

# SQL Fundamentals MySQL (Modulul 1)

## Cuprins

Cuprins .....	1
1. Introducere despre bazele de date .....	4
1.1 Introducere .....	4
1.1.1 Aplicații ale bazelor de date pe Web .....	4
<b>1.1.2 Aplicații ale bazelor de date în alte domenii .....</b>	<b>6</b>
1.2 Obiective .....	6
1.3 Prezentare lecții .....	8
1.4 Noțiuni teoretice .....	9
1.4.1 Date, informații, cunoștințe .....	9
1.4.2 Bază de date .....	9
<b>1.4.3 Sistem de gestiune a bazelor de date (Database Management System) .....</b>	<b>9</b>
<b>1.4.4 Regulile lui Codd .....</b>	<b>10</b>
<b>1.4.5 Exemple de SGBD-uri .....</b>	<b>11</b>
<b>1.4.6 Colectarea și analizarea datelor. Modelul conceptual .....</b>	<b>12</b>
<b>1.4.7 Entități. Instanțe. Atribute. Identificator unic. ....</b>	<b>13</b>
<b>1.4.8 Etapele realizării unei aplicații informatice .....</b>	<b>15</b>
<b>1.4.8.1 Analiza sistemului informatizat .....</b>	<b>15</b>
<b>1.4.9 Modelul relațional .....</b>	<b>16</b>
1.4.10 Operatorii modelului relațional .....	18
2.1 Proiectarea bazei de date relaționale .....	22
2.2 Realizarea componentelor logice .....	23
2.3 Punerea în funcțiune și exploatarea .....	24
2.4 Întreținerea și dezvoltarea sistemului .....	24
2.5 Relații. Tipuri de relații între tabele .....	24
2.7 Limbajul SQL. Introducere .....	30
2.8 Clienți MySQL .....	30
Tema Sedința 2 .....	34
3.1 Introducere .....	34
<b>3.2 Crearea unei baze de date .....</b>	<b>36</b>

<b>3.3 Ștergerea unei baze de date</b>	36
<b>3.4 Utilizarea unei baze de date</b>	36
<b>3.5 Crearea unei tabele</b>	36
<b>3.6 Ștergerea unei tabele</b>	36
<b>3.7 Modificarea unei tabele</b>	37
<b>3.8 Modificarea structurii unei tabele</b>	37
<b>3.9 Tipuri de date</b>	37
Tema Sedinta 3	43
4.2 Instrucțiunea INSERT	46
4.3 Instrucțiunea UPDATE	47
4.4 Instrucțiunea DELETE	48
4.5 Ștergerea tuturor datelor dintr-o tabelă și resetarea auto incrementului	49
4.10 Clauza HAVING	52
4.11 Alias	53
4.14 Clauza DISTINCT	56
4.15 Concluzii	56
Tema sedinta 4	57
5.1 Tipuri de operatori	58
5.2 Operatori matematici	58
5.3 Operatori logici	59
5.4 Operatori de comparare	60
5.5 Operatori de evaluare condiționată	61
5.6 Funcții predefinite MySQL	62
5.7 Funcții matematice	62
5.8 Funcții de comparare	63
5.9 Funcții condiționale	63
5.10 Valoarea NULL	64
5.11 Funcții pentru șiruri de caractere	64
5.12 Funcții pentru date calendaristice	65
5.13 Funcții de agregare	66
5.14 Concluzii	66
Tema Sedinta 5	67
6. Uniuni de tabele	68
6.2 Alias-uri de tabele	68
<b>6.3 Tipuri de join-uri</b>	69
<b>6.4 Asocieri de tabele fără restricții (CROSS JOIN)</b>	69

6.5 Asocieri de tabele cu restricții.....	70
6.5.1 Clasificare .....	70
6.6 INNER JOIN .....	70
6.6.1 Forme posibile de utilizare INNER JOIN .....	71
<b>6.7 OUTER JOIN .....</b>	<b>71</b>
6.7.1 LEFT OUTER JOIN.....	72
6.7.2 RIGHT OUTER JOIN .....	73
6.8 SELF JOIN .....	73
6.9 Reuniuni. UNION și UNION ALL.....	74
6.9.1 Operatorul UNION .....	75
<b>6.9.2 Operatorul UNION ALL.....</b>	<b>75</b>
<b>6.9.3 Utilizarea clauzei ORDER BY într-o instrucțiune compusă de reuniune.....</b>	<b>75</b>
<b>6.10 Concluzii .....</b>	<b>76</b>
7.1 Subinterogări .....	77
7.2 Tipuri de subinterogări .....	79
7.2.1 Subinterogări de tip scalar .....	79
7.2.2 Subinterogări de tip listă.....	80
7.2.3 Subinterogări de tip rând.....	80
7.3 Operatori folosiți pentru introducerea subinterogărilor .....	81
7.3.1 Operatorul ANY .....	81
7.3.2 Operatorul ALL .....	81
7.3.3 Operatorul EXISTS .....	82
7.4 Concluzii.....	82
Tema Sedinta 7 .....	82
8.1 Tabele virtuale (vederi, view-uri) .....	83
8.1.1 Crearea unei tabele virtuale .....	84
8.1.2 Redefinirea unei tabele virtuale.....	84
8.1.3 Ștergerea unei tabele virtuale .....	85
8.2 Tabele temporare .....	85
8.2.1 Noțiuni generale .....	85
8.2.2 Modalități de creare a unei tabele temporare .....	86
8.2.3 Ștergerea unei tabele temporare .....	87
8.2.4 Deosebiri între tabele temporare și tabele virtuale .....	87
8.3 Index .....	88
8.3.1 Conceptul de index .....	88
8.3.2 Deosebiri între PRIMARY KEY și constrângerea UNIQUE .....	89

8.3.3 Considerații generale despre indexare.....	89
8.3.4 Avantaje și dezavantaje întâlnite la utilizarea indecșilor.....	90
8.3.5 Utilitatea creării indecșilor.....	91
8.3.6 Metode de regăsire a datelor.....	91
8.3.7 Clasificare.....	92
8.4 Operații.....	93
8.4.1 Crearea unui index.....	93
8.4.2 Vizualizarea indecșilor existenți.....	94
8.5 Concluzii.....	94

# 1. Introducere despre bazele de date

## 1.1 Introducere

Aplicațiile informatice utilizate în prezent lucrează cu un număr foarte mare de date care trebuie stocate în așa fel încât să le putem accesa rapid și ușor. Astfel, majoritatea aplicațiilor, de la site-uri și alte aplicații web până la aplicații bancare sau de gestiune a clienților, folosesc baze de date *relaționale*.

În acest curs de **Fundamente Baze de date MySQL** ne propunem să parcurgem câteva noțiuni teoretice fundamentale pentru înțelegerea conceptelor folosite în lucrul cu baze de date relaționale, concepte tot mai des întâlnite în limbajul informatic curent (*informație, date, baze de date, etc.*) dar și să trecem în revistă etapele care sunt parcurse în realizarea aplicațiilor informatice care folosesc baze de date (de la proiectarea unei baze de date până la interogări avansate asupra bazei de date).

Cursul se adresează persoanelor fără experiență și cunoștințe în domeniul bazelor de date, dar și persoanelor care au cunoștințe și o minimă experiență în lucrul cu baze de date.

Noțiunile teoretice fundamentale despre bazele de date relaționale, precum și elementele limbajului de interogare **SQL**, sunt introduse pas cu pas și sunt însoțite de exemplificări și utilizări practice.

Am ales ca mediu de dezvoltare a aplicațiilor ce vor fi realizate în acest curs sistemul de gestiune a bazelor de date **MySQL**.

Acest sistem de gestiune a bazelor de date (**SGBD**), **MySQL**, este foarte cunoscut datorită utilizării sale în aplicațiile și site-urile web împreună cu limbajul de programare PHP.

### 1.1.1 Aplicații ale bazelor de date pe Web

Aplicațiile informatice utilizate în prezent lucrează cu un număr foarte mare de date care trebuie stocate în așa fel încât să le putem accesa rapid și ușor. Astfel, majoritatea aplicațiilor, de la site-uri și alte aplicații web până la aplicații bancare sau de gestiune a clienților, folosesc baze de date relaționale.

În acest curs de **Fundamente Baze de date MySQL** ne propunem să parcurgem câteva noțiuni teoretice fundamentale pentru înțelegerea conceptelor folosite în lucrul cu baze de date relaționale, concepte tot mai des întâlnite în limbajul informatic curent (informație, date, baze de date, etc.) dar și să trecem în revistă etapele care sunt parcurse în realizarea aplicațiilor informatice care folosesc baze de date (de la proiectarea unei baze de date până la interogări avansate asupra bazei de date).

Cursul se adresează persoanelor fără experiență și cunoștințe în domeniul bazelor de date, dar și persoanelor care au cunoștințe și o minimă experiență în lucrul cu baze de date.

Noțiunile teoretice fundamentale despre bazele de date relaționale, precum și elementele limbajului de interogare **SQL**, sunt introduse pas cu pas și sunt însoțite de exemplificări și utilizări practice.

Am ales ca mediu de dezvoltare a aplicațiilor ce vor fi realizate în acest curs sistemul de gestiune a bazelor de date **MySQL**.

Acest sistem de gestiune a bazelor de date (**SGBD**), **MySQL**, este foarte cunoscut datorită utilizării sale în aplicațiile și site-urile web împreună cu limbajul de programare PHP.

Orice site care are un modul de creare cont și login, orice magazin online, site de știri, blog, etc. are o bază de date în care este ținută informația în mod structurat. Cea mai mare parte a site-urilor de pe Internet care trec de nivelul de site de prezentare folosesc baze de date.

Astfel o persoană care dorește să lucreze în domeniul programării web, pe lângă cunoștințele de HTML și CSS folosite în partea de dezvoltare a aplicației ce interacționează cu utilizatorul, și cele de programare PHP, are nevoie și de cunoștințe de baze de date relaționale, întrucât aproape toate site-urile și toate aplicațiile web conțin informația în baze de date relaționale. Jobul de programator web este unul foarte interesant și există o cerere mare pe piață pentru persoane care au cunoștințe de programare web. Este un job provocator, pentru că fiecare proiect aduce ceva nou, la fiecare proiect se pot învăța lucruri suplimentare și aduce și satisfacția că produsul realizat este vizualizat de foarte mulți oameni (ne referim aici la site-urile web).

Tipologia site-urilor web pornește de la site-uri de prezentare și continuă cu magazine online, bloguri, ajungând până la site-uri complexe (portaluri).

În cadrul acestor tipologii avem site-uri web statice, realizate doar în HTML sau site-uri web dinamice cu informația introdusă dintr-o secțiune de administrare, în acest sens folosindu-se un limbaj de programare și baze de date relaționale.

Iată modul de funcționare și de interacțiune cu serverul web și cu serverul de baze de date MySQL la realizarea unui site web:

Browser-ul web interpretează doar cod HTML. Astfel că, dacă avem un site cu pagini statice, unde nu folosim nici un limbaj de programare, ci doar limbajul de marcare HTML, iar fișierele ce conțin aceste pagini au extensia .html sau .htm nu avem nevoie să instalăm altceva pe calculatorul nostru pentru a putea deschide acele pagini.

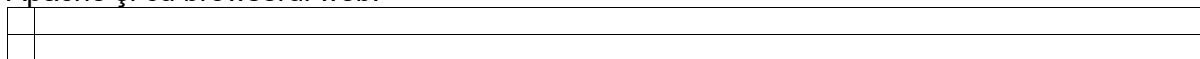
În schimb, dacă vom realiza un site web dinamic folosind limbajul PHP și baze de date **MySQL**, avem nevoie de instalarea pe calculator a unui server.

PHP este un limbaj care funcționează pe partea de server, în timp ce HTML este un limbaj pe partea de client.

Codul PHP este transmis către serverul de Apache, acesta interpretează codul primit și generează cod HTML pe care îl transmite către browser astfel că browserul web primește tot cod HTML, singurul pe care știe să îl interpreteze și să îl afișeze.

De aceea, dacă vizualizăm codul sursă al unui site realizat în PHP vom vedea că el este transformat în cod HTML și astfel este afișat în sursa paginii. Deci, acces la codul PHP nu putem avea, pentru a vedea cum a fost readactat codul, decât dacă accesăm fișierele cu extensia php de pe server.

În schema de mai jos vom reprezenta modul în care interacționează PHP-ul cu serverul Apache și cu browserul web:



Primul pas (1) este reprezentat de cererea pe care clientul (browser-ul web) o adresează serverului în momentul în care se accesează o pagină PHP printr-un URL. Serverul trimite pagina spre procesare interpretorului PHP (2).

Dacă avem și instrucțiuni MySQL, din PHP se face conectarea la baza de date MySQL și se trimite cererea către serverul MySQL (3).

Serverul MySQL execută instrucțiunile specifice și returnează rezultatele către PHP (4). Interpretorul PHP returnează aceste rezultate către serverul Apache (5).

Serverul Apache returnează clientului cod HTML, pe care acesta știe să îl interpreteze și să îl afișeze (6).

Cursul nu se ocupă de realizarea site-urilor web dar am prezentat acest mod de interacțiune dintre client – server – PHP – server MySQL pentru a înțelege modul în care sunt folosite bazele de date și în realizarea site-urilor web.

## 1.1.2 Aplicații ale bazelor de date în alte domenii

O mare parte dintre aplicațiile software existente folosesc baze de date pentru stocarea și extragerea informațiilor. Aceste baze de date se interoghează ulterior pentru obținerea de diverse statistici. De exemplu, informațiile despre clienții și facturile unei companii se țin într-o bază de date.

## 1.2 Obiective

Pe parcursul acestui curs de inițiere în baze de date vom parcurge noțiunile fundamentale teoretice și vom învăța cum se creează/șterg tabele în **MySQL**, cum se înserează/modifică/șterg înregistrări într-o tabelă din baza de date, precum și diverse interogări de regăsire a datelor din tabele pornind de la cele simple și ajungând la cele mai complexe. De asemenea, vom învăța să lucrăm cu **vederi (view-uri sau tabele virtuale)**, **join-uri**, **reuniuni**, **tabele temporare** și **tranzacții**.

Obiectivele cursului **Fundamente Baze de Date MySQL** sunt următoarele:

- să înțelegeți noțiunile teoretice folosite în domeniul bazelor de date (dată, informație, bază de date, sistem de gestiune a bazelor de date, tabelă, atribut, înregistrare, etc.);

- să înțelegeți principiile generale ale algebrei relaționale, să cunoașteți operațiile de reuniune, intersecție, diferență, produs cartezian folosite în teoria mulțimilor;
- să înțelegeți principiul normalizării bazelor de date
- să cunoașteți și să puteți instala mediile de lucru folosite pentru lucrul cu baze de date **MySQL** (serverul **WAMP** și **HeidiSQ/MySQL Workbench**);
- să înțelegeți noțiunea de relație între tabele și să cunoașteți tipurile de relații între tabele;
- să puteți proiecta o bază de date;
- să cunoașteți sintaxa **SQL** și instrucțiunile de bază (**INSERT**, **UPDATE**, **DELETE**, **SELECT**);
- să cunoașteți limbajul de descriere a datelor (**LDD**), limbajul de manipulare a datelor (**LMD**), precum și **tipuri de date** și **operatori MySQL**;
- să cunoașteți câteva **funcții predefinite MySQL** (pentru lucrul cu șiruri de caractere, cu date calendaristice, funcții matematice, etc.);
- să înțelegeți noțiunile de **join**, **union** și **subinterogări** și să le folosiți în interogări complexe;
- să înțelegeți utilizarea **vederilor (tabele virtuale)** și modul de folosire;
- să înțelegeți noțiunea de **index**;
- să știți cum **se definesc indecșii**, ce reprezintă, când se folosesc și cum se șterg;
- să cunoașteți **tipurile de index** ce pot fi declarate pe o tabelă și care sunt **avantajele** și **dezavantajele** folosirii lor;
- să cunoașteți **efectele** ce pot fi produse asupra **operațiilor de modificare** a datelor (**INSERT**, **UPDATE**) de către **indecșii de unicitate (indexul unic și cheia primară)** și ce **clauze speciale** se folosesc în **instrucțiunile de modificare** pentru a **preîntâmpina eventualele erori** ce pot fi cauzate de prezența indecșilor de unicitate;
- să înțelegeți noțiunea de **tabelă temporară**, cum se definesc tabelele temporare și care este utilitatea lor;
- să înțelegeți **diferența** dintre **tabelă temporară** și **tabelă virtuală (view, vedere)**;
- să înțelegeți noțiunea de **tranzacție**, la ce sunt utile și cum se folosesc tranzacțiile;
- să cunoașteți instrucțiunea **INSERT** prin care pot fi **inseminate înregistrări într-o tabelă preluate dintr-o instrucțiune SELECT**;
- să știți să realizați **o copie a unei tabele** din baza de date;
- să cunoașteți **clauzele** posibile ce pot fi întâlnite la **definirea constrângerilor de integritate referențială (FOREIGN KEY)**;
- să înțelegeți noțiunea de **motor de stocare (engine)** și să cunoașteți **principalele deosebiri** între cele mai cunoscute **motoare de stocare** precum și **modalitatea de modificare a motorului de stocare**;
- să cunoașteți modalitatea de **definire** și de **schimbare** a unui **set de caractere (CHARSET)** ce va fi utilizat la memorarea datelor din tabele, precum și **definirea și modificarea setului de reguli** ce stă la baza **comparării caracterelor din set (collation)**;
- să aprofundați cunoașterea **tipurilor de date MySQL**;
- să puteți realiza aplicații de complexitate medie cu baze de date **MySQL** (de la proiectarea bazei de date până la instrucțiuni complexe aplicate pe baza de date).

La finalul cursului, pentru a obține diploma finală, înainte de a susține testul final ce va conține întrebări de tip grilă, trebuie să prezentați o aplicație complexă în care să fie folosite noțiunile învățate. Este vorba despre o bază de date care asigură gestiunea unui anumit domeniu (de exemplu o bază de date pentru gestiunea cărților dintr-o bibliotecă sau o bază de date folosită la gestiunea angajaților unei companii, o bază de date folosită la gestiunea pacienților și medicilor dintr-un spital, etc.).

## 1.3 Prezentare lecții

Iată în continuare o descriere succintă a lecțiilor ce vor fi parcurse în cadrul cursului de **Fundamente Baze de Date MySQL**.

În **prima lecție** a cursului sunt prezentate pe larg **noțiuni teoretice** precum și câteva noțiuni fundamentale de **algebră relațională**. Înțelegerea conceptului de normalizare precum și prezentarea formelor normale reprezintă următorul subiect abordat în partea de început.

**Cea de-a doua lecție** se axează pe prezentarea concretă a mediului de lucru (**HeidiSQL/MySQL Workbench**) și pe prezentarea modului în care este proiectată o bază de date. De asemenea, vom aborda subiectul relaționării – concept fundamental în bazele de date relaționale – și vom explica și exemplifica tipurile de relații ce pot exista între tabelele unei baze de date. Continuăm această lecție cu o introducere în limbajul **SQL** (**Structured Query Language**).

În **Lecția 3, Limbajul de Descriere a Datelor (LDD)** sau, în limba engleză **Data Description Language (DDL)**, vom prezenta și exemplifica instrucțiunile de creare și ștergere a unei baze de date, creare și ștergere a unei tabele dintr-o bază de date precum și instrucțiunile de modificare a structurii tabelului. De asemenea, vom discuta despre tipurile de date **MySQL**.

În cea de-a **patra lecție** prezentăm **Limbajul de Manipulare a Datelor (LMD)**, în limba engleză **Data Manipulation Language (DML)**. Limbajul de Manipulare a Datelor se referă la cele patru instrucțiuni fundamentale ale limbajului **SQL**:

- **INSERT** – pentru introducerea înregistrărilor într-o tabelă;
- **UPDATE** – pentru modificarea înregistrărilor din tabele;
- **DELETE** – pentru ștergerea înregistrărilor din tabele;
- **SELECT** – instrucțiunea de regăsire a informațiilor din baza de date.

Lecția următoare (**lecția 5**) prezintă noțiuni despre **operatorii și funcțiile predefinite** ale **MySQL**. Este vorba de operatorii aritmetici, logici și de comparare și de funcții matematice, funcții pentru lucrul cu șiruri de caractere, cu date calendaristice, etc.

Cursul continuă, în **lecția 6**, cu noțiuni teoretice despre **JOIN-uri**, tipuri de **JOIN-uri** și **reuniuni** (operatorii **UNION** și **UNION ALL**).

Un alt capitol important al acestui curs este prezentat în **lecția numărul 7** și este cel în care sunt prezentate **subinterogările** și **tipurile de subinterogări**.

**Penultima lecție (lecția 8)** este dedicată prezentării **tabelului virtuale (view-uri sau vederi)** și **tabelului temporare**. Tot în cadrul acestei lecții vor fi menționate și anumite noțiuni de **optimizare**. Va fi introdus conceptul de **index** și vor prezentate și exemplificate **tipurile de indecși** ce pot fi aplicați pe coloanele unei tabele dintr-o bază de date.

**Ultima lecție** conține o referire amănunțită asupra **conceptului de tranzacție**. Vom prezenta utilitatea și scopul folosirii tranzacțiilor, proprietățile utilizarea lor. Va fi prezentată **instrucțiunea INSERT** având în interior o **subinterogare**, modalitatea de **realizare a copiei unei tabele**. Tot în cadrul acestei lecții vom discuta despre **seturi de caractere** și **reguli aplicate la compararea caracterelor**; despre **clauzele posibile** ce apar la **definirea unui FOREIGN KEY** ca și despre **motoare de stocare (engine)**.



## 1.4 Noțiuni teoretice

### 1.4.1 Date, informații, cunoștințe

Auzim adesea folosindu-se termenii „societate informațională”, „tehnologia informației” însă de multe ori cuvântul „informație” este folosit fără a înțelege clar sensul acestui cuvânt și faptul că este o diferență între **date**, **informații**, **cunoștințe** și **obiecte**.

În general, conținutul gândirii umane operează cu următoarele concepte:

**Date** – constau în material brut, fapte, simboluri, numere, cuvinte, poze fără un înțeles de sine stătător, neintegrate într-un context, fără relații cu alte date sau obiecte. Ele se pot obține în urma unor experimente, sondaje etc.

**Informații** – prin prelucrarea datelor și găsirea relațiilor dintre acestea se obțin informații care au un înțeles și sunt integrate într-un context. Datele organizate și prezentate într-un mod sistematic pentru a sublinia sensul acestor date devin informații. Pe scurt *informațiile sunt date prelucrate*. Informațiile se prezintă sub formă de rapoarte, statistici, diagrame etc.

**Cunoștințele** sunt colecții de date, informații, adevăruri și principii învățate, acumulate de-a lungul timpului. Informațiile despre un subiect reținute și înțelese și care pot fi folosite în luarea de decizii devin cunoștințe. Cu alte cuvinte, cunoștințele apar în momentul utilizării informației.

**Obiectele** reprezintă cunoștințe pentru care se știe comportamentul și proprietățile, referitoare la o entitate din lumea reală.

### 1.4.2 Bază de date

O *bază de date* (**BD**) conține toate informațiile necesare despre obiectele ce intervin într-o mulțime de aplicații, relațiile logice între aceste informații și tehnicile de prelucrare corespunzătoare.

O *bază de date* reprezintă o colecție de date organizate ce pot fi accesate simultan de mai mulți utilizatori. Prelucrarea datelor se referă la operațiile de *introducere*, *ștergere*, *actualizare* și *interogare* a datelor. O *bază de date* este o colecție de înregistrări sau de informații introduse și stocate într-un calculator într-un mod sistematic (structurat).

O *bază de date* poate fi interogată (întrebată) de către noi sau de către un program prin intermediul unui limbaj relativ simplu (în general **SQL**) și răspunde cu informație, în funcție de care se iau decizii. Pentru valorificarea informației ce poate fi extrasă dintr-o bază de date, este esențial modul în care organizăm și stocăm datele într-o bază de date.

### 1.4.3 Sistem de gestiune a bazelor de date (Database Management System)

Un *sistem de gestiune a bazelor de date* (**SGBD**) reprezintă un sistem de programe care permite construirea unor baze de date, introducerea informațiilor în bazele de date și dezvoltarea de aplicații privind bazele de date.

Cu alte cuvinte, aplicația care este folosită pentru a realiza, a administra și a interoga o bază de date este numită sistemul de gestiune sau de management al bazei de date (**SGBD**). În limba engleză denumirea pentru sistemul de management al bazei de date este **Database Management System (DBMS)**.

Printr-un **SGBD** se realizează interacțiunea utilizatorului cu baza de date. Orice **SGBD** conține, printre alte componente un limbaj de descriere a datelor (**LDD**) care permite descrierea structurii bazei de date, a fiecărei componente a ei, a relațiilor dintre componente, a drepturilor de acces ale utilizatorilor la baza de date, a restricțiilor în reprezentarea informațiilor.

O altă componentă a unui **SGBD** este limbajul de manipulare a datelor (**LMD**) ce permite operații asupra datelor aflate în baza de date, cum ar fi: inserarea unui element, ștergerea unui element, modificarea unui element, căutarea (regăsirea) unor elemente.

Teoria relațională, foarte bine pusă la punct într-un domeniu de cercetare distinct, a dat o fundamenteare solidă realizării de **SGBD**-uri performante. La sfârșitul anilor '80 și apoi în anii '90 au apărut, în special o dată cu pătrunderea în masă a microcalculatoarelor, numeroase sisteme de gestiune bazelor de date relaționale (**SGBDR**-uri).

Aceasta a însemnat o evoluție de la **SGBD**-urile de generația întâi (arborescente și rețea) spre cele de generația a doua (relaționale). Această evoluție s-a materializat, în principal în: oferirea de limbaje de interogare neprocedurale, îmbunătățirea integrității și securității datelor, optimizarea și simplificarea acceselor.

Teoria relațională este un ansamblu de concepte, metode și instrumente care a dat o fundamenteare riguroasă realizării de **SGBDR** performante.

**SGBD** relațional este un ansamblu de produse software complex și complet, care implementează modelul logic de date relațional, precum și cel puțin un limbaj de programare relațional.

Elementele necesare evaluării unui **SGBDR** sunt prezentate prin regulile lui Codd.

## 1.4.4 Regulile lui Codd

Edgar F. Codd (cercetător la IBM) a formulat 13 reguli care exprimă cerințele maxime pentru ca un **SGBD** să fie relațional.

Regulile sunt utile pentru evoluarea performanțelor unui **SGBDR**. Acestea sunt:

R<sub>0</sub>. *Gestionarea datelor se face la nivel de relație*: limbajele utilizate trebuie să lucreze cu relații (tabele). Relația trebuie să fie unitatea de informație pentru operații.

R<sub>1</sub>. *Reprezentarea logică a datelor*: toate informațiile din **BDR** trebuie stocate și prelucrate ca relații (tabele).

R<sub>2</sub>. *Garantarea accesului la date*: **LMD** trebuie să permită accesul la fiecare valoare atomică din **BDR** (tabelă, coloană, cheile de diferite tipuri).

R<sub>3</sub>. *Valoarea NULL*: trebuie să se permită mai întâi declararea și apoi prelucrarea valorii de tip NULL ca date lipsă sau inaplicabile. Deoarece valoarea NULL înseamnă date lipsă ea nu poate fi prelucrată în expresii aritmetice, dar se pot face alte prelucrări, se pot aplica alți operatori (de exemplu, operatorul existențial IS și operatorul logic NOT).

R<sub>4</sub>. *Metadatele*: informațiile despre descrierea **BDR** se stochează în dicționar ca tabele, la fel ca datele propriu-zise.

R<sub>5</sub>. *Limbajele utilizate*: **SGBDR** trebuie să permită utilizarea mai multor limbaje, dintre care cel puțin unul să permită definirea tabelor (de bază și virtuale), definirea restricțiilor de integritate (constrângeri), manipularea datelor, autorizarea accesului, tratarea tranzacțiilor.

R<sub>6</sub>. *Actualizarea tabelelor virtuale*: trebuie să se permită ca tabelele virtuale (view-uri) să fie și efectiv actualizabile, nu numai teoretic actualizabile.

R<sub>7</sub>. *Actualizările în baza de date*: manipularea unei tabele trebuie să se facă prin operații de regăsire (interogare) dar și de actualizare.

R<sub>8</sub>. *Independența fizică a datelor*: schimbarea structurii fizice a datelor (modul de reprezentare (organizare) și modul de acces) nu afectează programele.

R<sub>9</sub>. *Independența logică a datelor*: schimbarea structurii de date (logice) a tabelor nu afectează programele.

R<sub>10</sub>. *Restricțiile de integritate*: acestea trebuie să fie definite prin **LDD** și stocate în dicționarul (catalogul) **BDR**.

R<sub>11</sub>. *Distribuirea geografică a datelor*: **LMD** trebuie să permită ca programele de aplicație să fie aceleași atât pentru date distribuite cât și pentru date centralizate (alocarea și localizarea datelor vor fi în sarcina **SGBDR**-ului).

R<sub>12</sub>. *Prelucrarea datelor la nivel de bază (scăzut)*: dacă **SGBDR** posedă un limbaj de nivel scăzut (prelucrarea datelor se face la nivel de înregistrare), acesta nu trebuie utilizat pentru a evita restricțiile de integritate.

Regulile lui Codd sunt greu de îndeplinit în totalitate de către **SGBDR**. Pornind de la cele 13 reguli de mai sus, au fost formulate o serie de criterii (cerințe) pe care trebuie să le îndeplinească un **SGBD** pentru a putea fi considerat relațional într-un anumit grad. S-a ajuns astfel, la mai multe grade de relațional pentru **SGBDR**: cu interfață relațională (toate datele se reprezintă în tabele, există operatorii de selecție, proiecție și joncțiune doar pentru interogare), pseudorelațional (toate datele se reprezintă în tabele, există operatorii de selecție, proiecție și joncțiune fără limitări), minimal relațional (este pseudorelațional și în plus, operațiile cu tabele nu fac apel la pointeri observabili de utilizatori), complet relațional (este minimal relațional și în plus, există operatorii de reuniune, intersecție și diferență, precum și restricțiile de integritate privind unicitatea cheii și restricția referențială).

Cele 13 reguli ale lui Codd pot fi grupate în cinci categorii:

- reguli de bază (fundamentale): R<sub>0</sub> și R<sub>12</sub>;
- reguli structurale: R<sub>1</sub> și R<sub>6</sub>;
- reguli pentru integritatea datelor: R<sub>3</sub> și R<sub>10</sub>;
- reguli pentru manipularea datelor: R<sub>2</sub>, R<sub>4</sub>, R<sub>5</sub> și R<sub>7</sub>;
- reguli pentru independența datelor: R<sub>8</sub>, R<sub>9</sub> și R<sub>11</sub>.

În concluzie, **SGBDR** este un sistem software complet care implementează modelul de date relațional și respectă cerințele impuse de acest model. El este o interfață între utilizatori și baza de date.

## 1.4.5 Exemple de SGBD-uri

**Oracle**. Este realizat de firma Oracle Corporation USA. Sistemul este complet relațional, robust, se bazează pe **SQL** standard extins. Arhitectura sistemului este client/server, permițând lucrul, cu obiecte și distribuit. Sistemul respectă teoria relațională, este robust și se bazează pe **SQL** standard. Permite lucrul distribuit și are modul de optimizare a regăsirii.

**MS SQL Server**. Este realizat de firma Microsoft. Se bazează, ca și alte sisteme de management al bazelor de date relaționale, pe standardul **SQL** și rulează în arhitectura client/server. Este un **SGBD** foarte popular utilizat în multe aplicații, atât aplicații web cât și aplicații desktop.

**MySQL**. Este un **SGBD** produs de compania suedeză MySQL AB și distribuit sub Licența Publică Generală **GNU** (în engleză GNU General Public License, prescurtat **GNU GPL** – este o licență software care are scopul de a da dreptul oricărui utilizator de a copia, modifica și redistribui programe și coduri sursă ale programatorilor care își licențiază operele sub

tutela **GPL**). Ulterior a fost preluat de compania Sun Microsystems, iar apoi a fost cumpărat de compania Oracle, astfel că, în prezent este un produs al companiei Oracle. Este cel mai popular **SGBD** open-source la ora actuală. Pe lângă licența open-source, mai există și licența comercială MySQL Enterprise. Se bazează pe standardul pe **SQL**.

**Visual FoxPro**. Este realizat de firma Microsoft. Are un limbaj procedural propriu foarte puternic, o extensie orientată obiect, programare vizuală și nucleu extins de **SQL**.

**MS Access**. Este realizat de firma Microsoft. Se bazează pe **SQL**, are limbajul procedural gazdă (Basic Access) și instrumente de dezvoltare.

**Modelul unei baze de date** este o specificație tehnică acceptată de mai mulți furnizori de programe de baze de date (**DBMS**) ce se referă la modul în care sunt stocate informațiile în baza de date și modul în care sunt folosite.

Exemple de modele sunt: **modelul relațional**, **modelul orientat-obiect**, **modelul ierarhic**, etc.

Cel mai răspândit în prezent este **modelul relațional**. Bazele de date relaționale au informațiile organizate în tabele, iar între informațiile din aceste tabele pot fi stabilite legături.

## 1.4.6 Colectarea și analizarea datelor.

### Modelul conceptual

Primul pas în realizarea unei aplicații de baze de date este analiza datelor și realizarea unei **scheme conceptuale (model conceptual)** a acestor date.

**Modelul conceptual** al datelor este o reprezentare vizuală clară și corectă a activității unei organizații.

Modelul conceptual al datelor include entitățile (informațiile) majore și relațiile dintre acestea și nu conține informații detaliate privind atributele (caracteristicile) entităților. Este generat prin identificarea cerințelor afacerii modelate, așa cum rezultă din documente, discuții cu personalul implicat, cu analiști și experți ai domeniului studiat, cu beneficiarii finali. Modelul conceptual realizat va fi prezentat echipelor funcționale în vederea revizuirii.

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

Analiza datelor este un proces uneori dificil, care necesită mult timp, însă este o etapă absolut obligatorie. Fără o analiză atentă a datelor și a modului de utilizare a acestora, vom realiza o bază de date care putem constata în final că nu întrunește cerințele beneficiarului. Costurile modificării acestei baze de date sunt mult mai mari decât costurile pe care le-ar fi implicat etapa de analiză și realizare a modelului conceptual. Modificarea modelului conceptual este mult mai ușoară decât modificarea unor tabele deja existente, care eventual conțin și o mulțime de date. Informațiile obținute în etapa documentării vor fi reprezentate într-o formă convențională care să poată fi ușor înțeleasă de toată lumea.

O astfel de reprezentare este diagrama entități-relații numită și harta relațiilor **ERD (Entity Relationship Diagram)**. Aceste scheme sunt un instrument util care ușurează comunicarea dintre specialiștii care proiectează bazele de date și programatori, pe de o parte, și beneficiari, pe de altă parte. Aceștia din urmă pot înțelege cu ușurință o astfel

de schemă, chiar dacă nu sunt cunoscători în domeniul IT. Diagramele **ERD** sunt ușor de creat și de actualizat, însă, avantajul major al lor este dat de simplitatea reprezentărilor ce facilitează înțelegerea și de către nespecialiști.

În concluzie, putem sublinia câteva caracteristici ale **ERD**-urilor:

- sunt un instrument eficient de proiectare;
- sunt o reprezentare grafică a unui sistem de date;
- oferă un model conceptual de înalt nivel al bazelor de date;
- sprijină înțelegerea de către utilizatori a datelor și a relațiilor dintre acestea;
- sunt independente de implementare.

Iată în continuare principalele elemente care intră în componența unui **ERD** precum și convențiile de reprezentare a acestora.

## 1.4.7 Entități. Instanțe. Atribute.

### Identificator unic.

O entitate este un lucru, obiect, persoană sau eveniment care are semnificație pentru afacerea modelată, despre care trebuie să colectăm și să memorăm date. O entitate poate fi un lucru real, tangibil precum o clădire, o persoană, poate fi o activitate precum o programare sau o operație, sau poate fi o noțiune abstractă. Entitatea reprezintă un obiect concret sau abstract care aparține spațiului problemei de rezolvat, are o existență de sine stătătoare, poate fi identificată în raport cu celelalte obiecte.

O entitate este reprezentată în ERD printr-un dreptunghi cu colțurile rotunjite. Numele entității este întotdeauna un substantiv la singular și se scrie în partea de sus a dreptunghiului cu majuscule, ca în figura de mai jos:

Fig.1.1

Entitățile sunt clase de obiecte de același tip, un obiect al clasei reprezentând o instanță a entității.

O instanță a unei entități este un obiect, persoană, eveniment, particular din clasa de obiecte care formează entitatea. De exemplu, elevul X din clasa a IX-a A de la Liceul de Informatică din localitatea Y este o instanță a entității ELEV.

După cum se vede pentru a preciza o instanță a unei entități, trebuie să specificăm unele caracteristici ale acestui obiect, să-l descriem (în cazul entității ELEV precizăm de exemplu numele, clasa, școala etc).

Așadar, după ce am identificat entitățile trebuie să descriem aceste entități în termeni reali, adică să le stabilim atributele.

*Entitățile* sunt descrise folosind **atribute**, fiecare atribut având fie o singură valoare, fie niciuna. Valorile atributelor pot fi de tip numeric (ex. 100; -5; 14.3), șir de caractere (ex. Popescu, XII A), dată calendaristică (ex. 12/02/2010) etc.

Un *atribut* este orice detaliu care servește la identificarea, clasificarea, cuantificarea, sau exprimarea stării unei instanțe a unei entități. Atributele sunt informații specifice ce trebuie cunoscute și memorate.

În cadrul unui **ERD**, attributele se vor scrie imediat sub numele entității, cu litere mici. Un *atribut* este un substantiv la singular.

*Atributele* cărora le poate lipsi valoarea se numesc *attribute opționale*, iar cele cărora trebuie să le atribuim o valoare se numesc *attribute obligatorii*. Dacă un atribut este obligatoriu, pentru fiecare instanță a entității respective trebuie să avem o valoare pentru acel atribut, de exemplu este obligatoriu să cunoaștem numele elevilor.

Pentru un atribut opțional putem avea instanțe pentru care nu cunoaștem valoarea atributului respectiv. De exemplu atributul email al entității ELEV este opțional, un elev putând să nu aibă adresă de email. Un atribut obligatoriu este precedat în **ERD** de un asterisc „\*”, iar un atribut opțional va fi precedat de un cerculeț „o”.

