta
  - **ABS(valoare)** – returneaza valoarea absoluta (modulul) valorii expresiei primite ca argument
- trunchieri si rotunjiri
  - **CEIL(valoare)/CEILING(valoare)** - returneaza cel mai apropiat intreg mai mare sau egal cu valoarea primita ca argument
  - **FLOOR(valoare)** - returneaza cel mai mare intreg mai mic sau egal cu valoarea primita ca argument
  - **ROUND(valoare)** - returneaza intregul cel mai apropiat de valoarea primita ca argument, prin indepartare de 0 (ex: 13.2 devine 13, 13.7 devine 14, insa -13.2 devine -13 (spre deosebire de cazul pozitiv, de aceasta data a fost returnat intregul superior valorii) iar -13.7 devine -14). Exista si forma ROUND(valoare,nr_zecimale) care face rotunjirea la numarul de zecimale cerut.

  - **TRUNCATE(valoare,nr_zecimale)** – trunchiaza valoarea pastrand numai numarul de zecimale cerut. Nota: nu este acelasi lucru cu ROUND(valoare,nr_zecimale), deoarece acesta din urma efectueaza si rotunjirea rezultatului trunchierii
- ridicare la putere
  - **POW(baza,exponent)/POWER(baza,exponent)** – returneaza rezultatul ridicarii bazei la puterea exponent
  - **EXP(valoare)** – ridicarea lui *e* la puterea *valoare*
- logaritmi
  - **LOG(valoare), LOG10(valoare), LOG2(valoare)** - logaritm natural/ in baza 10/ in baza2
- radacina patrata (radical)
  - **SQRT(valoare)** – returneaza un numar fractionar ce reprezinta rezultatul operatiei. Daca valoarea primita ca argument este negativa, rezultatul returnat va fi NULL
- functii trigonometrice
  - **PI()** - returnează valoarea numărului π
  - **COS(), SIN(), TAN(), COT(), ACOS(), ASIN(), ATAN()** - functiile trigonometrice uzuale (sinus, cosinus, tangenta, cotangenta, arcsinus, arccosinus, arctangenta)
- generarea de numere aleatoare
  - **RAND()** - returneaza un numar de tip fractionar cu valoare cuprinsa intre 0 (inclusiv) si 1 (exclusiv). Poate fi folosita pentru a produce numere in orice plaja, prin inmultirea cu o alta valoare: expresia 100*RAND() va produce valori in intervalul [0, 100)
- semnul unei expresii
  - **SIGN()** - functia matematica signum. Returneaza -1 pentru numere negative, 1 pentru pozitive si 0 pentru numere nule
- conversie
  - **CONV(numar, baza_initiala, baza_dorita)** – returneaza un sir de caractere ce reprezinta rezultatul conversiei numarului din baza de numeratie initiala in cea dorita

```
CREATE TABLE numere (n1 DECIMAL(4,2), n2 DECIMAL(4,2));

INSERT INTO numere VALUES (13.68, -11.23);

SELECT ABS(n2), CEIL(n1),FLOOR(n1), CEIL(n2), FLOOR(n2), ROUND(n1), ROUND(n2),  TRUNCATE(n1,1),
CONV(n1,10,16) FROM numere\G

*************************** 1. row ***************************

       ABS(n2): 11.23

      CEIL(n1): 14

     FLOOR(n1): 13

      CEIL(n2): -11

     FLOOR(n2): -12

     ROUND(n1): 14

     ROUND(n2): -11

TRUNCATE(n1,1): 13.6

CONV(n1,10,16): D
```

5.4.3.3. Functii conditionale

MySQL pune la dispozitia clientului functii predefinite care ii permit acestuia sa ia decizii bazate pe valorile unor expresii. Exista urmatoarele functii de acest fel:

- **IF(expresie_evaluata,expresie1,expresie2)** – daca *expresie_conditie* se evalueaza ca true (nu are valoarea 0 sau NULL) functia returneaza valoarea lui *expresie1*, in caz contrar valoarea lui *expresie2*
- **IFNULL(expresie1, expresie2)** – daca expresie1 are valoare non-NULL este returnata valoarea lui *expresie1*, in caz contrar se returneaza valoarea lui *expresie2*
- **NULLIF(expresie1,expresie2)** – returneaza NULL daca *expresie1=expresie2*; in caz contrar returneaza valoarea lui *expresie1*

```
-- produce valoarea expresiei FLOOR(RAND()*2) in litere (unu sau doi)
SELECT IF(FLOOR(RAND()*2), 'unu', 'zero');
```