Fig. 1.2

De exemplu attributele entității ELEV sunt nume, prenume, adresa, număr de telefon, adresa de email, data nașterii etc.

Atributele care definesc în mod unic instanțele unei entități se numesc identificator unic (**UID**). **UID**-ul unei entități poate fi compus dintr-un singur atribut, de exemplu codul numeric personal poate fi un identificator unic pentru entitatea ELEV. În alte situații, identificatorul unic este compus dintr-o combinație de două sau mai multe atribute.

De exemplu combinația dintre titlu, numele autorului și data apariției poate forma unicul identificator al entității CARTE. Oare combinația titlu și nume autor nu era suficientă? Răspunsul este nu, deoarece pot exista de exemplu mai multe volume scrise de Mihai Eminescu având toate titlul Poezii, dar apărute la date diferite.

Fig. 1.3

Atributele care fac parte din identificatorul unic al unei entități vor fi precedate de semnul diez „#”. Atributele din **UID** sunt întotdeauna obligatorii, însă semnul „#” este suficient, nu mai trebuie pus și un semn asterisc în fața acestor atribute.

Valorile unor atribute se pot modifica foarte des, ca de exemplu atributul vârstă. Spunem în acest caz că avem de a face cu un atribut volatil. Dacă valoarea unui atribut însă se modifică foarte rar sau deloc (de exemplu data nașterii) acesta este un atribut non-volatil. Evident este de preferat să folosim atribute non-volatile atunci când acest lucru este posibil.

*Atributul sau ansamblul de atribute* ce identifică în mod unic o instanță a entității (valoarea sau combinația de valori ale atributelor este unică pentru fiecare instanță) se numește **identificator unic (Unique Identifier - UID)**.

Identificatorii unici sunt obligatorii și sunt precedați de caracterul # (diez). Se pot folosi identificatori unici *naturali* (atribute ce au o semnificație concretă pentru entitatea respectivă) sau *artificiali* (atribute create și menținute în mod artificial, arbitrar de noi).

Entitatea CANDIDAT este descrisă de unsprezece atribute: id\_candidat, nume, initiala, prenume, cnp, mediul, serii\_anterioare, taxa, adresa, telefon, data\_inscriere.

Identificatorul unic al entității este *id\_candidat*, un **UID** de tip artificial. Un **UID** natural al entității poate fi stabilit atributul *cnp*, întrucât nu există două persoane cu același cod numeric personal, sau o pereche de atribute ce ar asigura unicitatea: perechea (nume, inițiala, prenume, adresa).

În continuare aveți reprezentarea grafică a entității CANDIDAT:

Fig 1.4 Entitatea CANDIDAT

Atributele obligatorii sunt nume, initiala, prenume, cnp, mediul, iar cele opționale serii\_anterioare, taxa, adresa, telefon, data\_inscriere. Se recomandă evitarea folosirii atributelor ale căror valori se modifică frecvent (de exemplu vârsta unei persoane), așa-numitele atribute volatile. Acestea pot fi înlocuite cu atribute non-volatile (de ex. data nașterii, ce este o constantă pentru fiecare persoană).

De asemenea sunt de evitat atributele ale căror valori pot fi deduse prin diferite prelucrări din atribute definite anterior. În entitatea CANDIDAT nu are sens să adăugăm atributele data nașterii, sex, vârstă, întrucât ele pot fi obținute prin prelucrări elementare din atributul cnp.

## 1.4.8 Etapele realizării unei aplicații informatice

Realizarea unei aplicații informatice ce utilizează baze de date necesită parcurgerea unei succesiuni de etape: analiza sistemului, proiectarea bazei de date, realizarea componentelor logice, punerea în funcțiune, dezvoltarea și întreținerea (mentenanța).

### 1.4.8.1 Analiza sistemului informatizat

Scopul analizei sistemului este de a evidenția cerințele aplicației și resursele utilizate, precum și de a evalua aceste cerințe prin modelare (analiza). Studiul situației existente se realizează prin: identificarea caracteristicilor generale ale unității, identificarea activităților desfășurate, identificarea resurselor existente (informaționale, umane, echipamente), identificarea necesităților de prelucrare.

Analiza este o activitate de modelare (conceptuală) și se realizează sub trei aspecte: structural, dinamic și funcțional.

a) *Analiza structurală* evidențiază, la nivel conceptual, modul de structurare a datelor și a legăturilor dintre ele. Cea mai utilizată tehnică este entitate-asociere. Aceasta conține:

- Identificarea entităților: fenomene, procese, obiecte concrete sau abstracte;
- Identificarea asocierilor dintre entități ca fiind legăturile semnificative de un anumit tip;
- Identificarea atributelor ce caracterizează fiecare entitate în parte;
- Stabilirea atributelor de identificare unică a instanțelor entităților;
- Identificarea și eliminarea anomaliilor de date și minimizarea redundanței datelor prin procesul de normalizare a entităților.

Rezultatul analizei structurale este modelul *conceptual static* al datelor, numit și diagrama **ERD** – **E**ntity **R**elationship **D**igram. Pornind de la o astfel de diagramă, se pot construi, în activitatea de proiectare, schemele relațiilor (tabelelor).

b) *Analiza dinamică* evidențiază comportamentul elementelor sistemului la anumite evenimente. Una din tehnicile utilizate este diagrama stare-tranziție. Aceasta presupune:

- Identificarea stărilor în care se pot afla componentele sistemului;
- Identificarea evenimentelor care determină trecerea unei componente dintr-o stare în alta;
- Stabilirea tranzițiilor admise între stări;
- Construirea diagramei stare-tranziție.

Rezultatul analizei dinamice este *modelul dinamic*.

c) *Analiza funcțională* evidențiază modul de asigurare a cerințelor informaționale (fluxul prelucrărilor) din cadrul sistemului, prin care intrările sunt transformate în ieșiri. Prin analiza funcțională se delimitează:

- Aria de cuprindere a aplicației informatice;
- Se identifică sursele de date;
- Se identifică modul de circulație și prelucrare a datelor;
- Se identifică apoi rezultatele obținute.

Rezultatul analizei funcționale este *modelul funcțional*

## 1.4.9 Modelul relațional

**Modelul relațional** a fost propus de către IBM și a revoluționat reprezentarea datelor făcând trecerea la generația a doua de baze de date. Modelul este simplu, are o solidă fundamentare teoretică fiind bazat pe teoria seturilor (ansamblurilor) și pe logica matematică. Pot fi reprezentate toate tipurile de structuri de date de mare complexitate, din diferite domenii de activitate.

*Modelul relațional* este definit prin: structura de date, operatorii care acționează asupra structurii și restricțiile de integritate.

Conceptele utilizate pentru definirea structurii de date

sunt: domeniul, tabela (relația), atributul, tuplul, cheia și schema tabelului.

*Domeniul* este un ansamblu de valori caracterizat printr-un nume. El poate fi explicit sau implicit. *Domeniul* reprezintă o mulțime de tip similar (de exemplu: luni calendaristice, orașe, etc.).

*Tabela/relația* este un subansamblu al produsului cartezian al mai multor domenii, caracterizat printr-un nume, prin care se definesc atributele ce aparțin aceleiași clase de entități. Mai simplu spus, *relația* sau *tabela* reprezintă o mulțime de atribute. De obicei, relațiile se exprimă grafic sub formă de tabele în care distingem: *gradul relației*, adică *numărul de atribute* (coloane) folosite în relație și *cardinalitatea relației*, care reprezintă *numărul de tupluri* (înregistrări). Cardinalitatea relației este variabilă în timp datorită operațiilor de adăugare, ștergere.

*Atributul* este coloana unei tabele, caracterizată printr-un nume. *Atributul* reprezintă o submulțime a unui domeniu căreia i s-a atribuit un nume. Numele atributului (coloanei) reprezintă rolul sau semnificația atribuită elementelor subdomeniului respectiv (de exemplu: din domeniul localități pot fi definite atributele aeroport-plecare, aeroport-sosire, aeroport-escală, etc.)

*Cheia* este un atribut sau un ansamblu de atribute care au rolul de a identifica un tuplu dintr-o tabelă. Tipuri de chei: *primare*, *externe*.



*Tuplul* este linia dintr-o *tabelă* și nu are nume. Ordinea liniilor (tupluri) și coloanelor (atribute) dintr-o tabelă nu trebuie să prezinte nici o importanță.

*Schema tabelii* este formată din numele tabelii, urmat între paranteze rotunde de lista atributelor, iar pentru fiecare atribut se precizează domeniul asociat.

*Schema bazei de date* poate fi reprezentată printr-o diagramă de structură în care sunt puse în evidență și legăturile dintre tabele. Definirea legăturilor dintre tabele sau a relațiilor dintre tabele se face logic construind asocieri între tabele cu ajutorul unor atribute de legătură.

*Cardinalitatea relației* este egală cu numărul de linii sau tupluri conținute de un tabel.

*Cheia primară* a unei tabeli reprezintă un *atribut* sau un *ansamblu de atribute* care *identifică în mod unic* o înregistrare dintr-o tabelă a unei baze de date.

*Cheia primară* formată dintr-un singur atribut este o *cheie simplă*, iar cea formată dintr-un ansamblu de atribute se numește *cheie compusă*. Orice tabelă are o cheie primară care trebuie să aibă valori unice și nenule.

Deci, două înregistrări (linii) nu pot avea aceeași valoare a cheii primare, fiecare înregistrare trebuie să aibă o valoare în câmpul (coloana) care este cheie primară. Coloana care conține valorile cheilor primare nu poate fi modificată sau actualizată. Valorile cheilor primare nu pot fi refolosite, dacă o înregistrare este ștearsă din tabelă cheia ei nu va fi atribuită altor înregistrări noi.

Dacă avem o *cheie primară compusă* atunci regulile enunțate mai sus se aplică asupra ansamblului de atribute ce alcătuiesc cheia primară luată laolaltă.

Un *atribut* ce îndeplinește condițiile necesare cheii primare se numește *cheie candidat*. Într-o entitate pot exista mai multe atribute ce pot fi cheie primară. Dintre aceste chei candidat se va alege, de fapt, *cheia primară*.

Atributele implicate în realizarea legăturilor se găsesc fie în tabelele asociate, fie în tabele distincte construite special pentru legături. Atributul din tabela inițială se numește *cheie externă* iar cel din tabela finală este *cheie primară*.

*Cheia externă* (foreign key) este atributul sau ansamblul de atribute ce servește la realizarea legăturii cu o altă tabelă în care acest atribut sau ansamblu de atribute este cheie primară. Valorile asociate atributului cu rol de cheie externă pot fi duplicate sau nule.

#### **Condițiile pe care trebuie să le îndeplinească cheile externe sunt următoarele:**

- fiecare valoare a unei *chei externe* trebuie să se regăsească printre mulțimea valorilor *cheii primare corespondente*, reciproca nu este valabilă;
- o *cheie externă* este simplă dacă și numai dacă *cheia primară* corespondentă este simplă și este compusă dacă și numai dacă *cheia primară* corespondentă este compusă;
- fiecare atribut component al unei *chei externe* trebuie să fie definit pe același domeniu al componentei corespondente din *cheia primară*;
- o valoare a unei *chei externe* reprezintă o referință către o înregistrare care conține aceeași valoare pentru *cheia primară corespondentă*, deci această valoare trebuie să existe.

Această ultimă condiție este cea a *integrității referinței*, deci, dacă avem o cheie externă A care face referire la o cheie primară B, atunci B trebuie să existe. *Constrângerile de integritate* reprezintă reguli pe care valorile conținute într-o tabelă trebuie să le respecte. Aceste constrângeri previn introducerea de date eronate în tabele.

Deci *cheia externă* este o *constrângere*, prin aplicarea acestei constrângeri valorile unei coloane sunt forțate să fie doar dintre cele ale cheii primare corespondente. Aceasta poartă numele de *constrângere de integritate referențială*.

*Constrângerile de integritate* sunt verificate automat de **SGBD** atunci când au loc operații de modificare a conținutului unei tabele (introducere, modificare sau ștergere de înregistrări). În cazul în care valorile nu sunt valide operația nu se efectuează și se generează o eroare.

Fiecare constrângere de integritate poate avea asociat un nume, în cazul în care nu se asociază un nume, sistemul generează automat unul. O constrângere poate fi definită în descrierea unei coloane dacă se referă doar la acea coloană sau la finalul listei de descrieri a coloanelor.

Tipuri de constrângeri:

- **NOT NULL** – valorile nu pot fi nule;
- **PRIMARY KEY** – definește cheia primară;
- **UNIQUE** – definește uncitatea;
- **FOREIGN KEY** – definește o cheie externă;
- **CHECK** – introduce o condiție (expresie logică).

Avem următoarele legături posibile între tabele:

- *unu-la-unu* (*one-to-one*, 1:1);
- *unu-la-mai mulți* (*one-to-many*, 1:m);
- *mai-mulți-la-mai-mulți* (*many-to-many*, m:n).

Potențial, orice tabelă se poate lega cu orice tabelă, după orice atribut. Legăturile se stabilesc la momentul descrierii datelor prin limbaje de descriere a datelor (**LDD**), cu ajutorul restricțiilor de integritate. Practic, se stabilesc și legături dinamice la momentul execuției.

## 1.4.10 Operatorii modelului relațional

Operatorii modelului relațional sunt operatorii din *algebra relațională* și operatorii din *calculul relațional*.

*Algebra relațională* este o colecție de operații formale aplicate asupra tabelelor (relațiilor), și a fost concepută de E.F.Codd. Operațiile sunt aplicate în expresiile algebrice relaționale care sunt cereri de regăsire. Acestea sunt compuse din operatorii relaționali și operanzi.

Operanzii sunt întotdeauna tabele (una sau mai multe). Rezultatul evaluării unei expresii relaționale este format dintr-o singură tabelă.

Algebra relațională are cel puțin puterea de regăsire a calcului relațional. O expresie din calculul relațional se poate transforma într-una echivalentă din algebra relațională și invers. Codd a introdus șase operatori de bază (reuniunea, diferența, produsul cartezian, selecția, proiecția, joncțiunea) și doi operatori derivați (intersecția și diviziunea). Ulterior au fost introduși și alți operatori derivați (speciali).

În acest context, **operatorii** din algebra relațională pot fi grupați în două categorii: **pe mulțimi** și **speciali**.

*Operatori pe mulțimi* ( $R_1$ ,  $R_2$ ,  $R_3$  sunt relații (tabele)) sunt:

- **Reuniunea**.  $R_3 = R_1 \cup R_2$ , unde  $R_3$  va conține tupluri din  $R_1$  sau  $R_2$  luate o singură dată;
- **Diferența**.  $R_3 = R_1 \setminus R_2$ , unde  $R_3$  va conține tupluri din  $R_1$  care nu se regăsesc în  $R_2$ ;
- **Produsul cartezian**.  $R_3 = R_1 \times R_2$ , unde  $R_3$  va conține tupluri construite din perechi ( $x_1, x_2$ ), cu  $x_1$  aparținând  $R_1$  și  $x_2$  aparținând  $R_2$ ;

- **Intersecția.**  $R3 = R1 \cap R2$ , unde R3 va conține tupluri care se găsesc în R1 și R2 în același timp;

*Operatori relaționali speciali* sunt:

- **Selecția.** Din R1 se obține o subtabelă R2, care va conține o submulțime din tuplurile inițiale din R1 ce satisfac o condiție. Numărul de atribute din R2 este egal cu numărul de atribute din R1. Numărul de tupluri din R2 este mai mic decât numărul de tupluri din R1.
- **Proiecția.** Din R1 se obține o subtabelă R2, care va conține o submulțime din atributele inițiale din R1 și fără tupluri duplicate. Numărul de atribute din R2 este mai mic decât numărul de atribute din R1.
- **Joncțiunea** este o derivație a produsului cartezian, ce presupune utilizarea unui calificativ care să permită compararea valorilor unor atribute din R1 și R2, iar rezultatul în R3. R1 și R2 trebuie să aibă unul sau mai multe atribute comune care au valori comune.

*Proiectarea schemei conceptuale* pornește de la identificarea setului de date necesar sistemului. Aceste date sunt apoi integrate și structurate într-o schemă (exemplu: pentru **BDR** relaționale cea mai utilizată tehnică este *normalizarea*).

## 1.4.11 Normalizarea

Tehnica de *normalizare* este utilizată în activitatea de proiectare a structurii **BDR** și constă în eliminarea unor anomalii (neajunsuri) de actualizare din structură.

Anomaliile de actualizare sunt situații nedorite care pot fi generate de anumite tabele în procesul proiectării lor:

- *Anomalia de ștergere* semnifică faptul că ștergând un tuplu dintr-o tabelă, pe lângă informațiile care trebuie șterse, se pierd și informațiile utile existente în tuplul respectiv;
- *Anomalia de adăugare* semnifică faptul că nu pot fi incluse noi informații necesare într-o tabelă, deoarece nu se cunosc și alte informații utile (de exemplu valorile pentru cheie);
- *Anomalia de modificare* semnifică faptul că este dificil de modificat o valoare a unui atribut atunci când ea apare în mai multe tupluri.

*Normalizarea* este o teorie construită în jurul conceptului de **forme normale (FN)** sau **normal form (NF)**, care ameliorează structura **BDR** prin înlăturarea treptată a unor neajunsuri și prin imprimarea unor facilități sporite privind manipularea datelor.

Normalizarea utilizează ca metodă descompunerea (top-down) unei tabele în două sau mai multe tabele, păstrând informații (atribute) de legătură.

În practică, în cele mai multe situații, tehnica normalizării este utilizată până când tabelele unei baze de date sunt aduse la **FN3**. Aceasta este o formă care elimină o parte semnificativă din anomalii (cele mai mari) și, în general, se urmărește aducerea tabelor la această formă normală.

### 1.4.11.1 Forma Normală 1 (FN1 sau 1NF)

O tabelă este în **FN1** dacă toate atributele ei conțin valori elementare (nedecompozabile), adică fiecare tuplu nu trebuie să aibă date la nivel de grup sau repetitiv. Structurile de tip arborescent și rețea se transformă în tabele cu atribute elementare.

O tabelă în **FN1** prezintă încă o serie de anomalii de actualizare datorită eventualelor dependențe funcționale incomplete. Fiecare structură repetitivă generează (prin descompunere) o nouă tabelă, iar atributele la nivel de grup se înlătură, rămânând doar cele elementare.

## 1.4.11.2 Forma Normală 2 (FN2 sau 2NF)

O tabelă este în **FN2** dacă și numai dacă este în **FN1** și fiecare atribut noncheie al tabelii este dependent funcțional complet de cheie. Un atribut B al unei tabelii depinde funcțional de atributul A al aceleiași tabelii, dacă fiecărei valori a lui A îi corespunde o singură valoare a lui B, care îi este asociată în tabelă. Un atribut B este dependent funcțional complet de un ansamblu de atribute A în cadrul aceleiași tabelii, dacă B este dependent funcțional de întreg ansamblul A (nu numai de un atribut din ansamblu).

O tabelă în **FN2** prezintă încă o serie de anomalii de actualizare, datorită eventualelor dependențe tranzitive. Eliminarea dependențelor incomplete se face prin descompunerea tabelii inițiale în două tabele, ambele conținând atributul intermediar (B).

O tabelă aflată în **FN1** și care are cheia primară simplă, adică formată dintr-o singură coloană, este și în **FN2**. Astfel, trebuie urmărită aducerea la **FN2** a tabelilor care se află în **FN1** și au cheia primară compusă (formată dintr-un ansamblu de două sau mai multe coloane).

## 1.4.11.3 Forma Normală 3 (FN3 sau 3NF)

O tabelă este în **FN3** dacă și numai dacă este în **FN2** și fiecare atribut noncheie depinde în mod netranzitiv de cheia tabelii. Într-o tabelă T, fie A,B,C trei atribute cu A cheie. Dacă B depinde de A ( $A \rightarrow B$ ) și C depinde de B ( $B \rightarrow C$ ) atunci C depinde de A în mod tranzitiv.

Eliminarea dependențelor tranzitive se face prin descompunerea tabelii inițiale în două tabele, ambele conținând atributul intermediar (B). O tabelă în **FN3** prezintă încă o serie de anomalii de actualizare, datorate eventualelor dependențe multivaloare.

## 1.4.11.4 Forma Normală Boyce Codd (FNBC sau BCNF)

O definiție mai riguroasă pentru **FN3** a fost dată prin forma intermediară **BCNF** (Boyce Codd Normal Form): o tabelă este în **BCNF** dacă fiecare determinant este un candidat cheie. Determinantul este un atribut elementar sau compus față de care alte atribute sunt complet dependente funcțional.

## 1.4.11.5 Forma Normală 4 (FN4 sau 4NF)

O tabelă este în **FN4** dacă și numai dacă este în **FN3** și nu conține două sau mai multe dependențe multivaloare. Într-o tabelă T, fie A,B,C trei atribute. În tabela T se menține dependența multivaloare A dacă și numai dacă mulțimea valorilor lui B ce corespunde unei perechi de date (A,C), depinde numai de o valoare a lui A și este independentă de valorile lui C.

## 1.4.11.6 Forma Normală 5 (FN5 sau 5NF)

O tabelă este în **FN5** dacă și numai dacă este în **FN4** și fiecare dependență joncțiune este generată printr-un candidat cheie al tabelii. În tabela T (A,B,C) se menține dependența

joncțiune (AB, AC) dacă și numai dacă T menține dependența multivaloare  $A \twoheadrightarrow B$  sau C. Dependența multivaloare este caz particular al dependenței joncțiune. Dependența funcțională este caz particular al dependenței multivaloare.

## 1.4.12 Concluzii

În concluzie, în această primă lecție au fost prezentate și explicate mai multe noțiuni teoretice fundamentale pentru a înțelege de ce folosim baze de date relaționale și ce noțiuni sunt întâlnite frecvent în lucrul cu baze de date relaționale. Pe lângă definirea unor noțiuni de bază, a fost abordat și subiectul normalizării unei baze de date, precum și prezentarea modelului relațional.

Lecția următoare se va axa, în continuare, pe prezentarea unor noțiuni teoretice legate de tipurile de relații întâlnite între tabele, dar va conține și exemple concrete de proiectare a bazelor de date, precum și o prezentare a mediilor de lucru folosite pentru a avea instalat un server de baze de date **MySQL** și pentru conectarea la baza de date, respectiv, efectuarea de comenzi **SQL**.

## 2. Design-ul bazei de date

### 2.1 Proiectarea bazei de date relaționale

Proiectarea bazelor de date se referă la fixarea structurii bazei de date și a metodelor de prelucrarea datelor, spre deosebire de utilizarea bazei de date, care se referă la informațiile stocate în baza de date.

Dacă o bază de date își schimbă frecvent conținutul, structura ei rămâne neschimbată pentru o perioadă lungă de timp.

Proiectarea unei baze de date urmărește obținerea următoarelor calități:

- **Corectitudine** – reprezentarea cât mai fidelă în baza de date a modului obișnuit de lucru cu datele în sistemul real;
- **Consistență** – informațiile corespunzătoare obiectelor cu care se lucrează în baza de date (nume, definire, relații) să nu conțină contradicții;
- **Distribuire** – informațiile să poată fi utilizate de aplicații multiple și să poată fi accesate de mai mulți utilizatori, aflați în diferite locuri, utilizând medii diverse
- **Flexibilitate** – facilități de adăugare de componente care să reflecte cereri noi de informații, să îmbunătățească performanțele sau să adapteze datele pentru eventuale modificări.

Un **model relațional** de baze de date cuprinde trei componente principale:

- Structura datelor prin definirea unor domenii și a relațiilor (atribute, tupluri, chei primare);
- Integritatea datelor prin impunerea unor restricții;
- Prelucrarea datelor prin operații din algebra relațională sau calculul relațional.

**Modelul relațional** se bazează pe noțiunea matematică de relație așa cum este ea definită în teoria mulțimilor, și anume ca o submulțime a produsului cartezian a unei liste finite de mulțimi numite domenii.

**Algebra relațională** constă dintr-o colecție de operatori ce au ca operanzi relații. Rezultatul aplicării unui operator la una sau două relații este tot o relație.

Noțiunile de model relațional și algebră relațională au fost discutate în primul capitol al acestui curs.

Proiectarea structurii bazei de date relaționale (**BDR**) se face pe baza modelelor realizate în activitatea de analiză. Înainte de proiectarea bazei de date se alege tipul de sistem de gestiune a bazei de date (**SGBD**). Alegerea **SGBD**-ului se face ținând cont de două aspecte: cerințele aplicației și performanțele tehnice ale **SGBD**-ului.

Cerințele aplicației se referă la: volumul de date estimat a fi memorat și prelucrat în **BDR**; complexitatea problemei de rezolvat; ponderea și frecvența operațiilor de intrare/ieșire;

condițiile privind protecția datelor; operațiile necesare (încărcare/validare, actualizare, regăsire etc.); particularitățile activității pentru care se realizează baza de date.

Performanțele tehnice ale **SGBD**-ului se referă la: modelul de date pe care-l implementează; ponderea utilizării **SGBD**-ului pe piață și tendința; configurația de calcul minimă cerută; limbajele de programare din **SGBD**; facilitățile de utilizare oferite pentru diferite categorii de utilizatori; limitele **SGBD**-ului; optimizările realizate de **SGBD**; facilitățile tehnice; lucrul cu mediul distribuit și concurența de date; elementele multimedia; posibilitatea de autodocumentare; instrumentele specifice oferite.

Proiectarea **bazelor de date relaționale (BDR)** se realizează prin proiectarea schemelor **BDR** și proiectarea modulelor funcționale specializate.

Schemele bazei de date sunt: *conceptuală*, *externă* și *internă*.

a) *Proiectarea schemei conceptuale* pornește de la identificarea setului de date necesar sistemului. Aceste date sunt apoi integrate și structurate într-o schemă a bazei de date.

Pentru acest lucru se parcurg următorii pași:

- Stabilirea schemei conceptuale inițiale rezultă din schema entitate-relații **ERD**. Pentru acest lucru, fiecare entitate din modelul conceptual este transformată (mapată) într-o colecție de date (tabel memorat în fișier), iar pentru fiecare relație se definesc cheile aferente.
- Ameliorarea progresivă a schemei conceptuale prin eventuale adăugări de tabele suport suplimentare, prin eliminarea unor anomalii.

b) *Proiectarea schemei externe* are rolul de a specifica vederile fiecărui utilizator asupra **BDR**. Pentru acest lucru, din schema conceptuală se identifică datele necesare fiecărei vederi. Datele obținute se structurează logic în subscheme ținând cont de facilitățile de utilizare și de cerințele utilizator. Schema externă devine operațională prin definirea unor vederi (view-uri) în **SGBD**-ul ales și acordarea drepturilor de acces. Datele dintr-o vedere pot proveni din una sau mai multe colecții și nu ocupă spațiul fizic.

c) *Proiectarea schemei interne* presupune stabilirea structurilor de memorare fizică a datelor și definirea căilor de acces la date. Acestea sunt specifice fie **SGBD**-ului (scheme de alocare), fie sistemului de operare. Proiectarea schemei interne înseamnă estimarea spațiului fizic pentru **BDR**, definirea unui model fizic de alocare (a se vedea dacă **SGBD**-ul permite explicit acest lucru) și definirea unor indecși pentru accesul direct, după cheie, la date.

Proiectarea modulelor funcționale ține cont de concepția generală a **BDR**, precum și de schemele proiectate anterior. În acest sens, se proiectează fluxul informațional, modulele de încărcare și de manipulare a datelor, interfețele specializate, integrarea elementelor proiectate cu organizarea și funcționarea **BDR**.

## 2.2 Realizarea componentelor logice

Componentele logice ale unei baze de date (**BD**) sunt programele de aplicație dezvoltate, în cea mai mare parte, în **SGBD**-ul ales. Programele se realizează conform modulelor funcționale proiectate în etapa anterioară.

Componentele logice țin cont de ieșiri, intrări, prelucrări și colecțiile de date. În paralel cu dezvoltarea programelor de aplicații se întocmesc și documentațiile diferite (tehnică, de exploatare, de prezentare).

## 2.3 Punerea în funcțiune și exploatarea

Se testează funcțiile **BDR** mai întâi cu date de test, apoi cu date reale. Se încarcă datele în **BDR** și se efectuează procedurile de manipulare, de către beneficiar cu asistența proiectantului. Se definitivează documentațiile aplicației. Se intră în exploatare curentă de către beneficiar conform documentației.

## 2.4 Întreținerea și dezvoltarea sistemului

Ulterior punerii în exploatare a **BDR**, în mod continuu, pot exista factori perturbatori care generează schimbări în **BDR**. Factorii pot fi: organizatorici, datorati progresului tehnic, rezultați din cerințele noi ale beneficiarului, din schimbarea metodologiilor, etc.

## 2.5 Relații. Tipuri de relații între tabele

În bazele de date relaționale una dintre cele mai importante noțiuni este cea de relație. Tabelele unei baze de date sunt relaționate între ele. Într-o bază de date relațională nu este indicat să avem tabele izolate, dar nu este nici interzis, pot exista situații în care anumite tabele să rămână izolate.

Există trei tipuri de relații posibile între tabelele unei baze de date:

- *unu-la-unu* sau *one-to-one* ( $1:1$ ) – unei înregistrări din prima tabelă îi corespunde o singură înregistrare în cealaltă tabelă;
- *unu-la-mai-mult* sau *one-to-many* ( $1:n$ ) – unei înregistrări din prima tabelă îi corespund mai multe înregistrări în cealaltă tabelă;
- *mai-mult-la-mai-mult* sau *many-to-many* ( $m:n$ ) – unei înregistrări din prima tabelă îi corespund una sau mai multe înregistrări din cealaltă tabelă și reciproc.

Primul caz, relația *one-to-one* este mai puțin utilizată în cazuri concrete. Un exemplu ar fi o tabelă în care avem persoane și o tabelă cu acte de identitate (o persoană are un singur act de identitate sau o persoană are o singură adresă de domiciliu).

Relația *one-to-many* este foarte răspândită (de exemplu, dacă avem o tabelă de clienți și una de facturi – un client poate să aibă mai multe facturi sau dacă avem o tabelă cu elevi și una cu clase atunci o clasă poate să aibă mai mulți elevi).

Relația *many-to-many* este o relație în care avem nevoie de o tabelă intermediară de legătură între cele două tabele (practic relația *many-to-many* se descompune în două relații *one-to-many*).



Un exemplu ar putea fi următorul: dacă într-o tabelă avem informații despre studenții unei facultăți iar într-o altă tabelă informații despre materiile (cursurile) disponibile în cadrul acelei facultăți avem următorul tip de relație între aceste 2 tabele: un student participă la mai multe cursuri iar un curs este frecventat de mai mulți studenți, rezultă că avem de-a face cu o relație *many-to-many* între cele 2 tabele.

Această relație se descompune în două relații de tip *one-to-many*, prin introducerea unei tabele suplimentare care păstrează informații de identificare pentru studenți și pentru cursurile pe care ei le frecventează.

Înțelegerea modului de relaționare al tabelelor dintr-o bază de date reprezintă un pas fundamental pentru a putea construi o bază de date optimă atunci când proiectăm o aplicație.

## 2.5.1 Exemplu

Prezentăm în continuare diagrama **entitate-relație (ERD)** corespunzătoare unei baze de date în care avem stocate informații despre candidații la un examen. Sunt reprezentate în această diagramă entitățile, attributele (proprietățile, caracteristicile) fiecărei entități, precum și relațiile existente între aceste entități.

Urmează, ulterior, o prezentare detaliată a fiecărei entități care este modelată în această diagramă.

Figura 2.1. Diagrama ERD inițială

Tabelele de mai jos prezintă detaliat entitățile modelate în diagramă:

Entitatea: <b>CALITATE</b>				
Atribut	Tipul de date folosit	Atribut obligatoriu	Identificator unic	Semnificația
id_calitate	Numeric	Da	Da	ID-ul ce este asociat calității
denumire	Șir de caractere	Da	-	Denumirea calității deținută de profesor în comisie: evaluator, secretar, președinte, etc
atributii	Șir de caractere	-	-	Atribuțiile ce decurg din calitatea deținută: ex: evaluarea elevilor, organizarea examenului etc.

Entitatea: <b>PROFESOR</b>				
Atribut	Tipul de date folosit	Atribut obligatoriu	Identificator unic	Semnificația

id_profesor	Numeric	Da	Da	Id-ul asociat profesorului (ex. 100)
nume	Șir de caractere	Da	-	Numele profesorului
prenume	Șir de caractere	Da	-	Prenumele profesorului
gradul	Șir de caractere	-		Gradul didactic al profesorului: definitiv, gradul II, gradul I

Entitatea: <b>SUBCOMISIE</b>				
<i>Atribut</i>	<i>Tipul de date folosit</i>	<i>Atribut obligatoriu</i>	<i>Identificator unic</i>	<i>Semnificația</i>
id_subcomisie	Numeric	Da	Da	Id-ul asociat comisiei (ex. 100)
denumire	Șir de caractere	Da	-	Denumirea comisiei (ex. Comisia 1, Comisia 2 etc.)

Entitatea: <b>CLASA</b>				
<i>Atribut</i>	<i>Tipul de date folosit</i>	<i>Atribut obligatoriu</i>	<i>Identificator unic</i>	<i>Semnificația</i>
id_clasa	Numeric	Da	Da	Id-ul asociat clasei (ex. 100)
denumire	Șir de caractere	Da	-	Denumirea clasei (ex. XII A, XII B etc.)

## 2.5.2 Relații între entități

Diagrama **ERD** evidențiază și relațiile existente între entitățile modelate. O relație între două entități arată că există o dependență, o legătură naturală între conceptele reprezentate de aceste entități. Este evidentă relația dintre entitățile CANDIDAT și CLASĂ : un candidat este un elev ce este înregistrat într-o anumită clasă, iar o clasă este formată din mai mulți elevi.

Relația dintre entitatea A și entitatea B se definește prin:

- denumirea relației: un verb ce sugerează dependența dintre cele două entități
- opționalitatea relației: este necesar să stabilim dacă *trebuie* sau *poate* să existe corespondență între cele două entități
- cardinalitatea relației: precizează numărul de instanțe ale entității B ce sunt puse în corespondență cu o instanță a entității A.

Relația dintre două entități este bidirecțională, dar nu simetrică: dacă există o relație între A și B, există și o relație între B și A, dar nu aceeași.

## 2.6 Convenții de reprezentare a relațiilor

1. Linia ce unește entitățile relaționate e formată din două segmente distincte. Tipul liniei ce pleacă de la entitatea A către entitatea B relevă opționalitatea relației A→B: dacă linia este continuă, relația este obligatorie – „trebuie”, iar dacă este discontinuă, relația este opțională – „poate”.
2. Denumirea relației A→B este poziționată lângă entitatea A, deasupra sau dedesubtul liniei de opționalitate.
3. Cardinalitatea relației A→B se reprezintă astfel: linia de A la B se termină cu o linie simplă, în cazul în care o instanță a entității A este pusă în corespondență cu o singură instanță a entității B, și are forma unui „picior de cioară” (o ramificație în 3 a liniei de relaționare) în cazul în care o instanță a entității A este pusă în corespondență cu mai multe instanțe ale entității B.

### Exemplu:

Figura 2.2 Relația dintre entitățile CLASA și CANDIDAT

Relația CLASA→ CANDIDAT

- denumirea: **are**
- opționalitatea: **trebuie** (segmentul ce pleacă dinspre CLASA este continuu)
- cardinalitatea: **una sau mai mulți** (linia relației se termină cu ramificația în 3)

Relația CLASA→ CANDIDAT se citește: „O CLASA trebuie să aibă unul sau mai mulți CANDIDATI”.

Relația CANDIDAT→ CLASA

- denumirea: **aparține**
- opționalitatea: **poate** (e posibil să existe candidați din seriile anterioare care să nu mai aparțină vreunei clase)
- cardinalitatea: **una și numai una**

Relația CANDIDAT →CLASA se citește: „Un CANDIDAT poate aparține unei singure CLASE”.

O clasificare a relațiilor dintre entitățile A și B folosește cardinalitatea relațiilor A →B și B →A.

Cardinalitate A →B	Cardinalitatea B →A	Tipul relației
una și numai una	una și numai una	„one to one” (1:1)
una și numai una	una sau mai multe	„one to many” (1:M)
una sau mai multe	una și numai una	„one to many” (1:M)
una sau mai multe	una sau mai multe	„many to many” (M:M)

Relația dintre entitățile CLASA și CANDIDAT este de tipul „one-to-many”.

Entitățile relaționate	Citirea relației	Tipul relației
PROFESOR, CALITATE	Un PROFESOR trebuie să aibă o singură CALITATE.	One to many

	O CALITATE poate fi deținută de unul sau mai mulți PROFESORI.	
SUBCOMISIE, PROFESOR	O SUBCOMISIE trebuie să fie formată din unul sau mai mulți PROFESORI. Un PROFESOR poate fi alocat unei singure SUBCOMISII	One to many
SUBCOMISIE, CLASA	O SUBCOMISIE trebuie să fie formată pentru una sau mai multe CLASE. O CLASA trebuie să fie repartizată unei singure SUBCOMISII.	One to many
CLASA, CANDIDAT	O CLASA trebuie să aibă unul sau mai mulți CANDIDATI.  Un CANDIDAT poate aparține unei singure CLASE.	One to many
CLASA, SPECIALIZARE	O CLASA trebuie să aibă o singură SPECIALIZARE. O SPECIALIZARE poate fi urmată de una sau mai multe CLASE.	One to many
CANDIDAT, PROIECT	Un CANDIDAT trebuie să dezvolte un singur PROIECT. Un PROIECT poate fi dezvoltat de unul sau mai mulți CANDIDAȚI.	One to many
CANDIDAT, PROBA	Un CANDIDAT poate susține una sau mai multe PROBE. O PROBĂ poate fi susținută de unul sau mai mulți CANDIDAȚI.	Many to many

Se observă relația de tip *many-to-many* dintre entitățile CANDIDAT și PROBA. Relațiile de acest tip nu pot fi implementate în practică în niciun sistem de gestiune a bazelor de date.

O relație *many to many* dintre entitatea A și entitatea B este descompusă prin adăugarea unei noi entități C denumită *entitate de intersecție* și construirea de relații *one to many* între noua entitate C și entitățile inițiale A și B.

Entității de intersecție i se poate atribui o denumire naturală, ce are o semnificație concretă în legătură cu entitățile inițiale, sau, în lipsa acesteia, o denumire artificială care să combine denumirile entităților inițiale.

În cazul nostru, entitatea de intersecție este denumită NOTA, deoarece legătura între un candidat și o probă a examenului este nota obținută de acesta la acea probă. O denumire artificială ce putea fi folosită este PROBA\_CANDIDAT, însă putem recunoaște că nu este cea mai fericită alegere.

*Figura 2.3. Descompunerea relațiilor „many to many”*

Se observă că se păstrează vechile opționalități în partea dinspre entitățile originale, iar relațiile care pleacă din entitatea de intersecție sunt întotdeauna obligatorii în această parte.

Noile relații sunt de tip *one-to-many*, partea cu many - „piciorul de cioară” (ramificația în 3) fiind întotdeauna înspre entitatea de intersecție.

Se pune problema alegerii identificatorului unic (**UID**) pentru entitatea de intersecție.

Există două posibilități:

- 1) Să se creeze un nou **identificator unic artificial**, de ex: *id\_nota*, atribut numeric. Valorile acestui identificator unic nu au o semnificație concretă pentru o instanță a acestei entități (o notă), singura preocupare fiind asigurarea unicității (evitarea duplicatelor). Această metodă este preferată deseori pentru simplitatea implementării, în condițiile utilizării secvențelor de numere generate automat (*sequences*).
- 2) Să se construiască un **identificator unic compus** care să includă identificatorii unici ai entităților relaționate, la care eventual să se adauge atribute proprii entității de intersecție. În acest caz relațiile cu entitățile de la care s-au utilizat identificatori unici în construcția UID-ului entității de intersecție trebuie barate către entitatea de intersecție, iar atributul propriu ce a fost inclus în **UID**-ul compus trebuie precedat de caracterul „#”.

Se observă că **UID**-ul entității NOTA este unul compus, format din **UID**-ul *id\_candidat* preluat de la entitatea CANDIDAT, **UID**-ul *id\_proba* preluat de la entitatea PROBA și atributul propriu *numar\_bilet*. Includerea atributului propriu *numar\_bilet* în **UID** ar fi necesară doar în eventualitatea în care un candidat ar putea schimba biletul cu subiecte primit la o anumită probă și se dorește înregistrarea acestei situații, altfel atributele împrumutate în **UID** de la entitățile relaționate ar fi suficiente pentru unicitate.

*Figura 2.4. Definirea UID-ului unic compus al entității NOTA*

Diagrama **ERD** în forma actuală permite înregistrarea unei note sau a mai multor note obținute de un candidat la o anumită probă, însă implică discuții: dacă se înregistrează o singură notă obținută de un candidat la o anumită probă, atunci această notă trebuie să fie media aritmetică a notelor acordate de cei doi profesori evaluatori. Atunci nu am ști ce notă a acordat fiecare profesor unui candidat la proba respectivă.

Rezultă necesitatea de a relaționa entitatea NOTA cu entitatea PROFESOR, pentru a întregii informațiile privitoare la notă.

Vom cunoaște cu exactitate cui i s-a acordat nota (prin relația cu entitatea CANDIDAT), la ce probă (prin relația cu entitatea PROBA) și de către cine (prin relația cu entitatea PROFESOR).

Adăugarea acestei relații creează în diagramă o buclă ce creează posibilitatea apariției unor relații ciclice, *redundante*, situație ce trebuie evitată.

O relație între două entități A și B este considerată redundantă dacă relația se poate deduce din două relații  $A \rightarrow C$  și  $C \rightarrow B$  create anterior.

Un exemplu de relație redundantă este prezentat în figura următoare:

*Figura 2.5. Relație redundantă*

Dacă un oraș e împărțit în unul sau mai multe cartiere, și un cartier e la rândul lui împărțit în mai multe zone rezidențiale, se poate deduce că orașul este împărțit în zone rezidențiale, fără a fi necesar să marcăm acest lucru în diagramă. Ar apărea o relație ciclică, redundantă.

Relația PROFESOR  $\rightarrow$  NOTA ar putea fi considerată redundantă, deoarece din diagramă rezultă faptul că un candidat aparține unei clase ce este repartizată unei subcomisii de examinare, formată din doi profesori evaluatori.

Se poate deduce deci ce profesori au evaluat un elev, nefiind necesară o relație directă între CANDIDAT și NOTA. Pentru a destrăma bucla ar trebui eliminată o relație.

Dacă am elimina relația PROFESOR → NOTA, neexistând o ierarhie între relațiile PROFESOR→SUBCOMISIE→CLASA→CANDIDAT→NOTA, nu am ști ce profesor a acordat nota.

Dacă eliminăm relația PROFESOR→SUBCOMISIE, nu am cunoaște (sau ar fi dificil de aflat) fiecare profesor cărei subcomisii aparține.

Soluția este păstrarea diagramei în forma prezentă, nefiind de fapt o situație de redundanță, deoarece numele relațiilor nu sugerează că dacă un profesor este alocat unei subcomisii, el evaluează neapărat.

## 2.7 Limbajul SQL. Introducere

Primele sisteme de baze de date relaționale au apărut în 1970. Cele mai populare **SGBD**-uri relaționale sunt: **Oracle**, **Microsoft SQL Server**, **MySQL**. Toate aceste sisteme de baze de date relaționale au în comun limbajul standard de interogare a bazei de date numit **SQL**. Faptul că toate aceste sisteme de gestiune sau de management al bazelor de date (**SGBD**) sunt bazate pe standardul **SQL** le face să aibă foarte multe asemănări, ceea ce reprezintă un avantaj deoarece cunoscând bine unul din aceste sisteme de baze de date, se poate trece destul de ușor către un alt sistem de baze de date care se bazează pe standardul **SQL**.

**SQL - Structured Query Language** este un limbaj de baze de date realizat pentru a extrage informații și a administra bazele de date relaționale. Limbajul **SQL** a devenit standard **ANSI** (**American National Standards Institute**) în 1986. Fiecare sistem de management al bazei de date (**RDBMS - Relational Database Management System**) are propria versiune de limbaj **SQL**, bazată pe standardul **SQL**. Astfel, limbajul **SQL** folosit în **MySQL**, față de limbajul **SQL** folosit în PostgreSQL sau Oracle, deși asemănătoare, au elemente distincte, specifice celui **RDBMS**.

**MySQL** este o aplicație comercială pentru managementul bazelor de date relaționale (pe scurt un **RDBMS**) foarte populară, mai ales în dezvoltarea aplicațiilor web. **MySQL** este dezvoltată de firma suedeză MySQL AB ce a fost ulterior cumpărată de compania Sun Microsystems. După preluarea companiei Sun Microsystems de către compania Oracle, **MySQL** a devenit un produs Oracle și oferă utilizatorilor atât o licență open-source, cât și o licență comercială.

Echipele ce au dezvoltat limbajul **PHP** și baza de date **MySQL** au colaborat cu succes de-a lungul timpului pentru a oferi o interoperabilitate ridicată între cele două programe, astfel încât prima preferință a programatorilor dezvoltatori în **PHP** pentru baze de date este **MySQL**.

În plus, **PHP** are extensii (set de funcții) pentru a lucra și cu alte baze de date: **PostgreSQL**, **Oracle**, **SQL Server**, etc.

## 2.8 Clienți MySQL

Sistemele de baze de date sunt concepute într-o arhitectura client-server. Astfel, serverul de baze de date este programul principal ce stochează și manipulează datele, și răspunde clienților ce se conectează la acesta pentru a cere informații sau pentru a trimite cereri de altă natură (adăugări, modificări, etc). Serverul **MySQL** și clientul **MySQL** folosit pentru interogare

pot fi instalate pe același calculator, dar nu neapărat. Dacă lucrăm local (pe calculatorul propriu) și folosim un program ca **WAMP** server, atât serverul **MySQL** cât și clientul **MySQL** pe care-l alegem, vor fi instalate pe calculatorul nostru.

În momentul în care este mutată baza de date pe un server de hosting, serverul **MySQL** va fi pe acel server de hosting iar clientul **MySQL** poate fi tot pe acel server (de exemplu **phpMyAdmin**) sau ne putem conecta cu un client **MySQL** instalat pe calculatorul nostru.

Așadar, **SQL** este un limbaj special conceput pentru comunicarea cu bazele de date.

## 2.9 Medii de lucru

În continuare vom prezenta programele pe care le vom folosi pentru a testa noțiunile pe care le vom învăța. Sistemele de baze de date funcționează într-o **arhitectură de tip client-server**, astfel că este necesar să avem un server la care să ne putem conecta sau program care să instaleze printre alte componente și un server de baze de date **MySQL**.

### 2.9.1 Serverul de baze de date MySQL

În primul rând avem nevoie de instalarea pe calculator a unui server de baze de date **MySQL**. În acest sens vom instala un program numit **WAMP** care instalează local, pe calculator, un server de **Apache** și unul de baze de date **MySQL**. Acronimul WAMP vine de la **Windows Apache MySQL PHP**.

Adresa de la care se poate descărca acest program este următoarea: <http://www.wampserver.com/en/>

Corepondentul **WAMP** pentru sistemul de operare Linux, este un program numit LAMP (**Linux Apache MySQL PHP**).

Există și alte programe care odată instalate pe calculator și lansate în execuție ne oferă un server de baze de date. De exemplu: **XAMPP** sau **EasyPHP**.

Toate aceste programe pot fi folosit și atunci când realizăm aplicații web pe calculatorul propriu în limbajul **PHP** și avem nevoie de un server **Apache** pentru a le testa. De altfel aceste programe (**WAMP**, **XAMPP**, etc.) instalează, pe lângă serverul de baze de date **MySQL**, un server **Apache**, precum și un interpretor de **PHP**. Pentru noi, în lucrul cu baze de date **MySQL**, prezintă interes doar componenta care instalează un server de baze de date. Deci, ne va interesa ca serviciul de **MySQL** să fie pornit pentru a se putea realiza conectarea la serverul de **MySQL**.

După instalarea **WAMP** (instalare care este foarte simplă, se va face printr-un Wizard), se lansează în execuție acest program.

La pornire, pictograma acestei aplicații se plasează în partea dreaptă a barei de start și, atunci când toate serviciile oferite sunt pornite, are culoarea verde.

În continuare iată și fereastra **WAMP** care se deschide la clic pe această pictogramă:

În această imagine se observă printre serviciile oferite de **WAMP** și **MySQL**.

În cazul în care serviciul este oprit se apasă opțiunea *Start All Services*. De asemenea, pot fi oprite serviciile prin opțiunea *Stop All Services* sau restartate prin opțiunea *Restart All Services*.

## 2.9.2 Clientul de baze de date HeidiSQL

De asemenea, vom instala și un program client care se va conecta la serverul de MySQL și care ne permite să lucrăm efectiv cu baze de date în **MySQL**. Este vorba de programul **HeidiSQL**.

Acest program poate fi descărcat de la următoarea adresă: <http://www.heidisql.com/download.php>

După lansarea în execuție se va deschide următoarea fereastră:

La Network Type opțiunea este **MySQL**, întrucât ne conectăm la un server local, la Hostname/IP este trecută adresa IP corespunzătoare localhost (127.0.0.1).

Conectarea la baza de date se face cu userul root, fără parolă (aceasta este conectarea standard, evident, conectarea fără parolă nu este recomandată pe un server real din motive de securitate).

Se apasă butonul Open și se deschide fereastra următoare:

În această fereastră, în partea stângă se observă bazele de date disponibile, iar în partea dreaptă vedem tabelele bazei de date selectate, dacă există sau avem tab-ul *Query* care permite scrierea de instrucțiuni **MySQL** și rularea acestora prin acționarea butonului *Execute SQL*.

**HeidiSQL** nu este singurul program de tip client de baze de date. De asemenea, este interesat că acest client nu este doar pentru baze de date **MySQL**, el oferă posibilitatea de conectare și la baze de date **SQL Server** și **PostgreSQL**. Astfel că este un program foarte util.

## 2.9.3 Alți clienți de baze de date MySQL cu interfață vizuală

Un alt program de acest fel, client de baze de date **MySQL** este **MySQL Workbench**. Acesta este un client specific bazelor de date **MySQL**. Se poate descărca de pe site-ul oficial **MySQL**, de la următoarea adresă: <https://dev.mysql.com/downloads/workbench/>

Un alt client utilizat, în special, pentru administrarea bazelor de date **MySQL** din aplicațiile web, este **phpMyAdmin**. Acest client se instalează împreună cu programul de tip server de baze de date (**WAMP**, **XAMPP**, **EasyPHP**, etc.) sau cu serverul de baze de date **MySQL**, dacă neconectăm la un server de **MySQL** și nu la un server instalat local printr-un program de acest fel.

Poate fi accesat doar într-un browser web, la adresa serverului (IP sau nume) urmată de **/phpMyAdmin**. Dacă avem un server instalat local, atunci aplicația se accesează în browser la adresa: **http://localhost/phpMyAdmin** sau **http://127.0.0.1/phpMyAdmin**, unde 127.0.0.1 este IP-ul localhost.

Mai există și alte programe de tip client de baze de date **MySQL**, precum și programe de tip client pentru conectarea la alte sisteme de baze de date (**SQL Server**, **Oracle**, etc.).



Avantajul acestor programe de tip client este acela că pun la dispoziția utilizatorului o interfață grafică care le fac foarte accesibile și ușor de înțeles și de utilizat. Sunt foarte intuitive și oferă posibilitatea de a vizualiza baza de date, tabelele componente, coloanele unei tabele și proprietățile și constrângerile aplicate lor, precum și înregistrările stocate în tabele.

## 2.9.4 Clientul linie de comandă

Bineînțeles că se poate utiliza și **clientul linie de comandă** pentru conectarea și lucrul cu baza de date **MySQL**, dar și cu alte sisteme de baze de date. Aceasta dă posibilitatea conectării directe la serverul **MySQL** din linia de comandă. De altfel, acesta este cel mai facil mod de conectare la **serverul MySQL**, deoarece nu avem nevoie de alte aplicații de tip **client MySQL** instalate pe calculatoare pentru a ne conecta la serverul de baze de date **MySQL**.

Însă, deși conectarea din linia de comandă este foarte ușoară, lucrul cu instrucțiuni **SQL** în acest mediu este mai dificil. Mult mai ușor este să utilizăm programe de tip **client MySQL** care permit conectarea la **serverul** de baze de date **MySQL** și, în plus, oferă o interfață care permite vizualizarea elementelor componente ale bazei de date (tabele, tabele virtuale, proceduri stocate, etc.).

Clientul de tip linie de comandă face parte din kit-ul de instalare **MySQL**. El se numește **mysql.exe** și permite conectarea atât la un server local (localhost) cât și la un server la distanță.

Pornirea clientului **mysql.exe** se realizează, în general, din linia de comandă oferită de sistemul de operare (în cazul sistemului de operare **Windows – Command Prompt**, iar în cazul sistemelor de operare **Linux/Unix – consola** sau **emulatorul de terminal**).

Pentru accesarea acestei aplicații (**mysql.exe**), în cazul folosirii sistemului de operare **Windows**, deci, dacă serverul de baze de date **MySQL** este instalat pe un sistem **Windows**, este necesară deschiderea utilitarului **Command Prompt** care se găsește în **Start – All Programs – Accesories – Command Prompt** pe un sistem de operare **Windows 7**.

Așa cum spuneam, este de preferat utilizarea unui client cu interfață vizuală, deoarece acesta ne permite vizualizarea bazelor de date și ale obiectelor stocate în bazele de date de pe server, precum și datele efective, concrete stocate în baza de date. Totuși, sunt situații în care este absolut necesară conectarea din linia de comandă la un server de baze de date (resetarea parolei de conectare la baza de date pentru un utilizator, vizualizarea interogărilor care îngreunează rularea procesului **MySQL**, realizarea unui **backup** al bazei de date (copie de siguranță) sau **restaurarea** unei astfel de copii de siguranță, etc.)

Important este să înțelegem deosebire dintre un program de tip **client** și **serverul** de baze de date. Deci, **HeidiSQL**, **MySQL Workbench** sau alte programe de tip client nu reprezintă baza de date **MySQL**, ci doar un client care se conectează la baza de date și permite atât vizualizarea obiectelor din ea, cât și scrierea de instrucțiuni **SQL**. Așadar, să nu confundăm un client care se conectează la un server de baze de date și oferă facilitatea de a interacționa cu baza de date prin modul vizual sau prin scrierea de comenzi **SQL**, cu baza de date.

## 2.10 Concluzii

Această lecție a dezvoltat conceptul de proiectare a unei baze de date relaționale. Am prezentat în cadrul ei exemple concrete de realizare a design-ului unei baze de date. Conceptul a fost prezentat și explicat pe larg, cu exemple concrete. Tot în cadrul acestei lecții s-a realizat și o introducere în limbajul **SQL**.

De asemenea, s-a făcut și o prezentare a aplicațiilor pe care le vom folosi mai departe pentru conectarea la o bază de date **MySQL** și pentru realizarea de operații pe baza de date.

În următoarea lecție se va trece la prezentarea sintaxei **SQL**. Vom discuta pe larg despre Limbajul de **Descriere a Datelor (LDD)** și despre comenzile (instrucțiunile) acestui limbaj care se referă la structura bazei de date și a tabelor componente. Va fi prezentată sintaxa precum și exemple concrete de utilizare a acestor comenzi. Vor fi prezentate, de asemenea, tipurile de date existente în **MySQL**.

## Tema Sedinta 2

Sa se proiecteze diagrama ERD pentru o baza de date. Puteti alege una dintre temele de mai jos sau puteti compune o tema similara:

- un magazin (entitatile: PRODUS, CATEGORIE, PRODUCATOR, TARA, FURNIZOR, CLIENT, COMANDA, LIVRARE, FACTURA, TIP\_PLATA, PROMOTIE, etc.)
- o biblioteca (entitatile: CARTE, AUTOR, CATEGORIE, CITITOR, IMPRUMUT, etc.)
- o companie (entitatile: ANGAJAT, DEPARTAMENT, PROIECT, CLIENT, FACTURA, etc.)

## 3. Limbajul de descriere a datelor (LDD)

### 3.1 Introducere

Recapitulând noțiunile prezentate în capitolele anterioare și raportându-ne concret la o bază de date MySQL, avem următoarele corespondențe:

- o *entitate* reprezintă o *tabelă* din baza de date;
- un *atribut* reprezintă un *câmp* sau o *coloană* din baza de date;
- un *tuplu* reprezintă o înregistrare din baza de date;
- **UID** reprezintă *cheia primară* a unei tabele.

**Limbajul de Desciere a Datelor (LDD, în limba engleză Data Description Language, prescurtat DDL)** conține comenzi pentru:

- crearea unei baze de date;
- ștergerea unei baze de date;
- crearea unei tabele;
- ștergerea unei tabele;
- modificarea structurii unei tabele.

În continuare vom prezenta elemente ce țin de sintaxa limbajului **MySQL**. *Sintaxa limbajului* se referă la un *set de reguli* ce trebuie respectate atunci când scriem o *instrucțiune*. Pentru *instrucțiune* vom mai întâlni denumirile de *comandă*, respectiv, de *frază SQL*.

Așadar, pe parcursul acestui curs vom întâlni în foarte multe rânduri termenii de **instrucțiune SQL**, **comandă SQL** sau **frază SQL**. De asemenea, trebuie precizat că, spunem la modul generic **comandă SQL** sau **frază SQL** sau **instrucțiune SQL**, deși, în cazul nostru, este limpede că ne vom referi la **instrucțiuni/comenzi/fraze MySQL**. După cum am precizat și în lecția precedentă, **MySQL** este un Sistem de Gestiune al Bazelor de Date (**SGBD**) bazat pe standardul **SQL (Structured Query Language)**.

Deși prezintă particularități, ca de altfel orice **SGBD**, totuși instrucțiunile **MySQL** sunt foarte asemănătoare cu cele ale standardului **SQL**. Din acest motiv, în mod uzual, vorbim de instrucțiuni **SQL**. Cu alte cuvinte, putem spune că **SQL** este un nucleu de la care s-au dezvoltat aceste aplicații complexe utilizate pentru interacțiunea cu bazele de date, denumite **SGBD-uri**.

Așa cum spuneam, în **MySQL**, ca de altfel în orice alt **SGBD**, pentru fiecare instrucțiune/comandă avem o sintaxă, adică un set de reguli de scriere care trebuie respectat pentru a nu fi generate erori la rularea instrucțiunilor.

Astfel, există un set de **cuvinte rezervate** ale limbajului care definesc anumite acțiuni și care trebuie specificate în cadrul unei instrucțiuni **MySQL**. Aceste cuvinte speciale (rezervate) se numesc **cuvinte cheie**. Atunci când scriem o instrucțiune **MySQL** trebuie, așadar, să folosim aceste cuvinte cheie, plus alte cuvinte (nume de tabele, nume de câmpuri, valori, operatori, etc.) scrise într-o anumită ordine.

Important de reținut este și faptul că orice instrucțiune **MySQL** se încheie cu **caracterul „;”**.

În general, clienții **MySQL**, adică programele cu ajutorul cărora se realizează conectarea la o bază de date **MySQL** și se execută instrucțiuni **MySQL**, recunosc aceste *cuvinte cheie* și le *colorează* diferit de celelalte cuvinte comune (nerezervate) utilizate în cadrul instrucțiunilor.

Acest fapt ușurează în mare parte scrierea unei instrucțiuni. O altă mențiune importantă este aceea că **nu este indicat** să denumim baze de date, tabele sau coloane (câmpuri) ale tabelelor cu nume care să fie cuvinte din limbajul rezervat al **MySQL**. Dacă se întâmplă acest lucru există riscul să avem erori la execuția unor comenzi ce conțin aceste denumiri. Un câmp denumit astfel ar trebui prefixat de numele tablei în fiecare instrucțiune pentru a nu fi eroare, deci s-ar complica mult codul instrucțiunii scrise.

## 3.2 Crearea unei baze de date

Comanda care se folosește pentru crearea unei baze de date în MySQL este:

```
CREATE DATABASE nume_bd;
```

unde *nume\_bd* reprezintă numele pe care vrem să îl aibă baza de date.

Denumirea bazei de date poate să conțină doar caractere alfanumerice și semnul „\_”.

## 3.3 Ștergerea unei baze de date

Comanda care se folosește pentru crearea unei baze de date în MySQL este:

```
DROP DATABASE nume_bd;
```

## 3.4 Utilizarea unei baze de date

Comanda pentru stabilirea unei baze de date ca fiind curentă este următoarea:

```
USE nume_bd;
```

Această comandă specifică faptul că din momentul executării acestei instrucțiuni se folosește baza de date specificată (se pot realiza operații pe această bază de date). Comanda este utilă atunci când avem mai multe baze de date.

Fiecare instrucțiune MySQL se încheie cu caracterul „;”.

## 3.5 Crearea unei tabele

Comanda care se folosește pentru crearea unei tabele într-o bază de date este următoarea:

```
CREATE TABLE nume_tabelă(  
    nume_atribut1 tip_dată(dimensiune) [modificatori],  
    nume_atribut2 tip_dată(dimensiune) [modificatori],  
    ...  
    nume_atributn tip_dată(dimensiune) [modificatori][,]  
    [restricții]  
);
```

În comanda aceasta, *nume\_atribut* reprezintă numele coloanelor (câmpurilor) tabeli, *tip\_dată* reprezintă tipul de dată al câmpului respectiv (de exemplu: numeric, șir de caractere, dată calendaristică, etc.), *modificatori* este opțional la fel ca și *restricțiile* ce se pot aplica pe anumite câmpuri. Modificatorii reprezintă anumite opțiuni sau restricții ce se pot aplica asupra coloanelor din tabelă.

## 3.6 Ștergerea unei tabele

Comanda care se folosește pentru a șterge o tabelă dintr-o bază de date este următoarea:

```
DROP TABLE nume_tabelă;
```

## 3.7 Modificarea unei tabele

Comanda care se folosește pentru modificarea numelui unei tabele dintr-o bază de date este următoarea:

```
ALTER TABLE nume_tabelă RENAME TO nume_nou_tabelă;
```

sau

```
RENAME TABLE nume_tabelă TO nume_nou_tabelă;
```

## 3.8 Modificarea structurii unei tabele

Comenzile folosite pentru modificarea structurii unei tabele sunt următoarele:

- pentru modificarea definiției unui câmp:

```
ALTER TABLE nume_tabelă CHANGE nume_câmp nume_câmp definiție_câmp;
```

sau

```
ALTER TABLE nume_tabelă MODIFY nume_câmp definiție_câmp;
```

- pentru adăugarea unui câmp într-o tabelă

```
ALTER TABLE nume_tabelă ADD nume_câmp definiție_câmp;
```

- pentru ștergerea unui câmp dintr-o tabelă

```
ALTER TABLE nume_tabelă DROP nume_câmp;
```

## 3.9 Tipuri de date

În **MySQL** întâlnim următoarele tipuri de date:

- numerice
- șiruri de caractere
- binare
- date calendaristice
- text

**Tipurile de date numerice sunt următoarele:**

- pentru numere întregi:

**TINYINT** – poate lua valori de la -128 până la 127 sau de la 0 la 255 unsigned

**SMALLINT** – valori în intervalul -32 768 până la 32 767 sau de la 0 la 65 535 unsigned

**MEDIUMINT** – de la -8 388 608 până la 8 388 607 sau de la 0 la 16 777 215 unsigned

**INT** – de la -2 147 483 648 până la 2 147 483 647 sau de la 0 la 4 294 967 295 unsigned

**BIGINT** – de la -9 223 372 036 854 775 808 până la 9 223 372 036 854 775 807 sau de la 0 la 18 446 744 073 709 551 615 unsigned

Cel mai folosit tip de date numeric întreg este **INT**, sau dacă avem valori numerice mici într-un câmp putem folosi **SMALLINT** sau **TINYINT** care ocupă spațiu mai puțin pe disc.

Fiecărui tip de dată i se specifică și lungimea, de exemplu dacă avem valori de la 1 la 100 într-un câmp putem alocă ca tip de dată **INT(3)**, adică numere întregi cu lungimea maximă 3.

- pentru numere cu zecimală:

**FLOAT** – folosit pentru numere mici cu virgulă;

**DOUBLE** – folosit pentru numere mari cu virgulă;

**DECIMAL** – permite alocarea unui număr fix de zecimale.

De exemplu, un câmp ce conține prețul unui produs poate fi definit de tip **DOUBLE(5,2)**. În paranteză este trecut numărul total de cifre al prețului (5) respectiv numărul obligatoriu de zecimale care va fi afișat (2). Prin urmare, datele introduse în câmpul preș definit de tipul **DOUBLE(5,2)** poate lua valori cuprinse în intervalul închis [-999.99,999.99]. Delimitatorul pentru un număr cu zecimale recunoscut de MySQL este „.”.

Tipurile de date folosite pentru șiruri de caractere sunt următoarele:

**CHAR** – lungime fixă de la 0 la 255 de caractere

**VARCHAR** – lungime variabilă de la 0 până la 65 535 de caractere. La versiunile mai vechi de 5.0.3 ale **MySQL** lungimea era variabilă de la 0 la 255 de caractere.

În practică, tipul **VARCHAR** este cel mai folosit pentru definirea câmpurilor de tip șir de caractere sau string. Într-o tabelă cu informații despre angajații unei companii, de exemplu, numele angajaților poate fi ținut într-un câmp nume de tip **VARCHAR(70)**, între paranteze fiind trecută dimensiunea maximă pe care o poate avea valoarea introdusă în acest câmp.

Așadar, câmpul nume poate avea maxim 70 de caractere ceea ce considerăm a fi suficient pentru a nu avea probleme de trunchiere a vreunui nume.

Putem defini, de asemenea, și un câmp prenume de tip **VARCHAR**, de dimensiune 100 de caractere, adică **VARCHAR(100)**.

Tipurile de date **CHAR** și **VARCHAR** acceptă și definirea unei valori implicite (default) pe care o va avea acel câmp în cazul în care nu se introduce nici o valoare în el.

Principala diferență între aceste două tipuri este că șirul dintr-un tip **CHAR** va fi stocat întotdeauna ca un șir cu lungimea maximă a coloanei, folosind spații pentru completare, dacă șirul introdus este mai mic decât lungimea coloanei.

Tipurile de date folosite pentru stocare text sunt următoarele:

**TINYTEXT** – un șir cu lungime maximă de 255 de caractere;

**TEXT** – un șir cu o lungime maximă de 65 535 de caractere;

**MEDIUMTEXT** – un șir cu o lungime maximă de 16 777 215 de caractere;

**LONGTEXT** – un șir cu o lungime maximă de 4 294 967 295 de caractere.

Pentru câmpuri în care este necesară stocarea unui text de mari dimensiuni, în general, se folosește tipul **TEXT**. Unui câmp de tip text nu i se poate specifica lungimea.

De exemplu, într-o tabelă a unei baze de date în care sunt stocate articole, câmpul ce conține conținutul (corpul) articolului poate fi de tip **TEXT**.

Spre deosebire de **VARCHAR**, tipul de date **TEXT** nu permite definirea unei valori implicite (default) pentru acel câmp.

La ultimele versiuni de **MySQL**, a fost mărită dimensiunea maximă a tipului de date **VARCHAR**, astfel încât poate fi folosit acesta și pentru câmpurile cu texte lungi.

Tipurile de date binare întâlnite în cadrul MySQL sunt următoarele:

**TINYBLOB** – stochează până la 255 bytes;

**BLOB (Binary Large Object)** – stochează până la 64 KB;

**MEDIUMBLOB** – stochează până la 16 MB;

**LOBLOB** – stochează până la 4 GB.

Aceste tipuri de date sunt folosite pentru stocarea obiectelor binare de mari dimensiuni cum ar fi imaginile. Valorile din câmpurile de tip **BLOB** sunt tratate ca șiruri binare.

Ele nu au un set de caractere iar sortarea și compararea lor se bazează pe valorile numerice ale octeților din valoarea câmpului respectiv definit cu acest tip.

Tipurile de date folosite pentru stocarea datelor calendaristice sunt următoarele:

**DATE** – stochează o dată calendaristică în formatul *an-lună-zi*;

**TIME** – stochează ora în formatul *oră-minut-secundă*;

**DATETIME** – stochează data și ora în formatul *an-lună-zi ora-minut-secundă*;

**TIMESTAMP** – este util la înregistrarea unor operații precum inserare sau actualizare pentru că reține implicit data efectuării ultimei operații.

Singurul format în care **MySQL** păstrează și afișează datele calendaristice este formatul *an-lună-zi (AAAA-LL-ZZ)*, sau, mai cunoscut acest format după denumirea în limba engleză *year-month-day*, sau prescurtarea *YYYY-MM-DD*.

Intervalul în care poate lua valori o dată calendaristică este foarte mare, de la '1000-01-01' până la '9999-12-31'. Dacă avem într-o tabelă a unei baze de date stocată data nașterii unei persoane care presupunem că este 20 martie 1981. Informația cu privire la data nașterii va fi stocată în baza de date în următorul format '1981-03-20'.

Formatul în care se salvează un câmp de tip **TIME**, câmp care păstrează ora în baza de date, este *oră-minut-secundă (HH-MM-SS)*, format mult mai cunoscut după denumirea în limba engleză, *hour-minute-second* sau după prescurtarea *HH-MM-SS*.

Formatul în care se salvează un câmp de tip **DATETIME** care stochează atât data cât și ora este *year-month-day hour-minute-second (AAAA-LL-ZZ HH-MM-SS sau YYYY-MM-DD HH-MM-SS)*.

Domeniul de valori este între '1970-01-01 00:00:00' până în '2037-01-01 00:00:00'. Formatul în care păstrează valorile pentru **TIMESTAMP** este *YYYYMMDDHHMMSS*.

În cazul în care o dată calendaristică nu este introdusă în formatul corect sunt convertite la valoarea zero, adică '0000-00-00' dacă este cu câmp de tip **DATE** sau, dacă este și ora, de exemplu, tipul de date **DATETIME**, '0000-00-00 00-00-00'.

## 3.10 Modificatori

Modificatorii sunt constrângeri ce pot fi definite pentru câmpurile tabelelor stocate în baza de date. Modificatorii se definesc prin utilizarea unor cuvinte cheie și a unei sintaxe specifice.

Modificatorii ce pot fi întâlniți în definițiile de descriere ale unui câmp sunt:

**NOT NULL** – modificador sau constrângere care stabilește pentru câmpul la care este definit să nu permită valoarea **NULL**;

**DEFAULT** – permite stabilirea unei valori implicite pentru acel câmp care are setat acest modifcator;

**AUTO\_INCREMENT** – constrângere care este stabilită pentru un câmp care este cheie primară, în general un id care este incrementat automat la fiecare inserare de înregistrări în tabelă;

**PRIMARY KEY** – constrângere care definește acel câmp ca fiind cheie primară a unei tabele;

**FOREIGN KEY** – constrângere care definește o cheie externă pentru o tabelă;

**UNIQUE** – constrângere de unicitate care impune valori unice pentru câmpurile care au definit acest modificador;

**INDEX** – constrângere care aplică un index pe un câmp al unei tabele.

În continuare prezentăm câteva exemple de folosire a instrucțiunilor prezentate în cadrul limbajului de descriere a datelor (**LDD**). Aceste instrucțiuni se referă doar la structura unei baze de date cu tabelele aferente și nu au legătură cu valorile din baza de date (cu datele propriu zise).

Considerăm crearea unei aplicații care gestionează cărțile aflate într-o bibliotecă. Primul pas pentru realizarea acestei aplicații ar fi crearea bazei de date, bază de date care va primi numele *biblioteca*.

Deci instrucțiunea de creare este următoarea:

```
CREATE DATABASE biblioteca;
```

Prezentăm în continuare comanda de creare a unei tabele, autori ce conține informații despre autorii cărților din această bibliotecă.

```
CREATE TABLE autori(  
    id_autor INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    nume VARCHAR(200)  
);
```

Așadar, avem o instrucțiune de creare a unei tabele din baza de date, este vorba de o tabelă simplă cu doar 2 câmpuri ce conține numele autorilor cărților. Primul câmp, *id\_autor*, conține valori care identifică în mod unic o înregistrare, deci acest câmp este utilizat drept cheie primară a tabelei.

În descrierea definiției acestui câmp se aplică modificatorii pentru cheie primară, valori nenule și incrementare automată.

Al doilea câmp, *nume*, conține un șir de caractere de lungime maximă 200 de caractere ce va stoca numele și prenumele fiecărui autor.

## 3.11 Chei externe

Instrucțiunea prin care se aplică restricția de cheie externă este următoarea:

```
FOREIGN KEY (nume_câmp1) REFERENCES nume_tabelă(nume_câmp2)
```

**FOREIGN KEY** și **REFERENCES** sunt cuvinte cheie în timp ce *nume\_câmp<sub>1</sub>* reprezintă câmpul din tabelă care este cheie externă în tabelă, în timp *nume\_câmp<sub>2</sub>* reprezintă câmpul la care face referire, adică cheia primară din tabela *nume\_tabelă*.

## 3.12 Index

Introducem în continuare noțiunea de index. Indecșii sunt folosiți pentru sortarea logică a datelor în vederea îmbunătățirii vitezei operațiilor de căutare și sortare.

Indecșii dintr-o bază de date funcționează în maniera următoare: datele din cheile primare sunt întotdeauna sortate; este o operație pe care programul **SGBD** o execută. Deci, regăsirea anumitor rânduri în funcție de cheia primară este întotdeauna o operație rapidă și eficientă.



Un index se poate defini pe una sau mai multe coloane. Indecșii îmbunătățesc performanțele operațiilor de regăsire dar le degradează pe acelea ale operațiilor de inserare, modificare și ștergere a datelor. Când sunt executate aceste operații, programul **SGBD** trebuie să actualizeze indexul în mod dinamic. Datele din index pot ocupa o cantitate mare de spațiu de stocare.

Atunci când se definește un index se creează un fișier de index, iar în momentul în care este executată o instrucțiune de interogare pe câmpul indexat, se face practic o căutare în fișierul de index, din acest motiv avem o viteză foarte mare de execuție. Vom reveni, într-un capitol separat care este special dedicat conceptului de indexare, pentru a discuta pe larg despre conceptul de indexare și despre tipurile de indecși ce pot fi definiți pe coloanele din tabelele bazelor de date.

## 3.13 Exemple

Următoarea comandă modifică numele tabelului *autori* în *autori\_noi*:

```
ALTER TABLE autori RENAME TO autori_noi;
```

Aceeași comandă poate fi scrisă și în felul următor:

```
RENAME TABLE autori TO autori_noi;
```

Comanda de mai jos modifică lungimea maximă a câmpului *nume*:

```
ALTER TABLE autori CHANGE nume nume VARCHAR(70);
```

Iată și o comandă care adaugă un nou câmp în această tabelă, câmpul *prenume*:

```
ALTER TABLE autori ADD prenume VARCHAR(100);
```

De asemenea, prezentă și instrucțiunea pentru ștergerea unui câmp:

```
ALTER TABLE autori DROP prenume;
```

Prezentăm în continuare alte exemple de utilizare a instrucțiunilor din limbajul de descieri a datelor. Astfel, vom realiza o instrucțiune de creare a unei baze de date, apoi vom crea 2 tabele în această bază de date și vom aplica diverse constrângeri.

Presupunem crearea unei baze de date în care se păstrează informații despre angajații și departamentele unei companii. Așadar, vom avea o bază de date pe care o vom denumi *companie* și în această bază de date vom crea două tabele pentru evidența departamentelor, respectiv a angajaților.

Instrucțiunea pentru crearea bazei de date *companie*:

```
CREATE DATABASE companie;
```

Instrucțiunea pentru crearea tabelului *departamente*:

```
CREATE TABLE departamente(  
    id_departament INT(4) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    denumire VARCHAR(100),  
    manager VARCHAR (100)  
);
```

Instrucțiunea pentru crearea tabelului *angajati*:

```
CREATE TABLE angajati(  
    id_angajat INT (6) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    nume VARCHAR (100),  
    prenume VARCHAR (100),  
    cnp VARCHAR (13),  
    data_nasterii DATE,
```

```

        data_angajarii DATE,
        id_departament INT (4),
        FOREIGN KEY(id_departament)REFERENCES departamente(id_departament)
);

```

Observăm în definiția de creare a tabelii *angajati* aplicarea unei constrângeri de tip **FOREIGN KEY** pe câmpul *id\_departament* cu referire la câmpul *id\_departament* din tabela *departamente*, câmp care este cheie primară în această tabelă.