5.4.3.4. Funcţii pentru şiruri de caractere

In cadrul functiilor care opereaza pe/produc siruri de caractere distingem urmatoarele categorii/operatii:

- lungimea unui sir. Ea poate fi exprimata in caractere sau in octeti. In functie de charset-ul asociat sirului, cele doua lungimi pot coincide sau putem avea un numar de octeti mult mai mare decat cel de caractere
  - **CHAR_LENGTH(sir)/CHARACTER_LENGTH(sir)** – returneaza numarul de *caractere* din sir
  - **LENGTH(sir)/OCTET_LENGTH()**- returneaza numarul de *octeti* ocupati de sir
- concatenare
  - **CONCAT(sir1,sir2,..)** - returneaza sirul rezultat din concatenarea argumentelor. Daca vreunul dintre argumente este NULL, rezultatul returnat este de asemenea NULL
  - **CONCAT_WS(separator, sir1, sir2,...)** - realizeaza concatenarea sirurilor ce constituie argumentele 2,3,etc. insa intercaland intre fiecare doua siruri consecutive sirul specificat in primul argument (separatorul). Spre deosebire de CONCAT(), argumentele NULL sunt ignorate
- padding - completarea cu caractere a unui sir pentru a atinge lungimea dorita
  - **LPAD(sir, lungime_dorita, sir_adaugat)** – adauga la inceputul sirului specificat ca prim argument cate exemplare ale lui sir_adaugat este nevoie astfel incat rezultatul sa ajunga la lungimea dorita
  - **RPAD(sir, lungime_dorita, sir_adaugat)** – analog, dar face adaugarea la sfarsit
- eliminarea spatiilor marginale
  - **LTRIM(sir)** – elimina eventualele spatii aflate la inceputul sirului primit ca argument
  - **RTRIM(sir)** - analog, dar pentru spatiile de la sfarsit
  - **TRIM(sir)** – elimina spatiile aflate la inceputul si sfarsitul sirului primit ca argument (efectul combinat al lui LTRIM() si RTRIM())
- conversii litere mici/mari
  - **LCASE(sir)/LOWER(sir)** – returneaza sirul primit ca argument, majusculele tranformate in litere mici
  - **UCASE(sir)/UPPER(sir)** – conversia inversa
- decuparea unui subsir dintr-un sir dat
  - functiile SUBSTR()/SUBSTRING(). Specificarea pozitiei subsirului se poate realiza in doua moduri:

    - **SUBSTR(sir, pozitie_inceput)/SUBSTR(sir FROM pozitie_inceput)** – returneaza subsirul ce se intinde de la pozitia specificata din sirul original pana la sfarsitul acestuia. **Pozitiile se numeroteaza de la 1**. Daca pozitia este negativa, punctul de pornire va fi considerat a se afla cu acel numar de caractere inainte de finalul sirului original
    - **SUBSTR(sir,pozitie_inceput,lungime)/SUBSTR(sir FROM pozitie_inceput FOR lungime)** – functioneaza pe acelasi principiu ca mai sus, insa sunt returnate doar *lungime* caractere numarate de la pozitia de inceput
  - **LEFT(sir,N)** – returneaza primele N caractere ale sirului (echivalent cu SUBSTR(sir,1,N))
  - **RIGHT(sir, N)** - returneaza ultimele N caractere ale sirului ((echivalent cu SUBSTR(sir,-N))
  - **MID(sir, pozitie, lungime)** – echivalent cu SUBSTRING(sir, pozitie, lungime)
- gasirea pozitiei unui subsir in cadrul unui sir dat
  - **LOCATE(subsir, sir)/POSITON(subsir,sir)** – returneaza pozitia primei aparitii a subsirului (cautand de la stanga la dreapta in sirul mare), sau 0 in caz de absenta. Sintaxa aditionala **LOCATE(subsir, sir, pozitie)** incepe cautarea cu pozitia specificata din sirul mare
  - **INSTR(sir,subsir)** – acelasi efect cu LOCATE, dar ordinea argumentelor este inversata
- modificarea unui sir de caractere
  - **INSERT(sir, pozitie, numar_caractere, inlocuitor)** – inlocuieste un subsir cu altul. Subsirul de inlocuit este specificat prin argumentele 2 si 3 (pozitie de inceput si lungime).
  - **REPLACE(sir, de_inlocuit, inlocuitor)** – inlocuieste *toate aparitiile* subsirului de inlocuit cu cel inlocuitor
- conversie de la un set de caractere la altul
  - **CONVERT(sir USING charset_nou)** – produce, pe baza sirului primit ca prim argument, un nou sir de caractere ce foloseste charset-ul cerut

```
CREATE TABLE t (s VARCHAR(100) CHARSET ucs2);

INSERT INTO t VALUES('InfoAcademy');

SELECT LENGTH(s),CHAR_LENGTH(s),CONCAT('Academia',' ',s),CONCAT_WS(' ', 'Academia',s),
CONCAT(LPAD(s,12,'*'),'*'),SUBSTR(s,5,4),RIGHT(s,7)=SUBSTR(s,-7),INSERT(s,5,7,'rmativ') FROM t;

*************************** 1. row ***************************
                   LENGTH(s): 22
               CHAR_LENGTH(s): 11
    CONCAT('Academia',' ',s): Academia InfoAcademy
CONCAT_WS(' ', 'Academia',s): Academia InfoAcademy
  CONCAT(LPAD(s,12,'*'),'*'): *InfoAcademy*
                SUBSTR(s,5,4): Acad
     RIGHT(s,7)=SUBSTR(s,-7): 1
      INSERT(s,5,7,'rmativ'): Informativ
```

5.4.3.5. Funcţii pentru prelucrarea valorilor de tip temporal

Vom prezenta in continuare cele mai importante si des folosite functii pentru operarea cu valori de tip temporal. Ele pot fi impartite in urmatoarele categorii:

- determinare data/ora curenta
  - **CURDATE()/CURRENT_DATE()/CURRENT_DATE** – returneaza data curenta, in format AAAA-LL-ZZ
  - **CURTIME(),CURRENT_TIME(), CURRENT_TIME** – raportarea orei curente, sub forma HH:MM:SS
  - **NOW()/CURRENT_TIMESTAMP()/CURRENT_TIMESTAMP** – returneaza momentul de timp curent, sub forma AAAA-LL-ZZ OO:MM:SS
- extragerea elementelor componente ale unei date si/sau ore
  - **YEAR(data)** – returneaza anul component al datei primite ca argument
  - **MONTH(data)** - returneaza luna componenta a datei primite ca argument, intre 1 si 12
  - **DAY(date)/DAYOFMONTH(data)** - returneaza ziua din luna a datei primite a argument, intre 1 si 31
  - **DAYOFWEEK(data)** - indexul zilei din spatamana, cu 1=Duminica, 2=Luni, ..., 7=Sambata
  - **WEEKDAY(data)** - acelasi lucru, dar cu 0=Luni, 1=Marti, ... , 6=Duminica
  - **DAYOFYEAR(data)** - returneaza numarul zilei din an (1-366)
  - **HOUR(timp)** – returneaza componenta „ora” a argumentului. Pentru ora din zi, acesta este in intervalul 0-23, dar pentru intervale de timp poate fi >=24
  - **MINUTE(timp)** – analog, dar pentru minute
  - **SECOND(timp)** – analog, dar pentru secunde
- operatii aritmetice cu date si intervale de timp
  - **DATE_ADD(data, INTERVAL expresie unitate_masura)** – adauga intervalul de timp specificat la data primita ca prim argument (date sau datetime), returnand noua data rezultata. Unitatea de masura poate fi HOUR, MINUTE, SECOND, YEAR, DAY etc.
  - **DATE_SUB(data, INTERVAL expresie unitate_masura)** – analog, dar scade intervalul de timp specificat
  - **ADDTIME(data,interval_timp)** – adauga intervalul de timp la data specificata producand un nou moment de timp. Intervalul de timp poate fi negativ, rezultatul fiind cel de asteptat - efectuarea unei scaderi. Diferenta fata de DATE_ADD()/DATE_SUB() este ca se pot specifica intervale care nu cuprind numai numere intregi de zile, ore, minute etc.
  - **SUBTIME(data,interval_timp)** – analog cu ADDTIME(), dar scade intervalul de timp cerut
- functii pentru determinarea sau impunerea formatului datelor de tip temporal
  - **DATE_FORMAT(data,format)** – returneaza o reprezentare a datei primite ca prim argument in formatul dorit. Formatul se specifica sub forma unui string ce contine secvente de forma %c (unde c este un caracter). Iata cateva dintre secventele utile:
    - %Y – anul, reprezentat pe 4 cifre. Daca se doresc numai doua se utilizeaza %y
    - %m – luna, reprezentata intotdeauna pe 2 cifre (01-12). Pentru eliminarea eventualei cifre 0 de inceput se foloseste %c
    - %d – ziua reprezentata pe 2 cifre. Pentru suprimarea eventualei cifre 0 de inceput se foloseste %e
    - %H – ora din zi, 2 cifre. Pentru o reprezentare care omite prima cifra 0 se poate folosi %k
    - %i – minutul, 2 cifre (00-59)
    - %s sau %S – secunda, 2 cifre (00-59)
  - **STR_TO_DATE(sir,format)** – produce o valoare de tip data calendaristica pe baza unui sir de caractere ce specifica data, primit ca prim argument, si a formatului in care se afla aceasta data, sub