Așa cum am precizat anterior, o cheie externă poate fi definită în instrucțiunea de creare a unei tabeli, după ce au fost enumerate toate coloanele tabelii sau poate fi creată prin instrucțiuni de modificare a structurii unei tabeli din baza de date.

Pentru a prezenta și cea de-a doua modalitate, mai întâi vom elimina (șterge) constrângerea de cheie externă aplicată câmpului *id\_departament* din tabela *angajati*.

Instrucțiunea pentru eliminarea unei constrângeri de tip **FOREIGN KEY** este următoarea:  
**ALTER TABLE angajati DROP FOREIGN KEY nume\_cheie;**

Fiecare **FOREIGN KEY** primește automat un nume dacă noi nu am asociat un nume în definirea constrângerii. Pentru a defini un nume unei chei externe, în fața instrucțiunii de creare și referire trebuie să mai avem cuvântul cheie **CONSTRAINT** urmat de numele dat cheii externe. De exemplu, în cazul nostru, vom denumi cheia externă *fk\_deptAng*. Deci am fi avut următoarea instrucțiune:

```

CONSTRAINT fk_deptAng FOREIGN
KEY(id_departament)REFERENCES departamente(id_departament)

```

Așadar, instrucțiunea de ștergere a cheii externe, în cazul nostru concret, va fi:  
**ALTER TABLE angajati DROP FOREIGN KEY fk\_deptAng;**

În continuare, vom crea din nou, constrângerea de tip **FOREIGN KEY** pentru tabela *angajati*:  
**ALTER TABLE angajati ADD CONSTRAINT fk\_deptAng FOREIGN**  
**KEY(id\_departament) REFERENCES departamente(id\_departament);**

De asemenea, avem și forma în care nu atribuim un nume constrângerii, iar, în acest caz, **SGBD-ul** va atribui un nume automat pentru constrângere:

```

ALTER TABLE angajati ADD FOREIGN
KEY(id_departament)REFERENCES departamente(id_departament);

```

Important de precizat este faptul că, dacă tabela *departamente* nu ar fi fost creată înaintea tabelii *angajati* ar fi aparut o problemă de integritate a datelor în momentul în care încercăm să creăm o constrângere de tip cheie externă care ar fi făcut referire la un câmp dintr-o tabelă care nu exista.

În continuare prezentăm alte câteva exemple de instrucțiuni de modificare a structurii unei tabeli din baza de date. De exemplu, eliminarea cheii primare din tabela *departamente* se face cu instrucțiunea:

```

ALTER TABLE departamente DROP PRIMARY KEY;

```

Adăugarea unei chei primare la o tabelă deja creată se poate face cu instrucțiunea:

```

ALTER TABLE departamente ADD PRIMARY KEY (id_departament);

```

Schimbarea dimensiunii unui câmp dintr-o tabelă se face cu instrucțiunea:

```

ALTER TABLE angajati CHANGE prenume prenume VARCHAR(150);

```

Ștergerea unui câmp dintr-o tabelă se poate face cu următoarea instrucțiune:

```

ALTER TABLE departamente DROP manager;

```

Adăugarea unui câmp într-o tabelă se poate face cu următoarea instrucțiune:

```
ALTER TABLE departamente ADD manager VARCHAR(150) NOT NULL;
```

De asemenea, observăm în instrucțiunea anterioară și stabilirea unei constrângeri de tip **NOT NULL** pentru câmpul adăugat în tabelă.

Redenumirea unui câmp dintr-o tabelă se face astfel:

```
ALTER TABLE angajati CHANGE data_angajarii data_ang DATE;
```

Adăugarea unei constrângeri pe un câmp deja existent se face cu instrucțiunea:

```
ALTER TABLE angajati CHANGE cnp cnp VARCHAR(13) NOT NULL;
```

## 3.14 Concluzii

Așadar, în această lecție, au fost prezentate instrucțiuni și am trecut în revistă, comenzi din limbajul de descriere a datelor (**LDD**). Aceste comenzi afectează structura bazei de date, în timp ce limbajul de manipulare a datelor (**LMD**), de care ne vom ocupa în lecția următoare, conține instrucțiuni care afectează înregistrările din tabelele unei baze de date. De asemenea, tot în cadrul acestei lecții s-a discutat și despre tipurile de date ale **MySQL** și au fost prezentate și exemplificate constrângerile ce pot fi aplicate asupra coloanelor din tabelele unei baze de date, precum și anumiți modificatori ce pot fi definiți pe coloanele unei tabele.

## Tema Sedinta 3

Creați o **bază de date** în care să se păstreze evidența clienților unei companii, precum și facturile emise clienților. Un client poate să aibă mai multe facturi, în timp ce o factură este emisă unui singur client. Baza de date se va numi **clienti\_companie**.

Tabela **clienti** reține informații de bază despre clienții companiei: **id\_client**, **nume**, **prenume**, **număr de telefon**, **localitate**. Tabela **facturi** conține informații despre facturile clienților: **id\_factura**, **data\_factura**, **valoare**, **clientul** caruia i-a fost emisă factura.

### Cerințe:

1. Scrieți instrucțiunea pentru crearea bazei de date.
2. Scrieți instrucțiunile pentru crearea tabelelor **clienti** și **facturi**.
3. Stabiliți tipurile de dată și dimensiunile pentru fiecare câmp al celor două tabele.
4. Scrieți o instrucțiune **SQL** prin care să modificați numele tablei **facturi** în **facturi\_clienti**.
5. Scrieți o instrucțiune **SQL** prin care să reveniți la numele **facturi** pentru tabela **facturi\_clienti**.
6. Scrieți o instrucțiune **SQL** prin care să adăugați o constrângere **NOT NULL** pe câmpul **data\_factura** din câmpul **facturi**.
7. Scrieți o instrucțiune **SQL** prin care să măriți dimensiunea câmpului **nume** din tabela **clienti**.

8. Scrieți o instrucțiune **SQL** prin care eliminați constrângerea **FOREIGN KEY** din tabela **facturi**.
9. Scrieți o instrucțiune **SQL** prin care adăugați o constrângere **FOREIGN KEY** pe tabela **facturi** cu numele **fk\_fact**.
10. Scrieți o instrucțiune **SQL** prin care adăugați valoarea implicită (**DEFAULT**) **0** pe câmpul **valoare** din tabela **facturi**.

## 4. Limbajul de manipulare a datelor (LMD)

### 4.1 Introducere

Limbajul de manipulare a datelor conține comenzile de actualizare a datelor în interiorul tabelelor dintr-o bază de date, precum și comanda de regăsire a datelor. Aceste patru comenzi de manipulare sunt cele mai folosite deoarece ele permit interacțiunea cu datele stocate efectiv într-o bază de date.

Cele trei comenzi de actualizare a datelor dintr-o tabelă sunt: **INSERT**, **UPDATE** și **DELETE**.

Iată în continuare prezentată sintaxa **MySQL** a acestor comenzi:

Comanda pentru inserarea (adăugarea/introducerea) datelor într-o tabelă a bazei de date este **INSERT** și are următoarea sintaxă:

```
INSERT INTO nume_tabelă (câmp1, câmp2, ..., câmpn) VALUES (valoare1,valoare2, ..., valoaren);
```

Instrucțiunea **INSERT** mai are și alte forme în care poate fi utilizată. Una dintre ele este aceea în care se vor insera valori în toate coloanele unei tabelă și are sintaxa:

```
INSERT INTO nume_tabelă VALUES (valoare1, valoare2, ..., valoaren);
```

Următoarea formă a instrucțiunii de introducere date va insera valorile default (implicite) în toate câmpurile. Această formă este foarte puțin folosită.

```
INSERT INTO nume_tabelă VALUES ();
```

Prezentăm și forma în care se inserează valori doar în anumite coloane specificate:

```
INSERT INTO nume_tabelă (câmp1, câmp3) VALUES (valoare1, valoare3);
```

Urmează o formă diferită de celelalte a comenzii **INSERT**, asemănătoare, vom vedea cu sintaxa comenzii de actualizare **UPDATE**:

```
INSERT INTO nume_tabelă SET nume_câmp1 = valoare1, ..., nume_câmpn = valoaren;
```

Ultima formă permite inserearea mai multor înregistrări printr-o singură instrucțiune **INSERT**:

```
INSERT INTO nume_tabelă (câmp1, câmp2, ..., câmpn) VALUES (valoare11, valoare12, ..., valoare1n),  
(valoare21, valoare22, ..., valoare2n), ..., (valoaren1, valoaren2, ..., valoarenn);
```

Comanda pentru actualizarea sau modificarea unei înregistrări este **UPDATE**.

Sintaxa comenzii de actualizare a datelor este următoarea:

```
UPDATE nume_tabelă SET nume_câmp1 = valoare1 [, ..., nume_câmpn = valoaren]  
[WHERE condiții]  
[ORDER BY coloane]  
[LIMIT număr_rânduri];
```

Dacă lipsește clauza **WHERE** înseamnă că se vor actualiza toate înregistrările din tabelă.

Comanda pentru ștergerea datelor dintr-o tabelă este **DELETE**.

Sintaxa comenzii de ștergere a datelor este următoarea:

```
DELETE FROM nume_tabelă  
[WHERE condiții]  
[ORDER BY coloane]  
[LIMIT număr_rânduri];
```

Dacă lipsește clauza **WHERE** se vor șterge toate înregistrările din tabelă.

Comanda de regăsire a datelor este **SELECT**. Aceasta este cea mai utilizată comandă a limbajului SQL. Dacă pentru alte operații efectuate asupra bazei de date, nu toți utilizatorii primesc drepturi (privilegii), pentru comanda de regăsire se dă acest privilegiu, deoarece ea nu afectează datele stocate în tabele, ci doar le afișează potrivit condițiilor impuse.

Sintaxa comenzii **SELECT** este următoarea:

```
SELECT [DISTINCT] câmp1, câmp2, ..., câmpn [FROM nume_tabelă]  
[WHERE condiții]  
[GROUP BY câmp1 [, câmp2 ...]]  
[HAVING condiții]  
[ORDER BY câmp1 [ASC | DESC] [, câmp2 [ASC | DESC], ... ]]  
[LIMIT nr_rânduri];
```

Așadar, acestea sunt cele 4 instrucțiuni ce compun limbajul de manipulare a datelor. Ele au o sintaxă relativ simplă și ușor de înțeles. În continuare vom explica fiecare comandă în parte și vom prezenta câteva exemple de utilizare practică.

## 4.2 Instrucțiunea INSERT

Reluăm, în continuare, sintaxa instrucțiunii de adăugare înregistrări într-o tabelă a unei baze de date. Este vorba despre instrucțiunea **INSERT**. De asemenea, vom prezenta, mai departe, și câteva exemple de utilizare:

```
INSERT INTO nume_tabelă (câmp1, câmp2, ..., câmpn) VALUES (valoare1, valoare2, ..., valoaren);
```

Instrucțiunea de adăugare de informații în baza de date mai are și alte forme pe care le-am menționat anterior, dintre care reluăm aici două care au o frecvență mai mare de folosire (formele în care se inserează date în toate coloanele unei tabeli și, în această situație, nu mai trebuie menționate coloanele tabeli, dar valorile trebuie inserate în ordinea în care se regăsesc coloanele în tabelă și varianta asemănătoare comenzii **UPDATE** în care se face inserarea prin specificarea explicită a coloanei și a valorii care se inserează în acea coloană):

```
INSERT INTO nume_tabelă VALUES (valoare1, valoare2, ..., valoaren);
```

```
INSERT INTO nume_tabelă SET nume_câmp1 = valoare1, ..., nume_câmpn = valoaren;
```

De asemenea, reluăm și sintaxa comenzii **INSERT** care permite inserarea mai multor înregistrări printr-o singură instrucțiune:

```
INSERT INTO nume_tabelă (câmp1, câmp2, ..., câmpn) VALUES (valoare11, valoare12, ..., valoare1n),  
(valoare21, valoare22, ..., valoare2n), ..., (valoaren1, valoaren2, ..., valoarenn);
```

În continuare, vom prezenta câteva exemple concrete de utilizare a comenzii **INSERT**, pentru a înțelege mai clar modul în care funcționează.

După ce am creat o tabelă într-o bază de date, următorul pas pe care îl facem este să populăm tabela respectivă cu date, deci vom insera înregistrări în tabelă. Așa cum am prezentat mai sus, comanda **INSERT** este folosită pentru această operație.

Considerăm tabela *angajați* ce conține informații despre angajații unei companii (nume, prenume, data nașterii, data angajării, salariul).

Comanda pentru crearea acestei tabeli este următoarea:

```
CREATE TABLE angajati(  
    id INT (11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    nume VARCHAR (70),  
    prenume VARCHAR (100),  
    data_nasterii DATE,  
    data_angajarii DATE,  
    salariu DOUBLE(7,2)  
);
```

În continuare vom introduce date în această tabelă:

```
INSERT INTO angajati (id, nume, prenume, data_nasterii, data_angajarii, salariu)  
    VALUES (null, 'Popescu', 'Maria', '1981-06-08', '2010-02-15', 2000);
```

Se observă că în câmpul *id* care este definit cu restricția **AUTO\_INCREMENT** nu este introdusă nici o valoare (apare null), deoarece se va insera automat pentru coloana id un număr întreg care crește la fiecare înregistrare (se auto incrementează). Altfel, dacă nu ar fi definită această proprietate de auto incrementare, în coloana care este cheie primară nu ar fi permisă inserarea de valori nule.

În cazul în care una din înregistrări este ștearsă nu se va alocă id-ul ei unei înregistrări nou introduse în tabelă. În memoria internă se păstrează valoarea la care a ajuns incrementul.

În câmpul *nume* și în câmpul *prenume* se introduc șiruri de caractere, în câmpurile *data\_nasterii* și *data\_angajarii* se introduc valori de tip **date**, iar în câmpul *salariu* valori de tip **double** (numere cu zecimală).

Însă, nu este obligatoriu ca toate câmpurile să fie prezente în instrucțiunea **INSERT** de introducere a datelor în tabelă, mai ales că nu am aplicat restricția **NOT NULL** pe câmpurile tabelii (doar câmpul **id**, care este cheie primară are această restricție, dar acest câmp se și auto incrementează și atunci nu trebuie specificat în comanda **INSERT**).

În continuare avem un exemplu în care introducem o înregistrare doar cu numele și prenumele unui angajat:

```
INSERT INTO angajati (nume, prenume)
VALUES ('Ionescu', 'George');
```

Iată și o instrucțiune care inserează mai multe înregistrări în tabelă (este omisă specificarea coloanelor, deci, se vor insera valori în toate coloanele tabelii, în ordinea corespunzătoare):

```
INSERT INTO angajati
VALUES
(null, 'Cristescu', 'Ionut', '1991-11-28', '2014-01-10', 1500),
(null, 'Georgescu', 'Elena', '1987-01-21', '2015-02-01', 1700),
(null, 'Popescu', 'Florin', '1986-04-16', '2014-07-01', 2000);
```

Această instrucțiune va introduce 3 înregistrări în tabelă. Observăm faptul că lipsește partea instrucțiunii în care sunt specificate câmpurile în care se introduc date. Acest lucru este posibil deoarece sunt inserate date în toate câmpurile tabelii. Atenție, însă, în momentul în care sunt completate valorile care se introduc, trebuie păstrată ordinea în care avem definite câmpurile (coloanele) în tabelă. În caz contrar, putem să avem erori la execuția comenzii (de exemplu, dacă încercăm să introducem un șir de caractere într-un câmp de tip **int**, sau o dată într-un câmp de tip **double**).

Dacă execuția instrucțiunii va genera o eroare, datele nu sunt inserate în tabelă, chiar dacă o parte din ele sunt date care corespund tipurilor specificate. Instrucțiunea este evaluată în întregime, dacă ea generează o eroare atunci nu se inserează nimic.

## 4.3 Instrucțiunea UPDATE

Modificarea (editarea, actualizarea) datelor stocate într-o tabelă a unei baze de date se realizează folosind instrucțiunea **UPDATE**.

Sintaxa unei instrucțiuni **UPDATE** este următoarea:

```
UPDATE nume_tabelă SET nume_câmp1 = valoare1 [, ..., nume_câmpn = valoaren]
[WHERE condiții]
[ORDER BY coloane]
[LIMIT număr_rânduri];
```

În continuare explicăm această instrucțiune folosită pentru actualizarea/modificarea datelor dintr-o tabelă a unei baze de date, precum și câteva exemple concrete de utilizare.

Considerând tabela *angajati* pe care am utilizat-o și la exemplul precedent, pentru actualizarea salariului angajatului cu id-ul 2 se va folosi comanda:

```
UPDATE angajati SET salariu = 2000
WHERE id = 2
```

În cazul în care, într-o instrucțiune de actualizare (modificare), lipsește clauza **WHERE** se vor modifica informațiile din toate înregistrările tabelii. Deci, trebuie să fim atenți atunci când



folosim această instrucțiune de actualizare deoarece, în cele mai multe din cazuri, nu se dorește actualizarea tuturor înregistrărilor dintr-o tabelă.

Pot fi actualizate valorile din mai multe câmpuri printr-o singură instrucțiune **UPDATE**. De asemenea, pot exista mai multe condiții care se doresc a fi îndeplinite pentru a realiza actualizarea (deci, clauza **WHERE** va avea mai multe condiții).

În comanda de actualizare **UPDATE** mai pot să apară clauzele **ORDER BY** și **LIMIT** care determină ordonarea sau sortarea (**ORDER BY**) ascendentă sau descendentă a înregistrărilor din tabelă în funcție de una sau mai multe coloane și aplicarea instrucțiunii de modificare la un număr limitat de înregistrări care este specificat în clauza **LIMIT**.

Astfel, considerând că tabela *angajati* conține câteva sute de înregistrări, dacă dorim actualizarea salariului la valoarea 2500 pentru primii 10 de angajați în funcție de vechimea lor vom folosi în instrucțiunea **UPDATE** clauza **ORDER BY** pentru ordonare ascendentă după data angajării și clauza **LIMIT**, după cum urmează:

```
UPDATE angajati SET salariu = 2500
    ORDER BY data_angajarii
    LIMIT 10;
```

Observăm că în instrucțiunea **UPDATE** nu mai există clauza **WHERE**, dar, totuși actualizarea nu se face pentru toate înregistrările tabelii, ci doar pentru primele 10 întrucât s-a specificat această limită prin clauza **LIMIT**. Recomandarea este, însă, ca în majoritatea situațiilor să utilizăm clauza **WHERE** într-o instrucțiune de actualizare, altfel, există riscul de a modifica valorile din toată tabela.

Așadar, cu toate că avem și această variantă de limitare la un anumit număr de înregistrări, totuși cea mai folosită formă a instrucțiunii **UPDATE**, este cea care conține una sau mai multe condiții care trebuie să fie îndeplinite pentru a realiza actualizarea datelor. În acest mod vom ști cu certitudine că nu au fost actualizate înregistrări care nu îndeplinesc condițiile dorite pentru a se realiza modificarea.

## 4.4 Instrucțiunea DELETE

Instrucțiunea folosită pentru ștergerea înregistrărilor din baza de date este **DELETE**.

Sintaxa instrucțiunii de ștergere a înregistrărilor dintr-o tabelă este următoarea:

```
DELETE FROM nume_tabelă
    [WHERE condiții]
    [ORDER BY coloane]
    [LIMIT număr_rânduri];
```

Iată un exemplu de folosire a acestei comenzi, considerăm că avem aceeași tabelă pe care am utilizat-o mai înainte, *angajati*, ștergerea înregistrării cu id-ul 3 se face prin următoarea comandă:

```
DELETE FROM angajati
    WHERE id = 3;
```

La fel ca în cazul instrucțiunii **UPDATE** și, într-o instrucțiunea **DELETE**, dacă lipsește clauza **WHERE**, care stabilește condiția ce trebuie să fie îndeplinită pentru a se executa ștergerea înregistrărilor, se vor șterge toate înregistrările din tabelă. Deci, trebuie folosită cu atenție această instrucțiune, astfel încât să fim siguri că am stabilit condițiile necesare a fi îndeplinite pentru a șterge anumite înregistrări.



În general, se evită folosirea instrucțiunii de ștergere din tabelele unei baze de date a unei aplicații aflată în utilizare. Comanda **DELETE** va fi utilizată atunci când știm sigur că datele respective nu ne mai sunt necesare în baza de date.

În concluzie, acestea sunt cele 3 instrucțiuni folosite pentru actualizarea datelor din tabelele unei baze de date. Sunt instrucțiuni cu sintaxă destul de simplă și cu o logică ușor de înțeles. Dacă instrucțiunea **INSERT** are mai multe forme, în schimb, instrucțiunile **UPDATE** și **DELETE** au o singură formă asemănătoare și ușor de înțeles și de utilizat.

Atenție, nu există o comandă care să anuleze rezultatul generat de executarea instrucțiunilor **INSERT**, **UPDATE** și **DELETE**. Astfel, mai ales, în cazul comenzilor **UPDATE** și **DELETE** trebuie să avem foarte mare grijă cum le utilizăm deoarece putem altera baza de date. Deci, neexistând o operație de anulare (undo), singura variantă pentru a reface baza de date dacă s-a executat greșit o comandă de actualizare sau de ștergere este să se restaureze o copie de siguranță (cea mai recentă), dacă există o astfel de copie de siguranță (backup) a bazei de date.

## 4.5 Ștergerea tuturor datelor dintr-o tabelă și resetarea auto incrementului

Comanda care se folosește pentru a șterge toate datele dintr-o tabelă este următoarea:  
**TRUNCATE TABLE** *nume\_tabelă*;

Această comandă, pe lângă ștergerea tuturor înregistrărilor din tabelă, va reseta și valorile din câmpul unei tabele care se incrementează automat. Astfel, în momentul în care se vor adăuga din nou informații, câmpul care are definită proprietatea de auto incrementare va începe să ia valori de la 1. Această instrucțiune poate fi utilă pentru curățarea datelor de test introduse într-o tabelă, înainte de a porni aplicația cu date reale în tabelele bazei de date.

Deși nu face parte dintre instrucțiunile de manipulare, ea fiind instrucțiuni de descriere sau de definire, deci face parte din **LDD**, se potrivește prezentarea ei, mult mai bine, în acest context, alături de instrucțiunea de manipulare **DELETE**, tocmai pentru a sesiza diferența dintre cele două – **DELETE** șterge înregistrările dar nu modifică alte valori care există definite pe anumite coloane din tabele în urma aplicării unor constrângeri sau modificatori, în timp ce **TRUNCATE**, pe lângă ștergere, va reseta și toate aceste valori.

## 4.6 Instrucțiunea SELECT

Cea de-a patra comandă care aparține limbajului de manipulare a datelor este comanda de regăsire a datelor din tabelele unei baze de date. Aceasta este comanda **SELECT** care realizează o selecție (regăsire) a datelor care îndeplinesc anumite condiții.

Sintaxa acestei comenzi a fost prezentată în prima parte a lecției, dar o vom relua și aici, urmând ca apoi să explicăm fiecare clauză care poate să apară într-o astfel de instrucțiune de interogare a tabelelor dintr-o bază de date.

Așadar, reluăm sintaxa instrucțiunii de regăsire a datelor, care a fost prezentată și în prima parte a lecției:

```
SELECT [DISTINCT] câmp1, câmp2, ..., câmpn [FROM nume_tabelă]  
    [WHERE condiții]  
    [GROUP BY câmp1 [, câmp2 ...]]  
    [HAVING condiții]  
    [ORDER BY câmp1 [ASC | DESC] [, câmp2 [ASC | DESC], ... ]
```

```
[LIMIT nr_rânduri];
```

După cum se observă din prezentarea completă a sintaxei, instrucțiunea **SELECT** are mai multe clauze pe care le vom explica în continuare. Ordinea în care apar clauzele este cea specificată în sintaxă, inversarea anumitor clauze va duce la apariția erorilor de sintaxă. Pot să lipsească din clauze, întrucât mare parte sunt opționale, dar atunci când apar în instrucțiune o parte dintre ele sau chiar toate, ele trebuie specificate, obligatoriu, în această ordine.

Clauzele care sunt plasate între paranteze drepte „[]” sunt opționale, pot să lipsească din instrucțiunea **SELECT**. Dacă aceste clauze sunt folosite ele sunt scrise fără paranteze drepte. Deci, parantezele sunt folosite doar în prezentarea sintaxei instrucțiunii cu înțelesul că acele clauze sunt opționale.

Pentru a utiliza comanda de regăsire a datelor, **SELECT**, trebuie să precizăm cel puțin două informații: ce anume dorim să selectăm și locația de unde dorim să selectăm. Deci, imediat după cuvântul cheie **SELECT** urmează enumerarea câmpurilor (coloanelor) din tabela din care dorim să le extragem.

Pentru extragerea datelor din toate câmpurile unei tabelă se folosește caracterul asterisc „\*” care reprezintă selectarea tuturor coloanelor dintr-o tabelă.

Instrucțiunea următoare va extrage și va afișa toate înregistrările din tabela *angajati*:

```
SELECT * FROM angajati;
```

Continuăm cu o instrucțiune în care este selectat un singur câmp (doar numele angajaților) dintr-o tabelă:

```
SELECT nume FROM angajati;
```

Specificarea mai multor coloane ale unei tabelă într-o instrucțiune **SELECT** se face prin separarea câmpurilor (coloanelor) tabelărilor prin virgulă. Pentru a selecta numele, prenumele și salariul angajaților stocate în tabela *angajati* se va utiliza următoarea instrucțiune **SELECT**:

```
SELECT nume, prenume, salariu FROM angajati;
```

## 4.7 Clauza WHERE

În continuare vom descrie clauza **WHERE** a instrucțiunii **SELECT**. Este o clauză opțională, dar este foarte des folosită și foarte importantă. În cazul în care lipsește clauza **WHERE** dintr-o interogare, atunci se vor afișa toate înregistrările din coloanele specificate în instrucțiunea **SELECT** din tabela respectivă.

În cele mai multe situații însă nu avem nevoie de extragerea tuturor înregistrărilor din tabelă, ci doar de acele înregistrări care îndeplinesc anumite condiții. Aceste condiții sunt specificate în clauza **WHERE** în cadrul instrucțiunii de regăsire a datelor. După clauza **WHERE**, într-o interogare sunt specificate diverse condiții ce se cer îndeplinite pentru a extrage anumite date.

## 4.8 Operatori folosiți în clauza WHERE

În interiorul clauzei **WHERE** putem folosi următorii operatori:

- **=** este operatorul de **egalitate**, poate fi egalitate între două coloane sau între valoarea dintr-o coloană și o valoare specificată;
- **!= sau < >** este operatorul **diferit de**, deci verifică dacă două coloane sunt diferite sau o valoarea dintr-o coloană este diferită de o anumită valoare specificată;
- **<** este operatorul **mai mic**, acest operator compară dacă valoarea dintr-o coloană este strict mai mică decât o valoare din altă coloană sau decât o valoare specificată;
- **<=** este operatorul **mai mic sau egal**, acest operator compară dacă valoarea dintr-o coloană este mai mică sau egală cu o valoare din altă coloană sau cu o valoare specificată;
- **>** este operatorul **mai mare**, acest operator compară dacă valoarea dintr-o coloană este strict mai mare decât o valoare din altă coloană sau decât o valoare specificată;
- **>=** este operatorul **mai mare sau egal**, acest operator compară dacă valoarea dintr-o coloană este mai mare sau egală cu o valoare din altă coloană sau cu o valoare specificată;
- **BETWEEN** – compară dacă valoarea dintr-o coloană se află în intervalul specificat în operatorul **BETWEEN**, practic, verifică dacă acea valoare din coloană se află între valorile specificate în **BETWEEN**; forma în care se folosește este **BETWEEN valoare\_minimă AND valoare\_maximă**;
- **IN** – acest operator testează dacă operandul se regăsește printre lista de valori care este specificată între paranteze; acest operator este folosit în forma următoare: **IN(valoare<sub>1</sub>, valoare<sub>2</sub>,...,valoare<sub>n</sub>)**; valorile testate cu operatorul **IN** pot fi obținute și printr-o instrucțiune **SELECT**, deci poate fi folosit în subinterogări;
- **IS NULL** – verifică dacă valoarea dintr-o coloană a tabelului este **NULL**;
- **IS NOT NULL** – verifică dacă valoarea dintr-o coloană a tabelului nu este **NULL**;

De asemenea, atunci când punem condiții pe anumite câmpuri (coloane) ce conțin date de tip șir de caractere, mai apare un operator, **LIKE**.

Acest operator este folosit pentru a verifica dacă valoarea de tip șir de caractere dintr-o coloană corespunde cu un șir de caractere specificat sau, putem folosi aici și caractere de înlocuire. Astfel avem caracterul de înlocuire „%” care are semnificația că găsește orice caracter, indiferent de câte ori apare.

De exemplu pentru a găsi toți angajații al căror nume începe cu litera **A** se poate scrie următoarea instrucțiune **SELECT**:

```
SELECT * FROM angajati WHERE nume LIKE 'A%';
```

Mai există un caracter de înlocuire a unui singur caracter de această dată. Este vorba de caracterul „\_”. Este mai rar folosit și acest caracter înlocuiește un singur caracter, nici mai mult, nici mai puțin. În schimb caracterul de înlocuire „%” poate să substituie un caracter, nici un caracter (zero caractere) sau oricât de multe caractere.

În continuare vom prezenta și alți operatori folosiți în clauze **WHERE** mai complexe în care punem mai multe condiții, deci combinăm mai multe condiții simple. Astfel, intervin operatorii logici:

- **AND (&&)** – operatorul „și” logic, va returna **adevărat (1)** dacă toți operandii sunt adevărați, respectiv **fals (0)** dacă cel puțin unul dintre operandii este fals;
- **OR (||)** – operatorul „sau” logic, va returna **adevărat (1)** dacă cel puțin unul dintre operandii este adevărat, respectiv **fals (0)** dacă toți operandii sunt falși;
- **NOT (!)** – operatorul de negare, va returna **adevărat (1)** dacă expresia negată este falsă, respectiv **fals (0)** dacă expresia negată este adevărată;

• **XOR** – operatorul „sau exclusiv” logic, dacă este folosit pentru compararea a doi operanzi va returna **adevărat (1)** dacă unul și numai unul din acești operanzi este adevărat iar celălalt fals, dacă ambii operatori sunt la fel rezultatul returnat va fi **fals (0)**; dacă avem mai mulți operanzi rezultatul returnat va fi **adevărat (1)** dacă avem un număr impar de operanzi a căror valoare de adevăr este **adevărat (1)**; în caz contrar rezultatul returnat va fi **fals (0)**.

## 4.9 Clauza GROUP BY

Clauza **GROUP BY** se folosește pentru a grupa datele din una sau mai multe coloane pe baza unor criterii. Scopul grupării datelor este calcularea de valori statistice pentru fiecare grup în parte. În acest caz rezultatul cererii va conține câte o linie pentru fiecare grup identificat.

În cazul în care în clauza **GROUP BY** apar mai multe coloane, un grup va fi construit din toate înregistrările care au valori comune pe toate coloanele specificate.

Datele dintr-o tabelă pot fi grupate în funcție de valorile dintr-o anumită coloană. Astfel, toate valorile egale dintr-o anumită coloană vor forma un grup. Prelucrările datelor din cadrul unui grup se pot face cu ajutorul funcțiilor agregate (funcții de grup), acestea acționând asupra datelor din fiecare grup.

Gruparea efectivă se realizează cu clauza **GROUP BY**, aplicată comenzii **SELECT**. În cazul în care dorim filtrarea interogării rezultate în urma unei grupări, nu se mai folosește clauza **WHERE**, ci există o nouă clauză, **HAVING**.

Datele din tabela rezultată în urma grupării după o anumită coloană, vor fi sortate după coloana care realizează gruparea.

Într-o instrucțiune **SELECT** putem avea:

- nume de câmpuri (sau expresii în funcție de acestea): în acest caz se va folosi valoarea primei linii din fiecare grup;
- funcții agregate: acestea vor acționa asupra tuturor valorilor coloanei din grup asupra cărora sunt aplicate;

În exemplul următor, dacă vrem să obținem numărul de angajați din fiecare departament (considerând că avem coloanele `id_dept` ce indică codul unui departament în care lucrează fiecare angajat și `id_angajat` ce păstrează codul fiecărui angajat) vom executa următoarea instrucțiune **SELECT**:

```
SELECT id_dept, COUNT(id_angajat) FROM angajati GROUP BY id_dept;
```

În acest exemplu am folosit și funcția **COUNT()** care numără toate înregistrările nenule din coloana `id_dept`, coloană ce conține id-ul fiecărui departament din baza de date.

## 4.10 Clauza HAVING

Dacă într-o instrucțiune **SELECT** folosim funcții de agregare și avem nevoie să punem condiții pe rezultatul obținut în urma utilizării acestor funcții, atunci vom folosi clauza **HAVING**. Mai simplu de reținut, clauza **HAVING** se folosește atunci când avem în instrucțiunea de regăsire **SELECT** funcții de grup.

Deci, așa cum clauza **GROUP BY** se utilizează atunci când este folosită o funcție de grup pentru a afișa rezultatul calculat (total, medie, minim, etc.) grupat în funcție de una sau mai multe coloane (în general, coloanele asupra cărora nu s-a aplicat o funcție de de grup și care au fost extrase în comanda **SELECT**), pentru impunerea de condiții asupra rezultatului returnat de o astfel de funcție este utilizată clauza **HAVING**.

Principalele funcții de grup sau funcții de agregare care se întâlnesc în limbajul **SQL** sunt următoarele:

- **COUNT()** – funcție de numărare;
- **SUM()** – funcție care returnează suma valorilor din coloana trecută ca argument;
- **MIN()** – funcție care returnează valoarea minimă din coloana trecută ca argument;
- **MAX()** – funcție care returnează valoarea maximă din coloana trecută ca argument;
- **AVG()** – funcție care returnează media aritmetică a valorilor din coloana primită ca argument.

Funcția **COUNT()** are mai multe forme:

- **COUNT(\*)** – întoarce numărul total de înregistrări din tabelă;
- **COUNT(expr)** – întoarce numărul de valori nenule pentru expresia primită ca argument;
- **COUNT(DISTINCT expr)** – întoarce numărul de valori distincte pentru expresia primită ca argument.

Funcția **SUM()** întoarce suma valorilor unor expresii care sunt primite ca argument de către funcție. Valorile nule nu sunt luate în considerare la calculul sumei. Dacă grupul pentru care se calculează suma este vid atunci rezultatul funcției **SUM()** va fi **NULL**.

Funcția **AVG()** întoarce media aritmetică a valorilor din expresia primită ca argument și poate primi ca argument o coloană a unei tabeli sau o expresie.

Funcția **MIN()** întoarce valoarea minimă dintr-o expresie primită ca argument.

Funcția **MAX()** întoarce valoarea maximă dintr-o expresie primită ca argument.

Funcțiile de grup **MIN()** și **MAX()** se pot aplica atât expresiilor numerice, cât și șirurilor de caractere. În cazul în care se aplică șirurilor de caractere se va folosi ordinea lexicografică pentru determinarea valorii minime, respectiv valorii maxime din expresie.

Dacă instrucțiunea **SELECT**, în care au fost utilizate funcții de agregare, nu conține clauza **GROUP BY**, atunci valoarea funcțiilor de agregare va fi calculată pentru întreaga tabelă specificată în clauza **FROM** a instrucțiunii de interogare a bazei de date.

Pentru exemplul anterior, dacă vrem doar afișarea departamentelor cu cel puțin 2 angajați, instrucțiunea **SELECT** precedentă se transformă astfel (apare în cadrul instrucțiunii și clauza **HAVING** după clauza **GROUP BY**):

```
SELECT id_dept, COUNT(id_angajat) FROM angajati
    GROUP BY id_dept
    HAVING COUNT(id_angajat) >= 2;
```

## 4.11 Alias

De asemenea, un câmp, o expresie sau o tabelă poate primi un **alias**. Un **alias** reprezintă o denumire prin care acea expresie poate fi utilizată în cadrul interogării. De exemplu, în instrucțiunea **SELECT** de mai sus, expresia **COUNT(id\_angajat)** poate primi un **alias**, adică îi putem asocia un nume pe care să-l folosim mai departe în comanda de regăsire a datelor.

Pentru a defini un **alias** unei expresii se folosește cuvântul cheie **AS** urmat de numele asociat acelei expresii, în cazul nostru putem asocia **alias-ul număr\_angajati** expresiei **COUNT(id\_angajat)**.

Prin urmare, instrucțiunea **SELECT** precedentă poate fi rescrisă astfel:

```
SELECT id_dept, COUNT(id_angajat) AS nr_angajati FROM angajati
    GROUP BY id_dept
    HAVING COUNT(id_angajat) >= 2;
```

Astfel, este mult mai clar de înțeles ce returnează funcția. În plus, rezultatul acestei interogări, care este o tabelă, va avea ca antet (cap de tabel) sau câmpuri ale tabelului rezultat coloanele **id\_dept** și **nr\_angajati**. Dacă nu asociem un **alias** expresiei de numărare, coloanele rezultate ar fi **id\_dept** și **COUNT(id\_angajat)**

## 4.12 Clauza ORDER BY

Rezultatele obținute în urma unei instrucțiuni **SELECT** pot fi ordonate în funcție de anumite câmpuri. Ordonarea acestor rezultate poate fi crescătoare sau descrescătoare. În cazul în care câmpurile folosite pentru ordonare sunt de tip șir de caractere, atunci ordonarea este alfabetică sau în ordine inversă a alfabetului.

Clauza utilizată pentru ordonarea datelor rezultate în urma unei selecții este **ORDER BY**, după această clauză se specifică numele câmpului după care se face ordonarea și tipul de sortare (crescător sau descrescător). Pentru sortare în ordine crescătoare avem cuvântul cheie **ASC**, iar pentru sortare descrescătoare avem cuvântul cheie **DESC**.

De asemenea, se poate face sortare după mai multe câmpuri. În cazul în care, în clauza **ORDER BY**, sunt specificate mai multe câmpuri sortarea se realizează astfel: se sortează datele după valorile din primul câmp, iar în cazul în care în acest câmp avem valori egale (identice) se trece la sortare după următorul câmp specificat în clauza **ORDER BY**, și așa mai departe pentru toate câmpurile din clauză. De asemenea, sortarea se poate face crescător după anumite câmpuri și descrescător după alte câmpuri. Sortarea implicită a unei interogări este crescătoare, deci, dacă dorim o sortare crescătoare nu este necesar să mai specificăm cuvântul cheie **ASC** după numele coloanei stabilită drept criteriu de sortare.

Dacă vrem să selectăm toți angajații din baza de date sortați după nume și prenume vom realiza următoarea interogare:

```
SELECT * FROM angajati
ORDER BY nume, prenume;
```

După cum se observă lipsește specificarea ordinii de sortare, deci, implicit, se consideră sortare în ordine crescătoare. Interogarea următoare este echivalentă cu cea anterioară, va returna aceleași rezultate:

```
SELECT * FROM angajati
ORDER BY nume ASC, prenume ASC;
```

În cazul în care se dorește o sortare descrescătoare a valorilor returnate de interogare, specificarea ordinii de sortare este obligatorie. Avem astfel, următorul exemplu:

```
SELECT * FROM angajati
ORDER BY nume DESC;
```

Următoarea interogare realizează o ordonare combinată, descrescătoare după nume și crescătoare după prenume, adică angajații sunt sortați după nume în ordine inversă, iar dacă există mai mulți angajați cu același nume se va realiza o ordonare a acestora după prenume, în ordine alfabetică:

```
SELECT * FROM angajati
ORDER BY nume DESC, prenume ASC;
```

## 4.13 Clauza LIMIT

Ultima clauză a unei instrucțiuni **SELECT** este **LIMIT**. Această clauză, dacă este folosită limitează numărul de înregistrări returnate de interogarea **SELECT**. În clauza **LIMIT** se poate



specifica fie un singur număr, care reprezintă numărul de înregistrări pe care instrucțiunea **SELECT** le va întoarce, în acest caz fiind returnate primele  $n$  înregistrări din totalul de înregistrări returnate, unde  $n$  este numărul specificat în cadrul clauzei **LIMIT**, fie se pot specifica 2 numere.

În acest caz, când se specifică două numere, primul reprezintă poziția de la care va începe returnarea înregistrărilor rezultate în urma interogării, iar cel de-al doilea număr reprezintă numărul de înregistrări care vor fi returnate (cu alte cuvinte poziția de unde începe și câte înregistrări vor fi returnate de interogare).

Clauza **LIMIT**, atunci când este utilizată, este întotdeauna ultima în cadrul unei instrucțiuni **SELECT**.

Afișarea primilor 10 angajați din tabela în care sunt salvați, ordonați alfabetic după nume, se realizează cu următoarea instrucțiune:

```
SELECT * FROM angajati
ORDER BY nume
LIMIT 10;
```

Sintaxa acesteia are una dintre următoarele două forme, așa cum am precizat și anterior:

- **LIMIT  $n$**  – din ceea ce s-ar afișa în mod normal, se afișează doar primele  $n$  linii (înregistrări);

- **LIMIT  $m,n$**  – din ceea ce s-ar afișa în mod normal, se afișează doar începând de la a  $m+1$ -a linie (înregistrare) un număr de  $n$  linii (înregistrări).

Important de reținut este faptul că prima linie este numerotată cu 0. Așadar, instrucțiunea exemplu prezentată mai sus ar putea fi rescrisă astfel:

```
SELECT * FROM angajati
ORDER BY nume
LIMIT 0,10;
```

Din tabela *angajati* vor fi selectate 10 înregistrări, începând de la poziția 0. Deci, prima înregistrare rezultată în urma unei selecții se află pe poziția 0.

Dacă am fi scris următoarea instrucțiune:

```
SELECT * FROM angajati
ORDER BY nume
LIMIT 1,10;
```

rezultatul întors ar fi tot 10 înregistrări, însă nu va fi afișat primul angajat, ci vor fi afișați angajații, începând cu al doilea în ordine alfabetică până la al 11-lea.

Dacă în tabela noastră presupunem că am avea 100 de înregistrări, afișarea ultimilor 10 angajați sortați în ordine alfabetică după nume s-ar realiza cu instrucțiunea:

```
SELECT * FROM angajati
ORDER BY nume
LIMIT 90,10;
```

Întrucât prima poziție este 0, dacă avem 100 de linii în tabela rezultat, atunci ultima înregistrare, cea de-a 100, se află la linia 99. Deci, forma corectă a clauzei **LIMIT** pentru cerința anterioară este **LIMIT 90,10**, iar nu **LIMIT 91,10**. A doua variantă ar fi afișat doar 9 înregistrări, întrucât începând cu linia 91 nu mai există 10 înregistrări în tabelă. Deci, atunci când numărul de înregistrări care ar trebui afișate, specificat în clauza **LIMIT** este mai mare decât numărul de înregistrări care există în tabelă, de la poziția (linia) dată, atunci se afișează toate înregistrările rămase. Nu va fi generată nici o eroare din faptul că nu mai sunt în tabelă

atâtea înregistrări câte au fost specificate în clauza **LIMIT** pentru afișare, ci vor fi afișate atâtea câte există.

## 4.14 Clauza DISTINCT

Într-o tabelă, unele coloane pot conține valori duplicate. Adică, pentru mai multe înregistrări, pe același câmp, vom avea aceeași valoare. Aceasta nu este o problemă, dar uneori vrem să extragem dintr-o tabelă doar valorile diferite (distincte) din tabelă. În acest caz se va folosi clauza **DISTINCT** în cadrul unei interogări **SELECT**. Astfel, în instrucțiunea **SELECT** mai apare un cuvânt cheie, și anume **DISTINCT**, plasat imediat după cuvântul cheie **SELECT**, după care trebuie specificat câmpul (sau câmpurile) pentru care valorile returnate trebuie să fie distincte (diferite).

Sintaxa este următoarea:

```
SELECT DISTINCT nume_câmp1 [nume_câmp2, ...] FROM nume_tabelă  
[WHERE ...];
```

Trebuie reținut și că această clauză **DISTINCT** poate fi utilizată și în cadrul funcțiilor de agregare. În acest caz, cuvântul cheie **DISTINCT** este utilizat ca argument al funcției de agregare. De exemplu, pentru a număra doar valorile distincte dintr-o coloană.

Un exemplu în acest sens ar fi următoarea instrucțiune **SELECT**, care afișează localitățile de domiciliu ale angajaților salvați în baza de date a unei companii, în tabela *angajati*. Este evident faptul că, există posibilitatea ca mai mulți angajați să aibă aceeași localitate de domiciliu. Deci, pentru a extrage toate localitățile din care avem angajați, vom folosi o instrucțiune **SELECT** în care vom avea specificată o clauză **DISTINCT** pentru câmpul *localitate*:

```
SELECT DISTINCT localitate FROM angajati;
```

Fără utilizarea clauzei **DISTINCT**, interogarea ar fi returnat un număr de rezultate egal cu numărul înregistrărilor din tabelă, iar localitățile care se regăsesc de mai multe ori în tabelă ar fi fost afișate de fiecare dată.

Următoarea instrucțiune va returna toate localitățile de domiciliu, la fel ca mai sus, dar va returna și județul pentru fiecare localitate în parte (în acest caz vor exista județe care se repetă):

```
SELECT DISTINCT localitate, judet FROM angajati;
```

Iată și un exemplu de folosire a clauzei **DISTINCT** ca argument într-o funcție de agregare. De exemplu, dacă într-o tabelă în care sunt salvate spre evidență facturile unor clienți ai unei companii, vrem să știm câți clienți au facturi emise de companie, avem nevoie de folosirea acestui argument, **DISTINCT**, în cadrul funcției **COUNT**:

```
SELECT COUNT(DISTINCT cod_client) FROM facturi;
```

Observăm că absența clauzei **DISTINCT** din cadrul funcției **COUNT** ar duce la numărarea tuturor înregistrărilor din tabela *facturi* unde câmpul *cod\_client* este nenul. Dar dacă am fi avut mai multe facturi emise aceluiași client, ceea ce este foarte posibil, rezultatul obținut ar fi fost alterat, adică nu ar fi corespuns cerinței noastre de a afla numărul de clienți unici pentru care există facturi emise de către companie.

## 4.15 Concluzii

În această lecție am tratat pe larg comenzile aparținând Limbajului de Manipulare a Datelor (**LMD**), iar accentul a fost pus pe instrucțiunea de regăsire a datelor pentru care au fost precizate și explicate toate clauzele posibile. În continuare vor fi tratate aspecte legate de



operatorii întâlniți în **MySQL**, o parte din ei au fost prezentați și în cadrul acestei lecții, precum și de funcțiile predefinite pe care **MySQL** le pune la dispoziția utilizatorilor.

## Tema sedinta 4

Creați o **bază de date** în care să se păstreze evidența profesorilor și a cursurilor pe care aceștia le predau. Presupunem că un profesor poate să predea mai multe cursuri, în timp ce un curs poate fi predat de mai mulți profesori. Baza de date se va numi **scoala**.

Tabela **profesori** reține informații de bază despre profesori: **id\_profesor**, **nume**, **prenume**, **localitate**, **data angajării** și **salariu**. Tabela **cursuri** conține informații despre cursurile predate: **id\_curs**, **denumire**, **durată (număr de lecții)**.

### Cerințe:

1. Scrieți instrucțiunea **SQL** pentru **crearea bazei de date**.
2. Stabiliți tipurile de dată și dimensiunile pentru fiecare câmp al celor două tabele și scrieți instrucțiunile **SQL** pentru crearea tabelelor **profesori** și **cursuri**.
3. Stabiliți tipul de relație care există între cele 2 tabele și realizați legătura între tabele.
4. **Introduceți înregistrări** în tabele (**minim 5** înregistrări în fiecare tabelă). Utilizați mai multe variante ale instrucțiunii **INSERT**.
5. Scrieți o instrucțiune **SQL** prin care să **modificați localitatea** profesorului cu **id-ul 3** din tabela **profesori**.
6. Scrieți o instrucțiune **SQL** prin care să **mutați cursurile predate de profesorul cu id-ul 1 la profesorul cu id-ul 4**.
7. Scrieți o instrucțiune **SQL** prin care să **majorați cu 20% salariul profesorilor din București**.
8. Scrieți o instrucțiune **SQL** prin care să **afișați profesorii din provincie** (câmpuri afișate: **nume**, **prenume**, **localitate**) **ordonați alfabetic după nume și prenume**.
9. Scrieți o instrucțiune **SQL** prin care să **afișați localitățile** existente în tabela **profesori**. **Nu afișați valorile duplicate** - (câmp afișat: **localitate**).
10. Scrieți o instrucțiune **SQL** prin care să **afișați primele 4 cursuri** existente în tabela **cursuri**, **ordonate după durată descrescător și după denumire alfabetic** (câmpuri afișate: **id\_curs**, **denumire**, **durată**).

Scrieți o instrucțiune **SQL** prin care să **ștergeți cursurile ce sunt alocate profesorului cu id-ul 2**.

## 5. Operatori si functii MySQL

### 5.1 Tipuri de operatori

În această lecție vom aborda subiectul referitor la **operatori** și **funcții predefinite** în **MySQL**. În ceea ce privește operatorii, în **MySQL** avem trei tipuri de operatori:

- **matematici;**
- **logici;**
- **de comparare;**

Expresiile care apar împreună cu acești operatori se numesc *operanzi*. Operanzii pot fi coloane ale tabelelor, valori sau expresii.

### 5.2 Operatori matematici

Operatorii matematici pe care îi întâlnim în **MySQL** sunt: **+**, **-**, **\***, **/**, **%**.

Astfel, „**+**” este operatorul pentru adunare, „**-**” este operatorul pentru scădere, „**\***” este operatorul pentru înmulțire, „**/**” este operatorul pentru împărțire iar „**%**” este operatorul pentru restul împărțirii a două numere (**modulo**). Dacă avem împărțire la 0, deci folosim operatorul „**/**”, atunci rezultatul va fi **NULL**. Pentru împărțire, pe lângă operatorul „**/**” mai poate fi utilizat și operatorul **DIV**. Diferența dintre operatorul „**/**” și operatorul **DIV** este următoarea: „**/**” va returna un număr fracționar dacă împărțirea nu este exactă, în timp ce **DIV** returnează câtul împărțirii primului operand la al doilea, deci, va returna împărțire întreagă.

Pentru obținerea restului împărțirii a două numere, pe lângă operatorul „%” mai poate fi utilizat și operatorul **MOD**. Cei doi operatori funcționează în același mod, nu sunt diferențe între ei.

## 5.3 Operatori logici

Operatorii logici utilizați în **MySQL** sunt: **AND**, **OR**, **XOR**, **NOT**.

În **MySQL** operatorii logici întorc rezultatul 1 pentru adevărat, respectiv rezultatul 0 pentru fals. O expresie evaluată din punct de vedere logic poate fi adevărată sau falsă.

Operatorul **AND** este operatorul „și logic”. În **MySQL** pentru „și logic”, mai avem și operatorul **&&**.

Dacă avem 2 expresii pe care le evaluăm logic, fiecare din aceste expresii poate fi adevărată sau falsă. Iată în continuare tabla de valori cu toate variantele posibile pentru 2 expresii, folosind operatorul **AND (&&)**.

**1 AND 1 = 1**  
**1 AND 0 = 0**  
**0 AND 1 = 0**  
**0 AND 0 = 0**

Cu alte cuvinte, tabla aceasta de valori a operatorului logic **AND** poate fi explicată astfel: dacă prima expresie este adevărată, deci întoarce rezultatul 1, iar cea de-a doua expresie este tot adevărată, atunci rezultatul **1 && 1** este tot 1, deci operatorul **AND** returnează 1 (adevărat) dacă ambele expresii sunt adevărate.

Dacă una dintre expresii este adevărată iar cealaltă este falsă, atunci rezultatul este fals, deci **1 AND 0 = 0** și **0 AND 1 = 0**.

Evident că, dacă avem ambele expresii false, **0 AND 0 = 0**.

Deci, operatorul **AND (&&)** va returna adevărat doar atunci când ambele expresii sunt adevărate, în orice alt caz rezultatul este fals.

Operatorul **OR** este operatorul „sau logic”. Mai avem pentru „sau logic” și operatorul **||**. În continuare prezentăm tabla de valori cu variantele posibile pentru 2 expresii evaluate împreună cu operatorul **OR**.

**1 OR 1 = 1**  
**1 OR 0 = 1**  
**0 OR 1 = 1**  
**0 OR 0 = 0**

Cu alte cuvinte, tabla aceasta de valori a operatorului logic **OR** poate fi explicată astfel: dacă prima expresie este adevărată, deci întoarce rezultatul 1, iar cea de-a doua expresie este tot adevărată, atunci rezultatul expresiei **1 || 1** este tot 1, deci operatorul **OR** returnează 1 (adevărat) dacă ambele expresii sunt adevărate.

Dacă una dintre expresii este adevărată iar cealaltă expresie este falsă, atunci rezultatul este adevărat, deci **1 OR 0 = 1** și **0 OR 1 = 1**. Practic, atunci când evaluăm valoarea de adevăr a două expresii logice între care am folosit operatorul **OR** rezultatul este 1 (adevărat) dacă cel puțin una din expresii este adevărată. Așadar, este suficient să fie una singură din expresii adevărată pentru ca rezultatul obținut să fie adevărat.

Evident că, dacă avem ambele expresii false, **0 OR 0 = 0**.

Deci, operatorul **OR** (||) va returna adevărat atunci când cel puțin una dintre expresii este adevărată, iar în cazul în care ambele expresii sunt false și rezultatul este fals.

Operatorul **XOR** este operatorul „sau exclusiv”. Acest operator returnează adevărat atunci când unul dintre operanzi este adevărat iar celălalt este fals. Atunci când ambii operanzi sunt adevărați sau ambii operanzi sunt falși, rezultatul este fals. Iată în continuare tabla de valori cu variantele posibile pentru 2 expresii evaluate folosind operatorul **XOR** (sau exclusiv):

**1 XOR 1 = 0**  
**1 XOR 0 = 1**  
**0 XOR 1 = 1**  
**0 XOR 0 = 0**

Deci, tabla de valori a operatorului logic **XOR** (sau exclusiv) poate fi explicată astfel: atunci când ambele expresii evaluate sunt fie adevărate, fie false, rezultatul este fals, în timp ce rezultatul adevărat va fi returnat doar atunci când una din expresii returnează adevărat iar cealaltă returnează fals.

Acest operator, **XOR**, este mai rar folosit în evaluarea valorii de adevăr a unor expresii.

Ultimul operator logic este **NOT**, operatorul de **negație**. În **MySQL** pentru negarea unei expresii mai avem și operatorul „!”.

Acest operator de negație este foarte simplu de folosit și de înțeles. Practic, dacă avem o expresie adevărată și o negăm (îi aplicăm acest operator **NOT**) ea devine falsă, și reciproc, dacă o expresie este falsă și ea este negată, atunci rezultatul expresiei va fi adevărat. Deci, tabla de valori posibile pentru acest operator este următoarea:

**NOT 1 = 0**  
**NOT 0 = 1**

## 5.4 Operatori de comparare

De asemenea, în **MySQL** mai avem și operatorii de comparare. În această categorie avem următorii operatori:

**<** - compară dacă o expresie este **mai mică** decât altă expresie;  
**>** - compară dacă o expresie este **mai mare** decât altă expresie;  
**<=** - compară dacă o expresie este **mai mică sau egală** decât altă expresie;  
**>=** - compară dacă o expresie este **mai mare sau egală** decât altă expresie;  
**=** - compară dacă două expresii sunt **egale**;  
**!=, < >** - compară dacă două expresii sunt **diferite**;

**LIKE** - testează dacă un șir de caractere are o anumită formă: dacă este prefixat respectiv postfixat sau nu de un anumit subșir, dacă acesta conține un anumit subșir. Important de reținut este și faptul că, în **MySQL**, simbolul „\_” (**underline**) ține loc unui singur caracter, în timp ce simbolul „%” ține loc oricâtor caractere. Acestea se mai numesc caractere de înlocuire.

**IS NULL** - testează dacă o valoare este **NULL**

**IS NOT NULL** - testează dacă o valoare nu este **NULL**

O mențiune importantă este aceea că atunci când se testează anumite expresii (valori, câmpuri ale unei tabeli) pentru a determina dacă valoarea lor este **NULL** nu putem folosi operatorii **=, !=, <, <=, >, >=**.

Pentru a testa dacă o expresie este nulă se folosesc exclusiv cei doi operatori prezentați anterior și anume **IS NULL**, respectiv **IS NOT NULL**.

**BETWEEN** - testează dacă o valoare se găsește între 2 valori date; forma este următoarea **expresie BETWEEN valoare\_minimă AND valoare\_maximă**; deci, **BETWEE**

**N** verifică dacă **expresie** se găsește în intervalul închis cu capetele **valoare\_minimă**, respectiv, **valoare\_maximă**;

Operatorul **BETWEEN** poate fi înlocuit cu operatorii **>=** și **<=**. Forma ar fi următoarea:

**expresie >= valoare\_minimă AND expresie <= valoare\_maximă**

Astfel, deducem mai limpede, că operatorul **BETWEEN** ia în considerare atunci când evaluează expresia inclusiv valorile limită ale intervalului, în acest caz denumite **valoare\_minimă**, respectiv, **valoare\_maximă**.

**IN(val<sub>1</sub>, ..., val<sub>n</sub>)** - testează dacă o valoare aparține unei mulțimi de valori trecută ca argumente între parantezele operatorului **IN**;

**NOT IN(val<sub>1</sub>, ..., val<sub>n</sub>)** - testează dacă o valoare nu aparține mulțimii de valori dată între parantezele operatorului.

Acești operatori au mai fost explicați la lecția anterioară la subcapitolul **Operatori folosiți în clauza WHERE**, unde puteți găsi prezentări lămuritoare pentru fiecare operator în parte.

## 5.5 Operatori de evaluare condiționată

Operatorul **CASE** poate returna diverse valori, în funcție de valorile unor expresii care sunt primite ca operand.

Important de reținut este să **nu** confundăm operatorul de evaluare condiționată **CASE** cu instrucțiunea decizională **CASE** pe care o vom reîntâlni în cadrul **rutinelor MySQL**, la lecțiile aferente **extensiei procedurale MySQL**, a procedurilor stocate.

Deși sunt asemănători ca formă, **operatorul CASE** se va utiliza în cadrul interogărilor **SQL**, pe când **instrucțiunea decizională CASE** este utilizată în **rutine** (programe) **MySQL**.

Operatorul de evaluare condiționată **CASE** are două forme pe care le prezentăm în continuare:

**CASE expresie**

```
    WHEN valoare1 THEN rezultat1
    WHEN valoare2 THEN rezultat2
    ...
    WHEN valoaren THEN rezultatn
    ELSE alt_rezultat
```

**END**

sau, cea de-a doua formă:

**CASE**

```
    WHEN expresie1 THEN rezultat1
    WHEN expresie2 THEN rezultat2
    ...
    WHEN expresien THEN rezultatn
    ELSE alt_rezultat
```

**END**

În prima formă valoarea returnată de **expresie** este comparată pe rând cu toate valorile din clauzele **WHEN**, adică valorile **valoare<sub>1</sub>**, **valoare<sub>2</sub>**, ..., **valoare<sub>n</sub>**. Dacă **expresie** este egală cu

una din aceste valori atunci operatorul **CASE** va produce rezultatul corespunzător ce urmează după clauza **THEN**, adică unul dintre *rezultat<sub>1</sub>*, *rezultat<sub>2</sub>*, ..., *rezultat<sub>n</sub>*. Dacă valoarea returnată de *expresie* nu este egală cu nici una dintre valorile cu care este comparată, atunci operatorul **CASE** va produce rezultatul prezent în clauza **ELSE**, adică *alt\_rezultat*, iar în caz de absență a clauzei **ELSE**, care nu este obligatorie atunci va returna **NULL**.

A doua formă verifică pe rând valoarea fiecărei expresii din clauza **WHEN**, adică *expresie<sub>1</sub>*, *expresie<sub>2</sub>*, ..., *expresie<sub>n</sub>*, și va returna rezultatul primei expresii care are valoarea de adevăr **TRUE** (este adevărată din punct de vedere logic).

În situația în care nici una dintre expresii nu va fi evaluată ca adevărată (**TRUE**), atunci se va returna rezultatul din clauza **ELSE**, iar în absența clauzei **ELSE**, care nu este obligatorie, se va returna **NULL**.

## 5.6 Funcții predefinite MySQL

În **MySQL** avem funcții simple care prelucrează fiecare înregistrare și pentru fiecare înregistrare returnează un rezultat și funcții de agregare (de grup) care prelucrează un set de înregistrări și returnează un singur rezultat pentru acel set de înregistrări.

O funcție primește și *parametri*. *Parametrii* unei funcții sunt valori pe care o funcție le primește pentru a le prelucra și a returna un rezultat. Parametrii unei funcții se mai numesc și *argumente*.

Putem clasifica funcțiile simple în mai multe tipuri:

- **matematice**;
- **de comparare**;
- **condiționale**;
- **pentru șiruri de caractere**;
- **pentru date calendaristice**.

În continuare, vom lua fiecare din aceste tipuri de funcții și vom prezenta câteva dintre cele mai utilizate funcții din fiecare tip.

## 5.7 Funcții matematice

Funcțiile matematice sunt folosite pentru efectuarea de operații matematice. Ele pot fi utilizate fie pentru realizarea de operații matematice cu numele coloanelor, fie cu valori specifice.

În continuare vom prezenta câteva dintre cele mai cunoscute funcții matematice disponibile în **MySQL**. Fiecare funcție primește unul sau mai mulți parametri care sunt trecuți între paranteze rotunde – „()”. Acești parametri pot reprezenta numele unor coloane sau numere. Astfel, avem funcțiile:

- **ABS(*n*)** – returnează modulul sau valoarea absolută a unui număr;
- **CEILING(*n*)** – returnează cea mai mică valoare întreagă mai mare ca *n*;
- **FLOOR(*n*)** – returnează cea mai mare valoare întreagă mai mică ca *n*;
- **POW(*a*,*b*)** – returnează rezultatul ridicării la putere, adică  $a^b$ , deci primul parametru reprezintă baza, iar cel de-al doilea exponentul;
- **ROUND(*n*)** – returnează valoarea rotunjită a numărului primit ca parametru, rotunjirea se face fără zecimale; dar această funcție poate primi și un al doilea parametru care reprezintă numărul de zecimale la care se face rotunjirea, deci,

mai avem forma **ROUND(*n,d*)** care va returna valoarea rotunjită a lui *n* cu *d* zecimale;

- **TRUNCATE(*n,d*)** – returnează valoarea tăiată (trunchiată) a lui *n* cu *d* zecimale; de exemplu **TRUNCATE(1.284,1)** va returna 1.2;
- **RAND()** – returnează un număr aleatoriu între 0 și 1; această funcție nu are parametri;
- **SQRT(*n*)** – returnează valoarea rădăcinii pătrate (radicalul) unui număr;
- **MOD(*a,b*)** – returnează restul împărțirii lui *a* la *b*; același rezultat se poate obține și folosind operatorul „%” – **modulo**;
- **CONV(*nr,bază\_inițială,bază\_transformare*)** – returnează un șir de caractere ce reprezintă rezultatul obținut în urma conversiei numărului *nr* din baza de numerație *bază\_inițială* în baza de numerație *bază\_transformare*.

## 5.8 Funcții de comparare

În continuare vom prezenta câteva funcții de comparare:

**LEAST(*val1,val2,...*)** – returnează cel mai mic parametru dintr-o listă de parametri; această funcție trebuie să aibă cel puțin 2 parametri; dacă unul din argumente este **NULL** atunci rezultatul este **NULL**;

**GREATEST(*val1,val2,...*)** – returnează cel mai mare parametru dintr-o listă de parametri; similar cu funcția **LEAST()** și această funcție trebuie să aibă cel puțin 2 parametri;

**INTERVAL(*n,n1,n2,...*)** – această funcție compară valoarea lui *n* cu setul de valori care urmează – *n1, n2, ...*; este necesar ca setul de valori ce va fi comparat cu *n* să fie ordonat crescător, deci *n1 < n2 < n3 ...*; funcția va returna 0 dacă *n < n1*, 1 dacă *n < n2*, deci va returna poziția pe care este găsită prima valoare mai mare decât primul parametru al funcției, considerând că primul parametru cu care se compară această valoare se află pe poziția 0, al doilea pe poziția 1; dacă *n* este **NULL** funcția va returna -1.

## 5.9 Funcții condiționale

Continuăm prezentarea cu funcțiile condiționale:

**IF(*expresie\_testată,expr1,expr2*)** – această funcție primește trei parametri, se verifică valoarea de adevăr a primului argument al funcției, **expresie\_testată**, dacă valoarea de adevăr este **TRUE (adevărat)** atunci funcția returnează cel de-al doilea parametru, **expr1**, altfel, adică valoarea de adevăr a evaluării expresiei de testat este **FALSE (fals)** atunci funcția va returna **expr2**.

**IFNULL(*expr1,expr2*)** – această funcție returnează **expr1** dacă **expr1** este diferit de **NULL** sau va returna **expr2** dacă **expr1** este **NULL**. Valoarea returnată de această funcție poate fi numerică sau șir de caractere, în funcție de contextul în care este folosită.

**NULLIF(*expr1,expr2*)** – această funcție compară cele 2 expresii primite ca parametri, dacă sunt egale funcția va returna **NULL**, iar dacă cele 2 expresii sunt diferite va returna **expr1**, deci va returna primul parametru primit.

## 5.10 Valoarea NULL

După cum am văzut în sintaxa comenzii care creează o tabelă (**CREATE TABLE**), după fiecare coloană a tabelului se pot trece niște specificatori. Ne vom ocupa aici de specificatorii **NULL** respectiv **NOT NULL**. Primul, care este și implicit (dacă nu-l trecem, se asumă automat că acea coloană are valoarea **NULL**), se referă la faptul că în coloana respectivă pot să apară și valori de tip **NULL**.

Astfel, dacă într-o comandă **INSERT** este omisă o valoare pentru o anumită coloană, în acea coloană se va trece automat valoarea **NULL**.

Al doilea specificator, **NOT NULL**, nu permite înregistrări cu valori **NULL** în acea coloană. În acest caz, dacă este omisă valoarea acelei coloane, în ea se va trece automat:

- 0 dacă este de tip numeric;
- șirul vid dacă acea coloană este de tip șir de caractere;
- 0000-00-00 dacă acea coloană este de tip dată calendaristică, etc.

Valoarea **NULL** reprezintă, de fapt, **lipsa unei valori**. Foarte important este de menționat aspectul că nu trebuie să se facă confuzie între valoarea **NULL** și 0, de exemplu pentru coloane numerice, sau șirul vid ("" ) pentru coloane de tip șir de caractere. Acestea sunt valori, și 0 și șirul vid sunt valori, pe când **NULL** specifică de fapt **lipsa unei valori** în acel câmp.

## 5.11 Funcții pentru șiruri de caractere

Funcțiile pentru șiruri de caractere sunt folosite atunci când se lucrează cu șiruri de caractere, deci parametrii acestor funcții pot fi coloane sau valori de tip șir de caractere. Astfel, prezentăm în continuare câteva dintre aceste funcții pentru text:

**CONCAT(s1,s2,s3,...)** – va concatena (alipi) toate șirurile date ca argumente ale funcției; dacă unul dintre argumente este **NULL** atunci rezultatul funcției va fi **NULL**.

**CONCAT\_WS(separator,s1,s2,s3,...)** – va concatena (alipi) toate șirurile date ca argumente ale funcției cu separatorul dat ca prim argument între ele; dacă separatorul este **NULL** atunci rezultatul funcției va fi **NULL**, dar dacă unul din celelalte argumente este **NULL**, atunci funcția va returna celelalte șiruri concatenate, ignorând **NULL**. **Atenție**, în **MySQL** **NU** există un operator de concatenare (ca în PHP, spre exemplu). Deci, în **MySQL**, pentru concatenare vom folosi una din aceste funcții.

**CHAR\_LENGTH(s)** – returnează lungimea șirului *s*;

**LENGTH(s)** - lungimea (nr. de caractere) șirului *s*;

**LPAD(s,lungime\_finală,șir\_completare)** – șirul *s* este completat la stânga cu șirul *șir\_completare* până ajunge la lungimea *lungime\_finală*;

**RPAD(s,lungime\_finală,șir\_completare)** – șirul *s* este completat la dreapta cu șirul *șir\_completare* până ajunge la lungimea *lungime\_finală*;

**LTRIM(s)** - întoarce șirul obținut din *s* prin eliminarea spațiilor inutile din stânga;

**RTRIM(s)** - întoarce șirul obținut din *s* prin eliminarea spațiilor inutile din dreapta;

**TRIM(s)** - întoarce șirul obținut din *s* prin eliminarea spațiilor inutile atât din dreapta cât și din stânga;

**SUBSTR(s,pos,nr\_caractere)** – întoarce din șirul *s*, un subșir începând de la poziția *pos*, subșir de lungime *nr\_caractere*; dacă lipsește al treilea parametru care reprezintă câte caractere vor fi extrase în subșir, atunci subșirul va extrage toate caracterele începând de la poziția *pos* până la final; și funcția **SUBSTRING** este similară funcției **SUBSTR**;

**LEFT(s,nr\_caractere)** – întoarce primele *nr\_caractere* din șirul *s*;



**RIGHT(s,nr\_caractere)** – întoarce ultimele *nr\_caractere* din șirul *s*;  
**LOCATE(s1,s)** – returnează poziția primei apariții a subșirului *s1* în șirul *s*, respectiv, 0 dacă *s1* nu se găsește în șirul *s*;  
**LOCATE(s1,s,pos)** – returnează poziția primei apariții a subșirului *s1* în șirul *s*, începând de la poziția *pos*; returnează 0 dacă *s1* nu se găsește în șirul *s*;  
**UPPER(s)** - întoarce șirul obținut din *s* prin convertirea tuturor literelor mici la litere mari;  
**LOWER(s)** - întoarce șirul obținut din *s* prin convertirea tuturor literelor mari la litere mici;  
**REPLACE(s,de\_înlocuit,înlocuitor)** – înlocuiește în șirul *s* toate aparițiile subșirului *de\_înlocuit* cu șirul *înlocuitor*;

## 5.12 Funcții pentru date calendaristice

O altă categorie de funcții **MySQL** este cea a funcțiilor pentru date calendaristice. Trebuie să ne amintim că singurul format de dată acceptat de **MySQL** este *an-lună-zi* (AAAA-LL-ZZ), sau, mai cunoscut acest format după denumirea în limba engleză *year-month-day*, sau prescurtarea YYYY-MM-DD. Formatul în care se păstrează ora în baza de date este *oră-minut-secundă* (HH-MM-SS), format mult mai cunoscut după denumirea în limba engleză *hour-minute-second* sau după prescurtarea HH-MM-SS. Formatul în care se stochează atât data cât și ora este *year-month-day hour-minute-second* (AAAA-LL-ZZ HH-MM-SS sau YYYY-MM-DD HH-MM-SS).

Astfel avem următoarele funcții în această categorie:

- **CURDATE()** – returnează data curentă;
- **CURRENT\_DATE()** - returnează data curentă;
- **CURTIME()** – returnează ora curentă;
- **CURRENT\_TIME()** – returnează ora curentă;
- **CURRENT\_TIMESTAMP()** – returnează data și ora curentă;
- **NOW()** – returnează data și ora curentă;
- **YEAR(data)** – returnează anul din data introdusă ca parametru;
- **MONTH(data)** – returnează valoarea numerică a lunii din data primită ca argument (valori posibile de la 1 la 12);
- **DAY(data)** - returnează valoarea numerică a zilei din data primită ca argument (valori posibile de la 1 la 31);
- **DAYOFMONTH(data)** - returnează valoarea numerică a zilei din data primită ca argument (valori posibile de la 1 la 31);
- **HOUR(time)** – returnează ora din valoarea parametrului funcției (intervalul de valori este de la 0 la 23 dacă parametrul reprezintă o oră dar poate lua și valori mai mari dacă parametrul reprezintă un timp contorizat – număr de ore, minute și secunde contorizate);
- **MINUTE(time)** - returnează minutul din valoarea parametrului funcției (intervalul de valori care poate fi returnat este de la 0 la 59);
- **SECOND(time)** - returnează seunda din valoarea parametrului funcției (intervalul de valori care poate fi returnat este de la 0 la 59);
- **DAYNAME(data)** - returnează numele, în limba engleză, al zilei săptămânii din data care este trecută ca parametru;

- **MONTHNAME(data)** - returnează numele, în limba engleză, al lunii din data care este primită ca parametru;
- **DAYOFWEEK(data)** – returnează indexul zilei din săptămână din data primită ca parametru (returnează 1 pentru duminică, 2 pentru luni, ..., 7 pentru sâmbătă);
- **DAYOFYEAR(data)** - returnează indexul zilei din anul din data primită ca parametru (valoare posibilă returnată de la 1 la 366);
- **LAST\_DAY(date)** – returnează ultima zi din luna datei primită ca parametru;

## 5.13 Funcții de agregare

De asemenea, în **MySQL** mai avem o categorie de funcții care operează asupra mai multor înregistrări (linii) dintr-o tabelă a unei baze de date, calculând și returnând o singură valoare. Aceste funcții poartă denumirea de funcții agregat sau funcții de agregare. Avem funcții de agregare matematice și o funcție agregat pentru numărare.

Funcțiile de agregare matematice sunt:

- **MIN(coloană)** – returnează valoarea minimă pentru un set de înregistrări din coloana primită ca parametru;
- **MAX(coloană)** – returnează valoarea pentru un set de înregistrări din coloana primită ca parametru;
- **SUM(coloană)** – returnează suma valorilor din coloana specificată ca parametru;
- **AVG(coloană)** – returnează media aritmetică a valorilor din coloana specificată ca parametru;

Prezentăm în continuare și funcția de agregare de numărare. Este vorba de funcția:

- **COUNT(coloană)** – returnează numărul de înregistrări nenule din coloana primită ca parametru pentru un set de înregistrări;

Aceste funcții de agregare au fost prezentate pe larg, cu exemple concrete de utilizare, în cadrul lecției numărul 4 la prezentarea clauzelor instrucțiunii **SELECT**. În instrucțiunea de regăsire a datelor, instrucțiunea **SELECT**, pentru a utiliza funcțiile de agregare trebuie folosită clauza **HAVING**, deci, funcțiile de agregare trebuie folosite împreună cu această clauză **HAVING**.

În acest capitol au fost prezentate doar o parte dintre funcțiile predefinite ale limbajului **MySQL**. Au fost selectate câteva dintre cele mai des utilizate funcții din fiecare categorie. Lista completă a operatorilor și a tuturor funcțiilor predefinite pe care îi pune la dispoziție limbajul **MySQL** se găsește în documentația oficială la adresa următoare: <http://dev.mysql.com/doc/refman/5.7/en/functions.html>

De asemenea, și celelalte sisteme de management al bazelor de date (**Oracle**, **SQL Server**) oferă propriile funcții predefinite specifice fiecărui limbaj în parte, unele dintre ele identice cu cele din **MySQL**, altele foarte asemănătoare cu acestea, deci, cunoscând operatorii și funcțiile predefinite ale **MySQL** va fi destul de ușoară acomodarea și înțelegerea operatorilor și a funcțiilor predefinite din alte limbaje de baze de date.

## 5.14 Concluzii

Pe parcursul acestei lecții am prezentat operatorii întâlniți în **MySQL** și câteva dintre cele mai cunoscute funcții predefinite ale acestui **SGBD**.

În lecția următoare ne vom ocupa de **uniunile** între tabele sau **join-uri**, tipuri de **join-uri**, precum și de reuniuni. Așadar, vom trece la noțiuni mai complexe ale **MySQL**.

## Tema Sedinta 5

În fișierul de la link-ul **Baza de Date Exercitii Lectia 5** sunt instrucțiunile pentru crearea bazei de date **classicmodels** și a tabelelor din această bază de date.