forma argumentului secund. Specificarea formatului respecta aceleasi reguli ca in cazul functiei DATE_FORMAT()

Exemplu:

```
SELECT NOW(), YEAR(CURRENT_DATE()), MONTH(CURRENT_DATE), DAY(CURDATE()), DATE_ADD(CURRENT_DATE,
       INTERVAL 1000 SECOND), ADDTIME(NOW(), '-104:23:59');

*************************** 1. row ***************************

                                      NOW(): 2009-08-06 13:01:59

                          YEAR(CURRENT_DATE()): 2009

                           MONTH(CURRENT_DATE): 8

                                DAY(CURDATE()): 6

DATE_ADD(CURRENT_DATE, INTERVAL 1000 SECOND): 2009-08-06 00:16:40

                ADDTIME(NOW(), '-104:23:59'): 2009-08-02 04:38:00
```

5.4.3.6. Alte funcţii de interes

- informatii despre server si sesiunea curenta
  - **USER()** - raporteaza username-ul folosit pentru a deschide sesiunea curenta
  - **DATABASE()/SCHEMA()** - raporteaza baza de date curenta (cea stabilita cu comanda USE)
  - **VERSION()** - raporteaza versiunea serverului
- ID-ul ultimei inregistrari introduse din sesiunea curenta
  - **LAST_INSERT_ID()** - raporteaza valoarea de AUTO_INCREMENT generata automat la precedenta operatie INSERT. Este necesara atunci cand avem de introdus doua inregistrari consecutive care depind una de alta – id-ul asignat primeia figureaza ca parte a celei de-a doua

5.4.4. Funcţii de agregare

5.4.4.1. Conceptul de agregare

Functiile simple, prezentate pana acum, operau asupra uneia sau mai multor valori primite explicit ca argument, sub forma de expresii, si produceau cate o valoare pentru fiecare înregistrare implicată in operaţia in care participau. Spre exemplu, scriind *SELECT CONCAT_WS(' ',Nume,Prenume) FROM Persoane*, functia CONCAT_WS se va aplica pe rand valorilor fiecarei inregistrari din tabela Persoane, producand cate o valoare pentru fiecare inregistrare si deci rezultand, in final, un numar de inregistrari returnate de SELECT egal cu numarul de inregistrari prelucrate. Putem spune ca o functie simpla compune valori „pe orizontala” - adica poate folosi valori multiple insa aflate pe acelasi rand (parte a aceleiasi inregistrari).