### Cerințe

1. Rulați instrucțiunile din fișier pentru crearea bazei de date și a tabelelor.
2. Scrieți o instrucțiune **SQL** prin care să **afișați numărul de produse pentru fiecare linie de produse**(câmpuri afișate: **productLine**, și **număr de produse**).
3. Scrieți o instrucțiune **SQL** prin care să **afișați ultima comandă anulată, care are statusul Cancelled**(câmpuri afișate: **toate coloanele din tabela orders**).
4. Scrieți o instrucțiune **SQL** prin care să **afișați numărul de comenzi pentru fiecare status** (câmpuri afișate: **status** și **număr de comenzi**).
5. Scrieți o instrucțiune **SQL** prin care să **afișați numărul de clienți unici care au efectuat comenzi în anul 2004** (câmpuri afișate: **număr de clienți**).
6. Scrieți o instrucțiune **SQL** prin care să **afișați numărul de clienți din fiecare țară** (câmpuri afișate: **country**și **număr de clienți**).
7. Scrieți o instrucțiune **SQL** prin care să **afișați clienții al căror nume (contactLastName) are unul din sufixele 'en' sau 'on'** (câmpuri afișate: **contactLastName**, **contactFirstName**, **phone**). Rezultatele vor fi ordonate alfabetic după **contactLastName** și **contactFirstName**.
8. Scrieți o instrucțiune **SQL** prin care să **afișați comenzile plătite vara** (câmpuri afișate: **orderNumber** și **shippedDate**).
9. Scrieți o instrucțiune **SQL** prin care să **afișați clientul cu numărul de telefon format din cele mai multe caractere** (câmpuri afișate: **contactLastName**, **contactFirstName**, **phone**, **numărul de caractere din numărul de telefon**).
10. Scrieți o instrucțiune **SQL** prin care să **afișați produsul cel mai ieftin de la fiecare furnizor** (câmpuri afișate: **productVendor** și **cel mai mic preț**).

## 6. Uniuni de tabele

Pentru a putea selecta date din două sau mai multe tabele trebuie să unim acele tabele, deci, introducem noțiunea de uniune de tabele. Întrucât ne ocupăm de baze de date relaționale, știm că tabelele din baza de date sunt legate între ele (sunt relaționate). Nu este indicat să avem tabele izolate într-o bază de date. Pe parcursul acestui curs am făcut referire în mai multe rânduri la relaționarea dintre tabelele unei baze de date. De asemenea, au fost prezentate și tipurile de relații posibile între tabele.

Așa cum am spus, extragerea datelor din două sau mai multe tabele ale unei baze de date se realizează prin crearea unei uniuni între aceste tabele. Uniunile dintre tabele poartă și numele de **join**. De altfel, sub această denumire, de **join**, sunt cel mai mult cunoscute.

Rezultatul returnat de o interogare **SELECT** în care se extrag date din mai multe tabele, deci un **join** sau o uniune de **tabele** va fi o singură tabelă. Deci, printr-o singură interogare extragem date din mai multe tabele, iar rezultatul returnat va fi o singură tabelă.

Există multe situații în care este necesară extragerea informațiilor din mai multe tabele. Pentru acest lucru trebuie realizată o relație între tabele pe baza unor informații comune. Această interogare a bazei de date în care se realizează relații între tabele pe baza unor informații comune (câmpuri comune, coloane comune) se numește **joncțiune (JOIN)**. Cu alte cuvinte, tabelele sunt unite în cadrul interogărilor de regăsire a datelor.

### 6.2 Alias-uri de tabele

Pentru tabelele care alcătuiesc uniunea se pot stabili **alias**-uri de nume, sub forma **nume\_tabelă as alias\_num**, introduse în clauza **FROM** a instrucțiunii **SELECT**. **Alias**-urile pot fi utilizate în orice parte a instrucțiunii **SELECT**. Adresarea unei coloane a unei tabele se va face, atunci când tabela are un **alias**, sub forma **nume\_alias.nume\_coloană**.

Este recomandat să asigurăm nume scurte pentru **alias**-urile tabelor. Un alt avantaj al **alias**-urilor este că putem realiza auto-uniuni, în cadrul cărora aceeași tabelă poate avea **alias**-uri diferite. Acest aspect l-am explicat și când am prezentat **SELF JOIN**. Utilizarea unui **alias** duce și la o scurtare a sintaxei **SQL** într-o comandă.

Pe lângă **alias**-urile pentru tabele se mai pot folosi **alias**-uri și pentru coloanele unei tabele, dar și pentru câmpuri cu valoare calculată sau pentru câmpuri rezultate prin aplicarea unei funcții de agregare.

Expresiile sau coloanele dintr-o interogare **SELECT**, vor trebui să conțină numele coloanelor din tabele, dacă nu există ambiguități ce pot crea confuzii (coloane care au în ambele tabele același nume) sau, în caz contrar, adică în caz că există confuzii, vor conține numele tabelor sau **alias**-urile acelor tabele, din care fac parte coloanele, urmate de caracterul „.” (**punct**) și de numele coloanelor.

## 6.3 Tipuri de join-uri

Există mai multe *tipuri de join-uri*. O clasificare a lor s-ar putea face în două mari categorii:

- **JOIN-uri fără condiții** sau **asocieri fără restricții** (sau **fără clauza WHERE** dacă ne referim la **condiția de asociere** din **clauza WHERE** a interogării);
- **JOIN-uri cu condiții** sau **asocieri cu restricții** (sau **cu clauză WHERE** dacă ne referim la **condiția de asociere** din **clauza WHERE** a interogării).

## 6.4 Asocieri de tabele fără restricții (CROSS JOIN)

În prima categorie, a *asocierilor de tabele fără restricții* sau a *joncțiunilor fără condiții*, avem **joncțiunea încrucișată** (sau **CROSS JOIN**) în care sunt extrase coloane din două sau mai multe tabele, fără însă a avea specificate condiții de egalitate pe coloanele comune acestor tabele. Acest tip de *join* va returna **produsul cartezian** obținut din încrucișarea înregistrărilor din acele tabele specificate în clauza **FROM** a instrucțiunii **SELECT**.

Astfel, observăm că acest **tip de join** nu este unul util, el trebuie evitat pentru că obținerea unui produs cartezian nu duce la un rezultat care să satisfacă anumite condiții ce au impus utilizarea asocierilor de tabele (JOIN-urilor).

Practic, o instrucțiune **SELECT** fără condiții de egalitate pe coloana comună (condiția de realizare a JOIN-ului) care extrage date din două tabele ale unei baze de date va duce la obținerea unui produs cartezian, adică, fiecare înregistrare din prima tabelă este combinată (încrucișată, împerecheată) cu toate înregistrările din cea de-a doua tabelă.

La uniunile de tip produs cartezian se realizează o selecție în care se trec toate câmpurile pe care dorim să le obținem, din cadrul ambelor tabele, iar la clauza **FROM** se trec ambele tabele, fiecare dintre ele putând avea definit și un **alias**, adică un nume mai scurt prin care pot fi accesate.

Nu există o limită în ceea ce privește numărul de tabele ce pot fi unite în cadrul unei instrucțiuni **SELECT**. Așadar, putem avea două sau oricât de multe tabele unite printr-o **joncțiune**. Însă, cu cât avem mai multe tabele legate într-un **join**, iar tabelele sunt populate cu date numeroase, timpul de răspuns la o anumită solicitare poate fi mai mare.

Această asociere se mai numește și **CARTESIAN JOIN**, adică **asociere tip produs cartezian**.

Sintaxa unei asocieri de tabele fără restricții (CROSS JOIN este următoarea):

```
SELECT column_name(s)
FROM table1 [AS t1]
CROSS JOIN table2 [AS t2];
```

O altă formă a instrucțiunii **SELECT** care realizează o asociere de tip cartezian este cea în care tabelele sunt enumerate și separate prin virgulă în clauza **FROM** a instrucțiunii **SELECT**, dar **lipsește clauza WHERE** a instrucțiunii (în care ar putea să apară condiția de asociere, de unire). De aici și denumirea de **JOIN fără clauză WHERE** sau fără condiții de asociere:

```
SELECT column_name(s)
FROM table1 [AS t1], table2 [AS t2];
```

## 6.5 Asocieri de tabele cu restricții

A doua categorie de *JOIN-uri*, după cum am realizat clasificarea de mai sus, și anume, *JOIN-uri cu clauză WHERE*, sau cu condiții de asociere între tabele sunt cele care au specificate condițiile de egalitate între coloanele comune ale tabelor extrase în clauza **FROM** a instrucțiunii **SELECT**.

Condițiile de egalitate (sau de realizarea a **join**-ului) pot fi plasate în cadrul clauzei **WHERE** (de unde și această clasificare simplistă în *join-uri cu clauză WHERE* și *fără clauză WHERE*) dacă se folosește această sintaxă pentru realizarea **join**-ului (**sintaxa cu clauză WHERE este utilizată de toate SGBD-urile**) sau în clauza **ON** dacă se folosește **sintaxa de ANSI SQL** (adică standardul general valabil acceptat și recunoscut de toate sistemele gestione a bazelor de date (**SGBD**) bazate pe **standardul SQL – Structured Query Language**).

### 6.5.1 Clasificare

În această categorie, a **asocierilor de tabele cu restricții (join-uri cu condiții de asociere)**, distingem următoarele **tipuri de join-uri**:

- **INNER JOIN** (**joncțiunea simplă**);
- **OUTER JOIN** (**joncțiunea externă**);
- **SELF JOIN** (**auto-uniunea** sau *joncțiunea unei tabele cu ea însăși*).

Toate aceste tipuri de **join-uri** au fost prezentate pe larg în prima parte a cursului în care au fost implementate și foarte multe exemple pentru înțelegerea fiecărui *tip de join* în parte. În acest modul vom trece în revistă, succint, aceste *joncțiuni*, scopul fiind acela de a ne reaminti ce reprezintă fiecare tip în parte, cum și în ce context se folosește.

## 6.6 INNER JOIN

Cel mai folosit tip de **join** este reprezentat de **joncțiunea simplă** sau **INNER JOIN** care reprezintă **selectarea înregistrărilor comune tabelor** specificate în clauza **FROM** a instrucțiunii **SELECT**.

Este vorba, practic, de **intersecția valorilor** din tabelele care sunt unite (deci, valorile comune). Dacă sunt linii (înregistrări) în prima tabelă care nu au corespondent în cealaltă, ele nu sunt extrase (selectate).

În continuare va fi prezentată o **reprezentare grafică a acestui tip de join (INNER JOIN)**, pentru o înțelegere mai bună a ceea ce reprezintă.

Există mai multe forme în care poate fi utilizată instrucțiunea **SELECT** în care se realizează **INNER JOIN** între două sau mai multe tabele ale unei baze de date relaționale.

## 6.6.1 Forme posibile de utilizare INNER JOIN

Avem, astfel, **sintaxa standard ANSI SQL**, sintaxă care **funcționează în orice sistem de gestiune a bazelor de date (SGBD)**:

```
SELECT column_name(s)
FROM table1 [AS t1]
[INNER] JOIN table2 [AS t2]
ON table1.column_name=table2.column_name;
```

*În cadrul sintaxei unei instrucțiuni SQL, părțile cuprinse între paranteze drepte sunt opționale. Așadar, **tabelele din care se extrag informații pot primi sau nu alias-uri iar cuvântul cheie INNER este optional. Este suficient să avem cuvântul cheie JOIN între tabelele relaționate și se subînțelege că avem de-a face cu un INNER JOIN.***

*Alias-urile de tabele sunt folosite, în general, ca prescurtări ale numelor tabelelor, fiind utile atunci când este necesară prefixarea anumitor coloane cu numele tabelului din care face parte.*

De asemenea, realizarea unei joncțiuni (**INNER JOIN**) se poate face și prin scrierea condiției de egalitate între coloanele comune (*condiția care realizează JOIN-ul, relaționarea*) în cadrul clauzei **WHERE**.

În acest caz, avem sintaxa următoare:

```
SELECT column_name(s)
FROM table1 [AS t1], table2 [AS t2]
WHERE table1.column_name=table2.column_name;
```

Mai există o formă prin care se poate scrie instrucțiunea **SELECT** în care se realizează **join-ul** între tabele, utilizând **cuvântul cheie USING** urmat de numele coloanei de legătură între tabele (este vorba de coloană ce conține valori comune în cele 2 tabele).

**Condiția necesară pentru ca să poată fi utilizată această formă a instrucțiunii SELECT este ca numele coloanei de legătură între tabele să fie același în ambele tabele.**

De altfel, *această formă a instrucțiunii SELECT simplifică scrierea în cazul în care aceea coloană comună are același nume în tabelele relaționate.*

```
SELECT column_name(s)
FROM table1 [AS t1]
JOIN table2 [AS t2]
USING (column_name);
```

Am prezentat *sintaxa implementării unui INNER JOIN pentru două tabele. Se poate realiza asocierea mai multor tabele în cadrul unei **singure instrucțiuni SELECT**, sintaxa modificându-se doar prin adăugarea celorlalte tabele și condiții de egalitate între coloanele tabelelor, dar modul de gândire a instrucțiunii și de realizare a asocierilor de tabele fiind același.*

## 6.7 OUTER JOIN

*Al doilea tip de join este reprezentat de **joncțiunea externă** sau **OUTER JOIN** care, la rândul ei, se împarte în două tipuri de **OUTER JOIN**:*

- **LEFT OUTER JOIN;**
- **RIGHT OUTER JOIN.**

Cuvântul cheie **OUTER** poate fi omis în ambele tipuri de joncțiuni externe, fiind suficient să specificăm, ca tip de join, **LEFT JOIN** sau **RIGHT JOIN**, subînțelegându-se faptul că este vorba de **OUTER JOIN**.

## 6.7.1 LEFT OUTER JOIN

**LEFT JOIN** reprezintă uniunea de tabele în care sunt returnate atât înregistrările din prima tabelă (tabela din stânga, prima în ordinea de scriere a tabelelor în instrucțiunea **SELECT**) care au corespondent în cea de-a doua tabelă, cât și cele care nu au corespondent în cea de-a doua tabelă. În cazul în care înregistrările din prima tabelă nu au corespondent în cea de-a doua, pentru coloanele extrase din a doua tabelă se va returna **NULL**.

Așadar, în cazul joncțiunilor externe contează **ordinea** în care sunt scrise tabelele în cadrul instrucțiunii **SELECT**. De altfel, din punct de vedere al optimizării instrucțiunii **SELECT**, și în cazul **INNER JOIN** contează ordinea tabelelor în instrucțiunea de realizare a join-ului.

Astfel, indiferent de ordinea de scriere a tabelelor, în cazul **INNER JOIN**, se va returna același rezultat, dar optimizarea constă în faptul că, dacă prima (deci, cea din partea stângă) este trecută tabela cu mai puține înregistrări, atunci la realizarea join-ului este parcursă integral această tabelă, urmând ca din a doua tabelă (cea din dreapta) să fie preluate doar valorile corespondente.

În cazul în care este trecută ca primă tabelă cea cu mai multe înregistrări, rezultatul va fi același, doar că va crește timpul de execuție al instrucțiunii deoarece sunt parcurse toate înregistrările din această tabelă mai mare, urmând să fie luate valorile corespondente din tabela cu mai puține înregistrări.

Evident, timpul de execuție mai mare se va reflecta în cazul în care tabelele sunt foarte mari, conțin foarte multe înregistrări. Dacă tabelele au dimensiune mică (conțin un număr mic de înregistrări) diferențele în ceea ce privește timpul de execuție sunt nesemnificative.

În termeni de teoria mulțimilor, considerând cele două tabele ca fiind două mulțimi **A** și **B**, **LEFT JOIN** între cele două tabele (**A** și **B**) reprezintă  $(A \setminus B) \cup (A \cap B)$ . Deci, **LEFT JOIN** reprezintă rezultatul diferenței dintre cele două mulțimi reunit cu rezultatul intersecției celor două mulțimi.

Prezentăm în continuare **sintaxa ANSI SQL** a instrucțiunii **LEFT OUTER JOIN**, sintaxă ce funcționează în orice sistem de gestiune a bazelor de date:

```
SELECT column_name(s)
FROM table1 [AS t1]
LEFT [OUTER] JOIN
table2 [AS t2]
ON table1.column_name=table2.column_name;
```

La fel ca în cazul **INNER JOIN**, și la folosirea unei joncțiuni externe, cuvântul cheie **OUTER** poate fi omis din sintaxa instrucțiunii, prin specificarea termenilor **LEFT JOIN** sau **RIGHT JOIN** se subînțelege că este vorba despre o joncțiune externă de tip **LEFT OUTER JOIN** sau **RIGHT OUTER JOIN**.

Observăm faptul că, în **MySQL**, nu putem scrie acest tip de join utilizând forma instrucțiunii **SELECT** cu clauză **WHERE**, deoarece într-o astfel de formă nu se poate specifica faptul că este vorba despre o joncțiune externă.



Avem mai jos și **reprezentarea grafică** a joncțiunii de tip **LEFT OUTER JOIN** în scopul de a înțelege mai bine, din această reprezentare, ce înregistrări întoarce acest join:

## 6.7.2 RIGHT OUTER JOIN

În mod similar, **RIGHT JOIN** reprezintă toate înregistrările din tabela din partea dreaptă (a doua tabelă) care au corespondent sau nu în tabela din partea stângă (prima tabelă).

Cu alte cuvinte, **RIGHT JOIN** returnează același rezultat, cu deosebirea că se vor extrage înregistrările din a doua tabelă care au sau nu înregistrări în prima. Este, practic, reciproca joncțiunii **LEFT JOIN**.

**RIGHT JOIN** este mult mai puțin folosită deoarece se poate transforma foarte simplu într-un **LEFT JOIN** prin schimbarea ordinii celor două tabele. De aceea, forma cea mai utilizată a joncțiunilor externe este **LEFT JOIN**.

Instrucțiunea **RIGHT OUTER JOIN** este foarte asemănătoare cu **LEFT OUTER JOIN**. În cazul acestei instrucțiuni vor fi extrase din **tabela din dreapta** (deci, a doua în ordinea de scriere a tabelelor în clauza **FROM** a instrucțiunii **SELECT**) **atât înregistrările care au corespondent în prima tabelă (tabela din stânga), cât și cele care nu au corespondent în prima tabelă.**

**Sintaxa ANSI SQL** a instrucțiunii **RIGHT OUTER JOIN** este următoarea:

```
SELECT column_name(s)
FROM table1 [AS t1]
RIGHT [OUTER] JOIN
table2 [AS t2]
ON table1.column_name=table2.column_name;
```

La fel ca în situația **LEFT OUTER JOIN**, nici instrucțiunea **SELECT** ce va conține o *joncțiune externă* de tip **RIGHT OUTER JOIN** nu poate fi implementată, în **MySQL** prin utilizarea clauzei **WHERE** pentru stabilirea condițiilor de asociere între tabele.

**Reprezentarea grafică** a joncțiunii externe de tip **RIGHT OUTER JOIN** este redată în continuare pentru a înțelege mai bine ce înregistrări întoarce acest tip de *join*:

Așadar, se observă faptul că o asociere de tabele de tip **RIGHT OUTER JOIN** se poate transforma foarte ușor într-o asociere de tabele de tip **LEFT OUTER JOIN** prin **inversarea ordinii tabelelor** în clauza **FROM** a instrucțiunii **SELECT**.

Este valabilă și reciproca, adică, o joncțiune de tip **LEFT OUTER JOIN** se poate transforma într-o joncțiune de tip **RIGHT OUTER JOIN** prin **inversarea ordinii tabelelor** în clauza **FROM** a instrucțiunii **SELECT**.

## 6.8 SELF JOIN

Sunt situații în care, pentru a putea extrage anumite informații dintr-o tabelă, este nevoie să se realizeze o uniune a acelei tabele cu ea însăși. Avem, astfel, un alt tip de join, și anume **SELF JOIN**, adică **realizarea unei asocieri a unei tabele cu ea însăși**. Acest tip de uniune de tabele mai poartă numele de **auto-uniune**.

**SELF JOIN**-ul este, de fapt, tot un **INNER JOIN** sau un **OUTER JOIN**, dar în care, în clauza **FROM** avem o singură tabelă, care pentru a putea fi diferențiată va primi alias-uri diferite. Este vorba, deci, despre o redenumire temporară a tabelului în cadrul instrucțiunii **SELECT**.

**Sintaxa ANSI SQL** a unei joncțiuni de tip **SELF JOIN** este similară cu sintaxa unei joncțiuni de tip **INNER JOIN** sau **OUTER JOIN**, deosebirea fiind aceea că o singură tabelă, redenumită temporar prin utilizarea de alias-uri, participă la realizarea join-ului:

```
SELECT column_name(s)
FROM table1 AS t1
[INNER] JOIN | LEFT [OUTER] JOIN | RIGHT [OUTER] JOIN
table1 AS t2 ON t1.column_name=t2.column_name;
```

Se observă, așa cum am precizat anterior, că este tot un **INNER JOIN** sau un **OUTER JOIN (LEFT sau RIGHT)** în care, însă, fiind vorba de *participarea unei singure tabelă la realizarea asocierii de tabelă* (joncțiunii) **prezența alias-urilor pentru tabelă în cadrul instrucțiunii **SELECT** devine obligatorie**.

La implementarea unei auto-uniuni, mai poate fi întâlnită următoarea formă a instrucțiunii **SELECT**, formă în care *condiția de egalitate (adică, de realizare a asocierii) va fi plasată în clauza **WHERE*** (formă posibilă, în **MySQL**, doar pentru implementarea **INNER JOIN**):

```
SELECT column_name(s)
FROM table1 [AS t1], table1 [AS t2]
WHERE t1.column_name=t2.column_name;
```

Pentru o mai bună înțelegere, prezentăm și o **reprezentare grafică** a acestui tip de join. După cum se observă, **reprezentarea grafică a **SELF JOIN** este aproape identică cu cea de la **INNER JOIN**** (în mod similar, reprezentarea pentru o joncțiune externă unei tabelă cu ea însăși este cea prezentată la **LEFT**, respectiv, **RIGHT JOIN**, deosebirea fiind dată de faptul că ambele mulțimi (tabelă) participante la joncțiune au același nume (pentru că e vorba de aceeași tabelă) dar alias-uri diferite), deosebirea constă în faptul că ambele mulțimi (tabelă) participante la realizarea asocierii sunt de fapt una și aceeași, dar ele primesc un alias diferit pentru a permite identificarea lor în cadrul asocierii (join-ului):

## 6.9 Reuniuni. UNION și UNION ALL

Majoritatea interogărilor conțin o singură instrucțiune **SELECT**, care returnează date din una sau mai multe tabelă. Dar este permisă și efectuarea mai multor interogări și returnarea rezultatelor sub forma unui singur set de rezultate ale interogării. Aceste interogări combinate se numesc reuniuni sau interogări compuse.

Pentru a combina rezultatele obținute în urma execuției a două sau mai multe instrucțiuni **SELECT** se utilizează operatorul **UNION**. O astfel de instrucțiune va concatena (alipi) rezultatele returnate de fiecare instrucțiune **SELECT** în ordinea în care apar în cadrul instrucțiunii compuse ce utilizează **UNION**. Se realizează, de fapt, o reuniune a rezultatelor instrucțiunilor **SELECT** participante la instrucțiunea compusă.

Dacă interogările vor genera înregistrări duplicate, atunci, în urma utilizării operatorului **UNION**, rândurile care se repetă (sunt duplicate) vor fi eliminate. Deci, funcționează întocmai ca operația de reuniune pe mulțimi. Este posibilă păstrarea înregistrărilor duplicate prin folosirea **UNION ALL** în loc de **UNION** între interogările componente ale comenzi compuse.

În cazul utilizării operatorilor **UNION** sau **UNION ALL**, este necesar ca instrucțiunile **SELECT** conținute de comanda compusă să returneze același număr de coloane. Altfel, dacă instrucțiunile **SELECT** vor genera un număr diferit de coloane, se va întoarce o eroare.

Instrucțiunile **SELECT** participante la realizarea reuniunii pot fi oricât de complexe. *Ele pot conține toate clauzele cunoscute ale comenzii **SELECT**.*

Excepție face clauza **ORDER BY** care poate fi plasată doar la finalul instrucțiunii compuse, deci, la finalul ultimei instrucțiuni **SELECT**, deoarece clauza **ORDER BY** realizează o ordonare a rezultatelor afișate, iar ordonarea trebuie să fi pentru rezultatele obținute după aplicarea operatorului **UNION**.

Instrucțiunile **SELECT** participante la o astfel de comandă compusă pot conține și *asocieri de tabele (join-uri)*.

Deci, o reuniune trebuie să fie alcătuită din două sau mai multe instrucțiuni **SELECT**, separate prin cuvântul cheie **UNION** sau **UNION ALL**, iar fiecare instrucțiune **SELECT** trebuie să conțină același număr de coloane, expresii sau funcții agregat, dar nu contează ordinea în care sunt specificate coloanele. Coloanele din instrucțiunile **SELECT** trebuie să aibă tipuri de date compatibile.

## 6.9.1 Operatorul UNION

Operatorul **UNION** elimină în mod automat toate rândurile duplicate din setul de rezultate al interogării. Deci, dacă sunt înregistrări returnate de mai multe instrucțiuni **SELECT** dintr-o reuniune, este afișată o singură dată această înregistrare.

Sintaxa unei instrucțiuni în care se realizează o reuniune a datelor din două tabele prin utilizarea operatorului **UNION** este următoarea:

```
SELECT column_name(s) FROM table1 [AS t1] [...]  
UNION  
SELECT column_name(s) FROM table2 [AS t1] [...];
```

## 6.9.2 Operatorul UNION ALL

Operatorul **UNION ALL** nu elimină rândurile duplicate din setul de rezultate al interogării. Deci, dacă sunt înregistrări returnate de mai multe instrucțiuni **SELECT** dintr-o reuniune în care s-a folosit operatorul **UNION ALL**, vor fi afișate toate înregistrările.

**Sintaxa** unei comenzi ce conține operatorul **UNION ALL** este similară:

```
SELECT column_name(s) FROM table1 [AS t1] [...]  
UNION ALL  
SELECT column_name(s) FROM table2 [AS t1] [...];
```

## 6.9.3 Utilizarea clauzei ORDER BY într-o instrucțiune compusă de reuniune

Pentru ordonarea rezultatelor obținute în urma efectuării unei reuniuni, clauza **ORDER BY** se trece la final, după ultima instrucțiune **SELECT**.

Reluăm sintaxa instrucțiunii de reuniune de interogări, adăugând și clauza **ORDER BY**:

```
SELECT column_name(s) FROM table1 [AS t1] [...]
```

**UNION | UNION ALL**

```
SELECT column_name(s) FROM table2 [AS t1] [...]  
ORDER BY column_name(s) [ASC | DESC];
```

## 6.10 Concluzii

În această lecție am abordat noțiunile de **joncțiune** sau **uniune de tabele** și **reuniune**. Am prezentat tipurile de joncțiuni (**join-uri**) care se pot întâlni în **SQL**, și, de asemenea am discutat despre reuniunea interogărilor folosind operatorul **UNION**, respectiv, **UNION ALL**.

Așadar, începând cu această lecție am început să expunem lucruri mai complexe din domeniul bazelor de date. Lecția următoare continuă să abordeze teme cu o complexitate mai mare, și anume, vom prezenta conceptul de **subinterogare** și **tipurile de subinterogări**

## Tema Sedinta 6

În fișierul de la link-ul **Baza de Date classicmodels de la Ședința 5** sunt instrucțiunile pentru crearea bazei de date **classicmodels** și a tabelelor din această bază de date.

### Cerințe

1. Rulați instrucțiunile din fișier pentru crearea bazei de date și a tabelelor.
2. Scrieți o instrucțiune **SQL** prin care să **afișați numărul de angajați din fiecare birou** (câmpuri afișate: cod birou (**officeCode**), oraș (**city**) și număr de angajați).
3. Scrieți o instrucțiune **SQL** prin care să **afișați comenzile cu statusul Shipped efectuate de clienții din Franța (France) în anii 2004 și 2005** (câmpuri afișate: nume client într-o singură coloană (**contactLastName+contactFirstName**), țară (**country**), data comenzii (**orderDate**) și statusul comenzii (**status**)).
4. Scrieți o instrucțiune **SQL** prin care să **afișați cantitatea totală comandată pentru comenzile anulate (cu status Cancelled)** (câmpuri afișate: total cantitate comandată și statusul comenzii(**status**)).
5. Scrieți o instrucțiune **SQL** prin care să **afișați produsele comandate în luna mai 2005; eliminați duplicatele** (câmpuri afișate: cod produs (**productCode**), denumire produs (**productName**), luna din data comenzii (luna din **orderDate**) și anul din data comenzii (anul din **orderDate**)).
6. Scrieți o instrucțiune **SQL** prin care să **afișați totalul comenzilor produselor pentru fiecare furnizor** (câmpuri afișate: nume furnizor (**productVendor**) și total comenzi; total comenzi: **totalizarea preț\*cantitate comandată: priceEach\*quantityOrdered**).

**Rezultatele vor fi ordonate descrescător după totalul comenzilor, iar totalul va fi rotunjit la 2 zecimale.**

7. Scrieți o instrucțiune **SQL** prin care să **afișați toți clienții fără comenzi efectuate** (câmpuri afișate: nume client (**contactLastName+contactFirstName**), oraș (**city**), țară (**country**) și număr comenzi).

## 7. Subinterogari. Tipuri de subinterogari

### 7.1 Subinterogări

*O instrucțiune **SELECT** imbricată (inclusă) într-o altă instrucțiune **SELECT** poartă numele de **subinterogare** (**subquery**). Instrucțiunea **SELECT** imbricată, cea din interior mai poartă numele de **interogare secundară** sau de **subcere**.*

**MySQL** permite crearea de **subinterogări**, adică interogări care sunt înglobate în alte interogări. **Subinterogările** sunt mereu prelucrate pornind de la interogarea interioară înspre exterior. Deci, prima dată este evaluată instrucțiunea **SELECT** din interior (**subinterogarea**) și apoi interogarea exterioară.

Fiecare interogare conține în mod obligatoriu o clauză **FROM**. *Fiecare subinterogare trebuie să fie inclusă între paranteze rotunde pentru ca serverul de baze de date să o execute mai întâi.* Aceasta înseamnă că subinterogarea este, de fapt, o interogare ale cărei rezultate sunt transmise altei interogări. Subinterogarea este o modalitate de a face legături între două sau mai multe interogări.

Există multe situații în care pentru rezolvarea unei anumite cerințe pot fi utilizate fie interogări în care sunt folosite join-uri, fie interogări care conțin subinterogări.

*O **subinterogare (subquery)** poate fi plasată în una din următoarele clauze ale instrucțiunii **SELECT** principale: **WHERE**, **HAVING** sau **FROM**. În clauza **HAVING** va fi folosită subinterogarea atunci când condiția se va aplica unei **funcții de agregare**. Cele 5 funcții de agregare sau de grup, prezentate și exemplificate în modulul anterior, sunt: **COUNT**, **SUM**, **MIN**, **MAX** și **AVG**.*

De asemenea, o **subinterogare** poate fi folosită pentru obținerea de câmpuri cu valoare calculată, cum ar fi numărul de înregistrări care îndeplinesc o anumită condiție dintr-o altă tabelă decât cea din care se extrag celelalte câmpuri în instrucțiunea **SELECT** principală. În acest caz, în instrucțiunea **SELECT** de tip **subinterogare** trebuie prefixate numele câmpurilor cu numele tabelului pentru a nu exista neclarități.

Rezultatul obținut din **subinterogare** va primi un nume care poartă denumirea de **alias** care va fi folosit în instrucțiunea principală.

Întrucât pe parcursul acestui curs am făcut, în cele mai multe cazuri, referire la termenul de **interogare**, trebuie precizat că, de multe ori, se poate întâlni și termenul în limbă engleză ce definește o astfel de instrucțiune, este vorba despre termenul **query**.

Evident, în mod similar, pentru o **subinterogare** se poate întâlni frecvent și denumirea în limba engleză, și anume, **subquery**.

În cadrul interogării principale, pentru introducerea unei subinterogări se folosește un operator de comparație. Deci, se compară rezultatul returnat de subinterogare cu un rezultat obținut de interogarea **SELECT** principală.

Putem spune că o subinterogare reprezintă mecanismul prin care rezultatul întors de o interogare poate fi folosit mai departe, pentru a efectua o nouă interogare.

În concluzie, o subinterogare reprezintă o interogare inclusă într-o altă interogare. Rezultatul subinterogării este utilizat de **SGBD**, în cazul nostru **MySQL**, pentru a determina rezultatele interogării de nivel mai înalt (principale) care conține subinterogarea. Subinterogarea apare, în principal, în clauzele **WHERE** sau **HAVING** ale altei interogări, dar, așa cum am precizat anterior, pot fi și subinterogări plasate în clauza **FROM** a unei interogări.

Subinterogarea este cuprinsă între paranteze rotunde, dar are forma unei instrucțiuni **SELECT**, cu o clauză **FROM**, poate avea, de asemenea, clauzele **WHERE**, **GROUP BY**, **HAVING**.

*Numărul de interogări imbricate (subinterogări) nu este limitat, singura limită vine de la timpul de execuție al instrucțiunii **SQL**, care crește cu cât sunt mai multe subinterogări în aceeași*

*frază SQL*. Numărul de interogări imbricate depinde și de dimensiunea memoriei buffer (temporare) folosite de subinterogare. Interogarea principală și subinterogarea pot extrage date din tabele diferite sau din aceeași tabelă.

*Dacă o subinterogare nu returnează nici un rezultat sau returnează NULL, atunci interogarea principală, cea care afișează rezultatele, nu va returna nimic.* Este evident acest lucru, deoarece rezultatele returnate de subinterogare sunt utilizate apoi în instrucțiunea **SELECT** principală, ori, dacă subinterogarea nu returnează nimic sau returnează **NULL**, pe cael de consecință același lucru se va întâmpla cu instrucțiunea principală.

Clauza **ORDER BY** poate fi menționată într-o subinterogare, dar nu va influența rezultatele afișate de interogarea principală, deoarece rezultatele subinterogării nu sunt afișate, ci sunt utilizate intern în cadrul interogării principale, deci rezultatele unei subinterogări nu sunt vizibile pentru utilizator. Clauza **ORDER BY** plasată într-o subinterogare va influența doar rezultatele întoarse de acea subinterogare, dar nu și rezultatele afișate de interogarea principală. Pot fi ordonate, în schimb, și rezultatul va fi vizibil la afișare, înregistrările returnate de interogarea principală **SELECT**, care au prelucrat și rezultatele returnate de subinterogare.

Sintaxa unei subinterogări plasate în cadrul clauzei **WHERE** (precizăm că sintaxa unei subinterogări aflată în clauza **HAVING** sau în clauza **FROM** a comenzii principale este aceeași, diferă doar poziția în cadrul interogării principale), cele mai utilizate, a instrucțiunii **SELECT** principale este următoarea:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE column_name OPERATOR
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE] ...)
[...]
```

```
[ORDER BY column_name [ASC | DESC] [, column_name [ASC | DESC]]];
```

Operatorul folosit este unul de comparare, fie a egalității sau inegalității unor valori, al apartenenței la o listă de valori (operatorii **IN** sau **NOT IN**).

## 7.2 Tipuri de subinterogări

Putem clasifica subinterogările, din punct de vedere al numărului de valori și al valorilor returnate, astfel:

- **de tip scalar** – returnează o singură valoare (un singur rând și o singură coloană);
- **de tip listă** – returnează mai multe rânduri, dar o singură coloană;
- **de tip rând** – returnează un singur rând, dar mai multe coloane;
- **de tip tabelă** – returnează mai multe rânduri și mai multe coloane.

### 7.2.1 Subinterogări de tip scalar

Subinterogările de **tip scalar** întorc un scalar, adică o valoare atomică ce poate fi folosită ca o constantă într-o expresie **SQL**. Acest tip de **subinterogare** returnează o **singură coloană** și un **singur rând** dintr-o tabelă. Condiția este pusă pe cheia primară.

Interogarea care întoarce valoarea se scrie inclusă între paranteze rotunde „()”, din acest moment ea se comportă ca și cum, în acel loc, din punct de vedere sintactic, am avea o singură valoare.

## 7.2.2 Subinterogări de tip listă

În cazul celor de **tip listă** se va returna o **însiruire de valori** pentru o **singură coloană**. Deci, poate să returneze **mai multe rânduri**, dar, **doar o singură coloană**. În cadrul acestor subinterogări se folosesc operatorii **IN** sau **NOT IN**, operatori care verifică apartenența la o listă de valori, listă care a fost returnată de subinterogare.

## 7.2.3 Subinterogări de tip rând

Interogarea subordonată (subinterogarea) de **tip rând** se folosește pentru a verifica dacă liniile (înregistrările) extrase din interogarea secundară sau subordonată (subinterogare) există printre liniile (rândurile) extrase din interogarea principală. În acest scop, se folosește cuvântul cheie **EXISTS** care poate fi și negat, deci, se poate folosi și sub forma **NOT EXISTS**.

Sintaxa unei astfel de subinterogări, ce utilizează operatorul **EXISTS** sau negația acestuia, **NOT EXISTS**, este următoarea:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE EXISTS | NOT EXISTS
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE] ...)
[...]
```

```
[ORDER BY column_name [ASC | DESC] [, column_name [ASC | DESC] ]];
```

În acest mod, se verifică existența unor înregistrări rezultate din subinterogare printre rezultatele interogării **SELECT** principale.

7.2.4 În cazul subinterogărilor de tip tabelă, adică acele instrucțiuni **SELECT** secundare care întorc mai multe rânduri și mai multe coloane, subcererea se regăsește în clauza **FROM** a instrucțiunii **SELECT** principale. Deci, această subinterogare, întrucât va returna o tabelă *este necesar să aibă asociat un nume*, adică un alias. Acest alias primit de subcerere reprezintă *numele tabelii care va prefixa fiecare utilizare a unei coloane extrase în cadrul subinterogării*.

*Alias*-ul se specifică tot cu ajutorul clauzei **AS**, așa cum am expus în prezentarea **alias**-urilor descrisă în lecțiile anterioare. Trebuie, de asemenea, inclusă între paranteze rotunde „()”. Câmpurile tabelii întoarse de subinterogare au numele exact ca în antetul care s-ar afișa dacă ar fi executată această subinterogare.

Tabela rezultată dintr-o astfel de subinterogare poate fi utilizată pentru realizarea de uniuni cu alte tabele existente în baza de date, adică poate participa la realizarea de join-uri. Atunci când sunt folosite de către interogarea care o subordonează, câmpurile interogării subordonate trebuie adresate prin **alias**-ul tabelii, urmat de caracterul „.” și apoi de numele câmpului (coloanei) din subinterogare.



Sintaxa unei subinterogări de tip tabelă este următoarea:  
`SELECT ... FROM (subquery) [AS] nume_alias ...`

## 7.3 Operatori folosiți pentru introducerea subinterogărilor

Pe lângă operatorii de comparare foarte cunoscuți și care sunt folosiți și în condițiile unei instrucțiuni **SELECT** ce nu conține subcereri (subinterogări), adică: `=`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, **IN**, **NOT IN**, **BETWEEN**, etc. mai există o serie de operatori specifici subinterogărilor. Este vorba despre operatorii: **ANY**, **ALL** și **EXISTS**, respectiv, **negațiile acestora** care presupune apariția operatorului **NOT** în fața oricăruia dintre aceștia.

De asemenea, operatorii **ANY** și **ALL** se folosesc în combinație cu operatorii de comparare. Operatorul **ANY** înseamnă oricare element din listă, în timp ce **ALL** înseamnă toate elementele din listă. Operatorii **ANY** și **ALL** sunt operatori de cuantificare, se mai numesc și cuantificatori. Aceștia extind, practic, operatorii de comparare.

### 7.3.1 Operatorul ANY

Operatorul **ANY** este folosit **împreună** cu operatorii `=`, `>`, `<`, `>=`, `<=`, `!=` sau poate fi **negat**. El este utilizat pentru compararea cu oricare dintre valorile ce urmează după el, deci, valori ce sunt returnate de subinterogare. *Vor fi returnate acele valori care îndeplinesc condiția pentru măcar una din valorile din lista generată de subinterogare.* Poate fi și negat acest operator.

Operatorul **ANY** compară o valoare cu fiecare valoare returnată de subcerere și este suficient ca măcar o singură valoare întoarsă de subcere să îndeplinească condiția pentru ca valoarea din instrucțiunea principală care a fost comparată cu lista de rezultate din subinterogare să fie returnată ca rezultat al comenzii **SQL**.

Dacă valoarea acestei coloane coincide cu **vreuna** dintre valorile furnizate de subinterogare, condiția este evaluată la valoarea de adevăr **true (adevărat)**. Altfel, este evaluată ca fiind **false (falsă)**.

### 7.3.2 Operatorul ALL

Operatorul **ALL** este, la fel, utilizat **împreună** cu operatorii de comparare **specificați la ANY**. Utilizarea lui înseamnă compararea cu toate rezultatele întoarse de subinterogare. Vor fi returnate acele valori care *îndeplinesc condiția, fiind comparate cu toate valorile din lista întoarsă din subinterogare*. Poate fi negat și acest operator.

Operatorul **ALL** compară o valoare cu fiecare valoare returnată de subinterogare și trebuie ca toate valorile întoarse de subcerere să îndeplinească condiția de comparare pentru ca valoarea din instrucțiunea principală care a fost comparată cu lista de valori întoarsă din subinterogare să fie returnată ca rezultat al comenzii **SQL**.

Dacă valoarea acestei coloane coincide cu **fiecare** dintre valorile furnizate de subinterogare, condiția este evaluată la valoarea de adevăr **true (adevărat)**. Altfel, este evaluată ca fiind **false (falsă)**.

Operatorii **ANY**, respectiv, **ALL** se pot regăsi în clauzele **WHERE** sau **HAVING** ale unei instrucțiuni **SELECT**.

### 7.3.3 Operatorul EXISTS

Operatorul **EXISTS** *verifică dacă o valoare se regăsește în mulțimea de valori întoarsă de o subinterogare*. Dacă **valoarea există**, comanda va întoarce **TRUE**, iar **dacă valoarea nu există** în lista de valori returnate de subinterogare, atunci comanda va întoarce **FALSE**. Și acest operator poate fi negat.

Cu alte cuvinte, operatorul **EXISTS** verifică dacă subinterogarea returnează vreo linie. O subinterogare ce folosește operatorul **EXISTS** poate fi implementată și cu ajutorul operatorului **IN**, însă, prin aplicarea operatorului **EXISTS** performanța comenzii **SQL** este mai mare deoarece operatorul **IN** compară fiecare valoare întoarsă de subcerere, pe când **EXISTS** verifică doar existența a cel puțin unei linii (înregistrări) întoarse de subinterogare, fără a face nici o comparație.

## 7.4 Concluzii

În această lecție am discutat despre noțiunea de **subinterogare** și despre **tipurile de subinterogări**. Sunt noțiuni complexe în lucrul cu baze de date, însă utile în foarte multe situații.

Următoarea lecție a acestui curs va conține prezentări ale noțiunilor de **tabelă virtuală (vedere sau view)**, **tabelă temporară** și **index**. De asemenea, vor fi prezentate **tipurile de index**, precum și avantajele și dezavantajele pe care le oferă indexarea unor coloane din tabelele unei baze de date

## Tema Sedinta 7

În fișierul de la link-ul **Baza de Date classicmodels de la Ședința 5** sunt instrucțiunile pentru crearea bazei de date **classicmodels** și a tabelelor din această bază de date.

### Cerințe

1. Rulați instrucțiunile din fișier pentru crearea bazei de date și a tabelelor.
2. Scrieți o instrucțiune **SQL** prin care să **afișați clienții care au interacționat cu angajați de la birourile din Australia** (câmpuri afișate: **contactLastName, contactFirstName, city, country**). Utilizați subinterogări.
3. Scrieți o instrucțiune **SQL** prin care să **afișați angajații de la birouri din afara USA** (câmpuri afișate: **lastName, firstName, officeCode**). Utilizați subinterogări.
4. Scrieți o instrucțiune **SQL** prin care să **afișați produsele comandate de clienți din Boston, Madrid și Los Angeles** (câmpuri afișate: **productCode, productName**).

Realizați 2 variante: una cu JOIN si una cu subinterogări.

5. Scrieți o instrucțiune **SQL** prin care să **afișați comenzile cu valoarea mai mare decât valoarea medie a tuturor comenzilor în care au fost comandate produse Ferrari** (câmpuri afișate: `orderNumber`, `productCode`, `valoare comandă` (`priceEach*quantityOrdered`) și `valoarea medie a tuturor comenzilor`). Utilizați subinterogări.

Valoarea comenzii și media vor fi afișate cu 2 zecimale iar rezultatele vor fi ordonate crescător după valoarea comenzii.

## 8. Tabele virtuale. Tabele temporare. Indexare

### 8.1 Tabele virtuale (vederi, view-uri)

**Vederile** sunt, de fapt, niște **tabele virtuale**, adică vederile conțin interogări care regăsesc în mod dinamic datele atunci când sunt utilizate. Ele nu conțin date, ci fac referire la date din tabelele bazei de date. **Vederile (tabelele virtuale)** sunt stocate alături de tabele în baza de date.

Vederile pot fi folosite pentru simplificarea operațiunilor **SQL** complexe. După ce a fost scrisă o interogare aceasta poate fi refolosită cu ușurință dacă a fost creată o vedere pe baza acelei interogări. În plus, rezultatele din tabela virtuală se actualizează dinamic, în funcție de ceea ce se întâmplă în tabelele din care au fost extrase datele în interogarea care a fost declarată la crearea tabelului virtual.

**Vederile** sau **tabelele virtuale** sunt adesea denumite cu termenul în limba engleză, este vorba de termenul **view**. Mai trebuie precizat și faptul că, **vederile** (**view**-urile) pot fi folosite pentru a folosi doar părți din anumite tabele, iar nu tabele complete.

După ce au fost create **vederile**, se pot efectua operații de interogare (instrucțiuni **SELECT**) și pe acestea. Ca și tabelele, vederile trebuie să aibă un nume unic în cadrul unei baze de date. O **vedere** nu poate primi **numele unei table** din acea bază de date.

O **tabelă virtuală** în **SQL** este creată utilizând comanda **CREATE VIEW**. De asemenea, trebuie specificat un nume pentru tabela virtuală imediat după comanda **CREATE VIEW**.

Putem spune că o **vedere** reprezintă o instrucțiune **SELECT** stocată în baza de date. **Vederile** reprezintă un mod de accesare a datelor, ele nu stochează date efectiv. Sunt legate de tabela sau tabelele pe baza cărora au fost create. Ele pot fi folosite din motive de securitate. De exemplu, se pot crea părți din tabelă vizibile anumitor utilizatori.

**Vederile** pot fi actualizabile sau neactualizabile. Cele actualizabile permit instrucțiunile **INSERT**, **UPDATE**, **DELETE**. Pe cele neactualizabile nu se pot efectua aceste operații. Pentru a fi actualizabile trebuie să fie o corespondență linie cu linie între o tabelă virtuală și tabela pe baza căreia a fost creată.

Este evident acest lucru, dacă nu există această corespondență între view și tabelă nu se pot face actualizări în tabela virtuală.

De asemenea, este important de menționat că nu trebuie să avem în instrucțiunea care creează tabela virtuală actualizabilă **funcții de grup**, clauzele **DISTINCT**, **GROUP BY** și **HAVING**. Nu putem avea **nicisubinterogări**, **reuniuni** sau **OUTER JOIN**. În schimb, dacă avem **INNER JOIN** în instrucțiunea de creare, atunci este permisă actualizarea.

## 8.1.1 Crearea unei table virtuale

Sintaxa folosită pentru a crea o vedere (view) este următoarea:

```
CREATE VIEW nume_vedere AS  
SELECT ....;
```

## 8.1.2 Redefinirea unei table virtuale

O **vedere** (**view**) poate fi redefinită. În acest sens, avem o instrucțiune specială, similară cu cea de creare a tabelului virtual.

Sintaxa folosită pentru **redefinirea** unui **view** este următoarea:

```
ALTER VIEW nume_vedere AS  
SELECT ....;
```

## 8.1.3 Ștergerea unei tabele virtuale

Evident, avem și posibilitatea de a șterge o tabelă virtuală din baza de date. Instrucțiunea care permite ștergerea unei vederi din baza de date este următoarea:

```
DROP VIEW nume_vedere;
```

Alte mențiuni pe care le vom face în legătură cu vederile se referă la faptul că o vedere poate fi creată și pe baza unei alte vederi, deci, nu este obligatoriu să fie realizată pe baza unei tabele. Tabelele virtuale și tabelele din baza de date sunt stocate în același loc pe disc, din acest motiv, este necesar ca ele să aibă nume diferite.

## 8.2 Tabele temporare

### 8.2.1 Noțiuni generale

Vom defini, în continuare, noțiunea de **tabelă temporară**. Vom prezenta sintaxa de creare a unei tabele temporare în bazele de date **MySQL** și vom explica ce înseamnă și la ce folosesc.

*O tabelă temporară stochează date doar pe durata unei tranzacții sau a unei sesiuni de lucru. Tabelelor temporare nu li se alocă spațiu la creare decât dacă în cadrul instrucțiunii **CREATE** de creare se specifică clauza **AS SELECT**. Altfel, spațiul va fi alocat la prima instrucțiune **INSERT** în această tabelă temporară.*

Subliniem faptul că nu trebuie făcută confuzia între tabele tempore și tabele virtuale sau vederi. Vederilor sau tabelelor virtuale au fost prezentate și exemplificate în modulul precedent. O tabelă virtuală este cunoscută și sub denumirea de view. *O tabelă virtuală nu poate face referire la tabele temporare, ea va face referire doar la tabele stocate permanent în baza de date.* De asemenea, nici trigger (declanșatoare) nu pot fi definite pentru tabele temporare.

La crearea tabelelor temporare, în sintaxa instrucțiunii **CREATE TABLE** apare cuvântul cheie **TEMPORARY** care specifică faptul că tabela creată este temporară. *Tabela temporară va fi ștearsă automat în momentul închiderii conexiunii sau sesiunii de lucru în care a fost creată.*

*Două conexiuni diferite la o bază de date pot crea tabele temporare cu același nume, ele nu vor interfera. Datele din tabelele temporare sunt vizibile doar sesiunii care le inserează (una singură la un moment dat).*

O tabelă temporară este utilă atunci când este imposibil sau foarte dificil să fi obținute date printr-o singură instrucțiune **SELECT**. Se folosesc în special în cadrul rutinelor memorate în baza de date (proceduri stocate) pentru a stoca seturi de rezultate imediate în vederea utilizării ulterioare.

Tabelele temporare sunt create prin utilizarea instrucțiunii **CREATE TEMPORARY TABLE**, astfel, se observă că față de crearea unei tabele permanente, între cuvintele cheie **CREATE** și **TABLE** apare cuvântul cheie **TEMPORARY** care specifică faptul că tabele ce va fi creată este temporară.

Așa cum am precizat, *tabelele temporare sunt șterse în mod automat la închiderea sesiunii de lucru*, dar ele pot fi și *șterse explicit, înainte de închiderea sesiunii în care au fost create, prin utilizarea comenzii **DROP TABLE***. O tabelă temporară este disponibilă și accesibilă doar utilizatorului care a creat-o.

Utilizatori diferiți pot crea tabele temporare cu același nume deoarece doar clientul care a creat o tabelă temporară are acces la ea și poate să o utilizeze, deci, ele nu vor interfera chiar dacă au același nume. În aceeași sesiune, însă, două tabele temporare create de același utilizator nu pot avea același nume.

De asemenea, o tabelă temporară poate avea același nume cu al unei tabele permanente stocate în baza de date. În această situație, pe parcursul sesiunii curente, tabela permanentă devine inaccesibilă. Fiecare instrucțiune care face referire la tabela cu acel nume se va referi la tabela temporară.

Prin urmare, nu este recomandată crearea tabelelor temporare cu același nume ca al tabelelor permanente deoarece se poate crea confuzie la utilizare.

## 8.2.2 Modalități de creare a unei tabele temporare

Definirea unei tabele temporare se poate realiza similar cu definirea unei tabele permanente, cu apariția suplimentară a cuvântului **TEMPORARY**, deci, în instrucțiunea **CREATE** se vor specifica numele coloanelor din tabela temporară, tipul lor de date, dimensiunea, constrângerile, etc.

O altă modalitate de a defini o tabelă temporară este utilizarea unei instrucțiuni **SELECT** după cuvintele cheie **CREATE TEMPORARY TABLE** și numele dat tabelii temporare. Aceasta înseamnă că tabela temporară va conține acele coloane returnate de instrucțiunea **SELECT** (cu tipurile de date aferente) și va fi și populată cu valorile corespunzătoare acelor coloanel ce au îndeplinit criteriile specificate în cadrul instrucțiunii **SELECT** din comanda **CREATE**.

Crearea unei tabele temporare poate fi realizată prin specificarea explicită a coloanelor componente ale tabelii, prin utilizarea instrucțiunii **SELECT** în comanda **CREATE**, astfel preluându-se date din tabelele de origine, sau, prin copierea structurii unei tabele cu ajutorul clauzei **LIKE**.

Avem, deci, cele trei forme posibile, a căror sintaxă este enunțată în cele ce urmează:

```
CREATE TEMPORARY TABLE nume_tabelă(  
    nume_coloană1 tip_dată[(dimensiune)],
```

```

        nume_coloană2 tip_dată[(dimensiune)],
        ....
    );

CREATE TEMPORARY TABLE nume_tabelă_nouă LIKE nume_tabelă_originală;

CREATE TEMPORARY TABLE nume_tabelă [AS]
    SELECT * | column_name(s)
    FROM tabelă_origine
    [WHERE ...];

```

### 8.2.3 Ștergerea unei tabele temporare

Am menționat faptul că o tabelă temporară poate fi ștearsă pe parcursul unei sesiuni prin utilizarea instrucțiunii **DROP TABLE** urmată de numele tabelului.

Recomandarea este să fie utilizată instrucțiunea **DROP TEMPORARY TABLE** urmată de numele tabelului temporare, tocmai pentru a nu se crea confuzie și pentru a nu exista riscul ca, în cazul în care există două tabele – una permanentă și una temporară cu același nume – să fie ștearsă tabela permanentă.

Comanda **DROP TEMPORARY TABLE** nu mai generează ambiguitate, este foarte limpede ce tabelă va fi eliminată din baza de date. Sintaxa comenzii de ștergere a unei tabele temporare este următoarea:

```
DROP [TEMPORARY] TABLE nume_tabelă;
```

### 8.2.4 Deosebiri între tabele temporare și tabele virtuale

Așa cum am menționat mai sus, nu trebuie să confundăm tabelele temporare cu tabelele virtuale, sunt noțiuni diferite, astfel că, mai departe vom încerca să explicăm deosebirile dintre ele.

Vom prezenta, în tabelul următor, o comparație între aceste două obiecte ce pot să apară în bazele de date. Comparația are menirea de a ne lămurii pe deplin asupra a ceea ce reprezintă fiecare dintre aceste obiecte, precum și pentru a vedea principalele elemente care le caracterizează pe fiecare dintre ele.

Tabele temporare	Tabele virtuale (vederi, view-uri)
Sunt stocate temporar în baza de date, până la încheierea sesiunii curente de lucru	Sunt stocate permanent în baza de date

Sunt create prin instrucțiunea <b>CREATE TEMPORARY TABLE</b>	Sunt create prin instrucțiunea <b>CREATE VIEW</b>
Conțin efectiv date atât timp cât sunt create și sesiunea curentă este activă	Nu conțin date efectiv ci fac referire la datele existente în tabela sau tabelele pe baza cărora au fost create. Tabelele virtuale sunt de două feluri: actualizabile și neactualizabile. Cele actualizabile prezintă avantajul actualizării automate și a vederii atunci când s-au făcut actualizări în tabela la care face referire vederea.
O tabelă temporară poate avea același nume cu al unei tabele permanente din baza de date	O tabelă virtuală nu poate să aibă același nume cu al unei tabele permanente stocate în baza de date
Pot exista două tabele temporare cu același nume în aceeași bază de date dacă sunt create de utilizatori diferiți	Nu pot exista două tabele virtuale cu același nume în aceeași bază de date, chiar dacă se încearcă crearea de utilizatori diferiți

## 8.3 Index

### 8.3.1 Conceptul de index

La bazele de date de dimensiuni mari este foarte utilă metoda indexării pe anumite coloane sau ansambluri de coloane, astfel ca, la executarea interogărilor timpul de execuție să scadă.

Conceptul de index definește o structură de date adițională, redundantă, care dacă este atașată unei tabele dintr-o bază de date poate crește viteza de căutare și ordonare a datelor din acea tabelă. Un index poate fi definit pe una sau mai multe coloane ale unei tabele.

În momentul în care este definit un index, informația din coloana sau ansamblul de coloane pe care a fost definit indexul este copiată într-o structură separată, fiecare element al acestei noi structuri având o referință către înregistrarea de la care provine. Într-un index elementele sunt ordonate, iar aceasta face ca găsirea unei informații să fie rapidă.

Un index este util atunci când se realizează o căutare după o coloană ce a fost indexată (căutarea realizându-se în structura unde se află indexul – fișierul de index și va găsi elementele care îndeplinesc condiția enunțată, iar apoi, se va folosi de referințele pe care indexul le are către înregistrările corespunzătoare din tabelă), dar și atunci când se realizează o ordonare după o coloană ce este indexată (în acest caz, informațiile din index fiind deja ordonate, serverul va obține imediat înregistrările care corespund referințelor elementelor din index și le va afișa în ordinea din index). O ordonare descrescătoare va reprezenta o parcurgere inversă a intrărilor din index.

Există și anumite constrângeri care atunci când sunt definite pe coloanele unei tabele, creează automat un index pe coloana sau ansamblul de coloane pe care a fost impusă restricția respectivă.



Definirea unui **PRIMARY KEY** pe o coloană sau pe un ansamblu de coloane, crează automat un index pe acea coloană sau pe ansamblul de coloane ce formează cheia primară.

Astfel că, o interogare în care condiția de căutare din clauza **WHERE** este pusă pe o coloană **PRIMARY KEY** va genera rezultatul într-un timp rapid.

Același lucru se întâmplă și la definirea unei constrângeri **UNIQUE** pe o coloană, adică se va genera automat un index pe coloana respectivă. Constrângerile de unicitate garantează faptul că toate datele dintr-o coloană sau dintr-un set de coloane sunt unice.

## 8.3.2 Deosebiri între PRIMARY KEY și constrângerea UNIQUE

Constrângerile de unicitate sunt asemănătoare cu cheile primare, dar între ele există și deosebiri. Astfel, o tabelă poate conține mai multe constrângeri de unicitate, deci constrângerea **UNIQUE** poate fi definită pe mai multe coloane, pe când **cheia primară** este unică într-o tabelă, chiar dacă este compusă din mai multe coloane (ansamblu de coloane) și nu este simplă (o singură coloană).

Deci, atunci când cheia primară este definită pe un ansamblu de coloane trebuie ca acele valori din coloanele respective luate împreună (combinația de valori) să fie unice.

O altă **deosebire** între coloanele care au definită constrângerea de **cheie primară** și cele care au definită **constrângerea de unicitate** este aceea că acele coloanele ce prezintă constrângeri de unicitate pot conține valori NULL. Cunoaștem faptul că în coloanele ce prezintă constrângerea cheie primară nu pot conține valori nule. Coloanele cheie primară conțin valori unice și nenule.

## 8.3.3 Considerații generale despre indexare

Există, însă, posibilitatea de a defini indecși și pe alte coloane ce nu au definite constrângeri de alt tip asupra lor. În momentul definirii unui index, sau al generaării automate a unui index, pe o coloană sau pe un grup de coloane se crează automat un fișier, numit fișier de index care conține datele din această coloană sau din grupul de coloane.

Astfel, atunci când o interogare conține condiția pe o coloană indexată căutarea se va face în fișierul de index, fapt ce va duce la returnarea rezultatului interogării într-un timp mult mai rapid decât dacă nu ar fi indexate coloanele, tabele ar fi foarte mari iar căutarea s-ar face prin parcurgerea întregii tabele.

Indexarea trebuie înțeleasă și, reprezintă, de fapt, o ordonare la nivel logic a datelor, deci, nu este o ordonare vizibilă așa cum se întâmplă cu rezultatul utilizării clauzei **ORDER BY** într-o interogare, ci este o ordonare sau sortare la nivel logic a datelor.

Principiul unui index este acela că este sortat în mod corect. După definirea unui index, programul **SGBD** păstrează o listă sortată a conținutului. În momentul adresării unei cereri de

regăsire către **SGBD** acesta caută în indexul sortat pentru a găsi locația eventualelor valori care corespund celor ce îndeplinesc criteriile de căutare și apoi regăsește înregistrările respective.

Nu există nici o regulă universală cu privire la ce coloane trebuie să fie indexate și când trebuie indexate. Menționăm, însă, și anumite aspecte cu privire la posibilitatea ca, în anumite cazuri, utilizarea de indecși să îngreuneze anumite instrucțiuni pe baza de date.

Trebuie să ținem cont și de faptul că, doar adăugarea de indecși, operație care se realizează foarte ușor, nu reprezintă mereu o soluție completă de optimizare. Într-adevăr, prin generarea acelor fișiere de indecși care se salvează pe server și care permit realizarea mult mai **rapidă** a instrucțiunilor **SELECT**, aceste fișiere duc la **îngreunarea** execuției instrucțiunilor **INSERT**, **UPDATE** și **DELETE**.

Duc la îngreunarea execuției acestor instrucțiuni deoarece la fiecare operație de acest tip (**INSERT**, **UPDATE** sau **DELETE**) se va produce o regenerare automată de către sistem a acestor fișiere de indecși.

Deși, așa cum am menționat mai sus, nu există nici o regulă generală cu privire la ceea ce trebuie indexat, totuși sunt coloane care nu se pretează la indexare. Practic, datele care nu prezintă o frecvență suficient de redusă nu vor beneficia la fel de mult de pe urma indexării.

Astfel, declararea unui index pe coloane cum ar fi un nume, un prenume, o denumire, un pret, o cantitate, o localitate nu sunt eficiente pentru că ar putea exista foarte multe valori identice astfel că nu ar fi relevante. Se pot indexa coloane ce conțin date calendaristice, de exemplu data nașterii stocată pentru anumite persoane ceea ce ar îmbunătăți căutarea după data nașterii pentru a identifica o persoană.

De asemenea, este recomandată reexaminarea periodică a indexărilor realizate într-o bază de date, deoarece anumite indexuri eficiente la un anumit moment, pot deveni ineficiente după o perioadă când baza de date a suferit diverse modificări (manipulări de date).

## 8.3.4 Avantaje și dezavantaje întâlnite la utilizarea indecșilor

Principalul **avantaj** al definirii de indecși este creșterea vitezei operațiilor de căutare și ordonare. Acest lucru se întâmplă, în cazul în care căutarea se realizează după o coloană indexată. Dacă se realizează o filtrare după alte coloane din tabelă care nu sunt indexate, nu are importanță faptul că în tabela respectivă există și coloane indexate. Deci, recomandarea este să fie indexate acele coloane ce sunt folosite frecvent pentru filtrarea datelor dintr-o tabelă (coloanele utilizate în clauza **WHERE** a instrucțiunii **SELECT**).

Un alt **avantaj** al indexării este oferit de faptul că indecșii pot introduce restricții asupra coloanei sau grupului de coloane pe care sunt definiți. Acest lucru se întâmplă, de exemplu, atunci când este definită o restricție **UNIQUE**, care va crea și un index pe coloana sau ansamblul de coloane respectiv..

Un **dezavantaj** constă în scăderea vitezei la celelalte operații de manipulare a datelor (inserare, modificare și ștergere) deoarece, pe lângă modificările produse în tabelă, este necesar să se actualizeze de fiecare dată și indexul (actualizarea aceasta se face automat dar îngreunează aceste operații).

Un alt **dezavantaj** este acela că la crearea de indecși se alocă spațiu suplimentar care va fi ocupat. Se întâmplă acest lucru deoarece indexul reproduce părți ale înregistrărilor din tabele. Un număr mare de indecși va crește dimensiunea tablei, fapt din care decurge că va fi ocupat mai mult spațiu pe disc.

Deci, faptul că pot ocupa o mare cantitate de spațiu reprezintă un alt aspect care poate face inefficientă utilizarea indexării.

Astfel, indexarea degradează operațiile de inserare, modificare și ștergere pentru că la execuția acestor instrucțiuni, programul **SGBD** trebuie să actualizeze fișierul de index (lista de index) în mod dinamic.

## 8.3.5 Utilitatea creării indecșilor

Fiind vorba despre optimizări aduse bazei de date care pot duce la creșterea vitezei de căutare, dar care pot produce și anumite încetiniri pe operații de actualizare a datelor, trebuie analizată cu atenție oportunitatea folosirii indexării. Nu întotdeauna avem nevoie de indexări de coloane pe tabelele unei baze de date.

Prezentăm mai jos o parte din situațiile în care este recomandat să fie folosiți indecși pentru optimizarea procesului de căutare a informațiilor:

- Coloana respectivă este utilizată frecvent în clauza **WHERE** a instrucțiunilor **SELECT** sau în condițiile de join;
- Coloana conține o gamă largă de valori;
- Coloana conține un număr mare de valori nule;
- Tabelele au dimensiuni foarte mari, iar interogările vor returna un număr mic de rezultate;
- Două sau mai multe coloane sunt utilizate frecvent împreună în clauza **WHERE** a instrucțiunilor **SELECT** sau în condițiile de join din interogări.

Expunem, în continuare, și o serie de situații în care nu este recomandată indexarea coloanelor, deoarece indexarea ar cauza o încetinire a vitezei de lucru:

- Tabelele au dimensiuni mic, deci, conțin un număr mic de înregistrări;
- Coloanele sunt folosite rar în condițiile ce se pun la realizarea interogărilor;
- Tabelele sunt modificate frecvent, deci, se execută foarte des instrucțiuni de manipulare ce afectează înregistrările din tabele;
- Cele mai multe interogări returnează un număr mare de înregistrări.

## 8.3.6 Metode de regăsire a datelor

Există două metode ce pot fi utilizate în SQL pentru regăsirea datelor. Aceste metode sunt următoarele:

- **Metoda secvențială;**
- **Metoda cu acces direct;**

În cazul utilizării metodei secvențiale se realizează o parcurgere a fiecărui element din tabelă pentru a găsi înregistrările ce îndeplinesc condițiile specificate în interogare. Aceasta este o metodă inefficientă de căutare.

Pentru a înțelege mai bine ce înseamnă indexarea se poate face asocierea cu indexul situat la sfârșitul unei cărți. Astfel, dacă se dorește găsirea tuturor aparițiilor unei anumite expresii sau al unui anumit concept, metoda cea mai simplă este să se verifice indexul de la sfârșitul cărții și se va găsi acolo la ce pagini este menționată expresia respectivă. Indexul este o listă alfabetică, deci, găsirea unui anumit concept în index, oricât de stufos ar fi, este rapidă. O altă metodă ar fi parcurgerea cărții pagină cu pagină și linie cu linie pentru a găsi locurile în care apare expresia căutată. În mod evident, această metodă este total inefficientă și consumă foarte mult timp.

În felul acesta funcționează și indexarea coloanelor în tabelele bazelor de date. Atunci când se realizează o căutare (**SELECT**) după o anumită coloană care este indexată se va parcurge fișierul de index (deci, facem analogia cu indexul de la finalul cărții). Avem de-a face, în situația aceasta, cu o metodă de acces direct la date.

Dacă nu există indecși definiți, atunci se va face o parcurgere linie cu linie a întregii tabele pentru a găsi înregistrările ce corespund criteriilor de căutare (deci, analogia cu parcurgerea pagină cu pagină a unei cărți). Este vorba, deci, de o metodă secvențială de acces la date,

Devine limpede faptul că este mult mai rapidă căutarea atunci când există indecși definiți iar căutarea se face în fișierul de index. Prin adăugarea de indecși se obțin performanțe superioare privind accesul la datele din baza de date.

## 8.3.7 Clasificare

Din punct de vedere al numărului de coloane pe care le cuprinde, un index poate fi clasificat astfel:

- **index simplu** sau **index uni-coloană** (creat pe o singură coloană);
- **index multiplu** sau **multi-coloană** (creat pe două sau mai multe coloane);
- **index parțial** (creat pe o anumită parte dintr-un șir de caractere în cazul **MySQL**);

În situația indexării multiple, deci, crearea unui index pe un ansamblu (grup) de coloane, **MySQL** realizează indexarea prin *concatenarea (alipirea) coloanelor în ordinea specificării acestora* în instrucțiunea de crearea indexului multiplu.

De asemenea, la acest tip de index, multi-coloană, ordinea în care sunt introduse coloanele în index contează. **MySQL** se folosește de un *index multi-coloană* doar în situația în care clauza **WHERE** a instrucțiunii **SELECT** conține valori din prima sau primele coloane din index, dacă, însă, lipsește prima coloană și sunt utilizate doar valori din celelalte, atunci nu va fi utilizat indexul.

O altă clasificare a indecșilor care se poate face este din punct de vedere al constrângerilor ce sunt impuse asupra datelor din coloana sau din coloanele pe care este definit indexul respectiv.

Astfel, distingem următoarele tipuri de indecși:

- **index simplu** – coloana respectivă este indexată fără a se impune restricții asupra ei;

- **index unic** – valorile acestei coloane trebuie să fie unice (constrângerea **UNIQUE**). În cazul introducerii de valori duplicate în coloane ce au definit acest tip de index, sistemul va genera o eroare. Însă, într-o coloană ce are definit acest index pot fi introduse valori nule (**NULL**). Pot exista mai multe valori nule pe coloana respectivă.
- **cheie primară (PRIMARY KEY)** – index care constrânge la introducerea de valori unice și nenule. Pe o tabelă poate fi definit un singur index de tip **PRIMARY KEY**, dar mai multe de tip **UNIQUE**.

Un **index unic** crește și mai mult viteza de execuție a interogărilor filtrate după o astfel de coloană, față de un index simplu. În cazul unui index simplu este parcurs întreg fișierul de index, deoarece poate exista un set de valori corespunzător căutării. În situația în care pe coloana respectivă este definit un index unic, în momentul găsirii în index a elementului respectiv, sistemul nu va căuta mai departe în index alte valori.

## 8.4 Operații

Operațiile ce se pot face cu indecși sunt: crearea, ștergerea și actualizarea lor (aceasta din urmă se realizează automat în cazul efectuării operațiilor de manipulare **INSERT**, **UPDATE**, **DELETE**); de altfel, așa cum am prezentat mai sus, și crearea se poate face automat în anumite cazuri, adică atunci când sunt definite constrângeri de tip **PRIMARY KEY**, **UNIQUE** se crează automat și indecși pe acele coloane.

Fiecare index definit trebuie să aibă un nume unic.

### 8.4.1 Crearea unui index

Indecșii pot fi creați în momentul creării tabelii, deci, în cadrul instrucțiunii **CREATE TABLE** sau, ulterior, prin executarea unei instrucțiuni **ALTER TABLE** sau **CREATE INDEX**.

La crearea tabelii, în cadrul sintaxei **CREATE TABLE** se adaugă clauze suplimentare: **PRIMARY KEY**, **UNIQUE** sau **INDEX**. Clauza se adaugă în dreptul coloanei respective, sau la final, după enumerarea coloanelor, în zona de restricții (în această zonă coloana sau coloanele vor fi trecute după numele constrângerii între paranteze).

Sintaxa instrucțiunii **ALTER TABLE** de adăugare a unui index după ce tabela a fost creată este următoarea:

```
ALTER TABLE nume_tabelă ADD tip_index(nume_coloană1[, nume_coloană2, ...]);
```

Dacă tabela este deja populată cu înregistrări pot să apară unele erori în cazul definirii unui index de tip **PRIMARY KEY** sau **UNIQUE**. Sistemul va genera eroare la definirea unui astfel de index, dacă în coloana respectivă există valori duplicate (**UNIQUE**) sau valori duplicate sau nule (**PRIMARY KEY**).

Comanda **CREATE INDEX** se folosește pentru crearea unui index la o tabelă, index care va conține câte o intrare pentru fiecare valoare ce apare specificată în tabelă în coloana respectivă. Această coloană poartă numele de coloană de index.

Sintaxa comenzii de creare a unui index este următoarea:

```
CREATE [UNIQUE] INDEX nume_index  
    ON nume_tabelă (nume_coloană1 [ASC|DESC][, nume_coloană2 [ASC|DESC],  
    ...]);
```

Cu ajutorul comenzii **CREATE INDEX** se pot crea indecși simpli sau unici, dar nu se pot crea indecși de tip **PRIMARY KEY**.

## 8.4.2 Vizualizarea indecșilor existenți

Vizualizarea indecșilor definiți asupra coloanelor unei table se poate face prin executarea comenzilor:

- **SHOW CREATE TABLE** – ce va afișa indecșii ca parte a definiției tablei;
- **DESCRIBE** – care afișează structura unei table, inclusiv indecșii;
- **SHOW INDEX** – care va afișa doar indecșii definiți.

Primele două comenzi au fost deja prezentate, așa că aici vom reda sintaxa instrucțiunii **SHOW INDEX** care este următoarea:

```
SHOW INDEX FROM nume_tabelă;
```

Această comandă este importantă mai ales pentru afișarea numelor pe care le are fiecare index. Numele unui index este important dacă se încearcă ștergerea lui, deoarece va fi utilizat în instrucțiunea de ștergere.

8.4.3 Ștergerea unui index se poate realiza fie printr-o instrucțiune **ALTER TABLE** care va îndepărta o constrângere care este și index, fie printr-o instrucțiune **DROP INDEX**.

Eliminarea unei constrângeri care este și index, prin executarea unei instrucțiuni **ALTER TABLE**, se face prin comanda următoarea:

```
ALTER TABLE nume_tabelă DROP tip_index [(nume_index)];
```

Instrucțiunea de ștergere a unui index simplu este următoarea:

```
DROP INDEX nume_index ON nume_tabelă;
```

## 8.5 Concluzii

În concluzie, *indexarea este o metodă deosebit de eficientă pentru creșterea eficienței execuției instrucțiunilor de regăsire a datelor (SELECT)*, dar, în același timp, poate duce la *îngreunarea execuției instrucțiunilor de manipulare a datelor ce afectează datele din tablele unei baze de date* (este vorba de instrucțiunile **INSERT**, **UPDATE** și **DELETE**).