Spre deosebire de functiile simple, functiile de agregare realizeaza compunerea valorilor „pe verticala” - ele opereaza asupra tuturor valorilor unei coloane dintr-un grup de inregistrari, returnand pe baza acestora o singura valoare. Deducem imediat ca, daca o functie simpla producea cate o valoare per inregistrare, una de agregare produce cate o valoare per grup de inregistrari. Este posibil ca interogarea sa fie scrisa de asa natura incat toate inregistrarile tabelei sa formeze un singur grup, si deci o functie de agregare ar produce o singura valoare pentru toata tabela. Spre exemplu, daca avem o tabela Magazin care contine, printre altele, pretul fiecarui

produs, putem afla cat valoreaza produsele in intregul magazin, cu TVA si fara, calculand suma tuturor valorilor de pe coloana Pret a tabelei:

```
SELECT SUM(Pret) as PretTotal, SUM(Pret*1.19) as PretTotal CuTVA FROM Produse;

/* interogarea va returna o singura inregistrare cu doua coloane, dar in calculul valorilor
coloanelor au participat toate valorile de pe coloana Pret ale inregistrarilor tabelei */
```

Nota: *in exemplul de mai sus, functia de agregare SUM() actioneaza asupra tuturor inregistrarilor tabelei producand o singura inregistrare. Este posibil de asemenea sa obtinem pretul cumulat al produselor per categorie, ceea ce presupune ca SUM() sa divida totalul inregistrarilor in grupuri corespunzatoare categoriilor si sa calculeze cate o valoare pentru fiecare grup (vezi mai jos clauza GROUP BY)*

Functiile de agregare primesc ca argument o expresie; valoarea expresiei este evaluata pentru fiecare inregistrare in parte, rezultand un set de valori ce constituie datele de intrare ale functiei. Functia se apeleaza o singura data si prelucreaza acest set de valori producand pe baza lui un singur rezultat. Acest lucru difera de cazul functiilor simple, unde datele de intrare erau constituite de valorile argumentelor pentru inregistrarea curenta, iar functia era apelata de un numar de ori egal cu numarul de inregistrari, producand tot atatea valori.

Nota: *functiile de agregare se aplica numai pt instructiuni SELECT, pe cand cele simple pot fi utilizate si in alte locuri (ex: in clauza WHERE, ORDER BY etc)*

5.4.4.2. Operaţii matematice pe seturi de rezultate

MySQL ofera functii de agregare ce pot realiza urmatoarele operatii:

- valorile extreme ale unei coloane sau expresii. Functiile prezentate pot lucra atît cu valori numerice, cat si cu valori temporale sau siruri de caractere; in ultimul caz, ordonarea valorilor – si implicit valorile extreme – vor depinde de combinatia charset/collation a sirurilor in cauza.
  - **MIN(expresie)** – returneaza valoarea minima a expresiei dintre toate valorile rezultate prin evaluarea expresiei pentru fiecare inregistrare in parte
  - **MAX(expresie)** - analog, valoarea maxima
- sume
  - **SUM(expresie)** - însumarea valorilor unei expresii pentru toate înregistrările unui set de rezultate
- valoarea medie a unei expresii
  - **AVG(expresie)** – returneaza valoarea medie a tuturor valorilor expresiei calculate pentru toate inregistrarile returnate, sau NULL in cazul in care nu exista inregistrari

Functie simpla/operator	Functie de agregare
LEAST()	MIN()
GREATEST()	MAX()
CONCAT(), CONCAT_WS()	GROUP_CONCAT()
+	SUM()

Nota: *observam cateva corespondente intre functii simple sau operatori si functii de agregare – ele sunt sintetizate in tabelul de mai sus.*

```
# cel mai ieftin produs din lista
SELECT MIN(Pret) FROM Produse


# media notelor unui anumit student dintr-o anumita clasa
SELECT AVG(Nota) FROM Catalog WHERE Clasa='SQL 29 februarie' AND NumeStudent='John Doe'
```

5.4.4.3. Numărarea valorilor din seturi de rezultate

Functia MySQL folosita in acest scop este COUNT(). Ea poate fi utilizata in 3 moduri:

- **COUNT(*)** - numara randurile unui set de rezultate
- **COUNT(expresie)** – returneaza numarul de valori nenule ale expresiei, expresia fiind evaluata pe rand pentru fiecare inregistrare din setul de rezultate
- **COUNT(DISTINCT expresie)** – numarul de valori nenule unice. Atunci cand valorile sunt de tip string, egalitatea valorilor depinde de character set-ul si collation-ul atasate.

Exemplu: o interogare SELECT aplicata pe o tabela ce contine valorile din tabelul alaturat:

```
SELECT COUNT(*),COUNT(Deimpartit),COUNT(DISTINCT Deimpartit),COUNT(Deimpartit/Impartitor) FROM t;
+----------+------------------+--------------------------+-----------------------------+
| COUNT(*) | COUNT(Deimpartit) | COUNT(DISTINCT Deimpartit) | COUNT(Deimpartit/Impartitor) |
+----------+------------------+--------------------------+-----------------------------+
|        5 |                3 |                        2 |                           1 |
+----------+------------------+--------------------------+-----------------------------+
```

In exemplul de mai sus, prima coloana are valoarea 5 deoarece sunt luate in considerare toate randurile (chiar si cele formate exclusiv din valori NULL); a doua coloana are valoarea 3 deoarece se iau in considerare numai valorile non-NULL de pe coloana Deimpartit; a treia coloana are valoarea 2 deoarece, din cele trei valori anterioare, doua sunt egale; ultima coloana are valoarea 1 deoarece numara numai valorile nenule ale expresiei, iar expresia are valoarea NULL atat in cazul in care unul dintre operanzi este NULL, cat si in cazul in care se efectueaza impartire la 0 si sql_mode nu include optiune ERROR_FOR_DIVISION_BY_ZERO.

Deimpartit	Impartitor
7	2
NULL	4
7	NULL
15	0
NULL	NULL

5.4.4.4. Funcţii de agregare pentru şiruri de caractere

Mentionam aici functia **GROUP_CONCAT()**. Sintaxa sa generala este:

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
      [ORDER BY {unsigned_integer | col_name | expr}
            [ASC | DESC] [,col_name ...]]
      [SEPARATOR str_val])
```

Functia poate primi ca argument una sau mai multe expresii, ale caror valori sunt evaluate pentru toate randurile setului de rezultate si concatenate. Valorile NULL rezultate sunt ignorate. Inaintea concatenarii, valorile pot fi ordonate dupa criteriile dorite. La conccatenare se poate specifica optional un separator, cu acelasi rol ca in cazul functiei CONCAT_WS().

Exemplu: aplicam functia GROUP_CONCAT() pe tabela prezentata la paragraful anterior:

```
SELECT GROUP_CONCAT(Deimpartit SEPARATOR ';') AS A,
       GROUP_CONCAT(DISTINCT Deimpartit SEPARATOR ';') AS B,
       GROUP_CONCAT(Impartitor +1) AS C,
       GROUP_CONCAT(Deimpartit+Impartitor*2 ORDER BY Deimpartit DESC SEPARATOR '#') AS D FROM t;
+---------+------+-------+-------+
| A       | B    | C     | D     |
+---------+------+-------+-------+
| 7;7;15 | 7;15 | 3,5,1 | 15#11 |
+---------+------+-------+-------+
```

5.4.4.5. Aplicarea funcţiilor de agregare pentru grupuri de înregistrări

Exista destule cazuri in care dorim aplicarea unor functii de agregare, insa nu pe toate inregistrarile unei tabele, ci pe grupuri de inregistrari. Spre exemplu, intr-o tabela cu cheltuieli, am putea face suma lunara a cheltuielilor – ceea ce presupune aplicarea functiei de agregare SUM() pentru fiecare grup de inregistrari ce tin de aceeasi luna.

Putem determina restrictiona efectul functiilor de agregare la grupuri de inregistrari folosind clauza GROUP BY, urmata de o una sau mai multe expresii separate prin virgula. Efectul va fi ca inregistrarile produse (si eventual filtrate de clauza WHERE) vor fi impartite in grupuri, fiecare grup continand inregistrarile care genereaza aceeasi valoare a expresiilor din GROUP BY, iar functiile de agregare vor produce cate o valoare pentru fiecare grup, prin prelucrarea valorilor continute in acesta. Sa consideram cateva exemple:

```
/* studentul cu nota cea mai mare din fiecare clasa. Fiecare grup va fi constituit din toate
inregistrarile care au aceeasi valoare pe coloana Clasa; din cadrul fiecarui grup va fi pastrata
valoarea cea mai mare de pe coloana Nota /*

SELECT NumeStudent, MAX(Nota) FROM Catalog GROUP BY Clasa;


/* studentii care au luat cea mai mare nota in fiecare clasa la fiecare examen  (altfel spus, de
data aceasta dorim ca in cadrul unei clase sa vedem cea mai mare nota pentru fiecare examen in
parte). Este necesara „subdivizarea” grupurilor din exemplul anterior in sub-grupuri, cate unul
pentru fiecare examen in parte. Reformuland, trebuie sa obtinem cate o valoare pentru fiecare
combinatie Clasa – Examen unica */

SELECT NumeStudent, MAX(Nota) FROM Catalog GROUP BY Clasa,Examen;
```

Clauza GROUP BY are un efect secundar – acela al ordonarii inregistrarilor dupa coloanele mentionate in expresiile atasate. Spre exemplu, GROUP BY Clasa,Examen va ordona ca si cum s-ar fi folosit clauza ORDER BY Clasa,Examen. Aceasta manifestare poate fi anulata adaugand explicit clauza ORDER BY NULL la instructiunea SELECT. In plus, in MySQL se pot folosi cuvintele cheie ASC si DESC direct dupa clauza GROUP BY:

```
SELECT NumeStudent,Examen,AVG(Nota) FROM Catalog GROUP BY NumeStudent,Examen DESC;
```

Nota: *clauza GROUP BY, atunci cand apare in cadrul unei instructiuni SELECT, trebuie plasata dupa WHERE (sau dupa FROM, daca nu exista WHERE) si inainte de ORDER BY.*

Atunci cand o interogare SELECT dispune de clauza GROUP BY, nu are in general sens ca in lista de coloane returnate sa se afle si alte coloane decat cea/cele dupa care se face gruparea. Sa consideram exemplul numarului de examene per student, in care includem in lista de coloane returnate si coloana Nume:

```
SELECT Examen, DataExamen, COUNT(Examen) GROUP BY NumeStudent;
```

Efectul este ca inregistrarile tabelei vor fi impartite in grupuri, fiecare grup avand in comun aceeasi valoare pentru StudentName, si functia COUNT va numara valorile non-NULL de pe coloana Examen din fiecare grup. Asadar, pentru fiecare grup se returneaza o singura inregistrare in rezultatul final al lui SELECT. Primele doua coloane din acest rezultat ridica o problema: pentru fiecare GRUP trebuie returnata o data de examen si un examen – ale carei inregistrari din grup sa fie acestea? De aceea, logic vorbind, in general nu are sens sa *cerem* astfel de coloane; daca totusi o facem, MySQL va returna valori impredictibile, sau, daca sql_mode contine optiunea ONLY_FULL_GROUP_BY, va produce in eroare in cazul unei astfel de interogari construite defectuos.

Aceeasi problema o intalnim si daca folosim clauza GROUP BY insa fara a exista in cadrul instructiunii nici o functie de agregare (ex: *SELECT Nume Student FROM Catalog GROUP BY Examene*): inregistrarile vor fi impartite in grupuri, si pentru fiecare grup trebuie returnata o singura valoare, numai ca, neavand functie de agregare, felul in care valorile din grup participa la crearea valorii unice per-grup este nespecificat.

5.4.4.6. Filtrarea în funcţie de o coloana calculata cu funcţie de agregare

Atunci cand dorim sa restrangem numarul de inregistrari returnate de către o interogare SELECT folosim in general clauza WHERE. In cazul in care in conditiile din WHERE apar coloane ale caror valori sunt calculate cu functii de agregare, nu putem folosi WHERE (explicatia urmeaza mai jos) ci trebuie sa apelam la clauza HAVING.

Fie o tabela care contine note la examene ale studentilor din diverse clase. Fiecare inregistrare contine numele studentului, clasa, numele examenului, nota obtinuta. Sa spunem ca dorim sa obtinem lista numelor studentilor clasei 'SQL 29 februarie' impreuna cu numarul de examene promovate al fiecaruia, conditia de promovare a unui examen fiind obtinerea unei note de cel putin 80 de puncte:

```
SELECT Nume, COUNT(Nota>=80) FROM Examene WHERE Clasa='SQL 29 februarie' GROUP BY Nume;
```

Sa spunem acum ca am vrea sa afisam doar studentii cu mai putin de 4 examene promovate, in ideea de a-i atentiona sa recupereze examenele restante. Primul impuls ar putea fi sa adaugam o conditie suplimentara in clauza WHERE; conditia fiind pusa chiar in functie de coloana ce participa la agregare, avem nevoie de un alias pentru coloana. Surpriza este insa ca primim o eroare:

```
SELECT Nume, COUNT(Nota>=80) AS ExPromovate FROM Examene
WHERE Clasa='SQL 29 februarie' AND ExPromovate<4 GROUP BY Nume;
ERROR 1054 (42S22): Unknown column 'ExPromovate' in 'where clause'
```

Explicatia este urmatoarea: conditiile din clauza WHERE sunt folosite pentru a filtra rezultatele incluse in setul final returnat de SELECT, iar functiile de agregare actioneaza abia dupa inregistrarile ce compun setul de rezultate au fost alese (functiile de agregare opereaza pe rezultatul filtrarii). De aceea nu am putea filtra in

WHERE dupa o coloana calculata cu functie de agregare, deoarece la momentul aplicarii lui WHERE valoarea acelei coloane nu exista inca. Este necesara o filtrare efectuata dupa aplicarea functiilor de agregare.

In astfel de cazuri, clauza „salvatoare” este HAVING, care filtreaza rezultatele dupa aceleasi principii ca si WHERE, insa actioneaza abia DUPA executarea functiilor de agregare:

```
SELECT Nume, COUNT(Nota>=80) AS ExPromovate FROM Examene
WHERE Clasa='SQL 29 februarie' GROUP BY Nume HAVING ExPromovate<4;
```

Etapele prin care trece executia unei interogari SELECT ce are clauze WHERE, GROUP BY si HAVING sunt urmatoarele:

- se aplica clauza WHERE: serverul parcurge inregistrarile tabelei si pastreaza doar acele inregistrari care corespund criteriilor din WHERE
- se aplica clauza GROUP BY: inregistrarile sunt grupate astfel incat fiecare grup sa contina acele inregistrari ce produc aceeasi valoare pentru combinatia de expresii din GROUP BY
- se aplica functiile de agregare, ce vor produce cate o valoare pentru fiecare grup creat anterior
- se genereaza setul de inregistrari pre-final, ce poate cuprinde atat date extrase direct din tabela, cat si rezultate ale functiilor de agregare
- dintre aceste inregistrari se pastreaza doar cele care corespund conditiilor din clauza HAVING

Nota: *o aplicatie simpla a clauzei HAVING este obtinerea listei de duplicate de pe coloana unei tabele:*

```
SELECT Coloana, COUNT(*) as nr FROM Tabela GROUP BY Coloana HAVING nr>1
```

5.4.4.7. Cuvîntul cheie DISTINCT

Cuvantul cheie DISTINCT poate fi folosit in majoritatea functiilor de agregare, efectul sau fiind aplicarea acestor functii numai pentru inregistrarile cu valori unice pentru coloana/expresia specificata ca argument, duplicatele fiind ignorate. Locul lui DISTINCT este imediat inaintea argumentelor functiei (dar dupa paranteza deschisa). El poate fi folosit in toate functiile de agregare prezentate, desi in cazul unora dintre ele nu va produce nici o diferenta (ex: in cazul lui MIN() si MAX()). Iata cateva exemple, aplicate pe datele din tabelul alaturat:

```
SELECT SUM(Deimpartit), SUM(DISTINCT Deimpartit),
       AVG(Deimpartit),AVG(DISTINCT Deimpartit) FROM impartiri;
*************************** 1. row ***************************
         SUM(Deimpartit): 29
SUM(DISTINCT Deimpartit): 22
         AVG(Deimpartit): 9.6667
AVG(DISTINCT Deimpartit): 11.0000
```

Deimpartit	Impartitor
7	2
NULL	4
7	NULL
15	0
NULL	NULL

5.5. BIBLIOGRAFIE

- MySQL 5 Certification Study Guide – Cap. 9.4 si 9.5, Cap. 10.1 – 10.6
- MySQL Reference Manual: operatori si functii - http://dev.mysql.com/doc/refman/5.0/en/functions.html