*Indexarea nu schimbă ordinea fizică a înregistrărilor din table. Prin definirea indecșilor se crează automat o corespondență între ordinea fizică a înregistrărilor și ordinea logică. Indecșii ordonează la nivel logic după anumite criterii stabilite la definirea (crearea) lor.*

Instalare My sql WorkBrench



mysql workbench download

mysql workbench download  
mysql workbench download for windows 7 32 bit  
mysql workbench documentation  
mysql workbench download ubuntu 16.04  
mysql workbench download ubuntu  
mysql workbench download ubuntu 14.04  
mysql workbench download 5.7  
mysql workbench docker  
mysql workbench download for windows 32 bit  
mysql workbench doesn't open

Google Search I'm Feeling Lucky

Google mysql workbench download

All Videos Images Books News More Settings Tools

About 364,000 results (0.49 seconds)

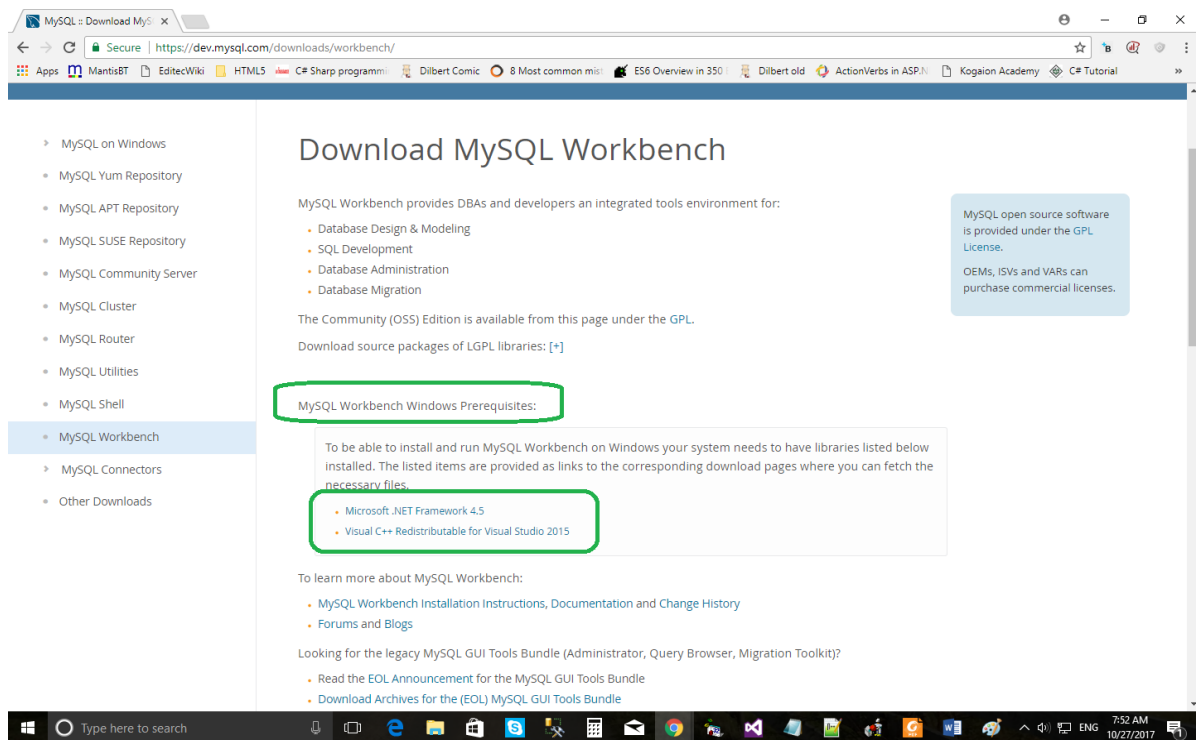
**MySQL :: Download MySQL Workbench**  
<https://dev.mysql.com/downloads/workbench/> ▼  
Download MySQL Workbench. MySQL open source software is provided under the GPL License. OEMs, ISVs and VARs can purchase commercial licenses.

<b>Download MySQL Workbench</b> MySQL Workbench provides DBAs and developers an integrated ...	<b>MySQL GUI Tools</b> Please note that development of MySQL GUI Tools has been ...
<b>MySQL :: Download MySQL ...</b> MySQL Workbench provides DBAs and developers an integrated ...	<b>Download</b> MySQL Workbench provides DBAs and developers an integrated ...
<b>Installation</b> The following sections explain the installation process for each of ...	<b>MySQL Workbench Release ...</b> Abstract. This document contains release notes for the changes in ...

[More results from mysql.com »](#)

Instalati .Net Framework 4.5 si Visual C++ 2015 – daca este nevoie.



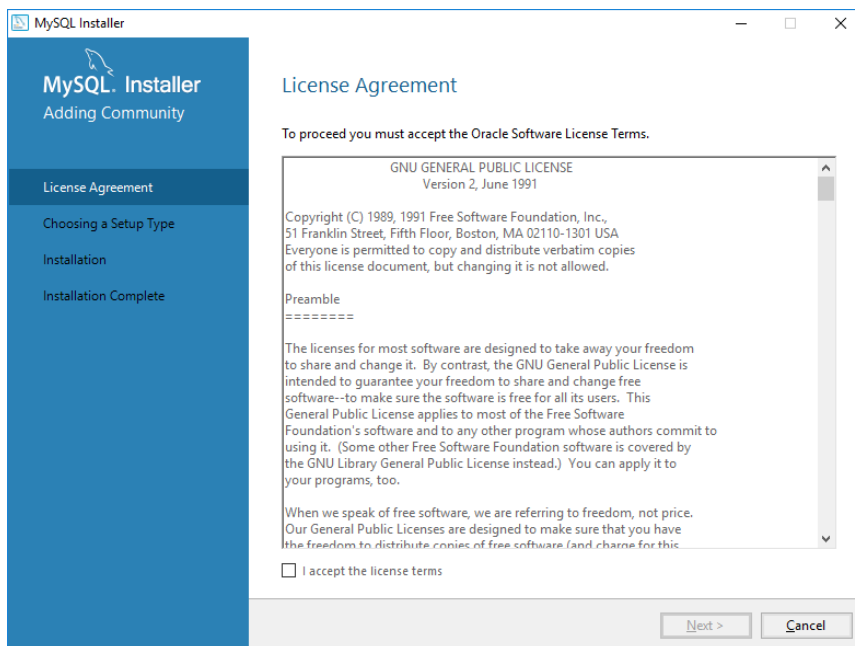
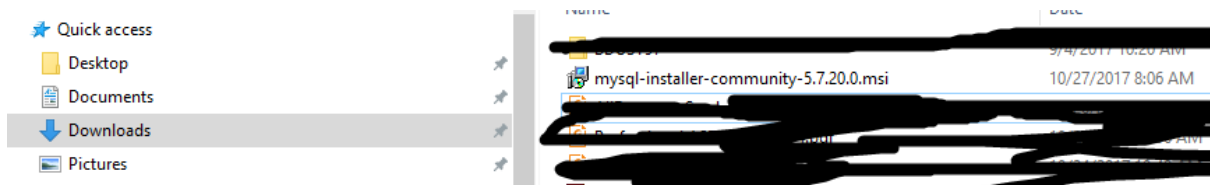
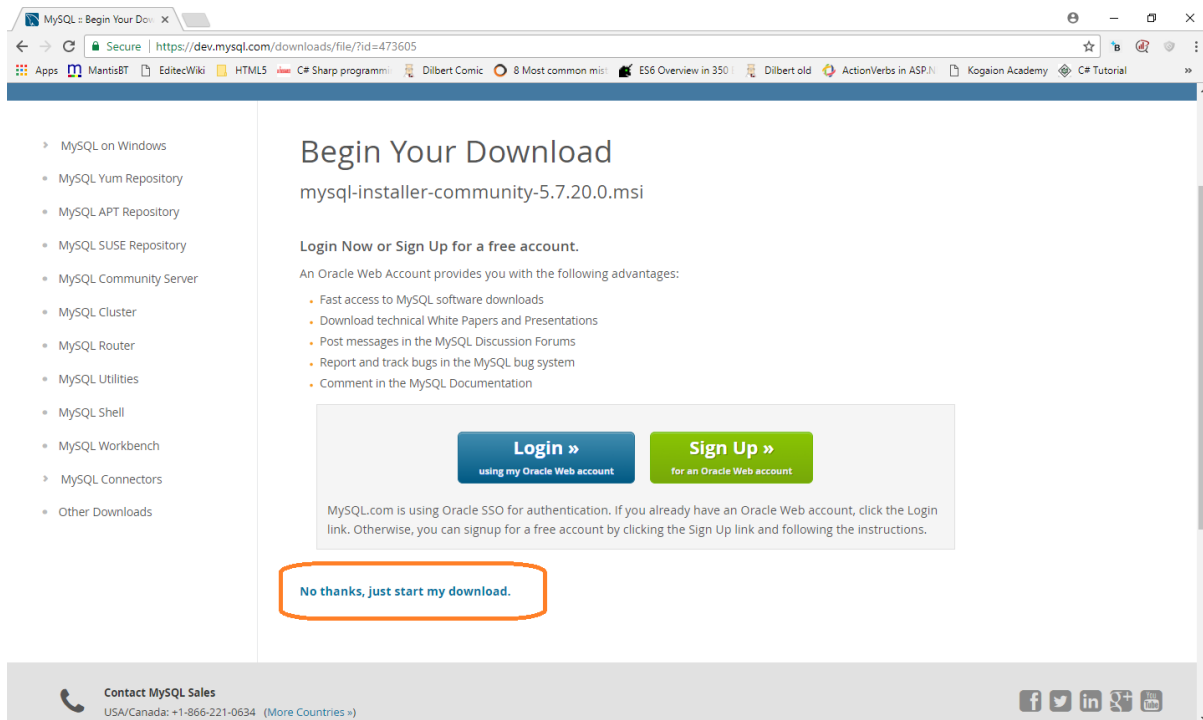


Dupa, in josul acestei pagini gasiti linkurile de instalare:

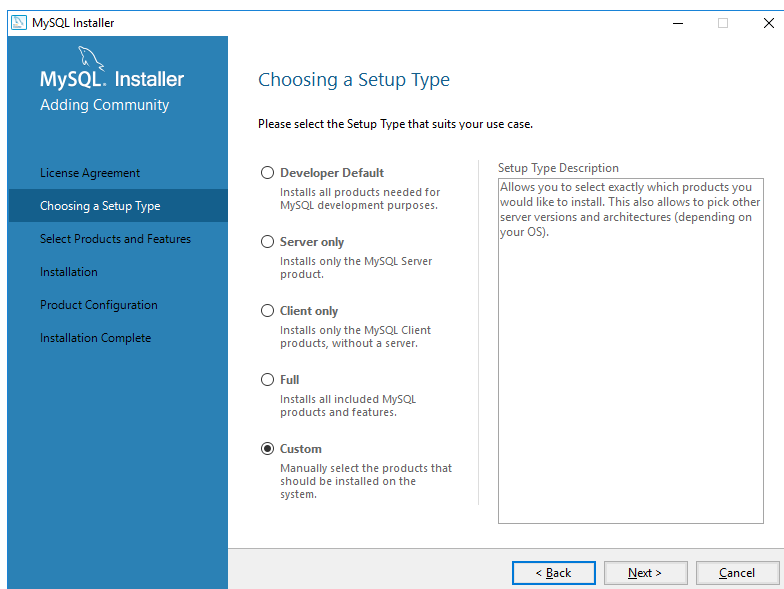
The image consists of two screenshots from the MySQL website. The top screenshot shows the 'MySQL Workbench 6.3.9' download page. It features a 'Recommended Download' section for Windows (x86, 32 & 64-bit) with a 'Go to Download Page' button. The bottom screenshot shows the 'MySQL Installer 5.7.20' download page. It features a table of download links for Windows (x86, 32-bit), MSI Installer, with the 376.3M version highlighted by an orange box. The table includes the version number, file size, and a 'Download' button.

Operating System	Version	File Size	Download Link
Windows (x86, 32-bit), MSI Installer	5.7.20	18.5M	<a href="#">Download</a>
Windows (x86, 32-bit), MSI Installer	5.7.20	376.3M	<a href="#">Download</a>

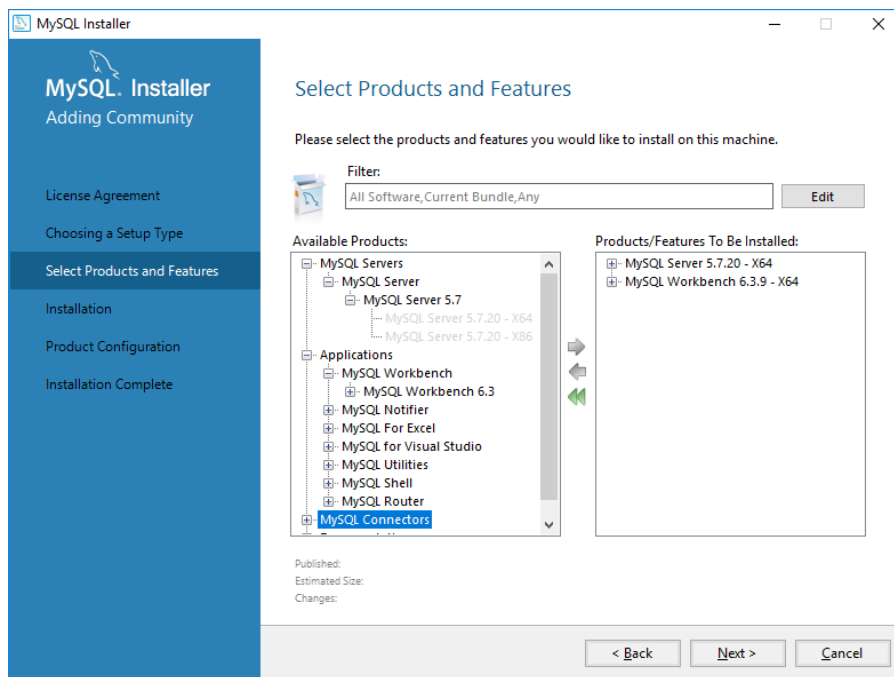
Alegeti pe cel mare la dimensiuni (376 M) – asta inseamna ca are si serverul in kitul de instalare.



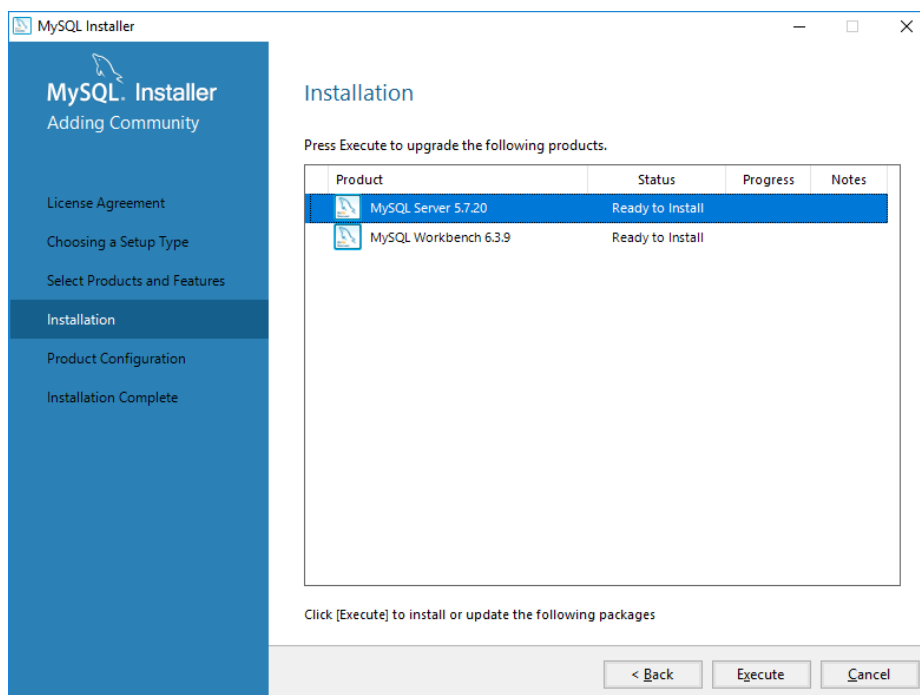
Checked "I accepted the license terms".



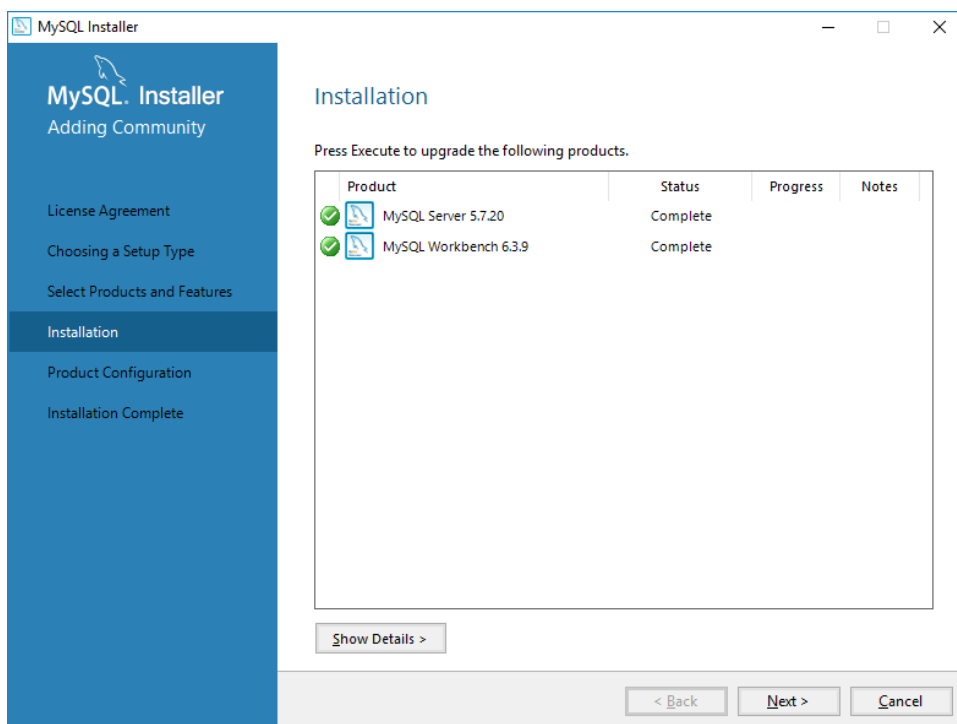
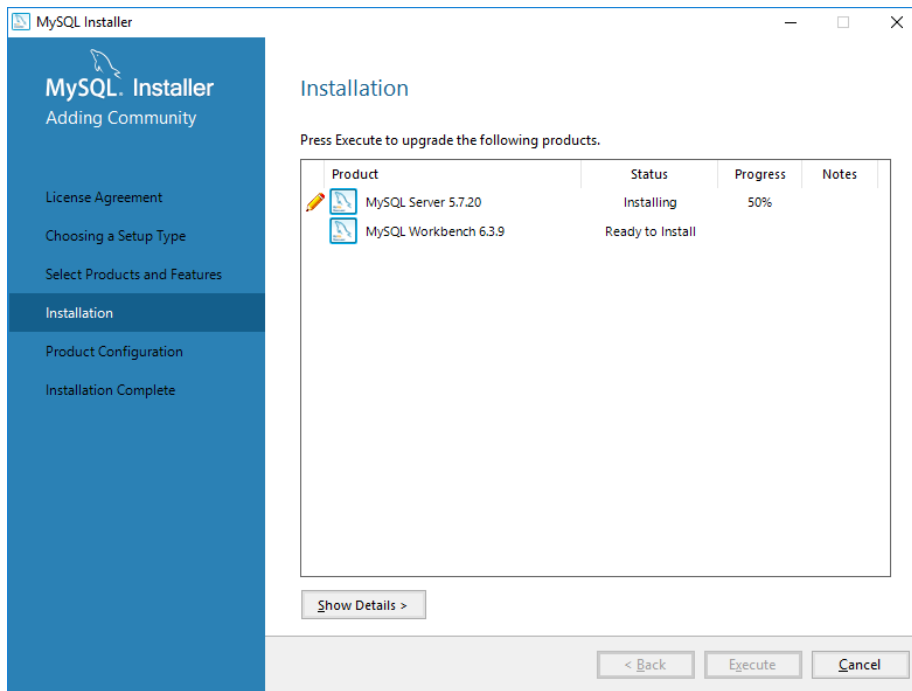
Alegeti de tip “Custom” – nu este necesara instalarea a tuturor produselor

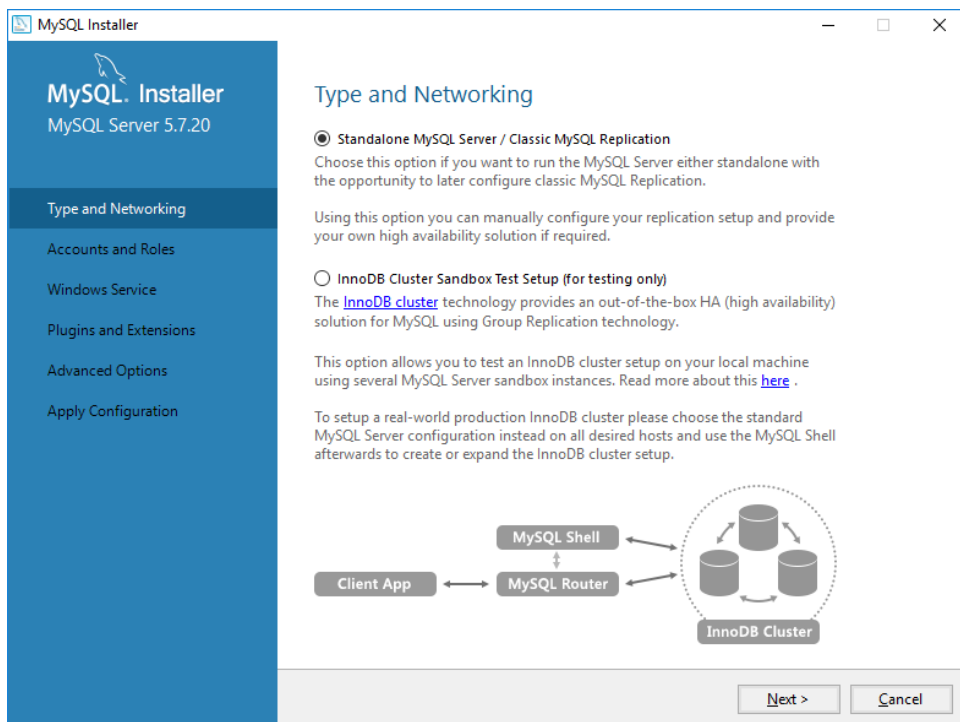
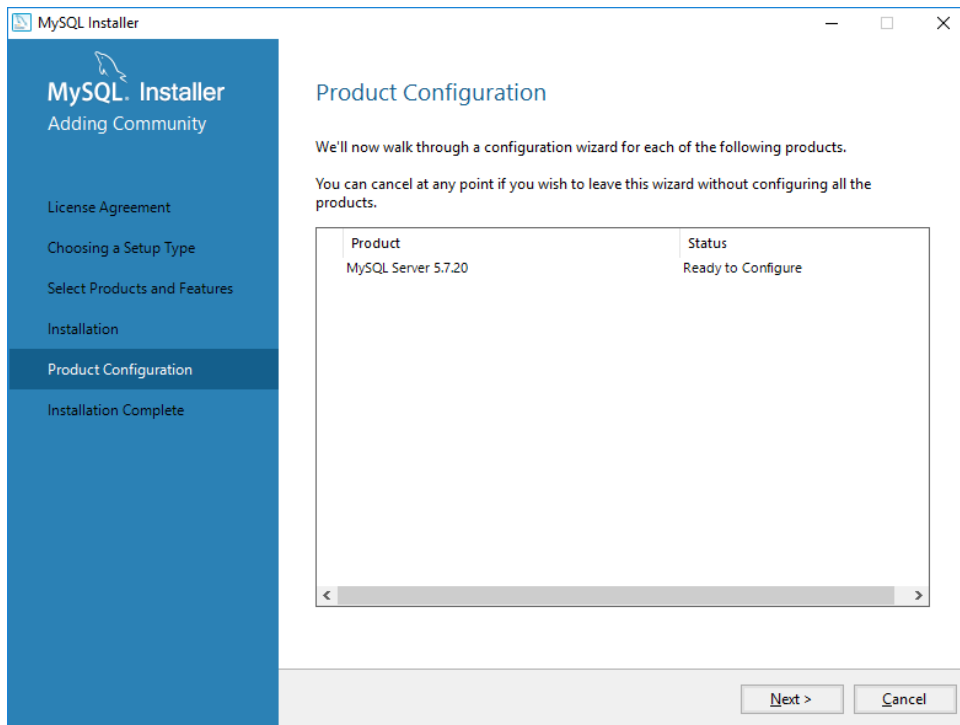


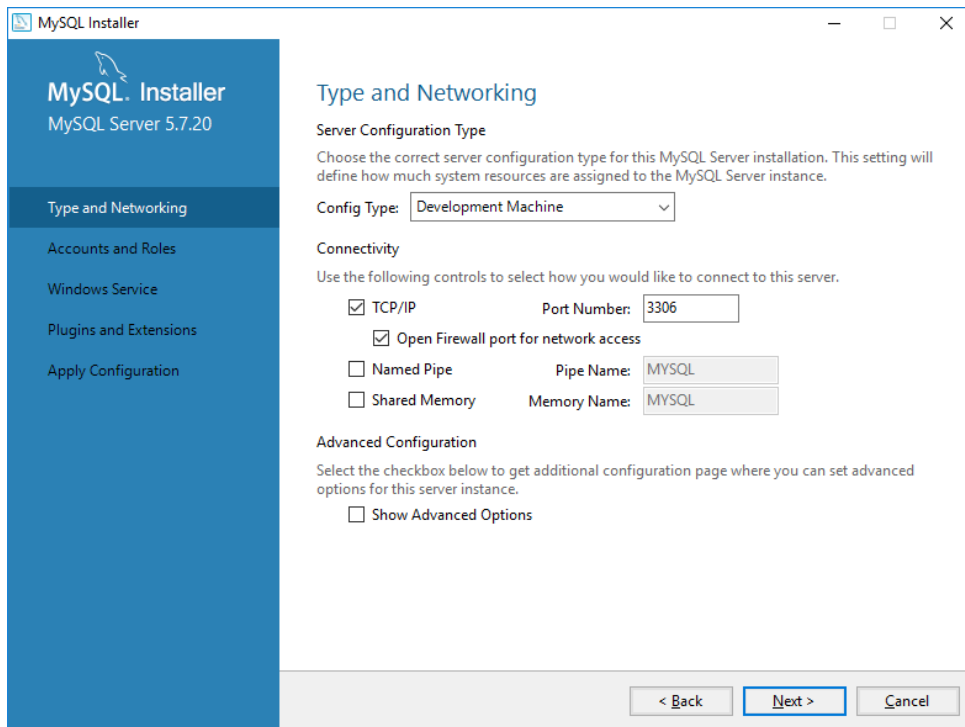
## Alegeți MySQL Server și MySQL WorkBench



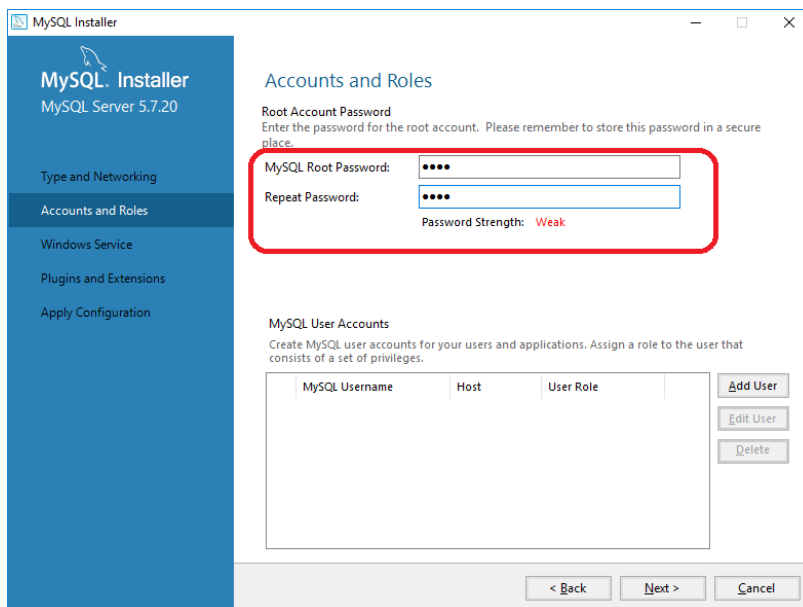
Click pe butonul “Execute”



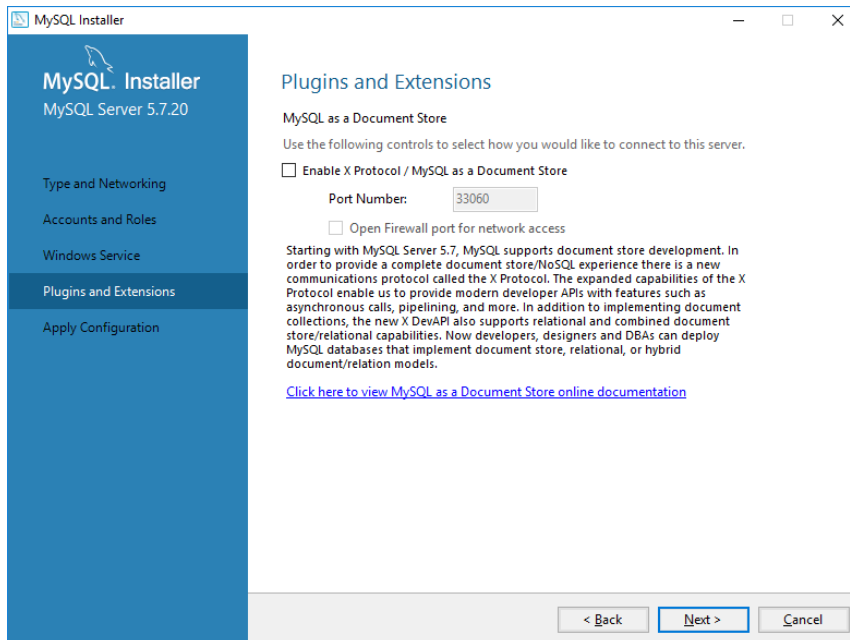
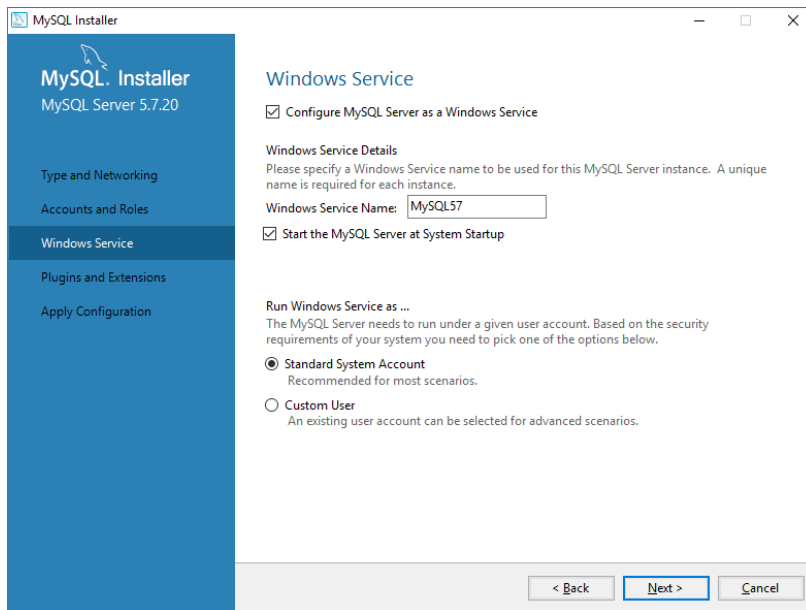


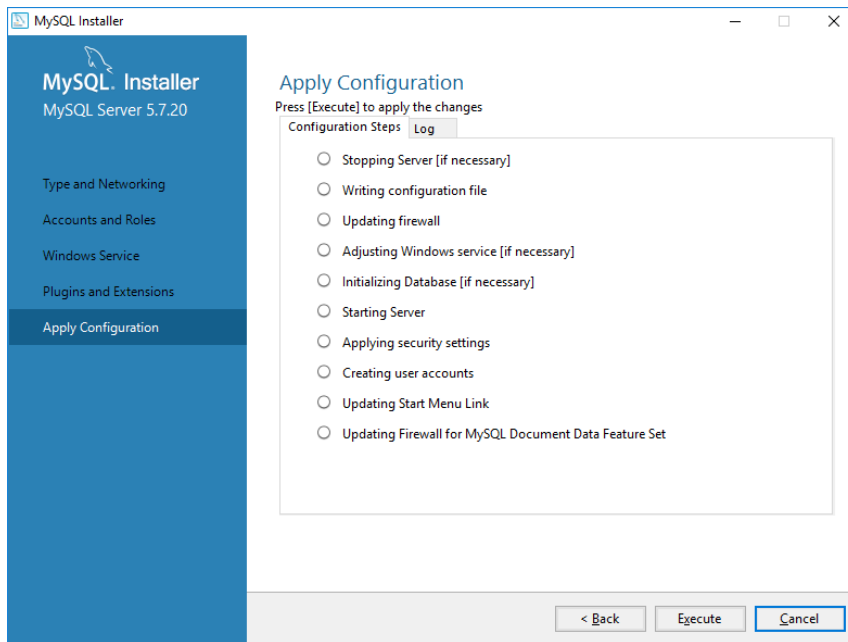


Alegeti o parola simpla, de exemplu eu am ales “root”

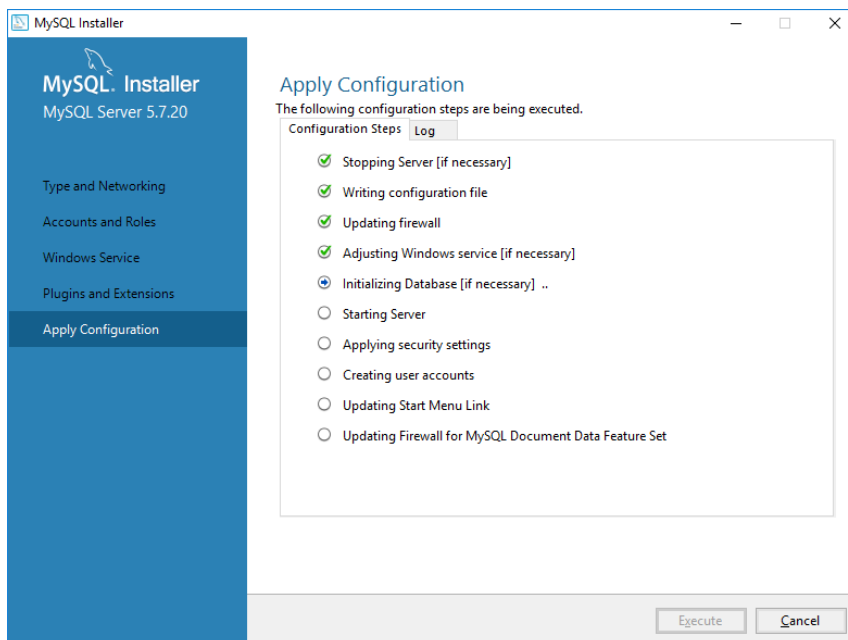


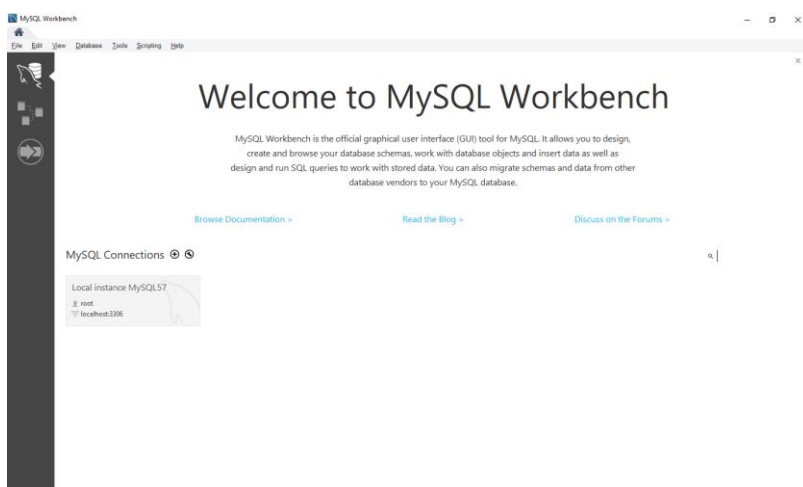
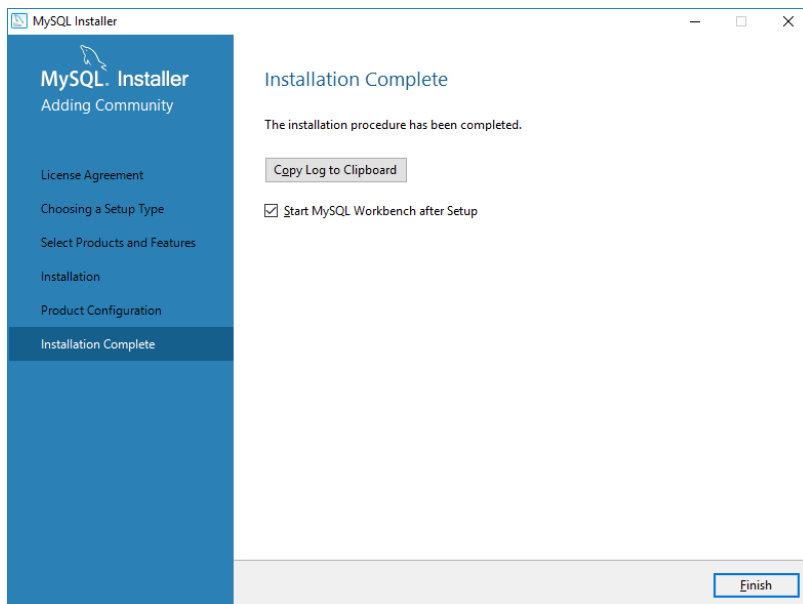
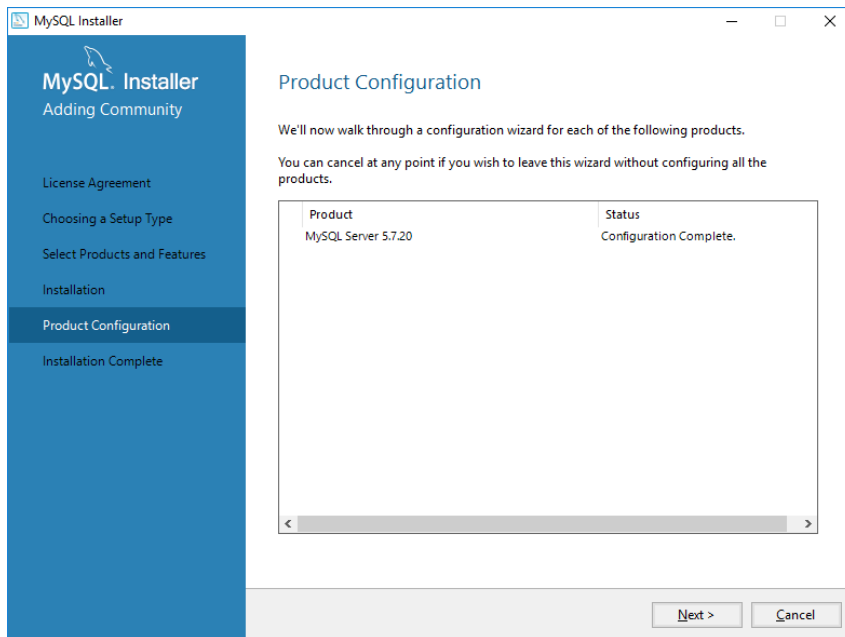






Click pe butonul “Execute”





Click pe “Local instance MySQL > root”. Vi se va cere parola:

